PROGRAMMER EN MQL4

Apprenez à automatiser vos stratégies sur MetaTrader 4

Veuillez excuser par avance les nombreuses erreurs qui subsistent probablement dans les pages suivantes malgré de nombreuses relectures et corrections. L'éditeur et l'auteur n'endossent par conséquent aucune responsabilité pour l'exactitude ou la précision des concepts étudiés dans ce livre et ne pourront pas être tenus pour responsables de toutes pertes éventuelles fictives ou réelles, que l'utilisation des informations contenues dans ce livre pourraient engendrer.

La programmation étant ce qu'elle est – le reflet de la façon de penser du programmeur -, il est fort probable qu'au fur et à mesure voire même au début de votre apprentissage, vous imaginiez des algorithmes (suite d'instructions, qui une fois exécutée correctement, conduit à un résultat donné) ou des méthodes plus simples pour arriver au même résultat que les codes qui figurent dans ce livre. Si tel est le cas, ce livre aura déjà rempli l'un de ses objectifs : vous inciter à utiliser les concepts du livre dans d'autres situations ou de manière différente.

« MetaTrader 4 », « MetaTrader », « MQL4 », « MQL » sont des marques de commerce de MetaQuotes Software Corp. MetaQuotes Software Corp n'est en aucun cas associé à cet ouvrage et n'endosse aucune responsabilité en relevant.

Pour obtenir plus d'informations sur ce livre ainsi que des mises à jour et les codes sources de l'expert consultant et de l'indicateur, rendez-vous sur http://www.eole-trading.com.

Pour contacter l'auteur, vous pouvez écrire à henri@eole-trading.com

ISBN : 978-1-4477-1407-1

Sommaire

PRÉFACE **11**

INTRODUCTION **12**

PLATEFORME 12

CONCEPT DE RENVOI 13

ICÔNES ET CONVENTIONS DE RÉDACTION 13

TERMINOLOGIE ANGLOPHONE 13

1 LES TYPES DE DONNÉES **14**

LES ENTIERS (INT) 14

LES RÉELS (DOUBLE) 15

LES BOOLÉENS (BOOL) 15

LES CHAÎNES DE CARACTÈRES (STRING) 16

LA COULEUR (COLOR) 16

DATE ET HEURE (DATETIME) 17

2 LES INSTRUCTIONS **19**

3 LES VARIABLES ET LES CONSTANTES **20**

DÉCLARER UNE VARIABLE 20

AFFECTATION ET INITIALISATION 22

LOCALES, GLOBALES OU EXTERNES 22

4 EXPRESSIONS ET OPÉRATEURS **25**

OPÉRATEURS ARITHMÉTIQUES 26

OPÉRATEURS D'INCRÉMENTATION ET DE DÉCRÉMENTATION 27

OPÉRATEURS D'AFFECTATION 28

OPÉRATEURS DE COMPARAISON 29

OPÉRATEURS LOGIQUES 30

OPÉRATEURS AU NIVEAU DU BIT 31

OPÉRATEUR VIRGULE 32

OPÉRATEUR D'APPEL DE FONCTION ET OPÉRATEUR RETURN 33

OPÉRATEURS D'INDEXATION DE TABLEAU 33

LA PRIORITÉ DES OPÉRATEURS 33

5 LES FONCTIONS 35
Définition et déclaration 35
Appel de fonction 37
Opérateur return 38
Imbriquer des fonctions 39
Fonctions spéciales 40
Fonction spéciale int init() 40
Fonction spéciale int start() 40
Fonction spéciale int deinit() 40
6 LES INSTRUCTIONS CONDITIONNELLES 43
Instruction conditionnelle if… 43
Instruction conditionnelle if… else 44
Multiples instructions if… et if… else 44
Instruction conditionnelle switch 45
7 LES BOUCLES 47
Boucle for 47
Boucle while 49
8 PRÉPROCESSEUR 51
Directive #define 51
Directive #property 52
Directive #include 53
Directive #import 53
9 STRUCTURE 55
10 COMMENTAIRES 57
Commentaires sur une seule ligne 57
Commentaires multi-lignes 57
Commentaires dans « Votre premier expert consultant » 58
11 VOTRE PREMIER EXPERT CONSULTANT #0 : PRÉPARER LE TERRAIN 59
12 VOTRE PREMIER EXPERT CONSULTANT #1 : CRÉATION DU FICHIER 61
Lancer MetaEditor 61
Créer un nouvel expert consultant 62
Compiler son expert 63
13 VOTRE PREMIER EXPERT CONSULTANT #2 : OBTENTION DU PLUS HAUT/BAS 66
14 FONCTIONS SUR LES BARRES 71

FONCTIONS BARS ET IBARS 71
FONCTIONS CLOSE[] ET ICLOSE() 73
FONCTIONS OPEN[] ET IOPEN() 75
FONCTIONS HIGH[] ET IHIGH() 75
FONCTIONS LOW[] ET ILOW() 76
FONCTIONS TIME[] ET ITIME() 77
FONCTIONS VOLUME[] ET IVOLUME() 78
FONCTIONS IHIGHEST() ET ILOWEST() 78

15 FONCTIONS TEMPORELLES 81
FONCTIONS DAY(), DAYOFWEEK() ET DAYOFYEAR() 81
FONCTIONS HOUR(), MINUTE(), SECONDS(), MONTH() ET YEAR() 82
FONCTIONS TIMEDAY(), TIMEDAYOFWEEK() ET TIMEDAYOFYEAR() 84
FONCTIONS TIMEHOUR(), TIMEMINUTE(), TIMESECONDS(), TIMEMONTH() ET TIMEYEAR() 85
FONCTIONS TIMELOCAL() ET TIMECURRENT() 86

16 VOTRE PREMIER EXPERT CONSULTANT #3 : FILTRE HORAIRE 88

17 FONCTIONS GRAPHIQUES 91
CRÉER UN OBJET 91
PARAMÉTRER UN OBJET 93
EXEMPLES 95

18 VOTRE PREMIER EXPERT CONSULTANT #4 : CRÉATION D'OBJETS GRAPHIQUES 101

19 FONCTIONS DE TRADING 104
FONCTION DE PASSAGE D'ORDRE 104
FONCTION DE GESTION DES ORDRES 106
OrderClose() 106
OrderCloseBy() 106
OrderDelete() 106
OrderModify() 107
FONCTIONS POUR SÉLECTIONNER ET OBTENIR DE L'INFORMATION SUR LES POSITIONS 108
Fonction de sélection 108
Fonctions pour obtenir de l'information sur les positions 109

20 VARIABLES PRÉDÉFINIES 112

21 VOTRE PREMIER EXPERT CONSULTANT #5 : FONCTIONS DE PASSAGE D'ORDRES 116

22 GESTION DES RISQUES 122

23 FONCTIONS DE CONVERSION 125

Fonction CharToStr() 125
Fonction DoubleToStr() 125
Fonction NormalizeDouble() 126
Fonction StrToInteger() 126
Fonction StrToDouble() 126
Fonction StrToTime() 127
Fonction TimeToStr() 127

24 LA GESTION DES ERREURS 129

25 FONCTIONS COMMUNES 133

Fonction void Alert() 133
Fonction void Comment() 134
Fonction int GetTickCount() 134
Fonction double MarketInfo() 135
Fonction int MessageBox() 135
Fonction void PlaySound() 138
Fonction void Print() 139
Fonction bool SendFTP() 139
Fonction void SendMail() 140
Fonction void Sleep() 141

26 FONCTIONS DE VÉRIFICATION 142

Fonction int GetLastError() 142
Fonction bool IsConnected() 142
Fonction bool IsDemo() 142
Fonction bool IsDllsAllowed() 143
Fonction bool IsExpertEnabled() 143
Fonction bool IsLibrariesAllowed() 144
Fonction bool IsTesting() 144
Fonction bool IsOptimization() 145
Fonction bool IsVisualMode() 145
Fonction bool IsStopped() 145
Fonction bool IsTradeAllowed() 146
bool IsTradeContextBusy() 146

27 VOTRE PREMIER EXPERT CONSULTANT #6 : GESTION DU RISQUE ET DES ERREURS 147

28 STOP SUIVEUR ET SEUIL DE RENTABILITÉ 153

Stop suiveur 153
Seuil de rentabilité 155

29 NUMÉRO MAGIQUE 157

30 VOTRE PREMIER EXPERT CONSULTANT #7 : STOP SUIVEUR 158

31 VARIABLES GLOBALES AU NIVEAU DE LA PLATEFORME 162

FONCTION DATETIME GLOBALVARIABLESET() 162

FONCTION BOOL GLOBALVARIABLECHECK() 162

FONCTION DOUBLE GLOBALVARIABLEGET() 163

FONCTION BOOL GLOBALVARIABLEDEL() 163

FONCTION STRING GLOBALVARIABLENAME() 163

FONCTION BOOL GLOBALVARIABLESETONCONDITION() 164

FONCTION INT GLOBALVARIABLESDELETEALL() 164

FONCTION INT GLOBALVARIABLESTOTAL () 164

EXEMPLES D'APPLICATION DES VARIABLES GLOBALES 165

Premier exemple 165

Deuxième exemple 166

32 VOTRE PREMIER EXPERT CONSULTANT #8 : VARIABLES GLOBALES 168

33 LE TRAITEMENT DES TABLEAUX EN MQL4 174

ACCÈS AUX DONNÉES D'UN TABLEAU 175

INITIALISATION D'UN TABLEAU 176

FONCTIONS DE TRAITEMENT DE TABLEAUX 176

Fonction int ArrayBsearch() 176

Fonction int ArrayCopy() 177

Fonction int ArrayCopyRates() 178

Fonction int ArrayCopySeries() 178

Fonction int ArrayDimension() 179

Fonction int ArrayGetAsSeries() 179

Fonction int ArrayInitialize() 179

Fonction int ArrayIsSeries() 179

Fonction int ArrayMaximum() 180

Fonction int ArrayMinimum() 180

Fonction int ArrayRange() 181

Fonction int ArrayResize() 181

Fonction bool ArraySetAsSeries() 182

Fonction int ArraySize() 182

Fonction int ArraySort() 182

CALCUL DE FRÉQUENCES RELATIVES 183

34 LES INDICATEURS 186

LES INDICATEURS PERSONNALISÉS 186
Fonction void IndicatorBuffers() 189
Fonction int IndicatorCounted() 189
Fonction void IndicatorDigits() 189
Fonction void IndicatorShortName () 189
Fonction void SetIndexStyle() 190
Fonction void SetIndexArrow() 191
Fonction bool SetIndexBuffer() 192
Fonction void SetIndexDrawBegin() 192
Fonction void SetIndexEmptyValue() 192
Fonction void SetIndexLabel() 192
Fonction void SetIndexShift() 193
Fonction void SetLevelValue() 193
Fonction void SetLevelStyle() 193
INTÉGRATION INDICATEUR-EXPERT 196

35 LES FONCTIONS MATHÉMATIQUES 198

36 LA GESTION DES FICHIERS 199
LOCALISATION DES FICHIERS 199
FONCTION DE GESTION DES FICHIERS 200
Fonction int FileOpen() 200
Fonction int FileOpenHistory() 200
Fonction int FileReadArray(), double FileReadDouble(), int FileReadInteger(), double FileReadNumber() et string FileReadString() 201
Fonction int FileWrite(), int FileWriteArray(), int FileWriteDouble(), int FileWriteInteger() et int FileWriteString() 201
Fonction void FileClose() 203
Fonction void FileDelete() 203
Fonction void FileFlush() 203
Fonction bool FileIsEnding() 203
Fonction bool FileIsLineEnding() 203
Fonction bool FileSeek() 204
Fonction int FileSize() 204
Fonction int FileTell() 204
EXEMPLES 204
Écriture de données diverses dans un fichier 205
Obtenir et utiliser un fichier depuis Internet 206

37 LES FONCTIONS SUR LES CHAÎNES DE CARACTÈRES 211

Fonction string StringConcatenate() 211
Fonction int StringFind() 211
Fonction int StringGetChar() 212
Fonction int StringLen() 213
Fonction string StringSetChar() 213
Fonction string StringSubstr() 213
Fonction string StringTrimLeft() et string StringTrimRight() 214

38 FONCTIONS SUR LES FENÊTRES 215

Fonction int Period() 215
Fonction bool RefreshRates() 215
Fonction string sPaire() 215
Fonction int iFenetresTotal() 215
Fonction void HideTestIndicators() 216
Fonction int iFenetreBarsPerChart() 216
Fonction string WindowExpertName() 216
Fonction int iFenetreFind() 216
Fonction int iFenetreFirstVisibleBar() 216
Fonction int iFenetreHandle() 217
Fonction bool WindowIsVisible() 217
Fonction int iFenetreOnDropped() 217
Fonction double WindowPriceOnDropped(), datetime WindowTimeOnDropped(), int iFenetreXOnDropped() et int WindowYOnDropped() 218
Fonction double WindowPriceMax() et double WindowPriceMin() 218
Fonction void WindowRedraw() 218
Fonction bool WindowScreenShot() 219

39 VOTRE PREMIER EXPERT CONSULTANT #9 : INCORPORER INDICATEUR ET CALENDRIER. 220

40 PROTÉGER SON CODE 227

Protection par mot de passe 227
Protection temporelle 228
Protection par numéro et type de compte 228

41 CONCLUSION 229

MQL5? 229
Qu'est-ce que le futur nous réserve? 229
Le mot de la fin 231
Quelques sites internet 231
Pour contacter l'auteur : 231

ANNEXE A **232**

ANNEXE B **236**

INDEX DES TABLEAUX **243**

INDEX DES FIGURES **244**

Préface

Vous avez entre les mains le premier manuel MQL4 en français digne de ce nom.

Cette bible est pour vous une porte d'entrée incroyable dans le monde fascinant du trading automatique.

Avec l'explosion du marché du forex pour les particuliers et la révolution Metatrader 4 qui a rendu accessible à tous le trading automatisé, jamais il ne vous a été aussi facile de déléguer vos prises de positions à des systèmes de programmation.

Vous vous demandez sans doute pourquoi sortir un manuel sur le MQL4 alors que le MQL5 fait déjà son apparition ?

Et bien rassurez-vous, le MQL4 a encore de beaux jours devant lui car il restera la façon la plus simple pour les néophytes et les apprentis trader d'automatiser des stratégies.

En lisant ce manuel, vous pourrez sans connaissance particulière et grâce à tous les exercices pratiques fournis et une approche pédagogique pertinente, réaliser vos premiers experts consultants et entrer dans le monde fantastique du trading automatisé.

Mes derniers mots seront pour l'auteur, trader passionné, ami et co-fondateur de ce qui deviendra Eole Trading.

Henri Baltzer, comme vous le découvrirez dans ce livre, ne triche pas. Il a su rendre simple ce qui ne l'est pas.

Il a su vous ouvrir un monde où tout est possible, où vous pourrez backtester, évaluer et mettre en œuvre vos idées les plus complexes.

Merci à Henri pour ce livre et le travail de longue haleine qu'il représente.

Bons Trades à tous !

Jérôme Revillier

Trader, Stratégiste et Directeur d'Eole Trading.

http://www.revillierjerome.com

OFFRE SPÉCIALE RÉSERVÉE AUX LECTEURS

Offrez la puissance du trading automatique à vos investissements !

PROFITEZ DE 15% DE RÉDUCTIONSUR LE PACK START.

Rendez vous sur :

http://www.eole-trading.com

Et indiquez le code

« MQL4 »

Pour plus d'informations et détails, contactez

infos@eole-trading.com ou appelez le +33 (0)5 62 247 524

Introduction

Ce « guide de programmation en MQL4 », loin d'être une encyclopédie complète sur le langage MQL4, se veut une introduction au monde du trading automatisé et de la programmation en MQL4 pour tous les non initiés qui veulent en savoir plus sur le sujet. Afin de faciliter l'apprentissage, ce livre a été conçu avec un objectif en tête : programmer « votre premier expert consultant ». Pour ce faire, vous ne trouverez pas comme dans certains livres toute la théorie en premier suivie d'exemples mais juste ce qu'il faut de théorie pour pouvoir débuter avec au fur et à mesure des chapitres, des exemples pratiques permettant d'introduire de nouveaux concepts avec les compléments théoriques nécessaires.

Cette façon de procéder devrait aider le lecteur à mieux visualiser comment utiliser certains concepts qui peuvent parfois sembler abstraits au premier abord et ainsi l'aider à appliquer ces concepts dans d'autres situations ou de manière différente. Avant de vous plonger dans l'univers du MQL4, il convient de préciser certains aspects.

Plateforme

Au fil des chapitres, nous ferons souvent référence à la plateforme MetaTrader et à certaines de ses fenêtres. Voici une figure vous permettant de visualiser cette dernière avec le nom associé à chaque zone.

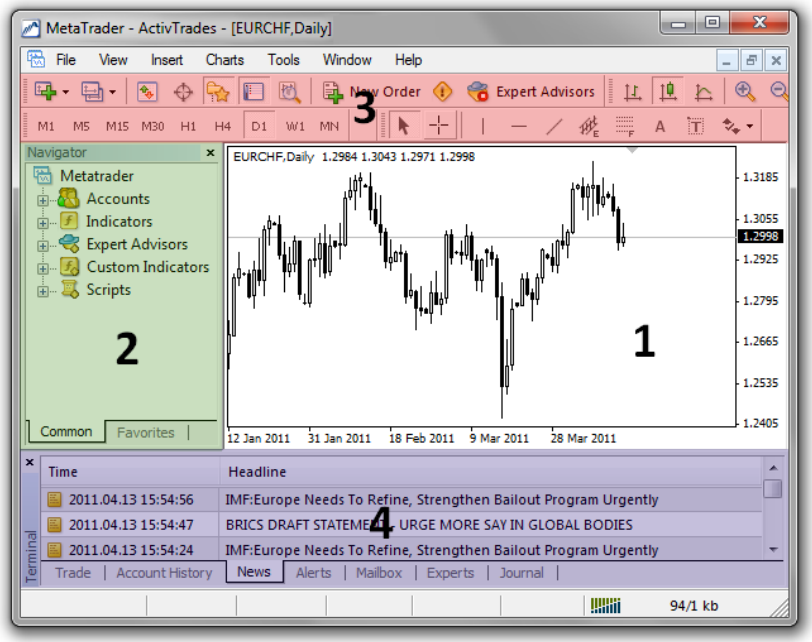

Figure 1 : Identification des zones de la plateforme MetaTrader 4

1 : Fenêtre graphique
2 : Fenêtre navigateur
3 : Barre d'outils
4 : Fenêtre Terminal (Onglets : Trade, Historique, Nouvelles, Alertes, Courrier, Experts et Journal)

Concept de renvoi

Le verbe *renvoyer* en programmation fait référence à l'obtention d'un résultat suite à l'exécution d'une fonction ou instruction : « la fonction renvoie le résultat au programme ou à l'utilisateur ».

Icônes et conventions de rédaction

Cette icône indique une remarque ou particularité à laquelle vous devriez porter attention.

Cette icône indique une procédure ou astuce.

Cette icône indique une précision ou explication supplémentaire sur le sujet de la section.

Le code source sera toujours présenté dans un cadre sous la forme suivante :

```
Code source
Code source mis à jour
```

Les guillemets « » sont utilisés pour identifier un élément particulier (nom, caractère) auquel le paragraphe fait référence. Tous les éléments référant au langage MQL4 ou à la plateforme (incluant les références à Windows) et présents dans le texte même (en dehors d'un cadre de code source) seront en *italique*.

Terminologie anglophone

En général, les termes anglophones ont tous été traduits dans leurs équivalents français (par exemple, « Expert Advisor » est devenu « Expert Consultant »).

1
Les types de données

Le MQL4, en tant que langage de programmation dérivé du C/C++, est par définition un langage « typé ». Cela signifie que le programme qui exécutera votre code ne peut interpréter et utiliser que des données appartenant à un type de données spécifiques et que certaines opérations ne peuvent être effectuées qu'entre données du même type. Il est donc primordial, comme nous le verrons par la suite, de définir un type lors de la création d'une nouvelle donnée par le biais d'une variable, constante ou fonction.

En MQL4, nous disposons des types suivants (le mot en caractère gras est le mot-clé utilisé pour désigner le type de la donnée dans le code source)

int	Nombre entier
double	Nombre réel
bool	Booléen
string	Chaîne de caractère
color	Couleur
datetime	Date et heure

Tableau 1 : Les types de données

Les entiers (int)

L'abréviation *int* vient du terme anglais *integer* dont la traduction littérale en français est *entier*. Le type *int* définit donc les nombres entiers. Ces nombres peuvent être négatifs ou positifs et ne comportent pas de décimales.

Pour indiquer si un nombre est négatif ou positif en MQL4, il suffit d'ajouter le signe « - » ou « + » devant le nombre. Ajouter le signe « + » n'est pas obligatoire compte tenu que, par défaut, tous les nombres sont considérés comme positifs.

L'intervalle de définition en MQL4 est [-2147483648 ; 2147483647]. Toute valeur en dehors de cet intervalle ne pourra être interprétée par le programme altérant de ce fait le fonctionnement de votre programme.

Le MQL4 est à même d'interpréter les valeurs entières au format décimal (base 10) ou encore hexadécimal (base 16).

Tous les entiers suivants sont valides : **2** | **3** | **4** | **-6** | **-78** | 0Xa3 | 0x7C7

Vous noterez aisément qu'il est plus facile d'utiliser le format décimal (en gras) que le format hexadécimal.

En MQL4, ce type de donnée est utilisé pour des quantités ou actions qui ne peuvent être fractionnées. Par exemple, le nombre de tentatives d'ouverture d'une position (vous allez demander au programme d'essayer 1 fois, 2 fois ou même 300 fois mais jamais 2.3 fois).

Un nombre entier ne peut commencer par 0. Vous ne verrez donc jamais une variable dont la valeur est **012345** mais bien **12345**.

Les réels (double)

Le type *double* définit le type des nombres réels. Un nombre réel est donc composé d'un nombre entier suivi d'un point et de chiffres après ce dernier. L'intervalle de définition en MQL4 est [-1.7 * e-308 ; 1.7 * e308].

Tous les réels suivant sont valides en MQL4 : 1.5000 | 1.35 | -6.54 | -78.00, | 32

Le dernier réel de notre exemple est 32 pour démontrer que vous pouvez utiliser le type *double* pour des entiers car un entier est tout simplement un réel dont les décimales sont une suite de 0. Dans notre cas, le programme interprétera 32 comme étant 32.000000…

Ce type est donc très utile lorsqu'il s'agit d'utiliser le cours d'une paire de devises ou encore désigner un volume en lots fractionnés.

MetaTrader utilise la notation américaine et par conséquent, contrairement au système européen qui utilise la virgule pour indiquer la séparation entre les entiers et les décimales, ici, vous devrez utiliser un point.

Les booléens (bool)

Les données de type *bool* ne peuvent avoir théoriquement que deux valeurs : *True* ou *False*. Dans la pratique, cela fait que l'on ne peut attribuer (ou qu'une variable booléenne peut prendre) que les valeurs suivantes : **TRUE, True, true, 1, FALSE, False, false ou 0**.

Toutes ces valeurs sont correctes et interprétables par le programme. Le 0 et le 1 viennent du système binaire (1 pour vrai et 0 pour faux).

MetaTrader interprétera dans le cas d'une variable booléenne tout nombre différent de 0 comme correspondant à l'état vrai également.

Ce type de variable est très utile lors du cheminement logique d'un programme afin de définir quelle action entreprendre lorsqu'une condition est vérifiée ou non.

Les chaînes de caractères (string)

La chaîne de caractère permet, comme son nom l'indique, de stocker une suite de caractères.

Ce type vous permet donc de stocker des mots ou phrases. Lorsqu'ils sont précédés par un antislash « \ », certains caractères ont une fonction de mise en page comme vous pouvez voir dans le tableau ci-dessous. S'agissant de caractères, il est important de les encadrer par des guillemets « " » pour que MetaTrader puisse les interpréter.

\\	Pour écrire le caractère « \ »
\n	Retour à la ligne
\t	Tabulation
\"	Pour écrire le caractère « " »
\'	Pour écrire le caractère « ' »
\r	Saut de ligne

Tableau 2 : Caractères spéciaux

La valeur de vos variables de type *string* ne peut pas excéder 255 caractères.

La couleur (color)

Ce type de donnée est propre au MQL4. Le type *color* vous permet donc de stocker une couleur qui sera utilisée par le programme lors de l'affichage d'une donnée, phrase, ligne ou objet graphique.

Vous avez trois possibilités lorsque vient le moment de préciser la couleur choisie :

1. En utilisant son nom tiré du tableau des couleurs internet dont un échantillon est fourni à la page suivante (voir tableau 3 à la page suivante).
2. En utilisant l'intensité des 3 couleurs (RGB). RGB vient de Red Green Blue - en mélangeant ces 3 couleurs en augmentant plus ou moins la quantité de chacune

d'entre elles, il est possible de créer toutes les couleurs que l'on désire. On obtient donc des codes de couleur ressemblant à ceci : **C'0×00 | 0×00 | 0xFF'**

3. En utilisant la valeur entière. Encore une fois, vous pouvez écrire cette valeur en décimal ou en hexadécimal. On obtient donc des couleurs du type : **16777215** ou **0xFFFFFF** (la couleur blanc respectivement en décimale puis hexadécimale).

Comme vous pouvez vous en rendre compte, il est relativement plus aisé d'utiliser la table des couleurs internet à moins de vouloir une couleur spécifique ne figurant pas sur ce tableau. Dans ce dernier cas, de nombreux logiciels de traitement d'image pourront vous fournir les codes nécessaires sans compter les nombreux outils disponibles sur internet.

Date et heure (datetime)

Le type *datetime* vous permet de stocker des informations au format date ou heure.

Nous utiliserons ce type de variable pour définir une date ou une heure pour initialiser le programme par exemple. Vous pouvez travailler aussi bien en seconde qu'en année avec ce type.

La particularité de ce type de donnée est que la valeur est calculée en secondes écoulées depuis le 1er janvier 1970 à minuit et ne peut aller plus loin que le 31 décembre 2037. Rassurez-vous, MetaTrader est bien fait et nous n'aurons jamais besoin de calculer nous-mêmes le nombre de seconde lorsque nous désirons utiliser une variable de ce type.

Black	DarkGreen	DarkSlateGray
Maroon	Indigo	MidnightBlue
SeaGreen	DarkGoldenrod	DarkSlateBlue
LightSeaGreen	DarkViolet	FireBrick
Goldenrod	MediumSpringGreen	LawnGreen
DarkOrange	Orange	Gold
DeepSkyBlue	Blue	Magenta
LightSlateGray	DeepPink	MediumTurquoise
IndianRed	MediumOrchid	GreenYellow
MediumPurple	PaleVioletRed	Coral
DarkSalmon	BurlyWood	HotPink
Plum	Khaki	LightGreen
PaleGreen	Thistle	PowderBlue
Moccasin	LightPink	Gainsboro
LemonChiffon	Beige	AntiqueWhite
Lavender	MistyRose	OldLace
LavenderBlush	MintCream	Snow
Olive	Green	Teal
DarkBlue	DarkOliveGreen	SaddleBrown
Sienna	MediumBlue	Brown
MediumVioletRed	MediumSeaGreen	Chocolate
CadetBlue	DarkOrchid	YellowGreen
Yellow	Chartreuse	Lime
Red	Gray	SlateGray
DodgerBlue	Turquoise	RoyalBlue
MediumAquamarine	DarkSeaGreen	Tomato
CornflowerBlue	DarkGray	SandyBrown
Salmon	Violet	LightCoral
Aquamarine	Silver	LightSkyBlue
PaleGoldenrod	PaleTurquoise	LightGray
PeachPuff	Pink	Bisque
PapayaWhip	Cornsilk	LightYellow
WhiteSmoke	Seashell	Ivory
White	SkyBlue	LightSalmon
Navy	Purple	LightGoldenrod
ForestGreen	OliveDrab	LightCyan
DarkTurquoise	DimGray	Honeydew
Crimson	SteelBlue	BlanchedAlmond
LimeGreen	OrangeRed	Linen
SpringGreen	Aqua	AliceBlue
Peru	BlueViolet	LightSteelBlue
SlateBlue	DarkKhaki	Wheat
RosyBrown	Orchid	LightBlue
MediumSlateBlue	Tan	NavajoWhite

Tableau 3 : Tableau des couleurs internet

2
Les instructions

Un code source est une suite d'instructions écrites dans le même langage. Ces dernières permettent de définir une action à réaliser lors de l'exécution du programme. Par exemples : comptabiliser le nombre de barres, effectuer un calcul ou encore savoir quelle heure il est.

Chacune de ces instructions doit se terminer par un point-virgule « **;** » pour indiquer la fin de celle-ci au programme. En l'absence de point-virgule, MetaEditor renverra des erreurs de compilation et il vous sera impossible d'utiliser votre programme dans MetaTrader.

Il est également possible de définir des blocs d'instructions. Ces derniers sont utiles lorsque le programme doit effectuer plusieurs actions si une condition est remplie ou si les instructions font partie de la même fonction comme vous le verrez par la suite. Dans ce cas, le bloc d'instructions sera encadré par des accolades « **{** » et « **}** ».

Les accolades ne dispensent par pour autant de rajouter un point-virgule à la fin de chaque instruction présente à l'intérieur du bloc.

3
Les variables et les constantes

Les variables sont les objets élémentaires du langage MQL4. Une variable est constituée de deux parties distinctes : un nom et une valeur correspondante. La valeur attribuée est en fait une information temporaire stockée dans la mémoire de l'ordinateur à laquelle le nom de la variable fait référence.

Bien que le nom de la variable ne puisse être changé une fois que le programme a démarré, la valeur associée, elle, et c'est pourquoi ces objets sont appelés variables, peut être modifiée à tout moment du programme.

Les constantes, comme le nom l'indique, sont des objets invariables. Il peut s'agir d'un entier, d'un réel, d'un caractère ou encore d'une phrase. Comme le fonctionnement des constantes fait appel à la notion de préprocesseur que nous verrons par la suite, nous allons volontairement sauter l'étude de ces dernières pour l'instant et nous concentrer sur les variables.

Déclarer une variable

La première étape pour créer une variable consiste à déclarer cette dernière. La déclaration est le processus par lequel vous allez attribuer un nom à une variable tout en indiquant à quel type de donnée elle appartient.

Pour déclarer une variable, il faut donc au préalable lui choisir un nom. Il est d'usage de choisir un nom qui fait référence à la donnée que la variable est censée stocker. Le MQL4, tout comme le langage C est sensible à la police, il faut donc prêter attention aux minuscules et majuscules. Pour vous simplifier la tâche, prenez de bonnes habitudes dès le début en utilisant des normes pour nommer vos variables.

Libre à vous de choisir les normes qui vous conviennent le plus mais vous trouvez ci-dessous une méthode pour nommer vos variables qui s'avère très pratique pour les débutants.

- Inclure un préfixe à vos noms de variables correspondant au type de la variable. De cette façon, lorsque vous utilisez la variable à l'intérieur de votre code source, vous pouvez facilement savoir de quel type de donnée il s'agit et éviter ainsi des erreurs de programmation que nous aborderons par la suite.

- Inscrire la première lettre du nom en majuscule afin de bien le différencier du type.

- Si le nom de la variable est composé de plusieurs mots, inscrire chaque première lettre en majuscule afin de rendre la lecture du code plus facile.

Nous aurions donc par exemple (type de donnée à gauche et exemple de nom à droite) :

int	iDateDuJour
double	dPrix
bool	bVerification
string	sNomExpert
color	cCouleur
datetime	dtDate
GlobalVariable	gvVariableGlobale

Tableau 4 : Comment nommer les variables

Essayer de toujours choisir des noms pertinents ni trop court ni trop long car le MQL4 impose un maximum de 31 caractères pour le nom des variables.

Vous ne pouvez pas non plus choisir n'importe quel nom - en effet, certains noms sont déjà réservés pour des usages spécifiques et nommer une variable de la sorte provoquerait un conflit au sein de votre programme. Vous trouverez ci-dessous la liste de tous les mots réservés que vous ne pouvez utiliser à d'autres fins que celles déjà prescrites par le langage.

Bool	color	datetime	int
String	void	Extern	static
Break	case	continue	default
Else	for	If	return
Switch	while	False	true
FALSE	TRUE	False	True

Tableau 5 : Mots réservés

Le MQL4 est sensible à la casse. « return » n'est donc pas la même chose que « Return » pour MetaEditor.

En résumé, pour nommer vos variables et constantes, vous devez suivre les règles suivantes :

1. Le nom ne doit pas commencer par un chiffre.
2. Le seul caractère spécial autorisé est « _ ».
3. Le nom ne doit pas excéder 31 caractères.
4. Ne pas utiliser les noms réservés.
5. Faire attention à la casse.
6. Inclure un préfixe en minuscule indiquant le type de la variable.

7. Première lettre de chaque mot composant le nom en majuscule.
8. Utiliser des noms pertinents.

Maintenant que vous avez vu comment nommer une variable, voyons comment faire pour la déclarer.

Pour déclarer une variable, la syntaxe est la suivante :

```
type tNomVariable;

// type est à remplacer par le type de la variable (int, double, string, bool, color ou datetime)
// t correspond à l'abréviation du type de la variable choisi (i, d, b, s, c ou dt)
```

Affectation et initialisation

Les deux opérations (affectation et initialisation) aboutissent au même résultat - on attribue une valeur à une variable. La différence réside dans le fait que l'affectation se produit au sein du programme tandis que l'initialisation se produit au moment de la déclaration.

Pour une affectation, la syntaxe est la suivante :

```
tNomVariable = valeur;
```

On remarquera que l'on n'a pas, comme lors de la déclaration, à préciser le type de la variable car celle-ci est censée avoir déjà été déclarée au préalable.

Pour une initialisation, la syntaxe sera donc la suivante :

```
type tNomVariable = valeur;
```

Initialiser une valeur signifie simplement que l'on affecte une valeur au moment de sa déclaration.

Locales, globales ou externes

Une variable peut être locale, globale ou externe. Nous allons passer chaque catégorie en revue afin de bien comprendre les différences et leurs implications.

Une variable locale est une variable qui ne peut être interprétée que par une portion spécifique du code. Visualisez le code de votre programme comme une suite d'instructions indépendantes les unes des autres. Chacune de ces instructions peut utiliser ses propres variables pour son fonctionnement. C'est à cela que l'on fait référence lorsque l'on parle de variable locale. Si vous déclarez une variable à l'intérieur d'un bloc d'instruction, celle-ci ne pourra pas être utilisée en dehors de ce bloc.

Globale car à la différence de locale, les variables de ce type peuvent jouer un rôle dans toutes les instructions du programme aussi indépendantes qu'elles soient. Toutes les instructions peuvent faire appel à la valeur de cette variable ou la modifier. Par défaut, une variable globale est déclarée en début de programme et a la valeur 0 à moins d'être initialisée.

En MQL4, vous verrez souvent dans la déclaration de variable, une syntaxe comme suit :

```
extern type tNomVariable = valeur;
```

Le mot *extern* sert à identifier les variables que l'utilisateur pourra modifier à partir des propriétés de l'expert consultant, indicateur ou script une fois le programme lancé dans MetaTrader.

Vous voyez dans l'image ci-dessous, la déclaration de 3 variables externes pour l'indicateur Ichimoku disponible par défaut avec MetaTrader.

```
extern int Tenkan=9;
extern int Kijun=26;
extern int Senkou=52;
```

Figure 2 : Variables externes de l'indicateur Ichimoku Kinko Hyo

Une fois dans la plateforme, si vous installez l'indicateur sur un graphique, la fenêtre suivante devrait s'ouvrir afin de vous permettre de paramétrer l'indicateur.

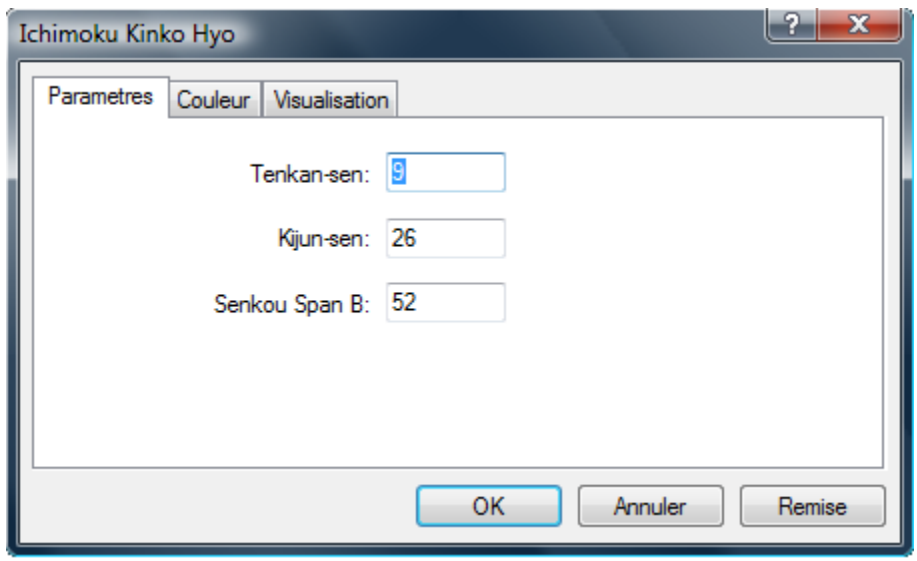

Figure 3 : Paramètres de l'indicateur Ichimoku Kinko Hyo

Les trois variables *extern* se retrouvent donc dans la fenêtre de paramétrage et vous pouvez les modifier sans nécessité de toucher au code source.

Nous avons vu ici les fonctions globales au niveau du programme (la variable peut être utilisée et sa valeur modifiée tout au long du même programme). Nous verrons également dans un autre chapitre qu'il existe également des variables globales au niveau de la plateforme (tous les programmes peuvent utiliser et modifier la variable – c'est-à-dire que plusieurs experts pourraient utiliser la même variable).

4
Expressions et opérateurs

L'expression la plus simple qui soit est tout simplement une variable. En effet, une variable est une expression au même titre qu'un ensemble de variables, d'instructions, d'opérations ou de fonctions retournant une valeur. En fait, tout ce qui possède une valeur peut être interprété comme une expression. Une expression peut également être composée de plusieurs expressions également.

Par exemple, **variable = 3** est une expression au même titre que **x + y** en est une. Ou encore **x + y = z** ou tout simplement **4** qui est également une expression.

Pour combiner plusieurs expressions entre elles comme dans l'exemple **x + y = z** où **x**, **y** et **z** sont des expressions, vous allez utiliser des opérateurs comme **+** et **=**.

Pour fonctionner correctement, votre programme a besoin de recevoir des instructions - il s'agit de commandes ou d'ordres que le programme devra exécuter en temps voulu. Les instructions les plus courantes sont les instructions d'opérations, c'est à dire des expressions qui utilisent des opérateurs.

En MQL4, il existe plusieurs types d'opérateurs que l'on peut classer dans les catégories suivantes :

- Opérateurs arithmétiques
- Opérateurs d'incrémentation et de décrémentation
- Opérateurs d'affectation
- Opérateurs de comparaison
- Opérateurs logiques
- Opérateurs au niveau du bit
- Opérateurs virgule
- Opérateurs d'appel de fonction
- Opérateurs d'indexation de tableau

Nous allons maintenant voir en détail les opérateurs composant chacune de ces catégories et leurs usages.

Opérateurs arithmétiques

Opérateur	Description	Syntaxe	Exemple	Équivalent
+	Additionne X et Y	X + Y	iA = 2 + 4;	A = 6
-	Soustrait Y à X	X – Y	iA = 42 – 20;	A = 22
-	Inverse de X	-X	dA = -dB; où dB = 2.3	A = -2.3
*	Multiplie X par Y	X * Y	dA = 4.5 * 2.0;	A = 9.0
/	Divise X par Y	X / Y	iA = 9 / 3;	A = 3
%	Renvoie le reste entier de la division de X par Y	X % Y	iA = 27 % 4;	A = 3 car 4*6 = 24 et 27 – 24 = 3

Tableau 6 : Opérateurs arithmétiques

La valeur renvoyée par une opération arithmétique peut être soit un entier soit un nombre réel. Nous allons utiliser la division « 5 / 2 » pour observer la valeur renvoyée par chacune des opérations suivantes :

Cas où les opérations sont insérées au sein d'une expression :

1) 5 / 2 = 2
2) 5.0 / 2.0 = 2.5
3) 5.0 / 2 = 2.5
4) 5 / 2.0 = 2.5

Cas où nous précisons une variable pour contenir le résultat des opérations :

5) int iA = 5.0 / 2.0 soit iA = 2
6) int iA = 5.0 / 2 soit iA = 2
7) int iA = 5 / 2.0 soit iA = 2
8) int iA = 5 / 2 soit iA = 2
9) double dB = 5.0 / 2.0 soit dB = 2.5
10) double dB = 5.0 / 2 soit dB = 2.5
11) double dB = 5 / 2.0 soit dB = 2.5
12) double dB = 5 / 2 soit dB = 2

Comme vous pouvez le voir, si la variable qui contiendra le résultat de l'opération est de type entier, alors peu importe la façon d'écrire les variables de l'opération, nous obtiendrons toujours une valeur entière. À l'inverse, dans le cas d'une variable de type réel, si l'opération n'est écrite qu'avec des entiers comme dans l'opération (12), alors le résultat sera également un entier malgré le type de la variable conteneur.

Il est donc primordial de s'assurer que la syntaxe choisie est correcte afin d'éviter d'éventuelles erreurs de calcul. Dans l'exemple ci-dessous, **dResultatCalcul = 200** tandis que **dResultatCalcul2 = 250**.

```
double dResultatCalcul = 100 * (5 / 2);
double dResultatCalcul2 = 100 * (5.0 / 2.0);
```

Les règles de calcul évoquées ci-dessus s'appliquent pour tous les opérateurs (« **+** », « **-** », « **/** », « ***** »).

Par exemple :

1) 5 + 2.5 = 7.5
2) int A = 5 + 2.5 soit A = 7
3) double B = 5 + 2.5 soit B = 7.5
4) 7 – 3.5 = 3.5
5) int A = 7 – 3.5 soit A = 3
6) double B = 7 – 3.5 soit A = 3.5
7) 2 * 2.32 = 4.64
8) int A = 2 * 2.32 soit A = 4
9) double B = 2 * 2.32 soit B = 4.64

L'opérateur « **+** » peut également servir à additionner des chaînes de caractères entre elles ou des chaînes de caractères avec des entiers ou réels. Le résultat d'une telle opération sera une nouvelle chaîne de caractères formée par les différentes variables.

```
int iDebutPhrase = 2;
string sDebutPhrase = "Deux";
string sFinDePhrase = " semaines";
string sNouvellePhrase1 = iDebutPhrase + sFinDePhrase;
string sNouvellePhrase2 = sDebutPhrase + sFinDePhrase;
```

On obtiendra donc pour sNouvellePhrase1, « **2 semaines** » et pour sNouvellePhrase2, « **Deux semaines** ».

Opérateurs d'incrémentation et de décrémentation

Opérateur	Description	Syntaxe	Exemple	Équivalent
++	Opérateur de post-incrémentation	X++	iA = iA++;	iA = iA + 1
--	Opérateur de post-décrémentation	X--	iA = iA--;	iA = iA – 1

Tableau 7 : Opérateurs d'incrémentation et de décrémentation

Ce type d'opérateur est souvent utilisé à l'intérieur d'une boucle permettant ainsi d'incrémenter (ajouter 1) ou décrémenter (soustraire 1) une variable à chaque exécution de la boucle.

Il n'est pas permis d'utiliser les opérateurs d'incrémentation et de décrémentation à l'intérieur d'une expression avec d'autres opérateurs. Cette caractéristique est illustrée dans l'exemple suivant.

L'expression ci-dessous est correcte.

```
int iA = 0;
iA++;
iNbrJours = iA * 7;
```

L'expression qui suit est incorrecte.

```
iNbrJours = (iA++) * 7;
```

Opérateurs d'affectation

Dans l'expression **X = Y + Z**, on affecte le résultat de l'expression **Y + Z** à **X**. C'est à dire que, dorénavant, **X** vaudra la valeur renvoyée par l'addition **Y + Z**. L'opérateur d'affectation utilisé est donc le signe **« = »**. En MQL4, il existe 6 opérateurs d'affectation.

Opérateur	Description	Syntaxe	Exemple	Équivalent
=	Affecte la valeur de l'expression à droite de l'opérateur à la variable située à gauche	Y = X;	iJour = 2 + 3;	iJour renverra la valeur 5 (2+3)
+=	On additionne la valeur de l'expression à droite de l'opérateur à la variable à gauche et on affecte cette nouvelle valeur à la variable de gauche	X += Y	iJour += 4; avec iJour = 5	L'opération effectuée sera donc iJour = 4 + 5 soit iJour = 9
-=	On soustrait la valeur de l'expression à droite de l'opérateur à la variable à gauche et on affecte cette nouvelle valeur à la variable de gauche	X -= Y;	iJour -= 4; avec iJour = 5	L'opération effectuée sera donc iJour = 4 – 5 soit iJour = -1
*=	On multiplie la variable à gauche de l'opérateur par l'expression à droite et on affecte cette nouvelle valeur à la variable de gauche	X *= Y;	iJour *= 2; avec iJour = -1	L'opération effectuée sera donc iJour = -1 * 2 soit iJour = -2
/=	On divise la variable à gauche de l'opérateur par l'expression à droite et on affecte cette nouvelle valeur à la variable de gauche	X /= Y;	dJour /= 4; avec dJour = -2	L'opération effectuée sera donc dJour = -2 / 4 soit dJour = -0.5
%=	On affecte le reste entier de la division de la variable à gauche de l'opérateur par l'expression à droite à la variable de gauche	X %= Y;	iJour %= 4; avec iJour = 15	L'opération effectuée sera donc iJour = 15 % 4 soit iJour = 3

Tableau 8 : Opérateurs d'affectation

Le mot expression a été volontairement utilisé pour désigner la partie de l'opération située à droite de l'opérateur car vous pouvez effectivement utiliser une variable ou une expression plus complexe si besoin est.

Dans les exemples, vous ne trouverez que des affectations avec deux variables uniquement, excepté pour le premier opérateur mais retenez que vous pouvez faire de même pour tous les opérateurs.

```
double dJour = 2;
iSemaine = 2;
dJour *= 5.0/ iSemaine;
```

Dans l'exemple ci-dessus, l'opération effectuée sera **dJour = 2 * (5.0 / 2)** soit **dJour = 5**. Attention une nouvelle fois à la syntaxe car si au lieu de **(5.0 /2)**, vous auriez écrit **(5 / 2)**, la valeur renvoyée n'aurait plus été **5** mais **4** car pour MetaTrader, **(5 / 2)** est égale à **4** vu que les chiffres sont sous la forme d'entiers et pas de réels.

L'opérateur « **=** » est compatible avec tous les types de variables. Par contre, les autres opérateurs d'affectation ne fonctionne qu'avec des entiers ou réels hormis « **%=** » qui n'est utilisable qu'avec des entiers.

Le code suivant est correct.

```
bool bBaissier;
bBaissier = 1;
```

Le code suivant est incorrect car vous ne pouvez pas utiliser un réel (4.5).

```
int iJour = 15;
iJour %= 4.5;
```

Opérateurs de comparaison

Les opérateurs de comparaison comme le nom l'indique permettent de comparer deux valeurs. La valeur obtenue à la fin d'une instruction de comparaison sera une valeur booléenne. Comme nous l'avons vu dans la section consacrée aux types, si l'instruction s'avère fausse, le programme renverra la valeur *False* sinon, ce dernier renverra la valeur *True*.

Opérateur	Description	Syntaxe	Exemple
==	Renvoi *True* si X est égal à Y sinon *False*	X == Y	iJour == (2 + 3)
!=	Renvoi *True* si X est différent à Y sinon *False*	X != Y	iJour != 4
<	Renvoi *True* si X est strictement plus petit que Y sinon *False*	X < Y	iJour < 4
>	Renvoi *True* si X est strictement plus grand que Y sinon *False*	X > Y	iJour > 2
<=	Renvoi *True* si X est plus petit ou égal à Y sinon *False*	X <= Y	dJour <= 4.5
>=	Renvoi *True* si X est plus grand ou égal à Y sinon *False*	X >= Y	iJour >= 4

Tableau 9 : Opérateurs de comparaison

La syntaxe de ces opérateurs ne comprend pas de « **;** » à la fin de l'instruction car ces opérateurs ne peuvent être employés tout seuls et doivent être insérés à l'intérieur d'une instruction conditionnelle ou d'une boucle comme dans l'exemple ci-dessous.

Il est important de noter que vous ne pouvez pas cumuler plus d'un opérateur de ce type dans la même expression.

Le code suivant est correct.

```
if(iNombreTrades < 1)
  bTrade = True;
```

Le code suivant est incorrect car il y a plus d'un opérateur dans la même expression.

```
if(iNombreTrades < 1 == iCompteurTrade)
  bTrade = True;
```

Opérateurs logiques

Les opérateurs logiques servent à lier entre elles des expressions de comparaisons car comme nous l'avons vu précédemment, il n'est pas permis de cumuler plus d'un opérateur de comparaison par expression. Le résultat renvoyé par des expressions utilisant ce type d'opérateur sera une valeur booléenne *True* ou *False*.

Opérateur	Description	Syntaxe	Exemple
&&	Correspond à « ET ». Renvoie *True* si les deux expressions sont vraies	X == Y && Y != Z	(X > 0) && (x < 9)
!	Correspond à « INVERSE ». Renvoie *True* lorsque l'expression est fausse et *False* lorsque l'expression est vraie	! (X > Y)	! (X > 0)
||	Correspond à « OU ». Renvoie *True* si au moins l'une des deux expressions est vraie sinon on obtient *False*	X == Y || Y <= Z	X == 0 || Y <= 2

Tableau 10 : Opérateurs logiques

Comme pour les opérateurs de comparaison, les opérateurs logiques ne peuvent être employés tout seuls et doivent être insérés à l'intérieur d'une instruction conditionnelle.

```
if(!X)
  Print("Faux");

if(X > 2 && Y == 2)
  Print("Conditions remplies");
```

Nous utiliserons la fonction *Print()* dans certains de nos exemples du début. Elle permet d'afficher l'information située entre parenthèse dans l'onglet expert de la fenêtre Terminal de la plateforme. Si vous désirez en savoir plus tout de suite, vous pouvez consulter la page 134.

Opérateurs au niveau du bit

Toute donnée informatique est stockée en mémoire sous la forme d'une combinaison de bits. Les instructions d'opérations au niveau du bit ne sont possibles qu'avec des entiers et renvoient des valeurs numériques.

Opérateur	Description	Syntaxe	Exemple
&	Correspond à « ET ». On compare bit à bit X et Y. Si le même bit de X et Y est à 1 alors la valeur renvoyée sera 1 sinon la valeur sera 0	X & Y	A = ((B & C) !=0);
|	Correspond à « OU ». On compare bit à bit X et Y. Si un bit de X ou de Y est à 1 alors la valeur renvoyée sera 1 sinon la valeur sera 0	X | Y	A = B | C;

^	Correspond à « OU EXCLUSIF ». On compare bit à bit X et Y. Si un bit de X est à 1 et le même bit de Y est à 0 alors la valeur renvoyée sera 1 sinon la valeur sera 0. L'inverse fonctionne également	X ^ Y	A = B ^ C;
~	Correspond à « INVERSE ». La valeur renvoyée est 0 lorsque le bit est à 1 et 1 lorsque le bit est à 0	~X	A = ~B;
>>	Décale les bits de X de Y fois vers la droite. Les espaces vides à gauches seront remplacés par des 0	X >> Y	A = A >> B;
<<	Décale les bits de X de Y fois vers la gauche. Les espaces vides à droite seront remplacés par des 0	X << Y	A = A << B;

Tableau 11 : Opérateurs au niveau du bit

Opérateur virgule

L'opérateur virgule est utilisé pour effectuer plusieurs instructions successivement de gauche à droite. Le type et la valeur du résultat renvoyée correspond donc à celui et celle de la dernière expression.

La syntaxe pour cet opérateur est :

```
expression1, expression2, expression3, etc…
```

Un exemple pourrait être :

```
for(X = 10, Y = 10 ; X > 0 ; X--, Y++)
  Print(X+" – "+Y+" = "+(X-Y));
```

La boucle ci-dessus renverra les résultats suivants :

10 – 10 = 0
9 – 11 = -2
8 – 12 = -4
7 – 13 = -6
6 – 14 = -8
5 – 15 = -10
4 – 16 = -12
3 – 17 = -14
2 – 18 = -16
1 – 19 = -18

Opérateur d'appel de fonction et opérateur return

Les fonctions sont traitée dans un chapitre qui leur est entièrement consacré, veuillez vous reporter au chapitre 5.

Opérateurs d'indexation de tableau

De la même façon que pour l'appel de fonction, les tableaux sont une part importante de l'apprentissage du MQL4 et il est donc préférable et logique de leur consacrer une section à part entière dans le chapitre 33.

La priorité des opérateurs

Lorsque vous utilisez plus d'un opérateur dans la même instruction, il convient de préciser au programme dans quel ordre les opérations doivent s'effectuer sinon ce dernier utilisera un ordre de priorité préétabli entre les différents opérateurs et effectuera les opérations de gauche à droite.

Par défaut, les expressions entre parenthèses sont prioritaires et ce, peu importe les opérations se trouvant à l'extérieur des parenthèses. C'est donc en utilisant ces dernières que vous pouvez forcer un ordre de priorité si vous le désirez.

Ci-dessous, vous trouvez l'ordre de priorité entre les différents opérateurs du plus prioritaire au moins prioritaire.

( )	:	Expressions entre parenthèses sont prioritaires par rapport au reste
[ ]	:	Opérateurs se rapportant à un tableau
!	:	Contraire
~	:	Inverse au niveau du bit
-	:	Changement de signe
*	:	Multiplication
/	:	Division
%	:	Modulo
+	:	Addition
-	:	Soustraction
<<	:	Décalage à gauche au niveau du bit
>>	:	Décalage à droite au niveau du bit
<	:	Inférieur à
<=	:	Inférieur ou égal à
>	:	Supérieur à
>=	:	Supérieur ou égal à
==	:	Égal à
!=	:	Différent de
&	:	Et au niveau du bit
^	:	Ou exclusif au niveau du bit
&&	:	Et logique
||	:	Ou logique
=	:	Affectation
+=	:	Addition d'affectation

-= : Soustraction d'affectation
*= : Multiplication d'affectation
/= : Division d'affectation
%= : Modulo d'affectation
, : Virgule

Lorsque vous avez plusieurs opérateurs de même type ou de priorité identique (« * » et « / » par exemple), l'ordre s'établit alors de gauche à droite.

5
Les fonctions

Les fonctions sont un ensemble d'instructions ou expressions mises en commun afin d'effectuer une fonction particulière.

Plus vous allez acquérir de l'expérience en programmation et plus vous allez être tenté de réaliser des programmes de plus en plus longs et complexes. De plus, il est fort probable que d'un programme à l'autre, vous retrouviez le même type d'instructions (instructions pour paramétrer et modifier un stop, instructions de money management, etc...) ou qu'à l'intérieur du même programme, vous deviez répéter plusieurs fois les mêmes instructions, il est alors plus pratique et facile de faire appel à une fonction. Une fonction vous permet donc de découper votre programme en sous-ensembles que vous pouvez réutiliser en cas de besoin et gagner ainsi en lisibilité et en temps.

Chaque portion de votre code doit faire partie d'une fonction. Celle-ci peut être une fonction que vous aurez spécialement créée pour l'occasion ou l'une des fonctions spéciales de MetaTrader que nous allons voir par la suite.

Une fonction est composée d'instructions en MQL4, s'identifie par un nom qui lui est propre, peut posséder des arguments propres et renvoie en général une valeur. Nous allons maintenant voir comment définir et déclarer une fonction.

Définition et déclaration

Pour définir une fonction, il vous faudra lui choisir un nom (essayez de toujours choisir des noms explicites sur ce que fait la fonction de façon à vous simplifier la vie ou celle des autres personnes qui auront accès à votre code), définir ses arguments (c'est-à-dire les données qu'elle va utiliser), lui attribuer un type de renvoi (en fait, il s'agit du type de la valeur que la fonction renverra si la fonction renvoie une valeur à la fin) et enfin les instructions qui la composeront.

Pour pouvoir utiliser une fonction, celle-ci doit au minimum avoir été déclarée au préalable. Vous n'êtes pas obligé de la définir dès le départ mais c'est généralement plus simple. Déclarer une fonction revient à simplement préciser le type de retour, son nom et ses arguments.

Définir une fonction revient à la déclarer et inclure les instructions qu'elle devra exécuter.

Si vous appelez une fonction pour l'utiliser au sein d'une autre fonction, la fonction appelée doit être définie avant la fonction qui l'appelle. Vous ne pouvez pas non plus définir une fonction à l'intérieur d'une autre fonction.

Tout ce qui précède étant très théorique, voici quelques exemples pour vous aider à mieux comprendre le concept de fonction.

Exemple de déclaration :

```
typeRetour fNomFonction(type tArgument1, type tArgument2);
```

Exemple de déclaration et définition :

```
typeRetour fNomFonction(type tArgument1, type tArgument2)
 {
 tArgument2 = tArgument 1 + 3;
 return(tArgument2);
 }

// typeRetour : type de la valeur renvoyée par la fonction (int, double, bool, color, datetime, ou string).

// fNomFonction : nom que vous choisirez pour votre fonction. Attention aux mots réservés.

// type tArgumentX sont les différents arguments de la fonction avec leur type respectif.

// return(tArgument2); est l'opérateur qui permet de signaler au programme la fin de la fonction et renvoie la //
variable entre parenthèse. Le type de la valeur renvoyée doit correspondre au typeRetour indiqué dans la //
déclaration. Lorsque vous utilisez void comme type de la fonction, vous pouvez simplement utiliser //
« return; » sans parenthèses ni expression pour signaler la fin de l'exécution de la fonction.

// { } sont les accolades qui servent à encadrer les instructions qui composeront votre fonction.
```

Lorsque vous déclarez une fonction, le « **;** » est de mise à la fin de l'instruction mais lorsque vous définissez la fonction, la ligne de déclaration ne prend pas de « **;** ».

Vous remarquerez que de la même façon que pour les variables, le préfixe *f* est ajouté devant le nom afin de pouvoir identifier facilement à l'intérieur du code qu'il s'agit d'une fonction.

Si la fonction ne renvoie aucune valeur, il faut alors utiliser *void* comme type.

Appel de fonction

Maintenant que vous savez ce qu'est une fonction et comment la définir, nous allons voir comment faire appel aux fonctions afin de pouvoir les utiliser dans votre programme.

Si vous reprenez l'exemple vu précédemment.

```
typeRetour fNomFonction(type tArgument1, type tArgument2)
 {
 tArgument2 = tArgument 1 + 3;
 return(tArgument2);
 }
```

La syntaxe pour appeler cette fonction serait la suivante :

```
tVariable = fNomFonction(arg1, arg2);
```

Grâce à *fNomFonction*, le programme sait à quelle fonction vous faites référence et est capable d'aller chercher les instructions. Ensuite, il va utiliser les variables *tArg1* et *tArg2* qui sont définis pour le programme au complet et les utiliser pour exécuter la fonction. Cela signifie que dans les instructions de la fonction, le programme va remplacer *tArgument1* par *tArg1* et *tArgument2* par *tArg2*.

Voici un exemple concret avec de vraies valeurs :

Déclaration et définition :

```
int fExempleFonction(int iNombre1, int iNombre2)
 {
 int iNombre3= iNombre1 + iNombre2;
 return(iNombre3);
 }
```

Appel de fonction :

```
iValeurFonction = fExempleFonction(1, 4);
```

La valeur renvoyée par la fonction et qui sera affectée à la variable *iValeurFonction* sera *iNombre3* et sera égale à 1+4 soit 5. Au lieu de *1* et *4*, des variables ou constantes auraient pu être utilisées tant et aussi longtemps que leur type correspond à celui des arguments de la fonction soit *int*.

Vous remarquez que la variable *iNombre3* a été définie à l'intérieur de la fonction car elle n'a d'utilité que dans cette dernière. Il s'agit donc d'une variable locale. Vous auriez parfaitement pu utiliser également une variable globale, par exemple *iValeurFonction*. Ceci aurait donc donné :

Déclaration et définition :

```
void fExempleFonction(int iNombre1, int iNombre2)
 {
 iValeurFonction = iNombre1 + iNombre2;
 }
```

Appel de fonction :

```
fExempleFonction(1, 4);
```

Il n'est plus nécessaire d'affecter la valeur renvoyée de la fonction à une variable puisque l'affectation se fait directement à l'intérieur de la fonction. L'opérateur *return* ne sert donc plus car la fonction ne renvoie aucune valeur et c'est pourquoi le type de la fonction est maintenant devenu *void*.

Les noms que vous donnez aux arguments d'une fonction ne servent que pour la portion du programme correspondant à la fonction (portion encadrée par les accolades). Cela signifie que vous pouvez très bien avoir le même nom de variable dans plusieurs fonctions sans que ces dernières ne rentrent en conflit. Il est néanmoins recommandé de ne pas répéter les noms pour une meilleure lisibilité.

Si vous avez déclaré une variable globale, il ne vous sera plus possible de déclarer une variable locale avec le même nom.

L'appel d'une fonction peut s'effectuer en tant qu'expression unique ou à l'intérieur d'une autre expression comme vous pouvez le voir dans l'exemple suivant.

```
tArg3 = fNomFonction(tArg1, tArg2) + 4;
```

Opérateur return

Nous avons vu brièvement l'opérateur return et ses usages précédemment mais s'agissant d'un opérateur important en MQL4, nous allons lui dédier une section à part entière.

L'opérateur *return* indique au programme que l'exécution de la fonction appelée doit être stoppée et indique que l'exécution du programme appelant doit être poursuivie.
Lorsque l'opérateur contient une expression entre parenthèses *return(InformationARenvoyer)*, l'exécution de la fonction appelée est stoppée et la valeur ou information indiquée entre parenthèses transmis au programme appelant.

Dans le cas de fonction de type *void*, l'opérateur *return()* devra être utilisé sans parenthèses : *return;*

Observons quelques exemples :

```
bool fExempleFonction(int iPremier, int iDeuxieme)
 {
 if(iPremier + iDeuxieme == 0)
   {
   return(False);
   }
 return(True);
 }
```

```
void fExempleFonction()
 {
 if(IsTradeAllowed == False)
   Print("Vous devez activer les experts consultants ");
 return;
 }
```

L'instruction *return(0);* sert à indiquer au programme la fin de la fonction et qu'il faut recommencer l'exécution depuis le début de la fonction au prochain tick.

Imbriquer des fonctions

Nous avons vu plus haut qu'il était possible de faire appel à une fonction à l'intérieur même d'une autre fonction. Par exemple, supposez que vous avez les deux fonctions suivantes :

```
typeRetour fNomFonction(type tArgument1, type tArgument2)
 {
 tArgument2 = tArgument1 + 3;
 }


typeRetour fNomFonctionBis(type tArgument1, type tArgument2)
 {
 type tArgument3 = fNomFonction(tArg1, tArg2);
 return(tArgument3);
 }
```

La deuxième fonction va utiliser la première pour effectuer l'opération et ensuite, la valeur obtenue suite à l'exécution de la première fonction sera affectée à la variable *tArgument3* par la deuxième fonction.

Vous ne pouvez pas définir une fonction à l'intérieur d'une autre fonction. MetaEditor serait tout simplement incapable de compiler votre code.

Fonctions spéciales

Il existe 3 fonctions spéciales en MQL4. Vous ne devez sous aucun prétexte utiliser les mêmes noms pour vos propres fonctions.

Fonction spéciale int init()

Si vous insérez des instructions entre les { } de cette fonction, ces instructions seront exécutées au moment de l'initialisation de votre programme (Expert Consultant, indicateur ou script). Pour un Expert Consultant ou indicateur, la fonction *init()* sera exécutée à chaque fois que les experts seront activés, à chaque changement d'unité temporelle, à chaque recompilation, à chaque fois que vous changerez les paramètres de l'expert ou que le compte d'utilisateur sera changé. Pour un script, la fonction sera exécutée au chargement du script sur le graphique.

La fonction *init()* étant la première à être exécutée. Il est déconseillé d'appeler la fonction *start()* ou toute autre fonction que vous aurez créée ayant pour but d'effectuer des opérations d'achat ou vente car la fonction *init()* est justement chargée de récupérer les informations utiles à la bonne exécution du code défini dans votre fonction *start()*. Il existe donc le risque d'exécuter une instruction alors que le programme n'a pas encore recueilli toutes les informations utiles ou tout simplement celui que votre programme ne fonctionne pas car il ne dispose pas des informations nécessaires.

Fonction spéciale int start()

À l'intérieur de la fonction *start()* figurera le code principal de votre programme. Son activation dépend principalement du type de programme que vous avez codé. Pour un Expert Consultant, cette fonction sera exécutée à chaque nouveau tick (c.-à-d. à chaque nouvelle cotation du prix d'une paire). Pendant l'exécution de la fonction suite à la réception d'un nouveau tick, le programme ignorera les ticks suivants tant que l'exécution de la fonction ne sera pas achevée.

En général, cela ne pose pas de problème car tout ceci se fait très rapidement mais en période de forte volatilité, MetaTrader pourrait vous afficher un message d'erreur selon lequel le prix d'entrée choisi par l'Expert Consultant est invalide. Pour un indicateur, la fonction sera activée aussitôt après l'apparition d'un nouveau tick, à l'ouverture d'une session du programme MetaTrader à condition que l'indicateur ait été chargé lors de la précédente session, au changement de paires ou d'unité temporelle. Pour un script, l'exécution de cette fonction aura lieu juste après celle de la fonction *init()*.

Fonction spéciale int deinit()

Cette fonction est l'antithèse de la fonction *init()*. Comme son nom l'indique, cette fonction est activée suite à la clôture du programme. En fait, lorsque vous fermez votre session MetaTrader, changez de compte, d'unité temporelle, de paires ou encore de paramètres pour un programme en particulier, cette fonction est automatiquement exécutée par MetaTrader.

La clôture d'un Expert Consultant ou d'un script va obligatoirement appeler l'exécution de cette fonction. On considère comme fermeture d'un programme le remplacement de celui-ci par un autre programme ou le chargement du même programme dans le même graphique. Cela vient du fait qu'il est impossible de cumuler plus d'un Expert Consultant ou script dans le même graphique. Pour un indicateur, la fonction *deinit()* n'est appelée que lorsque vous fermez MetaTrader ou la fenêtre de cotation.

Contrairement aux deux autres types de programmes, il est possible de cumuler plus d'un indicateur, que ce soit le même ou non, sur le même graphique. En cas de clôture d'un programme (peu importe la façon) et si la fonction *start()* est en cours d'exécution, MetaTrader limite le temps d'exécution de cette dernière à 2.5 secondes. Idem pour la fonction *deinit()* qui est limité à 2.5 secondes également. Si l'exécution de l'une ou l'autre excède le temps imparti, MetaTrader met fin automatiquement au programme.

Seule la fonction *start()* est obligatoire dans un programme - *init()* et *deinit()* peuvent être absentes et ne pas bloquer la bonne exécution de votre code-. Il est également important de ne jamais essayer d'exécuter *init()* et *deinit()* sans que la fonction *start()* soit appelée entre les deux. Votre programme ne fonctionnerait tout simplement pas car il serait inexistant étant donné que le code principal est censé se situer dans la définition de la fonction *start()*. Un programme qui ne contiendrait pas la fonction *start()* ne pourrait pas fonctionner car

MetaTrader n'est pas capable d'appeler des fonctions autres que les fonctions spéciales. Les fonctions que vous allez créer ne peuvent être appelées que par le programme lui-même et doivent donc par conséquent être appelées dans l'une des fonctions spéciales. Attention néanmoins, bien qu'appelées à l'intérieur des fonctions spéciales, les fonctions nouvellement créées devront être déclarées et définies à l'extérieur des fonctions spéciales.

L'ordre d'exécution d'un programme est donc le suivant :

int init() → int start() → int deinit()

L'ordre dans lequel vous placez vos fonctions à l'intérieur du code source n'a pas d'importance. Vous pourriez par exemple placer la fonction *start()* avant toutes les autres sans que cela n'affecte le bon fonctionnement de votre programme. Souvenez-vous néanmoins que vous ne devez sous aucun prétexte appeler une fonction spéciale ou définir une fonction à l'intérieur d'une autre fonction.

Lorsque vous utilisez l'opérateur *return* avec ou sans expression dans une fonction créée par vos soins, l'opérateur indique au programme de stopper l'exécution de la fonction en question et de débuter l'exécution du code qui suit l'appel de la fonction personnelle. Lorsque vous voyez *return(0);* à la fin des fonctions spéciales, l'opérateur indique au programme la fin d'exécution et l'attente d'un nouveau tick avant de réinitier l'exécution de la fonction.

Dans la figure 4 située à la page suivante, vous trouverez un exemple de code source simplifié vous permettant de visualiser les différentes sections du code mises en valeur par différentes couleurs.

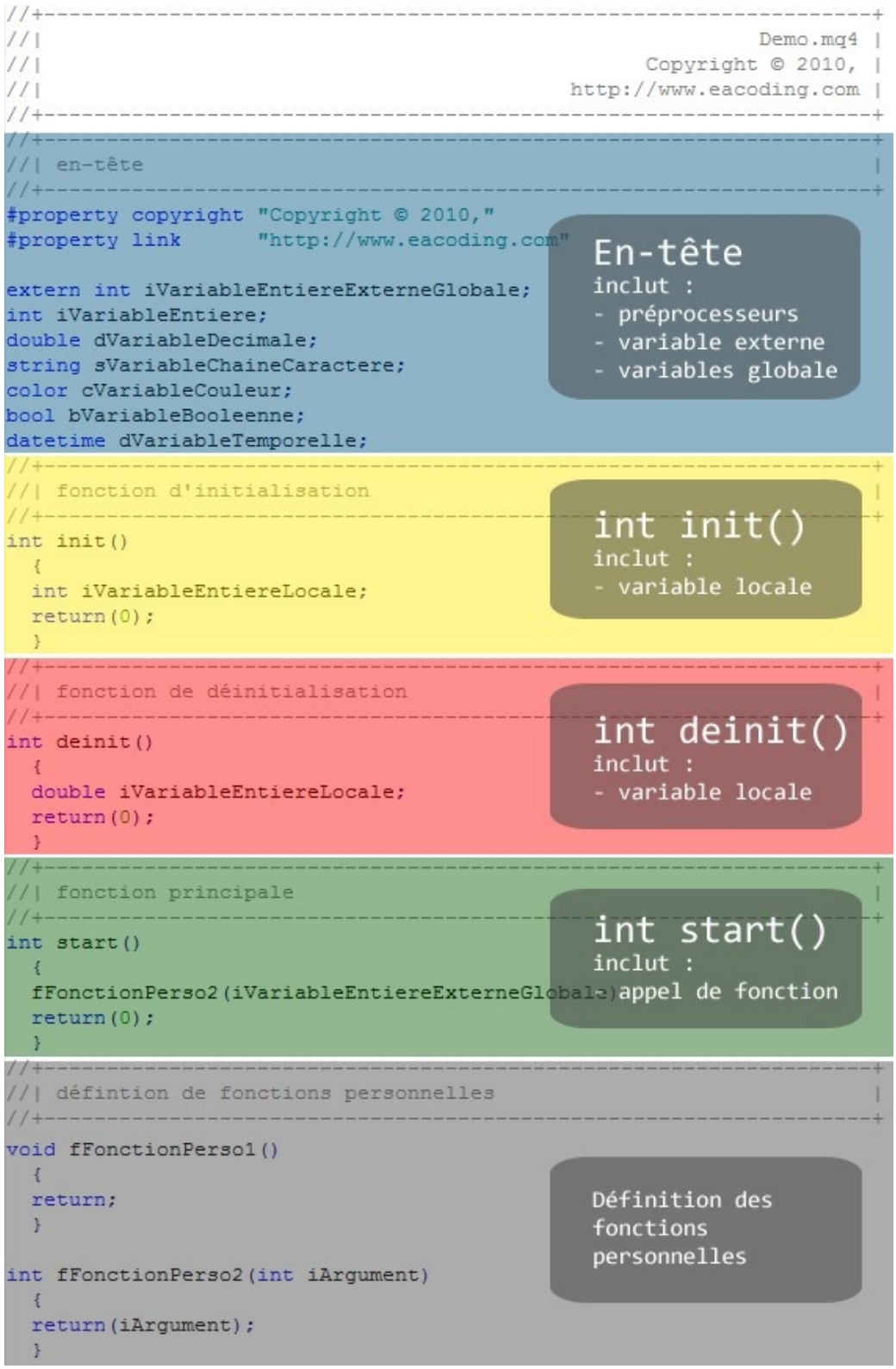

Figure 4: Différentes sections du code source

6
Les instructions conditionnelles

Une instruction conditionnelle sert à exécuter une portion de code selon qu'une condition est remplie ou non.

Exemple :

Si EURUSD > 1,5000
- **Alors** exécuter achat
- **Sinon** exécuter vente.

Si = Condition à vérifier
Alors = Action si la condition est remplie
Sinon = Action si la condition n'est pas remplie

En MQL4, nous avons 3 instructions conditionnelles : l'instruction *if* seule, l'instruction *if* suivi d'un *else* et enfin l'instruction *switch*. Les deux premières sont basiquement les mêmes mais la distinction existe car il n'est pas nécessaire d'avoir un *else* avec une boucle *if*.

Instruction conditionnelle if…

La syntaxe de cette instruction est la suivante :

```
if(condition(s) à vérifier)
  Action à exécuter;
```

Entre parenthèses, vous devez inscrire la ou les conditions à remplir. Il n'y a pas vraiment de limitation dans le nombre ou type de conditions. Vous pouvez cumuler plusieurs conditions à l'aide des opérateurs // (ou) et && (et).

Si vous désirez voir plusieurs actions s'exécuter si la partie condition est validée, il vous suffit simplement d'ajouter des crochets à l'ensemble de l'instruction comme ci-dessous :

```
if(condition(s) à vérifier)
  {
  action à exécuter #1;
  action à exécuter #2;
  action à exécuter #3;
  ...
  }
```

Lorsque vous utilisez l'instruction *if* seule et en cas de non validation des conditions, le programme se contentera de suivre son cours en lisant les instructions suivantes.

Il n'est pas nécessaire d'inclure des crochets si vous n'avez qu'une seule instruction à la suite de *if*. Par défaut, le programme lira l'instruction suivante comme celle à exécuter en cas de validation de la condition.

Instruction conditionnelle if… else…

L'ajout de *else* à l'instruction *if* permet d'attribuer une action à exécuter en cas de non validation de la condition.

```
if(condition(s) à vérifier)
  Action à exécuter si condition remplie;
else
  Action à exécuter si condition non remplie;
```

De la même façon que précédemment, il est possible d'inclure plusieurs instructions à la suite comme ci-dessous en les encadrant de crochets.

```
if(condition(s) à vérifier)
  {
  Action à exécuter si condition remplie #1;
  Action à exécuter si condition remplie #2;
  Action à exécuter si condition remplie #3;
  }
else
  {
  Action à exécuter si condition non remplie #1;
  Action à exécuter si condition non remplie #2;
  Action à exécuter si condition non remplie #3;
  }
```

Multiples instructions if… et if… else…

Si vous en avez la nécessité, vous pouvez également cumuler plusieurs if les uns à la suite des autres ou encore les uns dans les autres. Les possibilités sont vraiment multiples. Il faudra juste prendre garde à toujours bien délimiter les instructions pour éviter de possibles bugs ou erreurs de jugements de votre programme.

Pour vous faciliter la tâche, n'hésitez pas à inclure des crochets à chaque fois. De cette façon, vous aurez toujours bien à l'œil les limites de chacune des instructions. Nous reviendrons dans une prochaine partie sur la syntaxe et la façon d'écrire un programme mais vous remarquerez que dans les exemples, il y a toujours un décalage des instructions secondaires par rapport à l'instruction principale - il s'agit là d'une bonne habitude à prendre car il vous est alors beaucoup plus simple de repérer l'agencement de vos instructions.

```
if(condition à vérifier #1)
 {
 if(condition à vérifier #2)
   Action à exécuter;
 else
   {
   if(condition à vérifier #4)
     {
     Action à exécuter #1;
     Action à exécuter #2;
     }
   }
 }
```

Dans l'exemple ci-dessus, l'instruction conditionnelle #4 ne sera examinée que si la condition #1 est vraie et la #2 fausse.

Lorsque la condition est booléenne (vrai ou faux), si vous n'indiquez pas d'opérateur de comparaison, MetaTrader testera, par défaut, la condition pour voir si cette dernière est vraie.

Par exemple :

```
if(bCondition)
  Action à exécuter si condition remplie;
```

Équivaut à écrire :

```
if(bCondition == True)
  Action à exécuter si condition remplie;
```

Instruction conditionnelle switch

L'instruction *switch* équivaut à une suite de *if* mais permet de simplifier grandement la syntaxe. Cette instruction s'applique lorsque vous devez comparer la valeur d'une variable et appliquer une action différente selon la valeur.

La syntaxe de cette instruction est la suivante :

```
switch(x)
 {
 case 1 :  action à exécuter #1;
           action à exécuter #2;
           break;
 case 2 :  action à exécuter #3;
           action à exécuter #4;
           break;

 default : action à exécuter #5;
 }
```

L'exécution du code précédent se fera de la façon suivante : le programme va regarder quelle valeur est assignée à l'expression x. Si la valeur est 1, le programme exécutera les actions figurant au niveau de case 1. Même chose dans le cas de 2. Si x ne vaut ni 1 ni 2, le programme exécutera alors la portion de code figurant à droite de default.

L'opérateur *break* sert à mettre fin à l'instruction. En effet, supposons que x vaille 2, le programme exécutera donc les actions #3 et #4 qui correspondent au cas où x vaut 2 mais, en l'absence de *break*, le programme exécutera également l'action #5 qui suit. Le dernier cas, *default* ne nécessite pas de *break* puisque l'instruction s'achève avec lui. Le cas *default* n'est pas obligatoire pour le bon fonctionnement de l'instruction, vous pouvez donc l'omettre si vous n'en avez pas besoin.

Comme valeur de « x », vous pouvez également utiliser des caractères. Auquel cas, il faudra que l'expression « x » soit compatible avec le type de données « caractère » et la syntaxe sera la suivante : case 'A'. Si vous n'entourez pas le caractère de «' », le programme vous renverra une erreur.

Il est nécessaire d'encadrer tous les cas du *switch* par des accolades mais par contre, les accolades ne sont pas nécessaires à l'intérieur d'un cas et ce peu importe le nombre d'instructions.

7
Les boucles

Les boucles permettent de répéter l'exécution d'instructions tant et aussi longtemps que les conditions indiquées au départ de la boucle sont vérifiées. En MQL4, on compte deux types de boucles : les boucles *for* et les boucles *while*.

Boucle for

La syntaxe d'une boucle *for* est la suivante :

```
for(expr1 ; expr2 ; expr3)
  Action à exécuter;


// Expr1 : expression qui contiendra la variable clé de la boucle. C'est cette variable que l'on va utiliser pour
// déterminer quand il faudra sortir de la boucle. Expr1 correspond donc à l'initialisation de cette variable.
// L'initialisation ne se produira qu'une seule fois au démarrage de la boucle for.


// Expr2 correspond à la condition à tester avant chaque nouvelle boucle.


// Expr3 correspond à l'action à exécuter à la fin de chaque boucle.
```

Plus concrètement, une boucle *for* ressemblera à ceci :

```
for(int i = 0 ; i <= 10 ; i++)
  Print("i = "+i);
```

Vous n'êtes pas oblige de déclarer la variable *i* dans la parenthèse de la boucle *for* mais pour simplifier la lecture, c'est souvent préférable sauf si vous utilisez la même variable pour toutes vos boucles.

```
int i;
for(i = 0 ; i <= 10 ; i++)
  Print("i = "+i);
```

On déclare la variable *i* et on lui assigne la valeur 0. Avant chaque nouvelle boucle, le programme vérifie que *i* est bien inférieur à 10. Si i est supérieur ou égal à 10, alors le programme sortira de la boucle. À la fin de chaque boucle, le programme ajoute 1 à la valeur de *i*.

Il est possible de cumuler plusieurs expr1 et expr3 en les séparant par des virgules mais vous ne pouvez jamais avoir plus d'une expr2.

Le code suivant est correct.

```
for(int i = 0, int j =1 ; j <= 10 ; i++, j++)
  Action à exécuter;
```

Le code suivant est incorrect car il y a deux conditions dans la boucle.

```
for(int i = 0, int j =1 ; j <= 10, i <= 5 ; i++, j++)
  Action à exécuter;
```

Pour avoir les deux conditions, il faudrait les transformer en une seule et unique expression retournant la valeur *True* ou *False*. Par exemple :

```
for(int i = 0, int j =1 ; j <= 10 || i <= 5 ; i++, j++)
  Action à exécuter;
```

Vous pouvez aussi bien désincrémenter que incrémenter.

```
for(int i = 0, int j =10 ; j >= 1 ; i++, j--)
  Action à exécuter;
```

Si vous désirez exécuter plusieurs actions au sein de la boucle, il vous suffit comme pour l'instruction if de les encadrer par des crochets.

```
for(i = 0 ; i <= 10 ; i++)
  {
  Action à exécuter #1;
  Action à exécuter #2;
  }
```

Comme pour un l'instruction conditionnelle switch, il est possible d'introduire l'instruction *break* dans une boucle *for* afin de sortir de la boucle prématurément.

```
for(i = 0 ; i <= 10 ; i++)
  {
  Action à exécuter #1;
  if(Condition)
    break;
  Action à exécuter #2;
  }
```

Le programme va donc s'exécuter normalement et vérifier à chaque boucle si la condition est remplie ou non. Lorsque cette dernière sera remplie, le programme sortira immédiatement de la boucle sans exécuter l'action #2.

Il est parfois nécessaire de continuer une boucle sans pour autant achever l'itération en cours (boucle en cours), dans ce cas, nous introduirons un nouvel opérateur : *continue*.

```
for(i = 0 ; i <= 10 ; i++)
  {
  Action à exécuter #1;
  if(Condition)
    continue;
  Action à exécuter #2;
  }
```

Le programme va donc s'exécuter normalement et vérifier à chaque boucle si la condition est remplie ou non. Lorsque cette dernière sera remplie, le programme sautera à l'itération suivante sans exécuter l'action #2.

Il est possible d'éliminer *Expr1*, *Expr2*, *Expr3*, deux d'entre elles ou toutes en même temps dépendant de vos besoins. Dans le dernier cas, vous aurez créé une boucle perpétuelle.

Il est important de toujours laisser les « ; » en place.

```
for( ; ; )
  Action à exécuter;
```

Boucle while

La différence majeure entre les deux boucles est que pour une boucle *for*, vous savez combien d'itérations vont être exécutées tandis que pour une boucle *while*, vous ne possédez pas ce genre d'information. Ceci est possible par le fait qu'une boucle *while* ne possède pas d'informations du type *Expr1* et *Expr3* mais uniquement *Expr2* soit la condition.

La syntaxe de ce type de boucle est la suivante :

```
while(Condition)
  Action à exécuter;
```

Ou encore :

```
while (Condition)
  {
  Action à exécuter #1;
  Action à exécuter #2;
  }
```

Condition ne peut être comme son nom l'indique qu'une condition, il n'est pas possible de déclarer ou d'initialiser quoique ce soit.

Un exemple plus concret pourrait être :

```
int i = 0 ;
while(i <= 10)
  {
  i++;
  }
```

Le programme va vérifier la condition avant de commencer (i vaut 0 qui est inférieur à 10 donc la condition est remplie). À chaque itération, *i* va être incrémenté de 1. Lorsque *i* sera égal à 10, la condition ne sera plus rempli et le programme sortira alors de la boucle.

Comme pour la boucle *for*, vous pouvez utiliser les opérateurs *break* et *continue*.

8
Préprocesseur

Le préprocesseur permet d'opérer des transformations sur votre code avant l'étape de compilation. Le compilateur saura quelles instructions traiter en premier grâce aux directives de précompilation. En fait, ceci vous sera très utile si vous travaillez avec des fonctions propres à Windows que vous voudriez inclure dans votre programme ou encore du code provenant d'autres fichiers que votre fichier source. Ainsi, en utilisant la directive *#include <…>*, vous demandez en fait à ce que le contenu du fichier inclus entre <> soit inséré au début de votre code.

Il existe différentes directives en MQL4. Avant de passer chacune d'entre elles en revue, voici la syntaxe générale de ces directives : Il est d'usage de placer ces directives au début de votre programme. Lorsque vous utilisez le symbole « # » au début de votre programme et en début de ligne, le programme interprétera cette ligne comme une directive de précompilation. Contrairement à une instruction classique, les directives ne se terminent pas par un « ; ». Pour terminer l'instruction, il faut simplement aller à la ligne.

Directive #define

La directive *#define* sert à définir des constantes - autrement dit des variables dont la valeur ne change jamais au cours de l'exécution de votre programme.

La syntaxe pour la définition de constantes est la suivante :

```
#define NBRE_JOURS 100 // entier
#define PI 3.14 // décimal
#define NOM "Nom Prénom" // chaîne de caractères
```

Vous remarquez l'absence du « ; » à la fin de l'instruction ainsi que l'absence de signe « = » pour l'assignation de la valeur et enfin l'absence de type également. Cette directive va en fait analyser tout votre code source et, à chaque fois que vous aurez utilisé le nom *NBRE_JOURS*, remplacera *NBRE_JOURS* par sa valeur correspondante soit 100 dans l'exemple.

Le seul cas pour lequel *NBRE_JOURS* ne serait pas remplacé par la valeur définie serait si, dans votre code, *NBRE_JOURS* est dans une chaîne de caractère.

Comme la constante n'est pas typée dans la déclaration, vous augmentez de ce fait la probabilité de voir apparaître une erreur dans la compilation car la directive *#define* fait un chercher-remplacer mécanique donc faites attention aux endroits où vous allez insérer cette constante dans votre code.

Directive #property

Par défaut, il existe en MQL4 une série de constantes prédéfinies que vous n'avez pas besoin de déclarer en utilisant la directive précédemment vue. Vous allez spécifier la valeur de ces constantes en utilisant la directive *#property* que le compilateur interprétera et utilisera de lui-même.

Ces constantes prédéfinies peuvent être par exemple les informations de droits d'auteur de votre programme (nom de l'auteur, compagnie, site internet, etc.…) ou encore le positionnement de vos indicateurs (à même la courbe de prix, dans une fenêtre séparée, etc.…).

Constante	Type	Description
link	string	Lien vers un site internet (compagnie par exemple)
copyright	string	Informations sur les droits d'auteur
stacksize	int	Taille de la structure de donnée
library		Indique au compilateur qu'il s'agit d'une librairie
indicator_chart_window	void	Affiche l'indicateur à même la courbe de prix
indicator_separate_window	void	Affiche l'indicateur dans une fenêtre séparée
indicator_buffers	int	Indique le nombre de mémoires tampons (1 à 8)
indicator_minimum	double	Limite minimale de l'axe verticale de l'indicateur
indicator_maximum	double	Limite maximale de l'axe verticale de l'indicateur
indicator_colorX	color	Indique la couleur de la mémoire tampon X (où X vaut de 1 à 8)
indicator_widthX	int	Indique l'épaisseur de la mémoire tampon X (où X vaut de 1 à 8)
indicator_styleX	int	Indique le style de la mémoire tampon X (où X vaut de 1 à 8)
indicator_levelX	double	Indique le niveau horizontal pour afficher une ligne dans une fenêtre séparée (8 fenêtres possibles)
indicator_levelwidth	color	Indique l'épaisseur des niveaux horizontaux
indicator_levelcolor	int	Indique la couleur des niveaux horizontaux
indicator_levelstyle	int	Indique le style des niveaux horizontaux
show_confirm	void	Affiche une fenêtre de confirmation avant l'exécution d'un script
Constante	Type	Description
show_inputs	void	Affiche la fenêtre de paramétrage avant l'exécution d'un script

Tableau 12: Directives #property

Voici des exemples possibles d'utilisation de la directive *#property* vous permettant de voir la syntaxe dans chacun des cas.

```
#property link "www.siteinternet.com"
#property copyright "Nom de l'auteur"
#property stacksize 1024
#property library
#property indicator_chart_window
#property indicator_separate_window
#property indicator_buffers 3
#property indicator_minimum 0
#property indicator_maximum 100
#property indicator_color2 Red
#property indicator_width3 3
#property indicator_style1 DRAW_LINE
#property indicator_level6 30
#property indicator_levelwidth 1
#property indicator_levelcolor Blue
#property indicator_levelstyle STYLE_DOT
#property show_confirm
#property show_inputs
```

Directive #include

Cette directive peut être placée n'importe où dans votre code mais il est d'usage de placer cette dernière au début pour simplifier la lecture par la suite. Elle sert à indiquer au programme que vous désirez aller chercher le contenu d'un fichier et de le copier à l'endroit où figure votre directive. Elle sert le plus souvent à récupérer le contenu d'une librairie. Une librairie est un fichier contenant certaines informations non essentielles au fonctionnement et par conséquent non disponibles directement dans le code source de MetaTrader. Il peut s'agir par exemple comme nous le verrons dans le chapitre consacré à la gestion des erreurs, du fichier contenant la liste des codes de toutes les erreurs possibles et leur équivalent en chaîne de caractère.

La syntaxe est la suivante:

```
#include <stdlib.mqh>
#include "WinUser32.mqh"
```

Vous remarquerez que dans le premier cas, nous avons inséré le nom du fichier entre les caractères inférieur et supérieur « <> » et dans le deuxième cas entre guillemets « " ». La différence est que lorsque le nom est entre « <> », le programme va chercher ce fichier dans le dossier par défaut qui est *MetaTrader\experts\include*. Lorsque le nom du fichier est entre guillemets, ce dernier est cherché dans le dossier où se trouve le fichier contenant le code source de votre programme. Si le fichier n'est pas trouvé, le compilateur vous renverra une erreur.

Directive #import

La directive *#import* ressemble comme deux gouttes d'eau à *#include* mais la différence vient du fait qu'il faut utiliser *#import* lorsque vous désirez copier le contenu d'un fichier *.ex4 ou *.dll si vous désirez utiliser des fonctionnalités Windows.

La syntaxe est la suivante :

```
#import "FichierImport.ex4"

#import "FichierWindows.dll"
  int fFonctionImport(int iArg);
  double fFonctionImport2(int iArg2, int iArg3);
#import
```

Vous ne pouvez importer que des fichiers au format *.ex4 car il est obligatoire que ces derniers soit compilés avant d'être importés.

Dans le cas des *.dll, il sera nécessaire de déclarer les fonctions à importer - par conséquent, la syntaxe commence avec la directive #import contenant le nom du fichier puis lorsque vous avez terminé la déclaration des fonctions, vous fermez avec *#import*. Vous ne pourrez utiliser que les fonctions que vous aurez déclarées.

9
Structure

Nous ne reviendrons pas sur la construction du programme avec les fonctions *init()*, *start()* et *deinit()* qui ont déjà été abordées précédemment mais plutôt sur quelques astuces pour vous faciliter la vie ainsi qu'à tout ceux qui liront votre code source par la suite.

- Ne pas hésiter à sauter des lignes (idéalement, regrouper toutes les expressions qui ont un rapport et les séparer du reste). Par exemple, regrouper ensemble toutes les déclarations et ensuite sauter une ligne avant d'attaquer le reste de votre code.

- Utiliser des indentations. Ceci vous facilitera la tâche lorsque viendra le moment de retrouver une boucle ou autre à l'intérieur de votre code. Pour les indentations, la règle basique à suivre et de ne pas indenter si les instructions ne sont pas imbriquées les unes dans les autres. L'indentation est en général de deux espaces.

Par exemple :

```
extern int iPrix;
extern int iPrice;
```

Aucune raison d'indenter ici car les deux expressions sont indépendantes l'une de l'autre.

Par contre :

```
for(i = 0 ; i <= 10 ; i++)
  j++;
```

j++ dépend de l'expression du dessus et par conséquent, on l'indente pour bien identifier la dépendance.

Même chose avec les crochets, indenter au niveau de la fonction de départ.

Par exemple :

```
for(i = 0 ; i < 10 ; i++)
  {
  j++;
  for(l = 0 ; l < 5 ; l++)
    {
    h++;
    iPrice = 1 + h;
    }
  }
```

Nous avons ici deux boucles *for* imbriquées. Vous remarquerez que les crochets de fermeture se situent sur la même ligne verticale que ceux d'ouverture afin de bien délimiter chaque boucle.

Bien que vous puissiez quasiment écrire votre code comme bon vous semble, il est vivement conseillé d'aller à la ligne entre chaque expression. Toutes les expressions suivantes sont correctes mais le niveau de difficulté de lecture de chacune est plus ou moins élevé.

Dans le cas suivant, nous déclarons 3 variables décimales en même temps. Cette façon de déclarer ne permet pas de définir les variables.

```
double  dPrix, dPrice, dPrecio;
```

Même cas que précédemment sauf que cette fois-ci, un saut à la ligne sépare chaque variable. Tant que le programme ne trouve pas de « ; » lui indiquant la fin de l'instruction, il considère cette dernière comme valide sans tenir compte des sauts à la ligne.

```
double
dPrix,
dPrice,
dPrecio;
```

De la même façon que les sauts à la ligne, placer des espaces entre variables n'affecte pas la lecture de l'instruction.

```
double        dPrix,      dPrice    ,      dPrecio;
```

Tant que le programme ne lit pas de « ; » ou « } », cela signifie que l'instruction n'est pas terminée. Il est techniquement possible d'écrire un code entier sur une seule et unique ligne bien que ce ne soit pas du tout conseillé.

```
double dPrix;double dPrice;double dPrecio;
```

Il est relativement plus simple d'écrire votre code comme suit.

```
double dPrix;
double dPrice;
double dPrecio;
```

Même si avez le droit d'inclure des espaces ou sauts de ligne dans vos expressions, n'oubliez pas que les espaces et sauts de ligne ne peuvent pas couper un nom, une valeur ou autre.

Le code suivant est incorrect :

```
double dPri     x;
```

10
Commentaires

Comme tout langage de programmation, le MQL4 permet d'insérer des commentaires au sein du code. Cette possibilité vous permet d'inclure des indications ou remarques destinées à vous-même ou à tous ceux qui liront votre code.

Il existe deux types de commentaires en MQL4 : les commentaires sur une seule ligne ou les commentaires multi-lignes.

Commentaires sur une seule ligne

Pour insérer un commentaire sur une seule ligne, il vous suffit de l'introduire par « // ». Lorsque le programme verra ces deux caractères, il va interpréter tout ce qui suit sur la ligne comme un commentaire. Vous n'avez donc pas besoin de fermer la ligne de commentaire mais vous ne pouvez pas insérer de saut de ligne ou une expression de votre code à la suite de ces deux caractères.

Par exemple :

```
double dPrix; //Ceci est un commentaire

//Ceci est un autre commentaire
//Ceci est aussi un commentaire
```

Ce type de commentaire est spécialement utile lorsque vous désirez ajouter une indication concernant une expression précise.

Commentaires multi-lignes

Pour des commentaires plus généraux comme une présentation du programme qui nécessitera plusieurs lignes, vous pouvez bien évidemment utiliser une suite de commentaires sur une seule ligne ou encore utiliser les délimiteurs suivants : « /* » pour ouvrir le commentaire multi-lignes et « */ » pour le fermer. Entre ces deux délimiteurs, vous pouvez insérer autant de lignes que vous le désirez.

Par exemple :

```
/* première ligne de commentaire
deuxième ligne de commentaire
troisième ligne de commentaire
etc... */
```

Contrairement aux commentaires sur une seule ligne, le commentaire multi-lignes peut être inséré à l'intérieur d'une expression puisqu'il possède un délimiteur fermant.

Par exemple :

```
double /* Ceci est un commentaire */dPrix;
```

Commentaires dans « Votre premier expert consultant »

Afin d'alléger le code dans les chapitres suivant, il n'y aura pas de commentaires. Pour voir le code source complet largement commenté, veuillez-vous référer à l'annexe B.

11
Votre premier expert consultant #0

La première étape avant de se lancer dans la conception de tout expert consultant est de s'assurer que la stratégie est bel et bien programmable. En effet, parfois, certaines stratégies s'appuient sur un sentiment ou un sixième sens qu'il est malheureusement impossible de reproduire dans un programme. Il est donc nécessaire de s'assurer de la faisabilité du projet en se posant quelques questions :

- Est-ce que mon jugement personnel intervient dans ma stratégie?
- Est-il possible d'expliquer ma stratégie à quelqu'un d'autre?
- Est-il possible de dessiner un algorithme représentant la stratégie?

La réponse à la première question devra être non et la réponse aux deux suivantes oui. Même si vous savez déjà que votre stratégie est effectivement programmable, il est parfois judicieux et pratique de dessiner tout de même l'algorithme pour savoir dans quelle direction vous vous en allez. Cette représentation graphique peut vous permettre de détecter d'éventuelles failles dans le raisonnement et vous permettre de gagner un temps précieux lors de la transcription de la stratégie en code.

Avant de dessiner l'algorithme, voici une courte description de l'expert consultant que nous allons concevoir tout au long de ce livre (cette description inclut uniquement la logique et non pas toutes les étapes) :

Description en une phrase : *L'expert récupèrera tous les jours le prix le plus haut etle plus bas de la veille et se servira de ces derniers comme point d'entrée à l'achat et à la vente respectivement.*

Logique de l'expert (dans l'ordre d'apparition des instructions dans le code) :

1. Détermination des heures du filtre horaire en secondes.
2. Vérification si l'expert a déjà placé des ordres (protection en cas de redémarrage).
3. Affichage ou mise à jour du calendrier économique sur le côté gauche du graphique.
4. Détermination de l'heure actuelle en secondes.
5. Vérification si il y a lieu de modifier les stops des éventuelles positions ouvertes en se basant sur le stop suiveur.
6. Vérification si l'heure actuelle est dans la plage du filtre horaire.
    a. Vérification si les niveaux ont déjà été récupérées. Si non, récupération et traçage des niveaux sur le graphique.
    b. Vérification si l'expert peut passer un ordre. Si non, l'expert patiente une seconde avant de revérifier.
    c. Vérification si le prix du marché a atteint le niveau plus haut de la veille. Si c'est le cas, placement d'un ordre. Deux filtres pourront empêcher le passage d'ordre : le premier basé sur les conditions minimales requises par le courtier et le second basé sur un indicateur personnalisé.

d. Vérification si l'expert peut passer un ordre. Si non, l'expert patiente une seconde avant de revérifier.
e. Vérification si le prix du marché a atteint le niveau plus bas de la veille. Si c'est le cas, placage d'un ordre. Deux filtres pourront empêcher le passage d'ordre : le premier basé sur les conditions minimales requises par le courtier et le second basé sur un indicateur personnalisé.

7. Lors du passage à une nouvelle journée, l'expert récupèrera les nouveaux niveaux les plus haut et bas de la veille.

Algorithme de l'expert (réalisé à l'aide de http://live.gnome.org/Dia) :

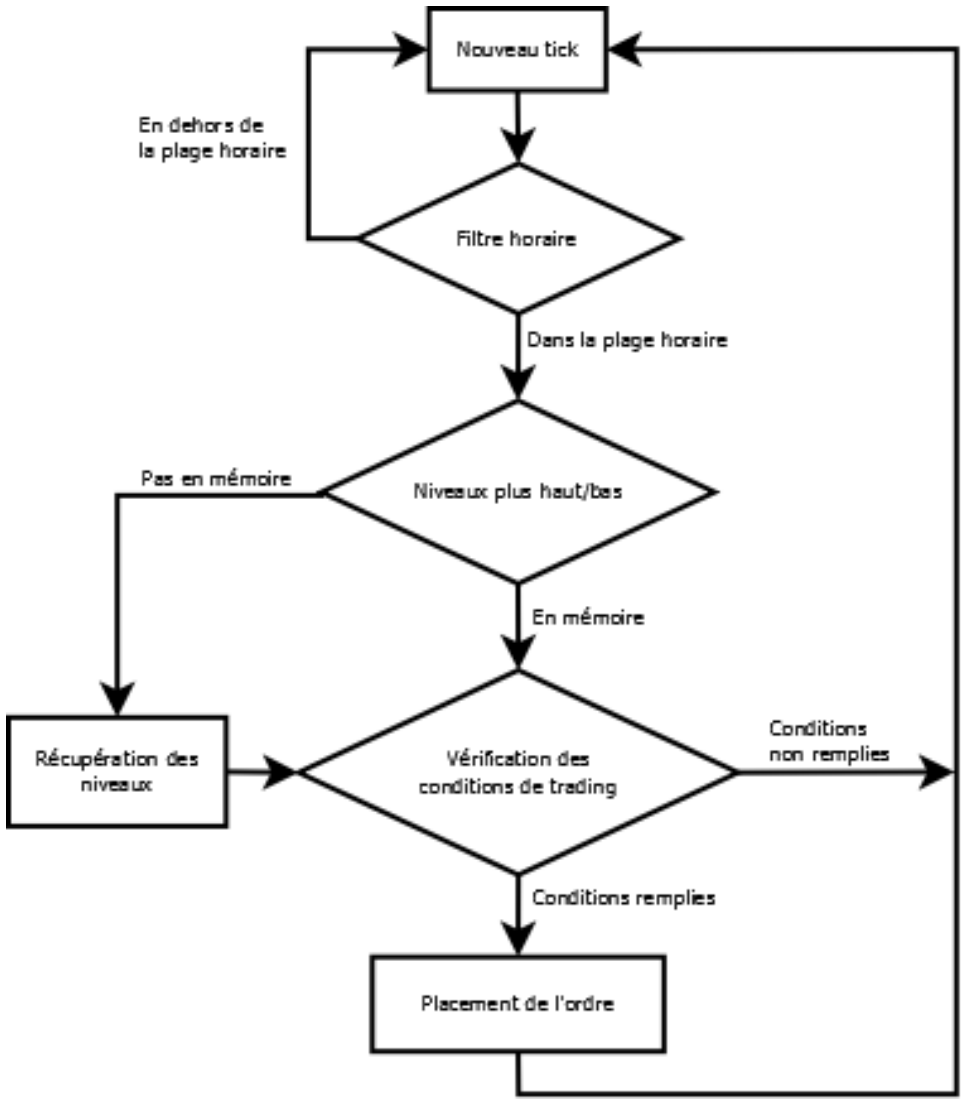

Figure 5 : Algorithme de l'expert "Votre Premier Expert"

12
Votre premier expert consultant #1

Lancer MetaEditor

Pour lancer MetaEditor, plusieurs choix s'offrent à vous, vous pouvez soit aller chercher le raccourci via le menu *démarrer* ou directement dans le dossier d'installation de MetaTrader, soit lancer MetaTrader et utiliser l'un des raccourcis suivants :

- Cliquez sur *F4*
- Aller dans le menu *Outils* et sélectionnez *MetaExpert Editor*
- Cliquez sur le raccourci dans la barre d'outils en haut (cerclé de rouge ci-dessous)

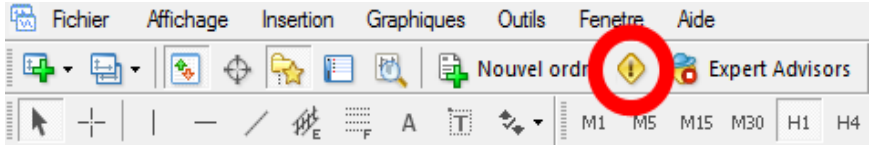

Figure 6 : Icône MetaEditor dans la barre d'outils MetaTrader

Lors du premier démarrage, la fenêtre MetaEditor devrait ressembler à la capture ci-dessous.

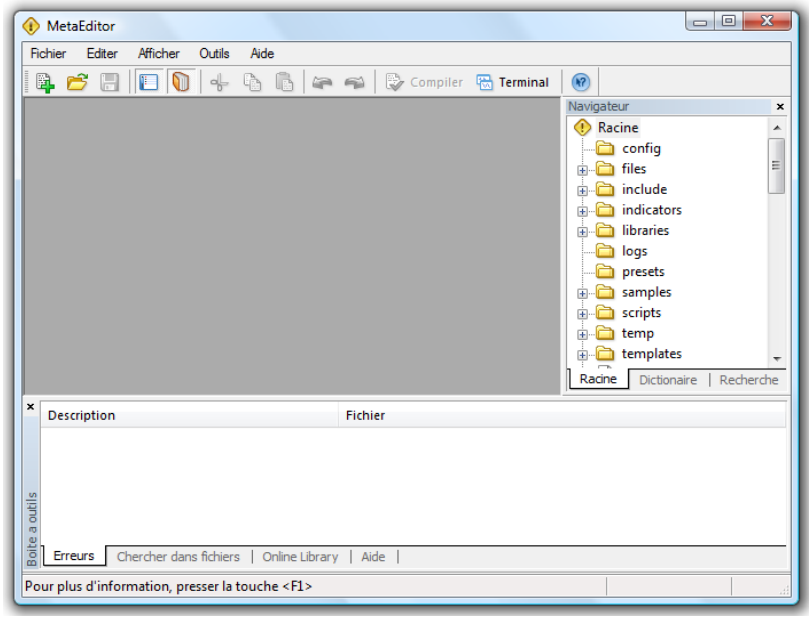

Figure 7 : MetaEditor

Créer un nouvel expert consultant

Pour créer un nouvel expert, indicateur ou script, il suffit de se rendre dans le menu *Fichier* et de sélectionner *Nouveau* (la plupart des raccourcis génériques présents dans les programmes Windows fonctionnent également dans MetaEditor, vous pouvez donc également faire Ctrl+N pour créer un nouveau document).

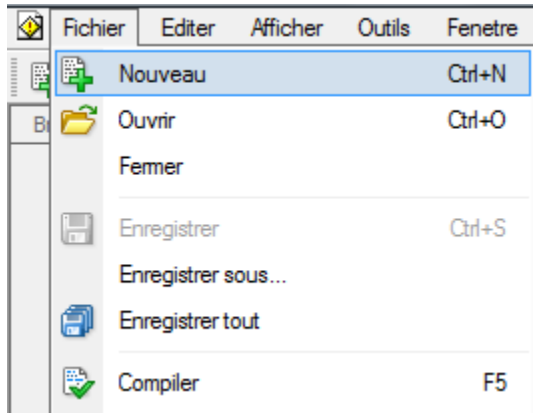

Figure 8 : Menu Fichier

La fenêtre suivante va alors s'ouvrir vous permettant de sélectionner le type de document que vous souhaitez créer. Dans notre cas, nous allons donc choisir *Programme Expert Advisor* avant de cliquer sur *Suivant >*.

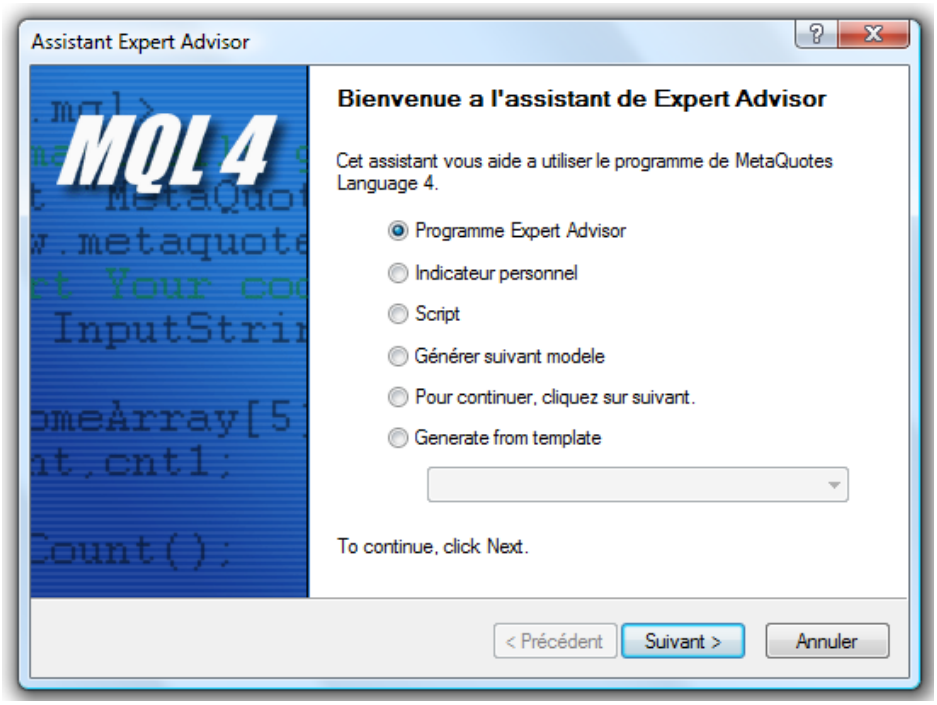

Figure 9 : Fenêtre de création d'un nouveau programme

Vous devez maintenant choisir les paramètres généraux de l'expert tels que le nom, l'auteur et un lien vers le site du créateur (les deux derniers paramètres sont facultatifs).

Vous pouvez voir également une option *Paramètres* - nous n'allons rien ajouter pour l'instant mais sachez simplement que cette option permet de déclarer par avance les variables externes de votre programme. Pour valider vos choix, cliquez sur *Terminer*.

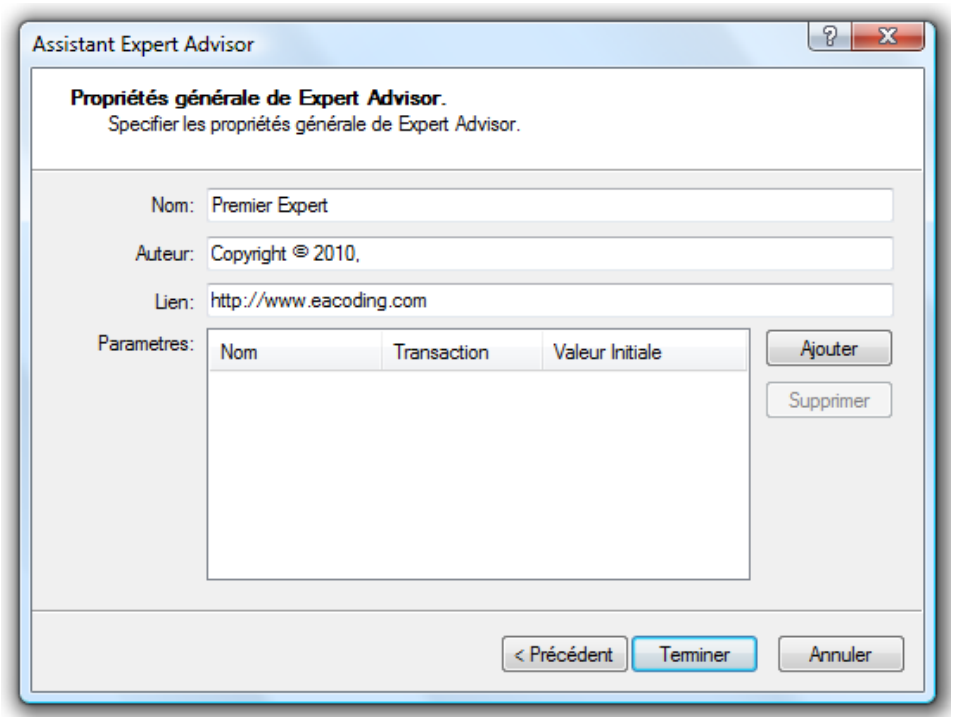

Figure 10 : Assistant création expert consultant

Le nouveau document va alors s'ouvrir. Vous noterez que ce dernier comporte déjà du code ainsi que des commentaires. En effet, MetaEditor a créé un canevas par défaut pour un expert consultant. C'est sur cette structure que vous allez insérer votre propre code.

Compiler son expert

Il ne nous reste plus qu'à compiler notre expert afin de valider le code et créer un fichier *.ex4 exploitable par MetaTrader. Pour ce faire, il suffit de cliquez sur le bouton *Compiler* présent dans la barre d'outils en haut.

Figure 11 : Bouton compiler dans la barre d'outils de MetaEditor

Vous devriez voir apparaître le message suivant dans la fenêtre du bas de votre éditeur (fenêtre *Boîte à Outils*) : « Compile 'Premier Expert.mq4'… » et à la ligne « 0 erreur(s), 0 alerte(s) ».

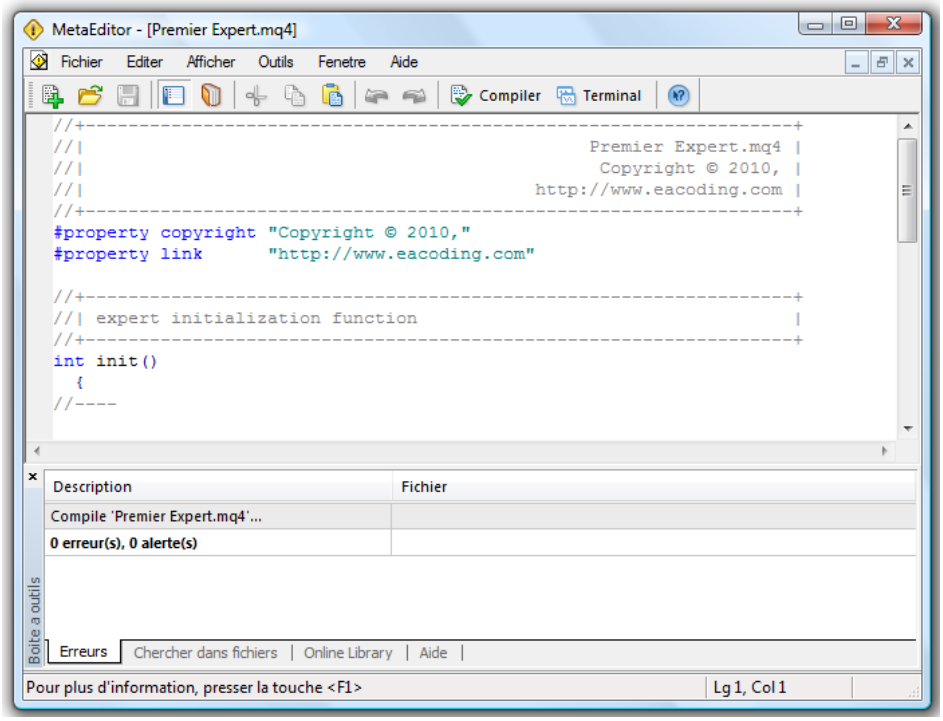

Figure 12 : Compilation de Premier Expert réussie

Même si votre programme ne peut rien faire pour l'instant- la syntaxe est correcte et donc compilable. Vous devriez maintenant voir dans la liste de vos experts dans MetaTrader, votre expert Premier Expert nouvellement compilé.

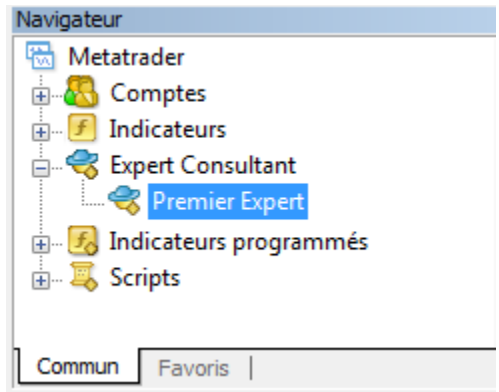

Figure 13 : Premier Expert dans la fenêtre navigateur de MetaTrader

Le code obtenu est le suivant :

```
//+------------------------------------------------------------------+
//|                                Votre Premier Expert.mq4          |
//|                                  Copyright © 2011,               |
//|                               http://www. eole-trading.com       |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www. eole-trading.com"


//+------------------------------------------------------+
//| expert initialization fonction                     |
//+------------------------------------------------------+
int init()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------+
//| expert deinitialization fonction                   |
//+------------------------------------------------------+
int deinit()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------+
//| expert start fonction                              |
//+------------------------------------------------------+
int start()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------+
```

13
Votre premier expert consultant #2

Maintenant que nous avons la base de l'expert, vous allez lui ajouter quelques lignes de codes lui permettant de repérer et garder en mémoire le plus haut et le plus bas de la journée précédente. À chaque nouvelle journée, l'expert réinitialisera les variables afin de toujours avoir le plus haut/bas de la journée précédente.

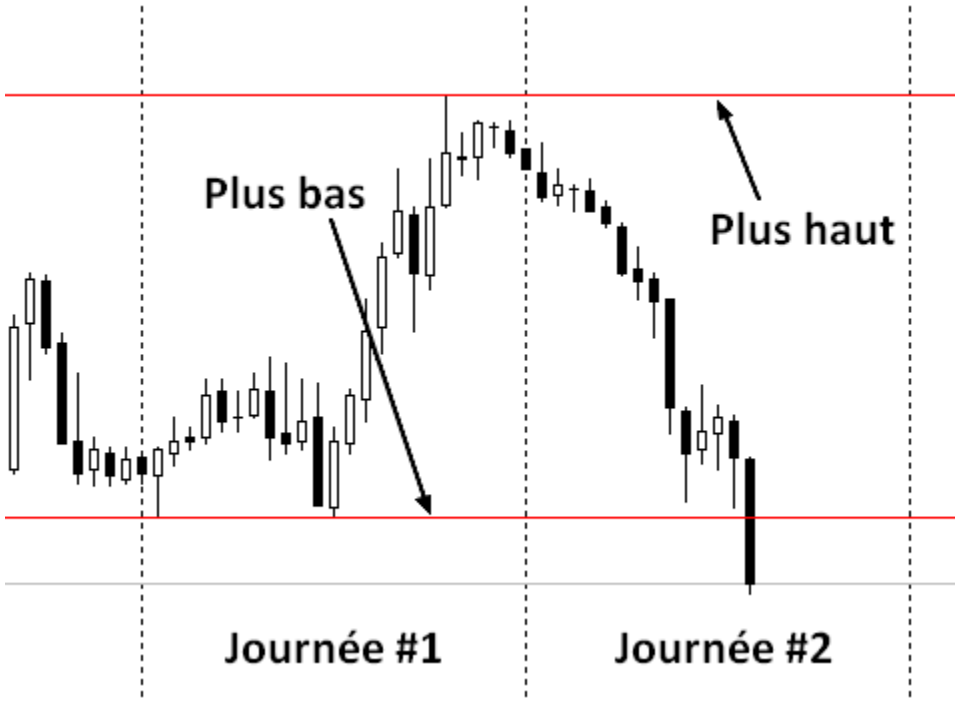

Figure 14 : Plus haut/Plus bas journée précédente

Dans la figure 14, Journée #2 est la journée actuelle en cours et Journée #1, la journée précédente. Le programme devra donc être capable de repérer le plus haut et le plus bas de la Journée #1 afin de les utiliser au cours de la Journée #2.

La fonction pour repérer le plus haut ou le plus bas d'une chandelle est la suivante :

```
double iHigh(string sPaire, int iUniteTemps, int iDecalage)

//remplacer iHigh par iLow pour le plus bas
```

Le type de la fonction est *double* (ce qui est logique compte tenu du fait que la valeur renvoyée sera soit un prix soit un réel). Après le type, vient le nom de la fonction puis les paramètres avec dans l'ordre, la paire, l'unité de temps et le décalage souhaité.

Dans l'idéal, l'expert devra être capable de s'adapter à toutes les paires. À la place de *string sPaire*, il faut donc indiquer *NULL* ou *Symbol()* comme nous le verrons dans le chapitre suivant pour signifier que la fonction doit utiliser la paire du graphique sur lequel est placé l'expert.

La fonction *iHigh* (ou *iLow*) ne peut que repérer le plus haut (le plus bas) d'une chandelle spécifique. Pour obtenir les données pour la journée précédente, il faudra donc indiquer à la place de *int iUniteTemps*, *PERIOD_D1* comme unité de temps de façon à ce que l'expert renvoie les bonnes valeurs et ce peu importe l'unité de temps du graphique. Enfin pour le paramètre *int iDecalage* correspondant au décalage, le réglage sera 1 afin que l'expert aille chercher les valeurs pour la chandelle précédantl'actuelle.

Le fait de préciser *PERIOD_D1* vous permet de placer votre expert sur un graphique avec une autre unité de temps qu'une journée (période de 24 heures par chandelle) et de le mettre en mesure de récupérer les bonnes valeurs quand même.

En langage MQL4, le décompte des barres se fait à l'envers. La dernière chandelle, soit celle en cours est numérotée 0 et le décompte est croissant au fur et à mesure que l'on va dans le passé. Vous trouverez ci-dessous un graphique pour illustrer ce phénomène.

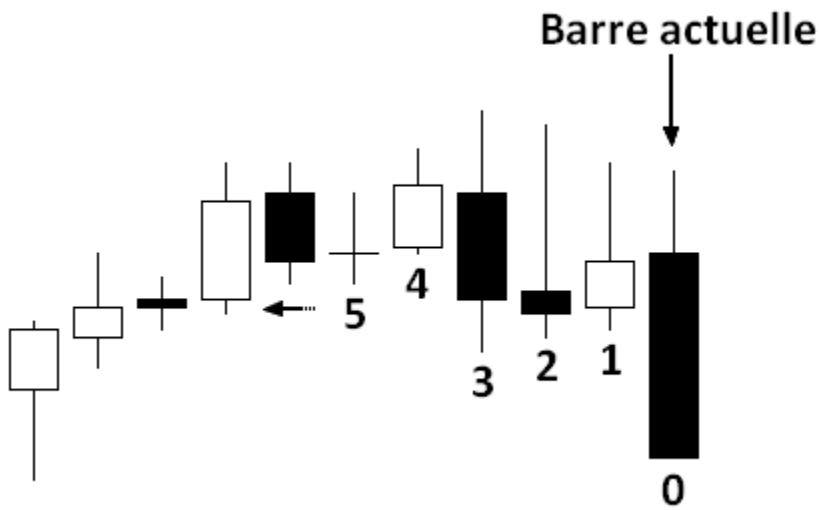

Figure 15 : Décompte des barres en MQL4

Le code pour obtenir le plus haut/bas et assigner ces derniers à des variables sera donc le suivant :

```
double dPlusHaut = iHigh(Symbol(), PERIOD_D1, 1);

double dPlusBas = iLow(Symbol(), PERIOD_D1, 1);
```

Vous remarquerez la nécessité d'avoir des variables de type *double* afin que celles-ci soient compatibles avec la valeur renvoyée par les fonctions *iHigh* et *iLow*. Dorénavant, il suffira donc de faire appel à ces deux variables pour avoir le plus haut/bas de la journée précédente.

L'opération de récupération du plus haut/bas n'ayant besoin d'avoir lieu que toutes les 24 heures, c'est à dire au moment où une nouvelle journée débute, des instructions conditionnelles vont être ajoutées à notre code afin de préciser à l'expert de n'effectuer les opérations qu'une fois que la date a changé et ce une seule fois (en effet, le plus haut/bas de la journée précédente ne va pas changer une fois la nouvelle journée entamée).

Le code incluant les instructions conditionnelles serait donc le suivant :

```
if(bRecuperation == False)
  {
  iDate = Day();
  dPlusHaut = iHigh(Symbol(), PERIOD_D1, 1);
  dPlusBas = iLow(Symbol(), PERIOD_D1, 1);
  bRecuperation = True;
  }

if(iDate != Day())
  bRecuperation = False;
```

La première instruction stipule que si la valeur de la variable de type booléen *bRecuperation* est *False*, l'expert va assigner la date de la dernière chandelle obtenue à l'aide de la fonction *Day()* à la variable *iDate*, le plus haut et le plus bas aux variables *dPlusHaut* et *dPlusBas* et enfin assigner la valeur *True* à la variable *bRecuperation*.

La deuxième instruction compare la date que nous avons stockée au cours de la première instruction à la date du jour en cours. Si les deux dates sont différentes, l'expert va assigner la valeur *False* à notre variable *bRecuperation*.

Les lignes de code ci-dessus doivent aller dans la fonction *start()* de notre programme afin que l'expert soit à même de vérifier la date à chaque tick (c'est-à-dire à chaque nouvelle cotation).

Le code source actualisé est donc (les ajouts ou modifications sont en gras) :

```
//+------------------------------------------------------------------+
//|                          Votre Premier Expert.mq4                |
//|                           Copyright © 2011,                      |
//|                        http://www. eole-trading.com              |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www. eole-trading.com"

int iDate;
double dPlusHaut;
double dPlusBas;
bool bRecuperation = False;

//+------------------------------------------------------------------+
//| expert initialization fonction                                   |
//+------------------------------------------------------------------+
int init()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------------------+
//| expert deinitialization fonction                                 |
//+------------------------------------------------------------------+
int deinit()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------------------+
//| expert start fonction                                            |
//+------------------------------------------------------------------+
int start()
  {

  if(bRecuperation == False)
    {
    iDate = Day();
    dPlusHaut = iHigh(Symbol(), PERIOD_D1, 1);
    dPlusBas = iLow(Symbol(), PERIOD_D1, 1);
    bRecuperation = True;
    }

  if(iDate != Day())
    bRecuperation = False;

  return(0);
  }
//+------------------------------------------------------------------+
```

Les ajouts sont donc :

```
int iDate;
double dPlusHaut;
double dPlusBas;
bool bRecuperation = False;
```

Déclaration des variables et initialisation de la variable *bRecuperation* afin que lors du premier lancement, la condition soit respectée et l'expert effectue les opérations voulues. Après le premier passage, *bRecuperation* est paramétrée sur *True* afin que la première boucle soit ignorée tant et aussi longtemps que la date sera la même.

```
if(bRecuperation == False)
  {
  iDate = Day();
  dPlusHaut = iHigh(Symbol(), PERIOD_D1, 1);
  dPlusBas = iLow(Symbol(), PERIOD_D1, 1);
  bRecuperation = True;
  }

if(iDate != Day())
  bRecuperation = False;
```

Vous noterez que toutes les variables ont été déclarées au début du programme et pas à l'intérieur des instructions. Si la déclaration avait eu lieu à l'intérieur des instructions conditionnelles, la valeur des différentes variables serait faussée car le programme n'aurait assigné les valeurs que tout au long de l'instruction mais pas pour tout le programme.

Par exemple, si nous avions écrit à l'intérieur de la première condition :

```
if(bRecuperation == False)
  {
double dPlusBas = iLow(NULL, PERIOD_D1, 1);
  }
```

Et plus loin dans le programme, nous décidions de faire appel à la variable *dPlusBas*, la valeur renvoyée au moment de la première itération aurait été la bonne valeur mais aux passages suivants, le programme aurait renvoyé 0.0000 et non plus le prix du plus bas de la chandelle.

Afin d'accélérer le processus de calcul, les variables auraient aussi pu être déclarées au début de la fonction *start()* et non pas au début du code source.

14
Fonctions sur les barres

Dans le chapitre « Votre expert consultant #2 », vous avez utilisé les fonctions *iHigh()* et *iLow()* pour permettre à votre code d'obtenir le plus haut et le plus bas d'une chandelle spécifique.

Il existe également une version simplifiée de ces fonctions qui s'écrivent tout simplement *High[]* et *Low[]*. La différence principale entre les fonctions avec un *i* devant et celles sans le *i* (par exemples *iHigh()* et *High[]*) est que ces dernières sont prédéfinies par défaut. C'est à dire que la majorité des paramètres que vous définissez lorsque vous utilisez la fonction avec un *i* tels que *symbol* ou *TimeFrame* sont déjà définis. En fait, vous ne pouvez indiquer que le décalage (shift) souhaité et la fonction s'adaptera automatiquement à la paire et l'unité de temps du graphique sur lequel votre expert est placé. Un changement effectué sur votre graphique influencera donc directement le résultat dans ce cas.

Alors pourquoi avons-nous utilisé *iHigh* et *iLow* dans notre code source ?

Et bien parce que nous désirions avoir les valeurs de la journée précédente et ce peu importe l'unité de temps du graphique. Si nous avions utilisé *High[]* et *Low[]* comme fonctions en indiquant un décalage de 1, nous n'aurions obtenu les bonnes valeurs que lorsque l'unité de temps du graphique aurait été D1 soit quotidienne.

Nous allons maintenant passer chaque fonction en revue en précisant à chaque fois la version simplifiée et la version en i et décrire une utilisation possible de la fonction.

Fonctions Bars et iBars

Ces fonctions permettent d'obtenir le nombre de chandelles (ou barres) présentes sur votre graphique. La fonction Bars peut s'utiliser telle quelle dans vos expressions. Le type de la valeur renvoyée est *int* soit un entier.

Par exemple :

```
if(Bars < 10 )
  return(0);
```

L'expression ci-dessous indique que si il y a moins de 10 barres sur le graphique, il faut arrêter l'exécution de l'instruction courante. Cette fonction peut être très utile si votre expert nécessite un nombre minimum de barres pour effectuer ses opérations (par exemple, la volatilité moyenne au cours des 30 dernières barres).

La syntaxe de la fonction *iBars* est la suivante avec à la place de *string sPaire* la paire de votre choix (*NULL* si paire du graphique courant) et l'unité de temps de votre choix à la place de *int iUniteTemps*.

```
int iBars(string sPaire, int iUniteTemps)
```

Vous pouvez écrire l'unité de temps sous forme écrite ou sous forme d'entier en indiquant le nombre de minutes. Le tableau 13 liste les choix possible.

Forme littérale	Forme numérique	Équivalent
PERIOD_M1	1	1 minute
PERIOD_M5	5	5 minutes
PERIOD_M15	15	15 minutes
PERIOD_M30	30	30 minutes
PERIOD_H1	60	1 heure
PERIOD_H4	240	4 heures
PERIOD_D1	1440	1 jour
PERIOD_W1	10080	1 semaine
PERIOD_MN1	43200	1 mois
NULL	0	Unité de temps du graphique

Tableau 13 : Unités de temps

Le nom de la paire doit être inséré entre guillemets - par contre si vous indiquez *NULL*, les guillemets ne sont pas nécessaires.

Voici quelques exemples :

```
// Syntaxe pour la paire GBPUSD en unité de temps H4
iBars("GBPUSD", PERIOD_H4)

// Syntaxe pour la paire et l'unité de temps du graphique
iBars(NULL, 0)

// Utilisation possible dans une instruction
if(iBars("GBPUSD",PERIOD_H4) < 10)
  return(0);

// Assignation à une variable
int iBarres = iBars("GBPUSD", PERIOD_M1);
```

Si il n'y a pas de graphique ouvert pour la paire dont vous souhaitez obtenir le nombre de barres, MetaTrader vous renverra l'erreur suivante : *#4066 - ERR_HISTORY_WILL_UPDATED*. Cette erreur signifie que les données étaient introuvables au moment de la tentative et que MetaTrader est en train de mettre à jour les données pour cette paire. Il faudra donc vous assurer d'insérer une instruction pour permettre à votre fonction de retenter l'obtention des données si cette erreur venait à se produire.

Fonctions Close[] et iClose()

Comme leurs noms l'indiquent, ces fonctions permettent d'obtenir le prix de clôture d'une chandelle. De la même façon que précédemment, la fonction *Close[]* peut être utilisée telle quelle si vous désirez obtenir la clôture d'une chandelle du graphique sur lequel votre programme est activé. Le type de la valeur renvoyée est *double* soit un réel puisqu'il s'agit d'un prix.

```
for(int i=1; i < 3; i++)
  {
  if(Close[1] < Close[i+1])
    bBaissier = true;
  }
```

L'expression ci-dessus analyse les deux barres précédant la barre courante de votre graphique pour savoir si la clôture de l'avant-dernière barre a eu lieu en dessous du niveau de la précédente ou non. Si tel est le cas, la valeur *True* va être attribuée à la variable booléenne *bBaissier* que vous aurez déclarée préalablement. Le graphique suivant illustre le fonctionnement de cette portion de code.

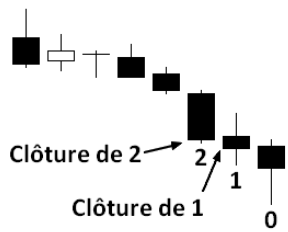

Figure 16: Analyse des 2 barres précédentes

La boucle *for* de l'expression va analyser les barres 1 et 2 étant donné que le décompte commence à 1 (i=1) et s'arrête à 2 puisque i doit être strictement inférieur à 3. Si le prix de clôture de 1 est inférieur au prix de clôture de 2, la variable booléenne *bBaissier* recevra la valeur *True*. Cette expression peut être utile si vous désirez implanter un filtre de tendance. Par exemple, si les 3 dernières barres ont toutes fermé en dessous de la précédente, vous déclarez que le marché est baissier et que votre expert ne doit prendre que des positions à la vente.

Une façon plus simple d'écrire le code précédent aurait pu être :

```
if(Close[1] < Close[2])
  bBaissier = true;
```

L'instruction ci-dessus vérifie si la clôture de la chandelle 1 est inférieure à celle de la chandelle 2 et si c'est le cas, assigne la valeur *True* à la variable *bBaissier*.

L'avantage de la première façon d'écrire le code comparé à celle-ci est de permettre d'étendre l'analyse à plus de deux barres. Vous pouvez par exemple choisir i=6 pour comparer la clôture de l'avant-dernière barre avec les 5 barres précédentes.

Si vous désirez aller chercher les informations d'une autre paire ou pour une unité de temps spécifique différente de celle du graphique, vous devrez alors utiliser la fonction *iClose()*.

Les syntaxes sont les suivantes :

```
double Close[int iDecalage]

double iClose(string sPaire, int iUniteTemps, int iDecalage)
```

À la place de *string sPaire*, *int iUniteTemps* et *int iDecalage*, vous devez respectivement indiquer la paire de votre choix, l'unité de temps choisie et le décalage souhaité.

```
// Déclaration des variables nécessaires à faire au début du code source

int iCompteur;
int iBarre=3;

// Portion de code incluant la fonction étudiée

iCompteur = 0;
for(int i = iBarre; i > 1; i--)
  {
  if(iClose("EURUSD", Period_H1, i) > iClose("EURUSD", Period_H1, i-1))
    count++;
  }

if(iCompteur == iBarre-1)
  bBaissier = true;
```

L'expression ci-dessus va analyser les barres 1, 2 et 3 (Référez-vous au graphique explicatif précédent pour identifier les barres).

La boucle commence avec i=3. Si la barre 3 a un prix de clôture supérieur à la barre 2, la variable *iCompteur* prend la valeur 1 (0+1). La boucle continuera ensuite avec i=2 et analyse les clôtures des barres 2 et 1 et ajoute 1 à *iCompteur* si encore une fois la clôture de 2 est supérieure à celle de la barre 1. *iCompteur* vaut maintenant 2. À ce stade, i=1 donc la boucle stoppe et on passe à la suite du programme. Si *iCompteur* = 2 (barre (=3) - 1 = 2), cela signifie que les clôtures des barres 1, 2 et 3 sont respectivement inférieures les unes par rapport aux autres (1<2<3) et donc que notre marché est baissier.

Fonctions Open[] et iOpen()

Le fonctionnement et le mode d'utilisation de *Open[]* et *iOpen()* sont similaires à ceux de *Close[]* et *iClose()* excepté que ces fonctions vous permettent d'obtenir le prix d'ouverture des barres et non le prix de clôture.
Les syntaxes sont les suivantes :

```
double Open[int iDecalage]

double iOpen(string sPaire, int iUniteTemps, int iDecalage)
```

À la place de *string sPaire*, *int iUniteTemps* et *int iDecalage*, vous devez respectivement indiquer la paire de votre choix, l'unité de temps choisie et le décalage souhaité.

Le code ci-dessous permet d'obtenir le prix d'ouverture et clôture de l'avant-dernière barre et calcule la distance en pips entre les deux. Le code ne fait pas encore la distinction entre une barre haussière ou baissière donc il est possible que la variable *iTailleBarre* soit négative.

```
// Déclaration des variables nécessaires à faire au début du code source

double dPrixOuverture;
double dPrixFermeture;
int iTailleBarre;

// Portion de code incluant les fonctions étudiées

dPrixOuverture =(iOpen("EURUSD", PERIOD_M15, 1));
dPrixFermeture =(iClose("EURUSD", PERIOD_M15, 1));
iTailleBarre = (dPrixOuverture - dPrixFermeture) * 10000;
```

Comme la paire choisie est EURUSD, la valeur obtenue de la soustraction pour la variable *iTailleBarre* sera multiplié par 10000 afin d'obtenir la différence en pips et non en décimales.

Fonctions High[] et iHigh()

Et voilà l'une des fonctions que nous avons utilisées dans notre expert - ces deux fonctions servent à obtenir le plus haut d'une barre.

Les syntaxes sont les suivantes :

```
double High[int iDecalage]

double iHigh(string sPaire, int iUniteTemps, int iDecalage)
```

À la place de *string sPaire*, *int iUniteTemps* et *int iDecalage*, vous devez respectivement indiquer la paire de votre choix, l'unité de temps choisie et le décalage souhaité.

Une application possible avec cette fonction serait d'écrire une instruction permettant de calculer la taille de l'ombre supérieure d'une chandelle (distance entre le plus haut et l'ouverture ou la clôture de la chandelle selon que si la chandelle est baissière ou haussière).

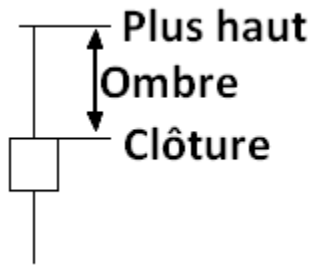

Figure 17 : Ombre d'une chandelle

Dans la figure 17, la chandelle est haussière, le programme serait donc censé calculer la distance entre le prix de clôture et le prix le plus haut.

```
// Déclaration des variables nécessaires à faire au début du code source

bool bBaissier;
int iTailleOmbre;

// Portion de code incluant la fonction étudiée

if(Open[1] > Close[1])
  bBaissier = 1;
else
  bBaissier = 0;

if(bBaissier == 1)
  iTailleOmbre = (High[1] - Open[1]) * 10000;
else
  iTailleOmbre = (High[1] - Close[1]) * 10000;
```

La première partie de l'expression sert à vérifier si la chandelle est baissière ou haussière et la deuxième partie effectue le calcul désiré.

Fonctions Low[] et iLow()

De la même façon que *High[]* et *iHigh()* permettent d'obtenir le plus haut d'une chandelle, *Low[]* et *iLow()* permettent d'obtenir le plus bas de cette dernière.

Les syntaxes sont les suivantes :

```
double Low[int iDecalage]

double iLow(string sPaire, int iUniteTemps, int iDecalage)
```

À la place de *string sPaire*, *int iUniteTemps* et *int iDecalage*, vous devez respectivement indiquer la paire de votre choix, l'unité de temps choisie et le décalage souhaité.

Deux exemples :

```
// Attribue le prix le plus bas de la chandelle située en position 10 à la variable dPlusBas
double dPlusBas = Low[10];


// Attribue le prix le plus bas de la chandelle située en position 2 de la paire // USDJPY de l'unité de temps du
// graphique à la variable plusBas
double dPlusBas = iLow("USDJPY", 0, 2);
```

Fonctions Time[] et iTime()

Les fonctions *Time[]* et *iTime()* servent à obtenir l'heure d'ouverture d'une chandelle spécifique.

Les syntaxes sont les suivantes :

```
datetime Time[int iDecalage]


datetime iTime(string sPaire, int iUniteTemps, int iDecalage)
```

À la place de *string sPaire*, *int iUniteTemps* et *int iDecalage*, vous devez respectivement indiquer la paire de votre choix, l'unité de temps choisie et le décalage souhaité.

Ces fonctions peuvent être utiles si vous désirez connaître l'heure d'ouverture de la chandelle et effectuer des actions en conséquence.

Par exemple :

```
// Déclaration des variables nécessaires à faire au début du code source

int iOpenBar;
double dPlusHaut;


// Portion de code incluant la fonction étudiée

if(iOpenBar != Time[0])
  {
  iOpenBar = Time[0];
  dPlusHaut = High[1];
  }
```

Le code ci-dessus vérifie l'heure d'ouverture de la chandelle en cours. Si celle-ci est différente de celle assignée à la variable *iOpenBar* (par défaut à 0 lorsque le programme est exécuté pour la première fois), l'heure d'ouverture de la chandelle sera assignée à la variable *iOpenBar* et le plus haut de la chandelle précédente assigné à la variable *dPlusHaut*. Ce code permet donc d'exécuter une série d'instructions lorsqu'une nouvelle chandelle débute.

Fonctions Volume[] et iVolume()

Volume[] et *iVolume()* permettent d'obtenir le volume correspondant au nombre de ticks qui se sont produits au cours de la formation de la chandelle spécifiée. Un tick correspond dans ce cas à un changement de prix se produisant au cours de la même chandelle. Cette fonction peut être utilisée pour vérifier l'existence d'une chandelle dans votre graphique.

Les syntaxes sont les suivantes :

```
double dVolume[int iDecalage]

double iVolume(string sPaire, int iUniteTemps, int iDecalage)
```

À la place de *string sPaire*, *int iUniteTemps* et *int iDecalage*, vous devez respectivement indiquer la paire de votre choix, l'unité de temps choisie et le décalage souhaité.

Fonctions iHighest() et iLowest()

iHightest() et *iLowest()* sont des fonctions très intéressantes qui permettent d'obtenir la position de la chandelle avec la donnée maximale/minimale de votre choix sur une période que vous définissez. La donnée peut être le plus grand/petit prix d'ouverture, de clôture ou encore le plus grand/petit maximum, minimum ou volume.

Les syntaxes sont les suivantes :

```
int iHighest(string sPaire, int iUniteTemps, int iType, int iDecompte=WHOLE_ARRAY, int iDebut=0)

int iLowest(string sPaire, int iUniteTemps, int iType, int iDecompte=WHOLE_ARRAY, int iDebut=0)
```

À la place de *string sPaire*, *int iUniteTemps*, *int iType*, *int iDecompte=WHOLE_ARRAY* et *int start=0*, vous devez respectivement indiquer la paire de votre choix, l'unité de temps choisie, la variable désirée (voir tableau 14), le nombre de barres à analyser et le numéro de la barre à partir de laquelle commencer l'analyse.

L'analyse s'effectue à l'envers - de la barre la plus récente vers le passé. Par conséquent, choisissez comme barre de départ pour votre analyse la barre la plus récente de votre période choisie.

Pour le paramètre *int iType*, les choix possibles sont détaillés dans le tableau ci-dessous, vous pouvez aussi bien utiliser la forme littérale que la forme entière équivalente à votre convenance.

Forme littérale	Forme numérique	Équivalent
MODE_OPEN	0	Prix d'ouverture
MODE_LOW	1	Prix minimum
MODE_HIGH	2	Prix maximum
MODE_CLOSE	3	Prix de clôture
MODE_VOLUME	4	Volume
MODE_TIME	5	Heure

Tableau 14: Identificateurs de variables des séries de temps

L'instruction ci-dessous va effectuer l'analyse des 5 barres précédant la barre en position 4 pour repérer la barre avec le prix maximum le plus élevé.

```
// Déclaration de la variable réel iPos
int iPos;

// Instruction permettant d'assigner la position de la barre à la variable iPos
iPos = iHighest(NULL, 0, MODE_HIGH, 8, 2);
```

Le graphique suivant illustre cette portion de code afin de mieux comprendre la fonction *iHighest*.

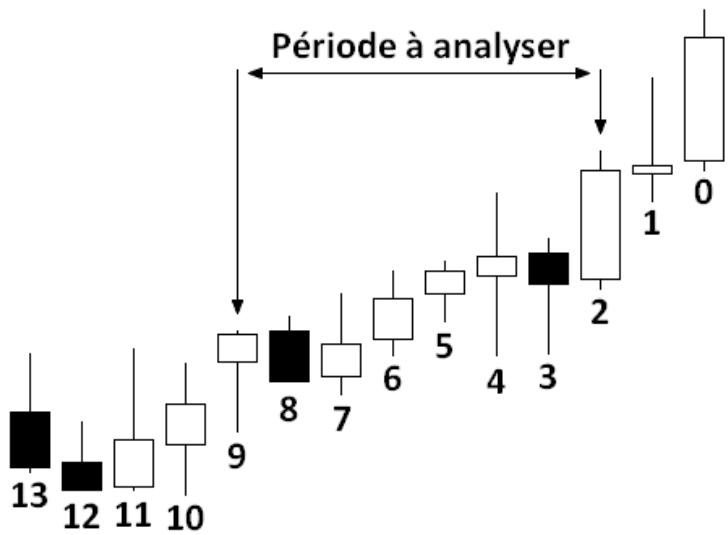

Figure 18 : Illustration du fonctionnement de la fonction iHighest()

La période d'analyse s'étend donc de la chandelle en position 2 jusqu'à la chandelle en position 9 inclusivement. Si vous faites le calcul, vous obtiendrez bien une période de 8 chandelles qui correspond à la période choisie dans le code.

Les fonctions *iHighest()* et *iLowest()* ne peuvent renvoyer qu'un entier correspondant à la position de la chandelle répondant aux critères de votre choix. Il vous faudra donc très souvent cumuler plusieurs fonctions pour obtenir l'information désirée.

Supposons que vous désiriez non pas la position de la chandelle mais bien le prix maximal au cours de cette période. Le code pour obtenir cette information sera :

```
// Déclaration de la variable réel dVal

double dVal;

// Instruction permettant d'obtenir le prix maximum d'une période précise

dVal = High[iHighest(NULL, 0, MODE_HIGH, 8, 3)];
```

La fonction *iHighest()* vous permet d'obtenir la position de la chandelle avec le prix maximum le plus élevé et la fonction *High[]* qui nécessitait la position pour savoir où chercher l'information vous renvoie donc ce prix.

15
Fonctions temporelles

Dans le dernier chapitre ont été traitées les fonctions sur les barres qui incluaient également les fonctions *Time[]* et *iTime()*. Pour compléter ces deux fonctions, voici en détails les fonctions temporelles qui sont au nombre de 18.

Fonctions Day(), DayOfWeek() et DayOfYear()

La fonction *Day()* renvoie la date du jour sous forme d'un entier de 1 à 31.

La syntaxe est la suivante :

```
int Day()
```

Vous pouvez par exemple utiliser cette fonction pour démarrer une instruction au début de chaque mois.

```
if (Day() == 1)
  // Inclure ici l'instruction à exécuter
```

La présence de parenthèses à la suite d'une fonction ne veut pas forcément dire que des paramètres sont requis. Il s'agit simplement de la syntaxe pour les fonctions (consulter le chapitre sur les fonctions pour plus d'informations à ce sujet).

La fonction *DayOfWeek()* renvoie le jour de la semaine sous forme d'un entier de 0 à 6.

La syntaxe est la suivante :

```
int DayOfWeek()
```

La numérotation fonctionne comme ceci : 0 correspond au dimanche, 1 au lundi, 2 au mardi, 3 au mercredi, 4 au jeudi, 5 au vendredi et 6 au samedi.

Il s'agit d'une fonction très utile, par exemple, pour arrêter le fonctionnement d'un programme en fin de semaine ou encore clôturer vos positions ouvertes pour la fin de semaine.

```
if(DayOfWeek() == 5)
  {
  // Inclure ici les instructions pour clôturer les positions ouvertes
  }
```

La fonction *DayOfYear()* permet de récupérer le jour de l'année sous forme d'un entier de 1 à 365 (366 dans le cas d'une année bissextile).

La syntaxe est la suivante :

```
int DayOfYear()
```

Un exemple d'utilisation possible :

```
iJourAn = DayOfYear();
iJourSemaine = DayOfWeek();


if(iJourSemaine != 0)
  iStartWeek = iJourAn - iJourSemaine;
else
  iStartWeek = iJourAn;

iEndWeek = iStartWeek + 5;
```

Le code ci-dessus permet de connaître les journées de l'année correspondantes au début et à la fin de la semaine en cours. Le raisonnement est le suivant : le programme va tout d'abord stocker la journée de l'année en cours dans la variable *iJourAn* puis le jour de la semaine en cours dans la variable *iJourSemaine*. Si *iJourSemaine* est différent de 0, autrement dit, il ne s'agit pas d'un dimanche, on va soustraire la valeur numérique correspondante au jour de la semaine à la journée de l'année afin d'obtenir le numéro de la journée de début de semaine. Si *iJourSemaine* est bien un dimanche, on attribue la valeur de la journée de l'année à la variable *iStartWeek*. On déterminera le numéro de la journée correspondant à la fin de la semaine, *iEndWeek*, en ajoutant 5 à la variable *iStartWeek*.

Fonctions Hour(), Minute(), Seconds(), Month() et Year()

Vous l'aurez certainement déjà deviné, les fonctions *Hour()*, *Minute()*, *Seconds()*, *Month()* et *Year()* servent à renvoyer sous forme d'entiers la valeur actuelle (au moment de l'exécution de la fonction) correspondant au nombre d'heures, de minutes, de secondes ou encore le mois ou l'année en cours.

Dans le tableau ci-dessous, sont résumées les valeurs que peuvent renvoyer ces fonctions.

Fonctions	Intervalles	Détails
Hour()	[0 ; 23]	0 correspond à minuit
Minute();	[0 ; 59]	

Seconds();	[0 ; 59]	
Month()	[1 ; 12]	1 correspond à janvier
Year()	[1970 ; 2037]	

Tableau 15 : Valeurs possibles pour les fonctions temporelles

Les syntaxes pour toutes ces fonctions sont listées à la suite avec un exemple possible d'utilisation.

```
int Hour()
int Minute()
int Seconds()
int Month()
int Year()
```

Un exemple utilisant toutes ces fonctions simultanément pourrait être :

```
int iHeure;
int iMinute;
int iSeconde;
int iJour;
int iMois;
int iAnnee;


iHeure = Hour();
iMinute = Minute();
iSeconde = Seconds();
iJour = Day();
iMois = Month();
iAnnee = Year();


 Comment("Heure : "+iHeure+":"+iMinute+":"+iSeconde+" "+"Date : "+iJour+"/"+iMois+"/"+iAnnee);
```

Le code de l'exemple précédent affiche l'heure et la date dans le coin gauche de votre graphique en dessous des informations du graphique comme dans l'exemple visuel ci-dessous.

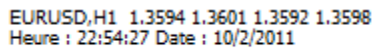

Figure 19 : Illustration du code permettant d'afficher l'heure et la date

Un autre exemple d'utilisation possible de la fonction *Hour()* uniquement mais la logique est la même avec les autres fonctions :

```
extern int iHeureDebut = 9;
extern int iHeureFin = 17;

int start()
  {
  if(Hour() <= iHeureDebut || Hour() > iHeureFin)
    return(0);
  // Suite du programme si les conditions précédentes ne sont pas vérifiées
  }
```

Le code ci-dessus est un filtre horaire très simple. Vous paramétrez sous la forme d'entiers, une heure de début et une heure de fin. À chaque nouveau tick, votre programme va devoir vérifier si l'on se situe avant, entre, ou après les deux limites horaires et selon la réponse, va continuer l'exécution du reste du programme ou renvoyer au début.

Fonctions TimeDay(), TimeDayOfWeek() et TimeDayOfYear()

De la même façon que le *i* devant le nom d'une fonction, le *Time* devant le nom d'une fonction sert à préciser que la fonction vous permet de renvoyer la valeur de la donnée recherchée pour une chandelle spécifique et pas uniquement pour la chandelle en cours.

Vous avez vu précédemment que la fonction *Day()* permettait d'obtenir la date du jour sous forme d'un entier de 1 à 31. La fonction *TimeDay()* permet d'obtenir la même information mais pour toutes les chandelles de votre graphique. Le même raisonnement est valable pour les fonctions *TimeDayOfWeek()* et *TimeDayOfYear()*.

Les syntaxes pour ces fonctions sont :

```
int TimeDay(datetime date)
int TimeDayOfWeek(datetime date)
int TimeDayOfYear(datetime date)
```

Le paramètre à indiquer entre parenthèses pour identifier la barre en question doit être de type *datetime*.

Par exemple :

```
int iJour = TimeDayOfWeek(Time[10]);
```

Le code ci-dessus va assigner à la variable *iJour*, le jour de la semaine actuelle de la chandelle numérotée 10 donc neuvième chandelle vers la gauche en partant de la chandelle actuelle.

Vous pouvez également utiliser des données temporelles en les indiquant directement entre parenthèses.

```
int iJour = TimeDay(D'2010.01.22');
```

Les données temporelles peuvent être indiquées sous les formes suivantes :

```
D'2010.01.01 00:00'          // Correspond au Nouvel An 2010
D'2010.07.19 12:30:27'       // Correspond au 19 juillet 2010 à 12:30:27
D'19.07.2010 12:30:27'       // Correspond au 19 juillet 2010 à 12:30:27
D'19.07.2010 12'             // Correspond au 19 juillet 2010 à midi
D'01.01.2010'                // Correspond au 1er janvier 2010 à minuit
D'12:30:27'                  // Correspond à l'heure 12:30:27
D''                          // Correspond à l'heure 00:00:00
```

Fonctions TimeHour(), TimeMinute(), TimeSeconds(), TimeMonth() et TimeYear()

Les fonctions *TimeHour()*, *TimeMinute()*, *TimeSeconds()*, *TimeMonth()* et *TimeYear()* sont très utiles pour extraire une information spécifique sur la date ou l'heure. En effet, la fonction *Time[]* vous renvoie l'information temporelle sur la chandelle dans sa totalité (heure complète plus date) et ce sous la forme d'une variable de type datetime difficile à manipuler. En cumulant la fonction *Time[]* et une fonction de *TimeHour()*, vous pouvez, par exemple obtenir l'heure actuelle sous la forme d'un entier beaucoup plus facile à comparer avec d'autres variables.

Les syntaxes pour ces fonctions sont les suivantes :

```
int TimeHour(datetime time)
int TimeMinute(datetime time)
int TimeSeconds(datetime time)
int TimeMonth(datetime time)
int TimeYear(datetime time)
```

Supposons que vous désiriez extraire l'heure de la chandelle actuelle. Le code serait alors :

```
int iHeure = TimeHour(Time[0]);
```

En utilisant le code précédent, vous obtiendrez une heure sous forme d'entier de 0 à 23 plus facile à manipuler que la valeur renvoyée par la fonction *Time[0]* retournant l'heure d'ouverture sous la forme du nombre de secondes écoulées depuis le 1er janvier 1970 à minuit (par exemple 23189823998).

Le filtre horaire vu plus haut peut être réécrit de la façon suivante :

```
// Déclaration de vos variables

extern int iHeureDebut = 9 ;
extern int iHeureFin = 17;

int start()
  {
  if(TimeHour(Time[0]) <= iHeureDebut || TimeHour(Time[0]) > iHeureFin)
    return(0);

  // Suite du programme si la condition précédente n'est pas vérifiée

  }
```

Fonctions TimeLocal() et TimeCurrent()

TimeLocal() et *TimeCurrent()* renvoient la même information mais provenant d'une source différente. Les deux fonctions vont renvoyer l'heure actuelle à chaque nouveau tick mais *TimeLocal()* comme son nom l'indique va utiliser l'heure locale soit celle de votre ordinateur tandis que *TimeCurrent()* va utiliser l'heure de la plateforme soit du serveur de votre courtier.

Les syntaxes sont les suivantes :

```
datetime TimeLocal()
datetime TimeCurrent()
```

Le filtre horaire peut encore une fois être réécrit comme ceci :

```
// Déclaration de vos variables

extern int iHeureDebut = 9 ;
extern int iHeureFin = 17;

// Instruction dans la fonction int start()

int start()
  {
  if(TimeHour(TimeCurrent()) <= iHeureDebut || TimeHour(TimeCurrent()) > iHeureFin)
  return(0);

  // Suite du programme si la condition précédente n'est pas vérifiée

  }
```

Une utilisation possible de ces fonctions serait de coder un filtre horaire qui tiendrait compte de la différence horaire possible entre l'heure locale et celle du serveur de votre courtier. Supposons que vous êtes en France et que votre courtier se trouve aux États-Unis sur la côte Est (la différence serait de +6 entre votre heure et celle du courtier).

```
// Déclaration de vos variables

int iHeureLocale;
int iHeureServeur;
int iDifferenceHeure;
extern iHeureDebut = 9;
extern int iHeureFin = 17;

// Instruction dans la fonction int init()


int init()
  {
  iHeureLocale = TimeHour(TimeLocal());
  iHeureServeur = TimeHour(TimeCurrent());


  iDifferenceHeure = iHeureLocale - iHeureServeur;


  if(iDifferenceHeure!= 0)
    {
    iHeureDebut = iHeureDebut + iDifferenceHeure;
```

```
    iHeureFin = iHeureFin + iDifferenceHeure;
    }
  }

// Instruction dans la fonction int start()

int start()
  {
  if(TimeHour(Time[0]) <= iHeureDebut || TimeHour(Time[0]) > iHeureFin)
  return(0);

  // Suite du programme si la condition précédente n'est pas vérifiée

  }
```

Au moment de l'initialisation, le programme va vérifier la différence éventuelle entre l'heure locale et l'heure du serveur. Si une différence existe, le nombre d'heures de différence va être ajouté aux variables externes définissant les heures utilisées pour le filtre horaire. Une fois initialisé, le programme suit son cours normalement avec les instructions de filtre horaire que nous avons vu précédemment.

16
Votre premier expert consultant #3

Pour mettre en application les fonctions nouvellement apprises, un filtre horaire va être ajouté à votre premier expert. Pour pouvoir paramétrer l'expert en cas de nouvelle économique, le filtre horaire prendra en compte l'heure mais aussi les minutes pour initialiser et arrêter le fonctionnement de l'expert.

Le premier problème auquel le filtre horaire sera confronté est le fait que contrairement aux heures qui ne se répètent pas (MetaTrader utilisant un système horaire de 24 heures), les minutes, elles, se répètent à chaque heure. Il faudra donc utiliser une conversion des heures en secondes afin d'éviter tout risque de répétition. En effet, une journée s'étend de 0 secondes (minuit pile) jusqu'à 86400 secondes (24 heures * 60 minutes * 60 secondes).

La première étape pour créer notre filtre consiste à convertir les heures du filtre en seconde.

```
// Déclaration et initialisation des variables

extern int iHeureDebut = 9;
extern int iMinutesDebut = 30;
extern int iHeureFin = 17;
extern int iMinutesFin = 30;
extern int iTimeDebut;
extern int iTimeFin;

// Instructions à exécuter à l'initialisation de l'expert

int init()
  {
  iTimeDebut = iHeureDebut * 3600 + iMinutesDebut * 60;
  iTimeFin = iHeureFin * 3600 + iMinutesFin * 60;
  }
```

Quatre variables ont donc été créées pour recevoir les valeurs du filtre horaire. *iHeureDebut* cumulée à *iMinutesDebut* correspondent à l'heure à partir de laquelle le filtre autorisera l'exécution de certaines instructions et *iHeureFin* cumulée à *iMinutesFin* détermine l'heure à partir de laquelle le programme ne sera plus autorisé à exécuter certaines instructions (la zone grise de la figure 20 correspond à la période de temps pendant laquelle le programme est autorisé à exécuter certaines instructions).

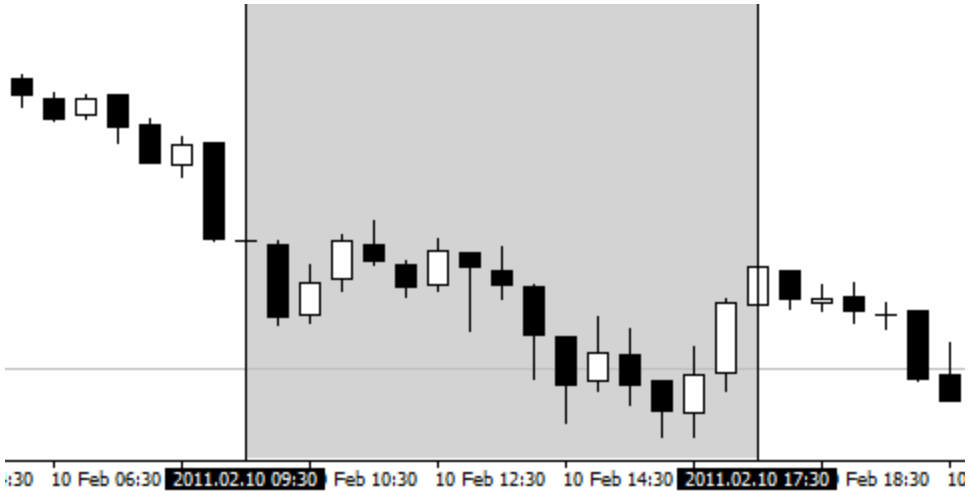

Figure 20 : Illustration du filtre horaire

Lors de l'initialisation du programme, les calculs suivants vont être effectués.

```
iTimeDebut = iHeureDebut * 3600 + iMinutesDebut * 60;
iTimeFin = iHeureFin * 3600 + iMinutesFin * 60;
```

Ces calculs permettent de convertir les heures et minutes du filtre en leurs équivalents en secondes. La seconde étape dans la création du filtre horaire consiste à déterminer l'heure actuelle en secondes et comparer celle-ci aux valeurs calculées précédemment. Ce code devra se situer dans la fonction *start()* afin d'être vérifié en permanence.

```
// Déclaration et initialisation des variables

int iTimeSeconds;

// Instructions à insérer dans la fonction int start()

iTimeSeconds = Hour()*3600 + Minute()*60 + Seconds();

if(iTimeSeconds >= iTimeDebut && iTimeSeconds < iTimeFin)
  {

  // Instructions à exécuter si les conditions du filtre sont respectés

  }
```

Le code source actualisé est donc (les ajouts ou modifications sont en gras) :

```
//+------------------------------------------------------------------+
//|                          Votre Premier Expert.mq4                |
//|                          Copyright © 2011,                       |
//|                        http://www. eole-trading.com              |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www. eole-trading.com"
```

```
extern int iHeureDebut = 9;
extern int iMinutesDebut = 30;
extern int iHeureFin = 17;
extern int iMinutesFin = 30;

int iDate;
int iTimeDebut;
int iTimeFin;
int iTimeSeconds;
double dPlusHaut;
double dPlusBas;
bool bRecuperation = False;


//+------------------------------------------------------+
//| expert initialization fonction                   |
//+------------------------------------------------------+
int init()
  {

  iTimeDebut = iHeureDebut * 3600 + iMinutesDebut * 60;
  iTimeFin = iHeureFin * 3600 + iMinutesFin * 60;

  return(0);
  }
//+------------------------------------------------------+
//| expert deinitialization fonction                 |
//+------------------------------------------------------+
int deinit()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------+
//| expert start fonction                            |
//+------------------------------------------------------+
int start()
  {

  iTimeSeconds = Hour()*3600 + Minute()*60 + Seconds();


  if(iTimeSeconds >= iTimeDebut && iTimeSeconds < iTimeFin)
    {
    if(bRecuperation == False)
      {
      iDate = Day();
      dPlusHaut = iHigh(NULL, PERIOD_D1, 1);
      dPlusBas = iLow(NULL, PERIOD_D1, 1);
      bRecuperation = True;
      }

    if(iDate != Day())
      bRecuperation = False;
    }

  return(0);
  }
//+------------------------------------------------------+
```

Le code situé à l'intérieur de l'instruction conditionnelle ne s'exécutera désormais qu'entre 9h30 et 17h30.

17
Fonctions graphiques

MetaTrader étant avant tout une plateforme de trading graphique, la possibilité d'afficher du texte ou des éléments graphiques est donc essentielle. En MQL4, les éléments graphiques sont assimilés à des objets. Pour chaque objet, il convient de définir le nom identifiant celui-ci, la position ainsi que les autres propriétés possibles telles que la couleur, la forme entre autres choix.

Créer un objet

La fonction à utiliser pour créer un objet est *ObjectCreate()*. Il s'agit d'une fonction booléenne dont la syntaxe est la suivante :

```
bool ObjectCreate(string sNom, int iType, int iFenetre, datetime dtCoord1, double dPrix1, datetime dtCoord2, double dPrix2, datetime dtCoord3, double dPrix3)
```

Cette fonction permet de définir le nom que vous désirez attribuer à la variable, le type d'objet (carré, rectangle ou autres), la fenêtre où vous désirez voir le nouvel objet apparaître et enfin les différents points de coordonnées qui serviront à déterminer la position de l'objet sur l'écran.

Les différents objets que vous pouvez créer sont résumés dans le tableau ci-dessous. La première colonne indique le nom officiel du type de l'objet, la deuxième colonne la valeur numérique associée au type précédemment mentionné, la troisième colonne le descriptif de l'objet et enfin la quatrième colonne vous indique combien de points de coordonnées sont requis pour positionner l'objet.

Type	Valeur numérique	Descriptif	Points de coordonnées requis
OBJ_VLINE	0	Droite verticale	dtCoord1
OBJ_HLINE	1	Droite horizontale	dPrix1
OBJ_TREND	2	Segment	dtCoord1, dPrix1, dtCoord2, dPrix2
OBJ_TRENDBYANGLE	3	Droite avec angle	dtCoord1, dPrix1
OBJ_REGRESSION	4	Droite de régression	dtCoord1, dtCoord2
OBJ_CHANNEL	5	Canal	dtCoord1, dPrix1, dtCoord2, dPrix2, dtCoord3, dPrix3
OBJ_STDDEVCHANNEL	6	Canal de déviation standard	dtCoord1, dtCoord2

OBJ_GANNLINE	7	Droite de Gann	dtCoord1, dPrix1, dtCoord2, dPrix2
OBJ_GANNFAN	8	Éventail de Gann	dtCoord1, dPrix1, dtCoord2, dPrix2
OBJ_GANNGRID	9	Grille de Gann	dtCoord1, dPrix1, dtCoord2, dPrix2
OBJ_FIBO	10	Retracement de Fibonacci	dtCoord1, dPrix1, dtCoord2, dPrix2
OBJ_FIBOTIMES	11	Zones horaires de Fibonacci	dtCoord1, dPrix1, dtCoord2, dPrix2
OBJ_FIBOFAN	12	Éventail de Fibonacci	dtCoord1, dPrix1, dtCoord2, dPrix2
OBJ_FIBOARC	13	Arc de Fibonacci	dtCoord1, dPrix1, dtCoord2, dPrix2
OBJ_EXPANSION	14	Expansions de Fibonacci	dtCoord1, dPrix1, dtCoord2, dPrix2, dtCoord3, dPrix3
OBJ_FIBOCHANNEL	15	Canal de Fibonacci	dtCoord1, dPrix1, dtCoord2, dPrix2, dtCoord3, dPrix3
OBJ_RECTANGLE	16	Rectangle	dtCoord1, dPrix1, dtCoord2, dPrix2
OBJ_TRIANGLE	17	Triangle	dtCoord1, dPrix1, dtCoord2, dPrix2, dtCoord3, dPrix3
OBJ_ELLIPSE	18	Ellipse	dtCoord1, dPrix1, dtCoord2, dPrix2
OBJ_PITCHFORK	19	Fourche de Andrews	dtCoord1, dPrix1, dtCoord2, dPrix2, dtCoord3, dPrix3
OBJ_CYCLES	20	Cycles	dtCoord1, dPrix1, dtCoord2, dPrix2
OBJ_TEXT	21	Texte	dtCoord1, dPrix1
OBJ_ARROW	22	Flèches	dtCoord1, dPrix1
OBJ_LABEL	23	Étiquette de texte	1 coordonnée en pixel

Tableau 16: Objets disponibles dans MetaTrader

La plupart des objets se positionnent en fonction de coordonnées temporelles et de prix mais l'objet de type étiquette de texte est positionné en fonction de coordonnées en pixel. Ceci signifie que l'objet se déplacera en même temps que le graphique pour rester en tout temps à la même position. Un exemple illustrant ce cas spécifique est disponible à la suite.

Comme vous avez pu le constater, la fonction *ObjectCreate()* nous permet uniquement de définir le nom, le type et la position de l'objet. Afin de modifier d'autres paramètres telles que couleur, type de ligne, police de caractère, etc.… une seconde fonction est requise : *ObjectSet()*.

Paramétrer un objet

La fonction *ObjectSet()* de type booléenne permet de définir les paramètres spécifiques à un objet en particulier. La syntaxe de la fonction est la suivante :

```
bool ObjectSet(string sNom, int iIndex, double value)
```

string sNom permet de spécifier le nom de l'objet sur lequel vous désirez appliquer des modifications. int iIndex permet de choisir le paramètre que vous désirez modifier et *double value* permet de choisir la valeur à attribuer au paramètre choisi précédemment.

Le nom dans *ObjectCreate()* et dans *ObjectSet()* doit être identique.

Les paramètres modifiables dans MetaTrader sont résumés dans le tableau suivant:

Nom	Valeur numérique	Type	Description
OBJPROP_TIME1	0	datetime	Valeur temporelle modifiant la première coordonnée temporelle de l'objet
OBJPROP_DPRIX1	1	double	Valeur réelle modifiant la première coordonnée de prix
OBJPROP_TIME2	2	datetime	Valeur temporelle modifiant la deuxième coordonnée temporelle de l'objet
OBJPROP_DPRIX2	3	double	Valeur réelle modifiant la deuxième coordonnée de prix
OBJPROP_TIME3	4	datetime	Valeur temporelle modifiant la troisième coordonnée temporelle de l'objet
OBJPROP_DPRIX3	5	double	Valeur réelle modifiant la troisième coordonnée de prix
OBJPROP_COLOR	6	color	Couleur de l'objet
OBJPROP_STYLE	7	int	Type de ligne (voir tableau 80)
OBJPROP_WITDH	8	int	Grosseur de la ligne (valeur comprise entre 1 et 5)
OBJPROP_BACK	9	bool	Place l'objet en avant ou arrière plan / remplit l'objet ou laisse uniquement le contour
OBJPROP_RAY	10	bool	Permet de définir l'objet en tant que segment ou droite
OBJPROP_ELLIPSE	11	bool	Permet de définir les arcs de Fibonacci comme ellipse
OBJPROP_SCALE	12	double	Échelle de l'objet
OBJPROP_ANGLE	13	double	Angle en degrés pour les objets nécessitant cette information
OBJPROP_ARROWCODE	14	int	Type de flèche (voir tableau 29)

OBJPROP_TIMEFRAMES	15	int	Définit pour quelle(s) unités de temps du graphique l'objet doit être visible
OBJPROP_DEVIATION	16	double	Paramètre de la déviation pour les objets utilisant la déviation standard
OBJPROP_FONTSIZE	100	int	Taille du texte
OBJPROP_CORNER	101	int	Coin pour positionner l'objet (0–3 avec 0=coin supérieur gauche)
OBJPROP_XDISTANCE	102	int	Distance horizontale en pixels du coin choisi
OBJPROP_YDISTANCE	103	int	Distance verticale en pixels du coin choisi
OBJPROP_FIBOLEVELS	200	int	Nombre de niveaux de Fibonacci (0 - 32)
OBJPROP_LEVELCOLOR	201	int	Couleur des niveaux de Fibonacci
OBJPROP_LEVELSTYLE	202	int	Type de ligne (voir tableau 80)
OBJPROP_LEVELWITDH	203	int	Grosseur de la ligne (valeur comprise entre 1 et 5)
OBJPROP_FIRSTLEVELn	210+n	int	Définit le prix pour le niveau n avec n = 0 à 31

Tableau 17 : Paramètres modifiables pour chaque objet dans MetaTrader

Il existe également deux fonctions de paramétrages spécifiques:

- *ObjectSetFiboDescription* → Permet de rajouter un texte descriptif sur un niveau précis de Fibonacci
- *ObjectSetText* → Permet de définir le texte d'un objet texte ou étiquette de texte

La syntaxe pour ces deux fonctions est comme suit :

```
bool ObjectSetFiboDescription(string sNom, int iIndex, string sTexte)

bool ObjectSetText(string sNom, string sTexte, int iTailleTexte, string sPolice, color cCouleurTexte)
```

De la même façon que pour la fonction *ObjectSet()*, il faut en premier lieu définir sur quel objet vous désirez voir appliquer les changements. Dans le cas de ces deux fonctions, il n'est pas nécessaire de préciser quel type de modification vous désirez apporter à l'objet vu que les deux fonctions ne servent qu'à une seule et unique modification.

Exemples

Comme premier exemple, nous allons nous contenter de tracer des lignes pour indiquer le point le plus haut/bas des barres affichées à l'écran.

La première étape pour être en mesure de tracer les lignes en questions est de définir les prix correspondants aux plus haut/bas des barres affichées à l'écran. Pour ce faire, nous allons utiliser les fonctions *High[]*, *Low[]*, *iHighest()* et *iLowest()* que nous avons déjà vues précédemment.

```
double dPrixPlusHaut = High[iHighest(Symbol(), 0, MODE_HIGH, Bars, 1)];
double dPrixPlusBas = Low[iLowest(Symbol(), 0, MODE_LOW, Bars, 1)];
```

Nous avons donc une ligne de code pour chacun des prix. On va attribuer le prix de la barre la plus haute à l'écran à notre variable *dPrixPlusHaut*. Même chose pour *dPrixPlusBas* pour le plus bas prix.

Nous utilisons *Bars* au lieu d'un nombre car nous désirons que toutes les barres affichées à l'écran soient analysées. La barre de départ pour l'analyse a été paramétrée sur 1 car nous voulons que le programme ne prenne en compte que les barres clôturées.

Maintenant que les prix maximum et minimum sont définis, il ne nous reste plus qu'à créer les lignes. Comme vous pouvez voir ci-dessus, nous retrouvons nos variables préalablement créées en tant que première et unique coordonnée prix. Pour ajouter un effet visuel, nous allons également paramétrer chacune des lignes pour qu'elles s'affichent dans une couleur différente à l'aide de la fonction *ObjectSet*().

```
ObjectCreate("Point plus haut",OBJ_HLINE, 0, 0, dPrixPlusHaut);
ObjectCreate("Point plus bas",OBJ_HLINE, 0, 0, dPrixPlusBas);
ObjectSet("Point plus haut", OBJPROP_COLOR, Blue);
ObjectSet("Point plus bas", OBJPROP_COLOR, Red);
```

Figure 21 : Lignes indiquant plus haut/bas du graphique

Supposons que vous désiriez que les lignes indiquent le plus haut/bas de la veille uniquement. Pour ce faire, il vous suffit de modifier légèrement le code chargé de récupérer les prix comme ceci :

```
double dPrixPlusHaut = iHigh(Symbol(), PERIOD_D1, 1);
double dPrixPlusBas = iLow(Symbol(), PERIOD_D1, 1);
```

Nous avons remplacé la période *0* par *PERIOD_D1* qui signifie que la fonction doit analyser les barres journalières puis nous avons indiqué que nous souhaitions analyser l'avant-dernière barre (la veille) Le code pour tracer les lignes demeure inchangé. De plus, nous n'utilisons plus la fonction *High[]/Low[]* mais *iHigh()/iLow()*.

Le fait d'utiliser *iHigh()* et *iLow()* permet d'éviter un risque d'erreur. En effet, si l'unité de temps du graphique n'est pas quotidienne, le code renverra un prix erroné.

Comme vous pouvez voir, les possibilités sont multiples selon l'information que vous désirez voir afficher.

Si vous placez le code de l'exemple ci-dessus dans la fonction *start()*, le code sera exécuté à chaque tick et par conséquent les lignes seront mises à jour si nécessaire. Certains ajouts sont nécessaires pour cela comme nous le verrons dans le chapitre suivant.

Traçons maintenant un rectangle sur notre graphique. À toutes fins utiles, nous allons paramétrer ce nouvel objet pour qu'il englobe une journée comme dans le graphique ci-dessous.

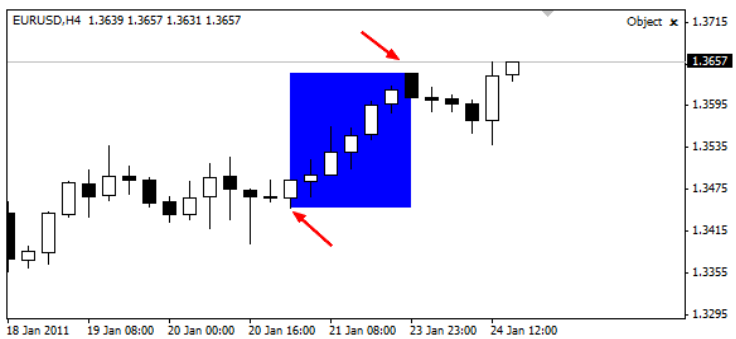

Figure 22 : Exemple de rectangle englobant une journée

Pour pouvoir tracer un rectangle similaire, nous avons besoin des coordonnées temporelles indiquées par les flèches.

Pour l'information sur les prix, il nous suffit de procéder de la même façon que dans l'exemple précédent.

```
double dPrixPlusHaut = iHigh(Symbol(), PERIOD_D1, 0);
double dPrixPlusBas = iLow(Symbol(), PERIOD_D1, 0);
```

Pour la deuxième étape, nous avons besoin de stocker l'information sur les dates dans deux variables.

```
datetime dtDateDebut = iTime(Symbol(), PERIOD_D1, 0);
datetime dtDateFin = Time[0];
```

Il ne reste plus qu'à intégrer toutes les informations dans la commande de création du rectangle.

```
ObjectCreate("Rectangle",OBJ_RECTANGLE, 0, dtDateDebut, dPrixPlusBas, dtDateFin, dPrixPlusHaut);
```

Nous obtenons le graphique suivant :

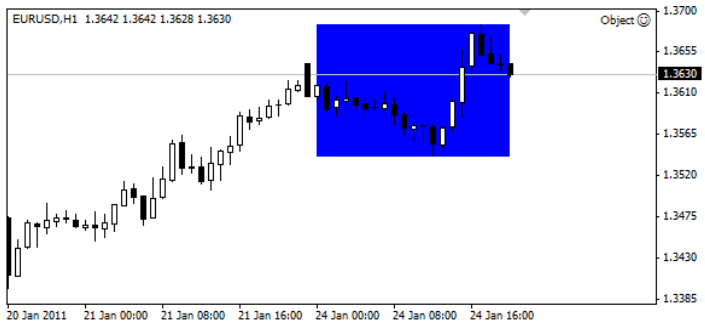

Figure 23 : Rectangle indiquant la journée écoulée

Supposons que vous vouliez que le rectangle couvre également le futur. Il vous suffit de pousser la deuxième coordonnée temporelle dans le futur. Pour ce faire, vous devez simplement déterminer combien de temps il reste avant la fin de journée et ajouter le nombre de périodes nécessaires.

Il suffit de remplacer le code pour obtenir la coordonnée temporelle *dtDateFin* par :

```
datetime dtDateFin = Time[0] + ((24 - TimeHour(Time[0]))*3600);
```

Nous prenons donc la barre actuelle à laquelle nous ajoutons la différence entre 24 et l'heure actuelle multipliée par 3600 pour obtenir la coordonnée en secondes (unité de temps utilisée pour les calculs par MetaTrader). Comme résultat, nous aurions un graphique similaire à ceci :

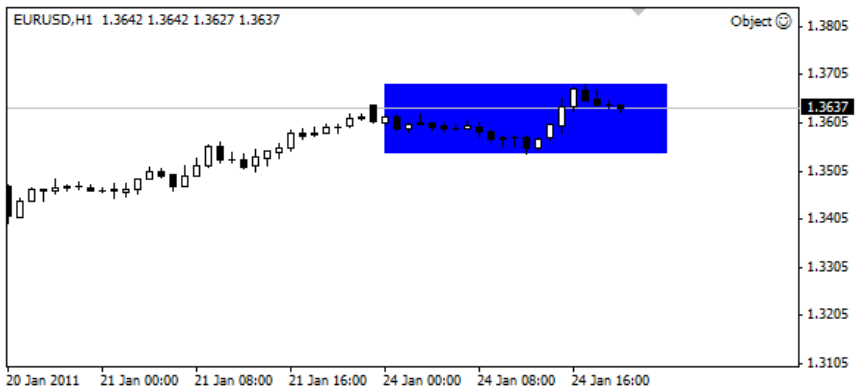

Figure 24 : Rectangle indiquant la journée en cours

Le code au complet maintenant modifié sera donc :

```
double dPrixPlusHaut = iHigh(Symbol(), PERIOD_D1, iHighest(Symbol(), PERIOD_D1, MODE_HIGH, 1, 0));
double dPrixPlusBas = iLow(Symbol(), PERIOD_D1, iLowest(Symbol(), PERIOD_D1, MODE_LOW, 1, 0));


datetime dtDateDebut = iTime(Symbol(), PERIOD_D1, 0);
datetime dtDateFin = Time[0] + ((24 - TimeHour(Time[0]))*3600);


ObjectCreate("Rectangle",OBJ_RECTANGLE, 0, dtDateDebut, dPrixPlusBas, dtDateFin, dPrixPlusHaut);
```

Comme dernier exemple, nous allons créer des objets textes en utilisant les deux possibilités offertes par MetaTrader : objet texte et étiquette de texte.

Supposons que vous désiriez ajouter du texte au dessus du rectangle créé précédemment.

Il vous suffit pour cela d'ajouter le code suivant :

```
ObjectCreate("Texte", OBJ_TEXT, 0, dtDateDebut, dPrixPlusHaut);
ObjectSetText("Texte", "Rectangle", 12, "Times New Roman", Black);
```

Nous obtenons le graphique suivant:

Figure 25 : Texte au-dessus du rectangle (position incorrecte)

Le texte est loin d'être idéalement positionné car nous avons indiqué comme coordonnée temporelle le début de la journée et comme coordonnée de prix le prix le plus haut de la journée. Le point fourni par ces deux coordonnées est en fait le point central du côté supérieur du rectangle contenant le texte comme vous pouvez voir dans le graphique ci-dessous.

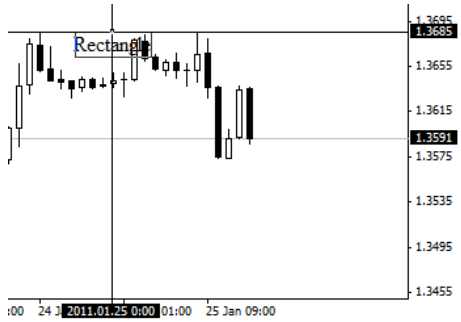

Figure 26 : Point de positionnement du texte

Pour corriger la position du texte, nous modifions le code précédent en plaçant le point de positionnement du texte au milieu du rectangle sur l'axe horizontal et ajoutons 20 pips sur l'axe vertical.

```
ObjectCreate("Texte", OBJ_TEXT, 0, dtDateDebut+(12*3600), dPrixPlusHaut+0.0020);
ObjectSetText("Texte", "Rectangle", 12, "Times New Roman", Black);
```

Le graphique obtenu sera maintenant :

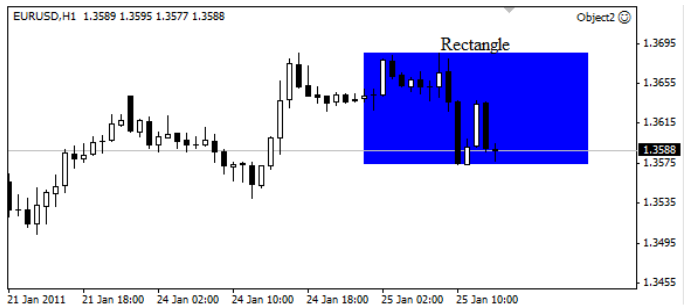

Figure 27 : Texte au-dessus du rectangle (position correcte)

La différence entre l'objet texte et l'objet étiquette de texte comme nous l'avons vu précédemment est que l'objet texte va avoir une position fixe par rapport aux chandelles (le texte bougera au fur et à mesure que le temps s'écoulera) tandis que l'objet de type étiquette de texte est positionné par rapport à la fenêtre graphique et sera par conséquent fixe indépendamment du temps.

Le code pour créer et positionner un objet étiquette de texte serait le suivant :

```
ObjectCreate("Texte", OBJ_LABEL, 0, 0, 0);
ObjectSetText("Texte", "Rectangle", 12, "Times New Roman", Black);

ObjectSet("Texte", OBJPROP_XDISTANCE, 100);
ObjectSet("Texte", OBJPROP_YDISTANCE, 100);
```

Vous remarquerez que contrairement à l'objet texte, dans ce cas-ci, les coordonnées dans la fonction ObjectCreate sont toutes à 0. Comme l'étiquette est positionnée en fonction de coordonnées en pixels, nous devons positionner l'objet dans ses propriétés en utilisant *OBJPROP_XDISTANCE* et *OBJPROP_YDISTANCE*.

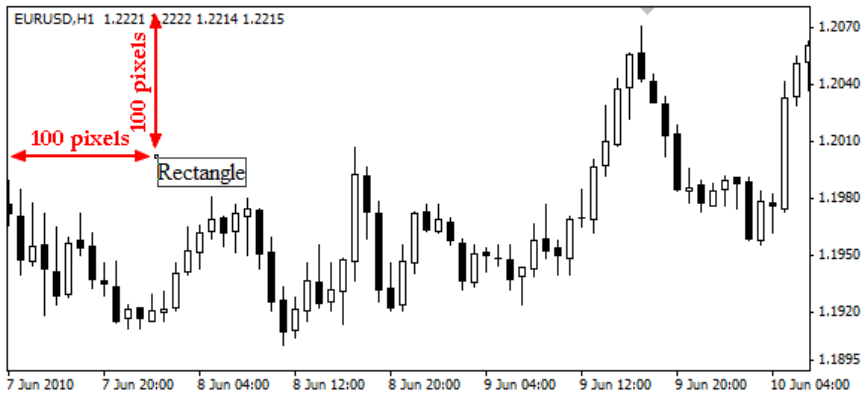

Figure 28 : Positionnement d'une étiquette de texte

Le point de positionnement n'est plus le milieu du côté supérieur mais le coin supérieur gauche. Le positionnement est calculé en fonction du coin supérieur gauche du graphique.

L'objet texte sert donc à afficher une information ponctuelle et ciblée sur un point précis du graphique tandis que l'objet étiquette de texte sert à afficher une information générale ou durable.

Les possibilités d'utilisation d'un objet seul ou en combinaison d'autres sont presque infinies.

Il est impossible sur MetaTrader de faire modifier les propriétés générales d'un graphique (couleur du fond, type de barre, etc.…) par un script, indicateur ou expert consultant. Cependant, grâce aux objets graphiques, vous pouvez outrepasser cette limitation avec quelques astuces. Par exemple, pour changer la couleur du fond, il suffit de demander à votre programme de créer un objet rectangle plus grand que le cadre du graphique (défini par le prix et la date). La seule limite est votre imagination…

18
Votre premier expert consultant #4

Nous allons maintenant incorporer quelques unes des dernières fonctions apprises à notre programme.

Nous avions précédemment créé les variables suivantes :

```
double dPlusHaut;
double dPlusBas;
```

Nous allons utiliser ces dernières pour tracer des lignes indiquant ces niveaux.

Le code pour créer les lignes sera donc le suivant :

```
ObjectCreate("Point plus haut",OBJ_HLINE, 0, 0, dPlusHaut);
ObjectCreate("Point plus bas",OBJ_HLINE, 0, 0, dPlusBas);

ObjectSet("Point plus haut", OBJPROP_COLOR, Blue);
ObjectSet("Point plus bas", OBJPROP_COLOR, Red);
```

Comme nous souhaitons que le programme retrace les lignes lorsque les niveaux deviennent obsolètes (lors du passage à une nouvelle journée par exemple), nous devons ajouter le code suivant qui servira à effacer les lignes anciennement tracées.

```
ObjectDelete("Point plus haut");
ObjectDelete("Point plus bas");
```

Comme vous pouvez voir, nous répétons deux fois le même code – la seule différence est au niveau de certains paramètres. Pour simplifier la lecture, nous allons donc créer une fonction personnalisée qui sera en charge de créer, paramétrer et effacer chaque ligne. Le code de cette fonction est donc :

```
void fTracageLignes(string sNomLigne, double dPrix, color cCouleur)
 {
 ObjectDelete(sNomLigne);
 ObjectCreate(sNomLigne,OBJ_HLINE, 0, 0, dPrix);
 ObjectSet(sNomLigne, OBJPROP_COLOR, cCouleur);
 }
```

Nous insérerons cette fonction à la fin du programme en dehors des fonctions spéciales et il faut maintenant ajouter le code suivant à l'intérieur de la fonction *start()*.

```
fTracageLignes("Point plus haut", dPlusHaut, Blue);
fTracageLignes("Point plus bas", dPlusBas, Red);
```

Le code source actualisé est donc (les ajouts ou modifications sont en gras) :

```
//+------------------------------------------------------------------+
//|                          Votre Premier Expert.mq4                |
//|                           Copyright © 2011,                      |
//|                        http://www. eole-trading.com              |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www. eole-trading.com"


extern int iHeureDebut = 9;
extern int iMinutesDebut = 30;
extern int iHeureFin = 17;
extern int iMinutesFin = 30;

int iDate;
int iTimeDebut;
int iTimeFin;
int iTimeSeconds;
double dPlusHaut;
double dPlusBas;
bool bRecuperation = False;


//+------------------------------------------------------+
//| expert initialization fonction                       |
//+------------------------------------------------------+
int init()
  {

  iTimeDebut = iHeureDebut * 3600 + iMinutesDebut * 60;
  iTimeFin = iHeureFin * 3600 + iMinutesFin * 60;

  return(0);
  }
//+------------------------------------------------------+
//| expert deinitialization fonction                     |
//+------------------------------------------------------+
int deinit()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------+
//| expert start fonction                                |
//+------------------------------------------------------+
int start()
  {

  iTimeSeconds = Hour()*3600 + Minute()*60 + Seconds();


  if(iTimeSeconds >= iTimeDebut && iTimeSeconds < iTimeFin)
    {
    if(bRecuperation == False)
      {
      iDate = Day();
      dPlusHaut = iHigh(NULL, PERIOD_D1, 1);
      dPlusBas = iLow(NULL, PERIOD_D1, 1);
      bRecuperation = True;
      fTracageLignes("Point plus haut", dPlusHaut, Blue);
      fTracageLignes("Point plus bas", dPlusBas, Red);
      }

    if(iDate != Day())
      bRecuperation = False;
```

```
   }

  return(0);
  }
//+------------------------------------------------------+


void fTracageLignes(string sNomLigne, double dPrix, color cCouleur)
  {
  ObjectDelete(sNomLigne);
  ObjectCreate(sNomLigne,OBJ_HLINE, 0, 0, dPrix);
  ObjectSet(sNomLigne, OBJPROP_COLOR, cCouleur);
  }
```

19
Fonctions de trading

Voici venu le moment d'un des chapitres essentiels de ce livre : « Les fonctions de trading ». Sans elles, MetaTrader serait tout bonnement très réussi visuellement mais totalement inutile si vous désirez automatiser toute la procédure de passage, fermeture, gestion des ordres.

Ces fonctions sont au nombre de 25. Nous allons comme précédemment les voir une à une et donner la syntaxe et un exemple pour chacune d'entre elles.

Fonction de passage d'ordre

OrderSend() est la fonction permettant de passer un ordre. Le type de la fonction est *int* car si un ordre est correctement placé, la fonction renvoie le numéro de ticket de l'ordre en question.

Le numéro de ticket correspond au numéro entier que votre courtier attribue à chaque position que vous placez. Ce numéro est unique.

La syntaxe de la fonction est :

```
int OrderSend(string sPaire, int iTypeOrdre, double dVolume, double dPrix, int iSlippage, double dStop,
double dLimite, string sCommentaire=NULL, int iNumeroMagique=0, datetime dtExpiration=0, color
cCouleurFleche=CLR_NONE)
```

string sPaire permet de spécifier sur quelle paire vous désirez passer un ordre.

int iTypeOrdre indique au programme quel type d'ordre vous désirez placer.

Constante	Type	Description
OP_BUY	0	Achat au prix du marché
OP_SELL	1	Vente au prix du marché
OP_BUYLIMIT	2	Achat à un prix limite
OP_SELLLIMIT	3	Vente à un prix limite
OP_BUYSTOP	4	Achat à un prix stop
OP_SELLSTOP	5	Vente à un prix stop

Tableau 18 : Différents types d'ordre

double dVolume vous permet de spécifier le nombre de lots que vous désirez. Il s'agit d'une variable de type *double* car il est possible de spécifier des fractions de lots dépendant du volume minimum imposé par le courtier (0.01 ou 0.1 en général).

double dPrix indique à quel prix vous désirez voir passer l'ordre. Si vous utilisez des achats stops ou limites, vous pouvez spécifier le prix ou la variable de votre choix. Dans le cas des ordres au marché, vous devrez utiliser *Ask* pour *OP_BUY* et *Bid* pour *OP_SELL*. Nous verrons dans le chapitre suivant les variables prédéfinies. *Ask* et *Bid* font partie de cette catégorie.

int iSlippage vous permet de vous assurer de voir votre ordre passer et ce même si le prix du marché a changé entre le passage de l'ordre et l'exécution de ce dernier (en général, le slippage est inexistant sauf dans le cas de nouvelles économiques importantes où la volatilité augmente). Cette valeur se situe en général entre 0 et 5.

double dStop comme son nom l'indique est le prix correspondant au niveau de perte maximale que vous acceptez d'encourir. Si vous ne désirez pas avoir de stoploss, vous pouvez laisser ce paramètre à 0.

double dLimite est similaire au paramètre précédent sauf que dans ce cas-ci, il s'agit du niveau de gain souhaité.

string sCommentaire est le commentaire qui apparaîtra dans la fenêtre Terminal de votre plateforme.

Order	Time	Type	Size	Sym...	Price	S / L	T / P	Price	Com...	Swap	Profit	Comment
1116...	2011.01.25 23:19	sell	0.10	gbpjpy	129.98	130.50	129.00	130.03	0.00	0.00	-5	Ici apparaît le commentaire

Figure 29: Commentaire de position ouverte

int iNumeroMagique est le numéro magique de votre position. Nous verrons plus en détail ce paramètre dans les fonctions de gestion d'ordre mais retenez simplement pour l'instant que ce numéro permet d'identifier une position ou un groupe de positions.

datetime dtExpiration permet de spécifier une date d'expiration pour les ordres de type stop ou limite.

color cCouleurFleche correspond à la couleur de la flèche indiquant la prise de position.

Voici quelques exemples de passage d'ordre en utilisant la fonction *OrderSend()* :

```
OrderSend("EURUSD", OP_BUY, 1, Ask, 3, 0, 0, " ", 0, 0, Blue);

OrderSend(Symbol(), OP_SELLSTOP, dVolume, dPrixBas, 5, dStopLoss, dTakeProfit,  "Vente", iMagic, 0, Red);

OrderSend("USDJPY", OP_BUYLIMIT, 0.2, 105.34, 1, 105.34 + 0.30, 105.34 - 0.50, "Achat Limite", 123456, dtExpiration, Green);
```

Quelques éléments importants à noter :

1) Si vous indiquez le nom d'une paire, vous devez impérativement encadrer celui-ci de guillemets.
2) Vous pouvez aussi bien utiliser des variables que des valeurs à la place des paramètres mais prenez garde à ce que ces dernières soient compatibles.

Fonction de gestion des ordres

Nous allons étudier dans cette section les fonctions suivantes : *OrderClose*(), *OrderCloseBy*(), *OrderDelete()* et *OrderModify()*.

OrderClose()

OrderClose() est une fonction booléenne permettant de clôturer une position ouverte. La syntaxe de la fonction est la suivante :

```
bool OrderClose(int iTicket, double dVolume, double dPrix, int iSlippage, color cCouleur=CLR_NONE)
```

int iTicket vous permet d'indiquer le numéro de ticket de la position que vous désirez clôturer.

double dVolume vous donne la possibilité d'indiquer au programme que vous ne désirez fermer qu'une portion de la position.

Si vous indiquez un volume plus grand que celui de la position, le programme n'exécutera pas votre requête.

Si vous indiquez un volume plus petit que le minimum requis par votre courtier, le programme sera à nouveau incapable d'exécuter votre requête.

double dPrix permet d'indiquer le prix auquel vous désirez voir votre position clôturée.

int iSlippage. De la même façon qu'il peut y avoir du slippage lors du passage de l'ordre, il peut y en avoir pour sa clôture. Il convient donc de ne pas négliger ce paramètre pour s'assurer de la fermeture de ses positions.

color cCouleur=CLR_NONE vous permet de choisir la couleur de la flèche qui indiquera le point de clôture temporel et le prix de la position.

OrderCloseBy()

OrderCloseBy() est de nouveau une fonction booléenne permettant de clôturer une position mais cette fois-ci uniquement si une position inverse similaire est ouverte. La syntaxe de la fonction est la suivante :

```
bool OrderCloseBy(int iTicket, int iTicketOppose, color cCouleur=CLR_NONE)
```

int iTicket est le numéro de ticket de la position que vous désirez voir clôturer.

int iTicketOppose correspond au numéro de ticket de la position inverse.

color cCouleur=CLR_NONE vous permet de choisir la couleur de la flèche qui indiquera le point de clôture temporel et prix de la position.

OrderDelete()

OrderDelete() est une fonction booléenne permettant de supprimer les ordres en attente (ordres stops ou limites). La syntaxe de la fonction est la suivante :

```
bool OrderDelete( int iTicket, color cCouleur=CLR_NONE)
```

int iTicket est le numéro de ticket de la position en attente que vous désirez voir supprimer.

color cCouleur=CLR_NONE vous permet de choisir la couleur de la flèche qui indiquera le point de suppression temporel et le prix de la position.

OrderModify()

OrderModify() est une fonction booléenne permettant de modifier les paramètres de positions ouvertes ou en attente. La syntaxe de la fonction est la suivante :

```
bool OrderModify(int iTicket, double dPrix, double dStop, double dLimite, datetime dtExpiration, color cCouleurFleche=CLR_NONE)
```

int iTicket est le numéro de ticket de la position ouverte ou en attente dont vous désirez modifier certains paramètres.

double dPrix permet de choisir un nouveau prix d'achat ou vente pour les positions en attente uniquement.

double dStop comme son nom l'indique est le prix correspondant au niveau de perte maximale que vous souhaitez encourir. Si vous ne désirez pas avoir de stoploss, vous pouvez laisser ce paramètre à 0.

double dLimite est similaire au paramètre précédent sauf que dans ce cas-ci, il s'agit du niveau de gain souhaité.

datetime dtExpiration permet de spécifier une date d'expiration pour les ordres de type stop ou limite.

color cCouleurFleche correspond à la couleur de la flèche indiquant la prise de position.

Il n'est malheureusement pas possible de modifier les paramètres suivants : la paire, le type d'ordre, le volume, le slippage, le commentaire ou le magic number. Pour ce faire, vous êtes obligé de passer un nouvel ordre.

Voici quelques exemples simples pour chacune des fonctions. Nous verrons des exemples plus complexes dans la section suivante (Fonctions pour sélectionner et obtenir de l'information sur les positions).

```
OrderClose(21322213, 1, Ask, 3, Red);
OrderClose(iTicket, dVolume, 123.45, Green);

OrderCloseBy(iTicket, iTicketInverse, Red);
OrderDelete(iTicket, Red);
OrderDelete(2313213, Blue);


OrderModify(iTicket, dPrixBas, dNouveauStop, dNouveauTakeProfit, 0, Red);
```

De plus en plus de courtiers offrant MetaTrader comme plateforme sont de type ECN/STP (signifiant que ces courtiers ne sont pas des faiseurs de marché). Si cela apporte plus de transparence au client, cela apporte également le désavantage de ne pas pouvoir placer de prix stop et limite en même temps que vous placez un ordre. Pour résoudre ce problème, il vous faudra utiliser la fonction *OrderModify()* une fois la position ouverte pour paramétrer un prix stop/limite.

Fonctions pour sélectionner et obtenir de l'information sur les positions

Fonction de sélection

Certaines fonctions permettent d'obtenir des informations sur les paramètres de positions ouvertes, clôturées ou en attente. Pour pouvoir utiliser ces fonctions, il faut impérativement que la position ait été sélectionnée préalablement par la fonction *OrderSelect()*. Nous allons maintenant voir plus en détail cette fonction particulière.

La fonction *OrderSelect()* est une fonction booléenne permettant de sélectionner une position en particulier afin de pouvoir appliquer des modifications ou obtenir des informations sur cette dernière. La syntaxe de la fonction est :

```
bool OrderSelect(int iIndex, int iSelection, int iListePositions=MODE_TRADES)
```

int iIndex permet d'indiquer un numéro de ticket en particulier ou encore le numéro de la position dans le groupe de position sélectionné dans le paramètre int iListePositions.

int iSelection indique au programme comment chercher la position. Ce paramètre doit être l'une de ces deux possibilités : *SELECT_BY_POS* si vous désirez sélectionner la position dans la liste des positions ou *SELECT_BY_TICKET* si vous avez indiqué un numéro de ticket dans le paramètre précédent.

int iListePositions=MODE_TRADES. Ce paramètre n'est utilisé que si vous avez indiqué *SELECT_BY_POS* dans le paramètre précédent. Les options possibles sont les suivantes : *MODE_TRADES* (possibilité par défaut) indique au programme de sélectionner la position dans la liste des positions ouvertes ou en attente. *MODE_HISTORY* indique au programme de sélectionner la position dans la liste des positions clôturées ou annulées.

Si vous connaissez le numéro de ticket de la position, vous pouvez utiliser directement cette fonction comme ci-dessous :

```
if(OrderSelect(iTicket,SELECT_BY_POS, MODE_TRADES) == True)
  {
  // Commandes que vous désirez effectuer sur la position avec le numéro de ticket iTicket.
  }


if(OrderSelect(iTicket,SELECT_BY_POS, MODE_HISTORY))
  {
  // Commandes que vous désirez effectuer sur la position avec le numéro de ticket iTicket.
  }
```

Dans le deuxième exemple, nous avons omis « == True ». En effet, même s'il est conseillé de l'écrire, par défaut dans une instruction conditionnelle, la condition testée est « == True » et il n'est donc pas nécessaire de l'écrire.

Dans la plupart des cas, soit vous ne connaîtrez pas le numéro de ticket d'ordre, soit vous désirerez effectuer une opération sur plusieurs positions sans avoir à garder constamment en mémoire les numéros de ticket. Vous pouvez donc insérer la fonction de sélection dans une boucle afin d'analyser un groupe de positions comme dans l'exemple ci-dessous :

```
for(int iCnt = OrdersTotal()-1; iCnt >= 0; iCnt --)
 {
 if(OrderSelect(iCnt, SELECT_BY_POS, MODE_TRADES))
   {
   // Commandes que vous désirez effectuer sur la position sélectionnée.
   }
 }
```

Dans l'exemple ci-dessous, le programme va sélectionner successivement toutes les positions ouvertes ou en attente. En effet, *OrdersTotal()* permet d'obtenir le nombre total de positions et à chaque itération, vous diminuez le numéro de 1 pour traiter la position suivante.

Dans MetaTrader, chaque position obtient un numéro entier correspondant à sa position dans le bassin auquel elle appartient (positions ouvertes ou en attente / positions clôturées ou annulées). Le décompte commence à 0 mais *OrdersTotal()* renvoie le nombre de position en commençant le décompte à 1, c'est pour cela que nous avons soustrait 1 à *OrdersTotal()* dans la boucle.

Fonctions pour obtenir de l'information sur les positions

Maintenant que vous savez comment sélectionner une position, nous allons voir quelles sont les informations que nous pouvons obtenir.

Deux fonctions ne nécessitent pas une sélection préalable et font en fait souvent partie du processus de sélection. Nous avons déjà vu *OrdersTotal()* dans l'exemple précédent mais il existe également *OrdersHistoryTotal()*. Ces deux fonctions permettent d'obtenir respectivement le nombre total de positions ouvertes ou en attente et le nombre total de positions clôturées ou annulées.

Dans le cas de *OrdersHistoryTotal()*, le nombre de positions retourné dépend de l'historique affiché dans le terminal MetaTrader dans l'onglet *Historique*.

La syntaxe pour ces deux fonctions est la suivante :

```
int OrdersTotal()

int OrdersHistoryTotal()
```

Voici quelques exemples simples :

```
int iTotal=OrdersTotal();

for(int iCnt = OrdersHistoryTotal()-1; iCnt >= 0; iCnt --)
 {
 if(OrderSelect(iCnt, SELECT_BY_POS, MODE_TRADES))
   {
   // Commandes que vous désirez effectuer sur la position sélectionnée.
   }
 }
```

Sans plus tarder, voici la liste des fonctions permettant d'obtenir ou de traiter des informations au sujet de la position préalablement sélectionnée à l'aide de la fonction *OrderSelect()* :

Fonction	Syntaxe	Description
OrderClosePrice()	double OrderClosePrice()	Renvoie le prix de clôture
OrderCloseTime()	datetime OrderCloseTime()	Renvoie la date et heure de clôture/annulation de la position
OrderComment()	string OrderComment()	Renvoie le commentaire de la position
OrderCommission()	double OrderCommission()	Renvoie la commission de la position
OrderExpiration	datetime OrderExpiration()	Renvoie la date et heure d'expiration de la position en attente
OrderLots()	double OrderLots()	Renvoie le volume de la position
OrderMagicNumber()	int OrderMagicNumber()	Renvoie le numéro magique de la position
OrderOpenPrice()	double OrderOpenPrice()	Renvoie le prix d'ouverture de la position
OrderOpenTime()	double OrderOpenTime()	Renvoie la date et heure d'ouverture de la position
OrderPrint()	void OrderPrint()	Affiche des informations au sujet de la position dans le journal du terminal
OrderProfit()	double OrderProfit()	Renvoie le profit de la position (en cours ou final)
OrderStopLoss()	double OrderStopLoss()	Renvoie le prix stop de la position
OrderSwap()	double OrderSwap()	Renvoie les intérêts engendrés par la position
OrderSymbol()	string OrderSymbol()	Renvoie le nom de la paire de la position
OrderTakeProfit()	double OrderTakeProfit()	Renvoie le prix limite de la position
OrderTicket()	int OrderTicket()	Renvoie le numéro de ticket de la position
OrderType()	int OrderType()	Renvoie le type d'ordre de la position (voir tableau 17)

Tableau 19 : Fonctions pour obtenir de l'information sur les positions

Les informations affichées par la fonction *OrderPrint()* apparaîtront sous la forme suivante dans le journal du Terminal : *numéro de ticket; date et heure d'ouverture; type d'ordre; volume; prix d'ouverture; prix stop; prix limite; date et heure de clôture; prix de fermeture; commission; intérêts; profit; commentaire; numéro magique; date d'expiration*

Voyons voir maintenant quelques exemples en utilisant les différentes fonctions étudiées.

```
for(int iCnt = OrdersTotal()-1; iCnt >= 0; iCnt --)
 {
 if(OrderSelect(iCnt, SELECT_BY_POS, MODE_TRADES))
  {
  if(OrderType() == OP_BUY && OrderSymbol() == "EURUSD")
   {
   if(Bid >= OrderOpenPrice() + 0.0050 && OrderStopLoss() != OrderOpenPrice())
    OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice(), OrderTakeProfit(), 0, Blue);
   }
  if(OrderType() == OP_SELL && OrderSymbol() == "EURUSD")
   {
   if(Ask <= OrderOpenPrice() - 0.0050 && OrderStopLoss() != OrderOpenPrice())
    OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice(), OrderTakeProfit(), 0, Blue);
   }
  }
 }
```

Le code ci-dessus permet de ramener le prix stop d'une position dès que le prix atteint un certain seuil (dans ce cas-ci, le prix d'ouverture plus 50 pips). Pour sélectionner la ou les positions, nous lançons une analyse sur les positions ouvertes ou en attente. Une fois une position est sélectionnée, nous vérifions certains critères à l'aide d'une instruction conditionnelle (la position doit être une position ouverte de type achat sur la paire EURUSD ou une position ouverte de type vente sur la paire EURUSD). Si la position remplit ces deux critères, nous vérifions à l'aide d'une deuxième boucle conditionnelle si le profit actuel est de 50 pips et si le prix stop est différent du prix d'ouverture. Si les deux critères sont vérifiés, la boucle modifiera l'ordre en question en ramenant le prix stop au niveau du prix d'ouverture.

```
if(OrderSelect(0, SELECT_BY_POS, MODE_HISTORY))
 {
 if(OrderType() == OP_BUY)
  {
  bBuy = False;
  }
 else bBuy = True;
 }

if(bBuy == True)
 {
 OrderSend("EURUSD", OP_BUY, 1, Ask, 3, 0, 0, "", 0, 0, Blue);
 }
```

L'exemple ci-dessus analyse la dernière transaction effectuée sur le compte. Le fait que le critère soit *OrderType() == OP_BUY* sous-entend qu'il s'agit d'un ordre réalisé et non pas annulé. Si la dernière transaction est un ordre d'achat, le programme interdira de repasser un ordre de ce type tant et aussi longtemps qu'il n'y aura pas eu une opération de vente.

20
Variables prédéfinies

Vous avez déjà découvert deux variables prédéfinies lors de l'étude de la fonction de passage d'ordre : *Ask* et *Bid*.

Vous trouverez dans le tableau ci-dessous la liste de toutes ces variables :

Variable	Syntaxe	Description
Ask	double Ask	Dernier prix d'achat connu pour la paire du graphique
Bid	double Bid	Dernier prix de vente connu pour la paire du graphique
Digits	int Digits	Nombre de décimales après le point du prix de la paire du graphique
Point	double Point	Valeur d'un pip pour la paire du graphique

Tableau 20 : Variables prédéfinies

Voici un exemple d'utilisation de ces variables :

```
extern int iNbrPips = 50;

OrderSend(Symbol(), OP_BUY, 1, Ask, 3, 0, Ask + iNbrPips*Point);
```

La variable prédéfinie *Point* permet de ne pas avoir à calculer vous-même l'équivalence en décimales d'un nombre de pips pour une paire en particulier. Le code ci-dessus fonctionnera aussi bien pour des paires à 2 décimales que celles à 4 décimales.

Certains courtiers ont instauré un système à 3 et 5 décimales au lieu de 2 et 4 pour déterminer les prix. Si votre courtier utilise 5 décimales au lieu de 4, le résultat de l'opération suivante : 50 * Point sera 0.0005 et non pas 0.005. Pour palier au problème lié à ce nouveau système, nous allons créer une fonction permettant de toujours avoir la bonne valeur pour un pip.

La fonction sera donc :

```
double fPoint(string sPaire)
 {
 if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 3)
  {
  double dPoint = 0.01;
  }
 if((MarketInfo(sPaire, MODE_DIGITS)) == 4 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
  {
  dPoint = 0.0001;
  }
 return(dPoint);
 }
```

Dorénavant, au lieu d'utiliser *Point* dans notre code, nous utiliserons la fonction *fPoint()* qui nous permettra de toujours avoir la valeur correcte pour un pip et ce quelque soit le système de cotation utilisé par le courtier.

Le même problème se pose pour le paramètre *slippage*. Nous allons donc créer une deuxième fonction chargée de résoudre ce problème.

```
double fSlippage(int iPipsSlippage)
 {
 if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 4)
  {
  double dSlippage = iPipsSlippage;
  }
 if((MarketInfo(sPaire, MODE_DIGITS)) == 3 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
  {
  dSlippage = iPipsSlippage * 10;
  }
 return(dSlippage);
 }
```

Les variables décrites antérieurement ne servent qu'à obtenir des informations sur la paire du graphique sur lequel est placé l'expert. Nous allons maintenant voir comment faire pour obtenir des informations sur n'importe quelle paire et ce quelque soit le graphique de l'expert.

Grâce à la fonction *MarketInfo()*, vous pouvez obtenir des informations sur n'importe quelle paire disponible dans votre plateforme. La syntaxe de la fonction est :

```
double MarketInfo(string sPaire, int iType)
```

string sPaire est à remplacer par la paire de laquelle vous désirez obtenir des informations.

int iType vous permet de sélectionner le type d'information que vous désirez obtenir. Vous trouverez la liste des choix disponibles dans le tableau suivant.

Forme littérale	Forme numérique	Équivalent
MODE_LOW	1	Prix minimum de la journée
MODE_HIGH	2	Prix maximum de la journée
MODE_BID	9	Dernier prix de vente connu
MODE_ASK	10	Dernier prix d'achat connu
MODE_TIME	5	Date et heure du dernier tick
MODE_POINT	11	Valeur d'un pip
MODE_DIGITS	12	Nombre de décimales après le point

MODE_SPREAD	13	Valeur du spread en pips
MODE_STOPLEVEL	14	Distance minimum du stop et limite en pips par rapport au prix d'entrée
MODE_LOTSIZE	15	Équivalent d'un lot standard dans la devise du compte
MODE_TICKVALUE	16	Valeur d'un pip dans la devise du compte
MODE_TICKSIZE	17	Valeur d'un pip dans la devise de base de la position
MODE_SWAPLONG	18	Intérêts pour une position à l'achat
MODE_SWAPSHORT	19	Intérêts pour une position à la vente
MODE_STARTING	20	Date de création d'un contrat futur
MODE_EXPIRATION	21	Date d'expiration d'un contrat futur
MODE_TRADEALLOWED	22	Renvoie l'information comme quoi il est possible de prendre position sur une paire spécifique
MODE_MINLOT	23	Volume minimal en lots autorisé
MODE_LOTSTEP	24	Fraction minimale en lots autorisée
MODE_MAXLOT	25	Volume maximal en lots autorisé
MODE_SWAPTYPE	26	Renvoie la méthode de calcul des intérêts (0 en pips, 1 dans la devise de base de la paire, 2 en taux d'intérêts ou 3 dans la devise du compte)
MODE_PROFITCALCMODE	27	Renvoie la méthode de calcul des profits (0 pour Forex, 1 pour CFD et 2 pour Futures)
MODE_MARGINCALCMODE	28	Méthode de calcul de la marge (0 pour Forex, 1 pour CFD et 2 pour Futures)
MODE_MARGININIT	29	Montant minimal requis de marge pour 1 lot
MODE_MARGINMAINTENANCE	30	Montant minimal requis de marge pour maintenir une position de 1 Lot ouverte
MODE_MARGINHEDGED	31	Montant de marge en cas de couverture calculée pour 1 Lot

MODE_MARGINREQUIRED	32	Montant de marge libre requis pour prendre position à l'achat pour 1 Lot
MODE_FREEZELEVEL	33	Distance minimale requise en pips entre un prix indiqué et le prix du marché pour modifier, annuler ou clôturer une position

Tableau 21 : Informations disponibles avec la fonction MarketInfo()

Quelques exemples d'utilisation de la fonction :

```
dMinLot = MarketInfo(Symbol(), MODE_MINLOT);
dMaxLot = MarketInfo(Symbol(), MODE_MAXLOT);


if(iStop < MarketInfo(Symbol(), MODE_STOPLEVEL))
  iStop = MarketInfo(Symbol(), MODE_STOPLEVEL);
```

Les codes ci-dessus permettent de s'assurer que les différentes variables respectent les minima requis par le courtier. Dans le dernier cas, le programme vérifie si le niveau de stop est suffisamment éloigné par rapport au minimum imposé par le courtier et, si ce n'est pas le cas, redéfinit la variable avec la valeur correcte.

21
Votre premier expert consultant #5

Nous allons maintenant incorporer quelques unes des dernières fonctions étudiées dans notre expert consultant.

En premier, nous allons ajouter des instructions pour placer des ordres stops pour acheter et vendre lorsque le prix atteint les niveaux plus haut/bas de la veille.

```
OrderSend(Symbol(), OP_BUYSTOP, 1, dPlusHaut + iDistance*fPoint(Symbol()), fSlippage(Symbol(), 3), 0,
0,  "Achat sur rupture du niveau plus haut de la veille", iMagic,Time[0] + (((iHeureFin -
TimeHour(Time[0]))*3600)+((iMinutesFin - TimeMinute(Time[0]))*60)), Blue);


OrderSend(Symbol(), OP_SELLSTOP, 1, dPlusBas - iDistance* fPoint(Symbol()), fSlippage(Symbol(), 3), 0,
0,  "Vente sur rupture du niveau plus bas de la veille", iMagic,Time[0] + (((iHeureFin -
TimeHour(Time[0]))*3600)+((iMinutesFin - TimeMinute(Time[0]))*60)), Red);
```

Vous remarquerez que nous avons utilisé notre fonction *fPoint()* pour déterminer la valeur *Point* afin de rendre notre programme compatible avec tous les courtiers. Nous ajouterons la fonction créée précédemment à la fin de notre programme ainsi que la fonction *fSlippage()* pour corriger le slippage.

Nous allons maintenant ajouter les variables nécessaires au début de notre programme.

```
extern int iDistance = 5;
extern int iMagic = 123456;
```

iDistance permet de paramétrer un décalage en pips par rapport au niveau. Si vous désirez rentrer au niveau du plus haut/bas, il suffit de paramétrer *iDistance* à 0.

Nous devons également ajouter une sécurité pour éviter que le programme n'ouvre les positions à répétitions. Pour cela, nous allons ajouter le code suivant :

```
if(bTrade == True)
  {
  // inclure ici les fonctions de passage d'ordre
  bTrade = False;
  }
```

Finalement nous modifions le code suivant dans le code source (ajout en gras) afin de réautoriser le passage d'ordre une fois la journée finie :

```
if(iDate != Day())
  {
  bRecuperation = False;
  bTrade = True;
  }
```

Nous ajouterons la déclaration de notre variable *bTrade* au début du programme également.

```
bool bTrade = True;
```

Notre code complet ressemble maintenant à ceci (ajouts et/ou modifications en gras):

```
//+------------------------------------------------------------------+
//|                                Votre Premier Expert.mq4          |
//|                                 Copyright © 2011,                |
//|                               http://www. eole-trading.com       |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www. eole-trading.com"


extern int iHeureDebut = 9;
extern int iMinutesDebut = 30;
extern int iHeureFin = 17;
extern int iMinutesFin = 30;
extern int iDistance = 5;
extern int iMagic = 123456;

int iDate;
int iTimeDebut;
int iTimeFin;
int iTimeSeconds;
double dPlusHaut;
double dPlusBas;
bool bRecuperation = False;
bool bTrade = True;


//+------------------------------------------------------+
//| expert initialization fonction                       |
//+------------------------------------------------------+
int init()
  {

  iTimeDebut = iHeureDebut * 3600 + iMinutesDebut * 60;
  iTimeFin = iHeureFin * 3600 + iMinutesFin * 60;

  return(0);
  }
//+------------------------------------------------------+
//| expert deinitialization fonction                     |
//+------------------------------------------------------+
int deinit()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------+
//| expert start fonction                                |
//+------------------------------------------------------+
int start()
  {

  iTimeSeconds = Hour()*3600 + Minute()*60 + Seconds();


  if(iTimeSeconds >= iTimeDebut && iTimeSeconds < iTimeFin)
    {
    if(bRecuperation == False)
      {
      iDate = Day();
```

```
    dPlusHaut = iHigh(NULL, PERIOD_D1, 1);
    dPlusBas = iLow(NULL, PERIOD_D1, 1);
    bRecuperation = True;
    fTracageLignes("Point plus haut", dPlusHaut, Blue);
    fTracageLignes("Point plus bas", dPlusBas, Red);
    }


  if(bTrade == True)
    {
    OrderSend(Symbol(), OP_BUYSTOP, 1, dPlusHaut + iDistance* fPoint(Symbol()),
    fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la veille", iMagic,Time[0] +
    (((iHeureFin - TimeHour(Time[0]))*3600) + ((iMinutesFin - TimeMinute(Time[0]))*60)), Blue);

    OrderSend(Symbol(), OP_SELLSTOP, 1, dPlusBas - iDistance* fPoint(Symbol()),
    fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la veille", iMagic,Time[0] +
    (((iHeureFin - TimeHour(Time[0]))*3600)+((iMinutesFin - TimeMinute(Time[0]))*60)), Red);

    bTrade = False;
    }

  if(iDate != Day())
    {
    bRecuperation = False;
    bTrade = True;
    }

  }

 return(0);
 }
//+------------------------------------------------------+


void fTracageLignes(string sNomLigne, double dPrix, color cCouleur)
 {
 ObjectDelete(sNomLigne);
 ObjectCreate(sNomLigne,OBJ_HLINE, 0, 0, dPrix);
 ObjectSet(sNomLigne, OBJPROP_COLOR, cCouleur);
 }


double fPoint(string sPaire)
 {
 if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 3)
   {
   double dPoint = 0.01;
   }
 if((MarketInfo(sPaire, MODE_DIGITS)) == 4 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
   {
   dPoint = 0.0001;
   }
 return(dPoint);
 }


double fSlippage(string sPaire, int iPipsSlippage)
 {
 if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 4)
   {
   double dSlippage = iPipsSlippage;
   }
 if((MarketInfo(sPaire, MODE_DIGITS)) == 3 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
   {
   dSlippage = iPipsSlippage * 10;
   }
 return(dSlippage);
 }
```

Nous allons maintenant ajouter les fonctions permettant d'ajouter un prix stop et limite à nos deux positions.

La première étape consiste à savoir si le passage d'ordre a été couronné de succès. Pour ce faire, nous allons stocker l'entier généré par la fonction *OrderSend()* grâce au code suivant (ajouts en gras) :

```
iTicket = OrderSend(Symbol(), OP_BUYSTOP, 1, dPlusHaut + iDistance* fPoint(Symbol()),
fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la veille", iMagic,Time[0] +
(((iHeureFin - TimeHour(Time[0]))*3600) + ((iMinutesFin - TimeMinute(Time[0]))*60)), Blue);

iTicket = OrderSend(Symbol(), OP_SELLSTOP, 1, dPlusBas - iDistance* fPoint(Symbol()),
fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la veille", iMagic,Time[0] + (((iHeureFin
- TimeHour(Time[0]))*3600)+((iMinutesFin - TimeMinute(Time[0]))*60)), Red);
```

Nous déclarerons la variable au début de notre programme.

```
int iTicket = 0;
```

Si le passage d'ordre est exécuté avec succès, la fonction *OrderSend()* renvoie un entier correspondant au numéro de ticket de la position. Si le numéro est *0*, cela signifierait que le passage d'ordre a échoué. Il ne reste plus qu'à créer la fonction pour modifier la position.

```
void fTicket(int iTicket)
 {
 if(iTicket != 0)
   {
   OrderSelect(iTicket, SELECT_BY_TICKET);

   if(OrderType() == 4)
     OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() - iStop * fPoint(Symbol()),
     OrderOpenPrice() + iLimite * fPoint(Symbol()), OrderExpiration());

   if(OrderType() == 5)
     OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() + iStop * fPoint(Symbol()),
     OrderOpenPrice() - iLimite * fPoint(Symbol()), OrderExpiration());
   }
 }
```

Il ne reste plus qu'à insérer la fonction à la fin du programme et ajouter un appel après chaque fonction *OrderSend()*.

Les autres modifications que nous apporterons au code source seront d'ajouter les variables *iStop* et *iLimite*.

Notre code source ressemble maintenant à :

```
//+------------------------------------------------------------------+
//|                          Votre Premier Expert.mq4                |
//|                          Copyright © 2011,                       |
//|                       http://www. eole-trading.com               |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www. eole-trading.com"


extern int iHeureDebut = 9;
extern int iMinutesDebut = 30;
extern int iHeureFin = 17;
```

```
extern int iMinutesFin = 30;
extern int iDistance = 5;
extern int iMagic = 123456;
extern int iStop = 50;
extern int iLimite = 50;

int iDate;
int iTimeDebut;
int iTimeFin;
int iTimeSeconds;
int iTicket;
double dPlusHaut;
double dPlusBas;
bool bRecuperation = False;
bool bTrade = True;


//+------------------------------------------------------+
//| expert initialization fonction                   |
//+------------------------------------------------------+
int init()
  {

  iTimeDebut = iHeureDebut * 3600 + iMinutesDebut * 60;
  iTimeFin = iHeureFin * 3600 + iMinutesFin * 60;

  return(0);
  }
//+------------------------------------------------------+
//| expert deinitialization fonction                 |
//+------------------------------------------------------+
int deinit()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------+
//| expert start fonction                            |
//+------------------------------------------------------+
int start()
  {

  iTimeSeconds = Hour()*3600 + Minute()*60 + Seconds();


  if(iTimeSeconds >= iTimeDebut && iTimeSeconds < iTimeFin)
    {
    if(bRecuperation == False)
      {
      iDate = Day();
      dPlusHaut = iHigh(NULL, PERIOD_D1, 1);
      dPlusBas = iLow(NULL, PERIOD_D1, 1);
      bRecuperation = True;
      fTracageLignes("Point plus haut", dPlusHaut, Blue);
      fTracageLignes("Point plus bas", dPlusBas, Red);
      }


    if(bTrade == True)
      {
      iTicket = OrderSend(Symbol(), OP_BUYSTOP, 1, dPlusHaut + iDistance* fPoint(Symbol()),
      fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la veille", iMagic,Time[0] +
      (((iHeureFin - TimeHour(Time[0]))*3600) + ((iMinutesFin - TimeMinute(Time[0]))*60)), Blue);
      fTicket(iTicket);

      iTicket = OrderSend(Symbol(), OP_SELLSTOP, 1, dPlusBas - iDistance* fPoint(Symbol()),
      fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la veille", iMagic,Time[0] +
      (((iHeureFin - TimeHour(Time[0]))*3600)+((iMinutesFin - TimeMinute(Time[0]))*60)), Red);
```

```
      fTicket(iTicket);

      bTrade = False;
      }

    if(iDate != Day())
      {
      bRecuperation = False;
      bTrade = True;
      }
    }
  return(0);
  }
//+------------------------------------------------+

void fTracageLignes(string sNomLigne, double dPrix, color cCouleur)
  {
  ObjectDelete(sNomLigne);
  ObjectCreate(sNomLigne,OBJ_HLINE, 0, 0, dPrix);
  ObjectSet(sNomLigne, OBJPROP_COLOR, cCouleur);
  }

double fPoint(string sPaire)
  {
  if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 3)
    {
    double dPoint = 0.01;
    }
  if((MarketInfo(sPaire, MODE_DIGITS)) == 4 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
    {
    dPoint = 0.0001;
    }
  return(dPoint);
  }

double fSlippage(string sPaire, int iPipsSlippage)
  {
  if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 4)
    {
    double dSlippage = iPipsSlippage;
    }
  if((MarketInfo(sPaire, MODE_DIGITS)) == 3 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
    {
    dSlippage = iPipsSlippage * 10;
    }
  return(dSlippage);
  }

void fTicket(int iTicket)
  {
  if(iTicket != -1)
    {
    OrderSelect(iTicket, SELECT_BY_TICKET);
    if(OrderType() == 4)
      OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() - iStop * fPoint(Symbol()),
      OrderOpenPrice() + iLimite * fPoint(Symbol()), OrderExpiration());

    if(OrderType() == 5)
      OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() + iStop * fPoint(Symbol()),
      OrderOpenPrice() - iLimite * fPoint(Symbol()), OrderExpiration());
      }
  }
```

Cet expert étant à but éducatif, le filtre horaire a volontairement été laissé tel quel même si il est inutile compte tenu que les positions seront annulées automatiquement lorsque l'heure paramétrée comme heure de fin est atteinte.

22
Gestion des risques

L'un des facteurs d'échec les plus fréquents pour les nouveaux venus dans la spéculation sur le marché des changes est la prise excessive de risque. Celle-ci se traduit en général par des volumes trop grands par rapport à la balance du compte.

L'une des meilleures façons d'éviter cette erreur est de définir le volume de la position en fonction d'un risque maximal en pourcentage de la balance du compte.

Pour ce faire, nous avons besoin de définir le stop en pips et le pourcentage de risque souhaité.

```
extern int iStop = 50;
extern double dPourcentageRisque = 3;
```

La première étape consiste à calculer l'équivalent du pourcentage de la balance du compte.

```
double dRisqueMax = (dPourcentageRisque/100) * AccountBalance();
```

Nous allons en profiter pour présenter les fonctions permettant de renvoyer des informations sur le compte. Vous trouverez la liste complète dans le tableau ci-dessous.

Fonction	Syntaxe	Description
AccountBalance	double AccountBalance()	Renvoie la balance du compte
AccountCredit	double AccountCredit()	Renvoie le crédit du compte
AccountCompany	string AccountCompany()	Renvoie le nom du courtier
AccountCurrency	string AccountCurrency()	Renvoie la devise du compte
AccountEquity	double AccountEquity	Renvoie l'équité du compte
AccountFreeMargin	double AccountFreeMargin()	Renvoie la marge libre du compte
AccountFreeMarginCheck	double AccountFreeMarginCheck(string sPaire, int iTypeOrdre, double dVolume)	Renvoie la marge libre du compte après que la position spécifiée ait été ouverte (symbol : paire ; cmd : type d'opération ; volume : volume de la position)
AccountLeverage	int AccountLeverage()	Renvoie le levier du compte
AccountMargin	double AccountMargin()	Renvoie la marge utilisée du compte
AccountName	string AccountName()	Renvoie le nom du compte
AccountNumber	int AccountNumber()	Renvoie le numéro du compte

AccountProfit	double AccountProfit()	Renvoie le profit réalisé du compte
AccountServer	string AccountServer()	Renvoie le nom du serveur
AccountStopoutLevel	int AccountStopoutLevel()	Renvoie la valeur du compte pour laquelle le trading est stoppé
AccountFreeMarginMode	double AccountFreeMarginMode( )	Renvoie la méthode de calcul de la marge libre (0 : ne prend pas en compte des profits/pertes des positions ouvertes ; 1 : prend en compte les profits/pertes des positions ouvertes ; 2 : ne prend en compte que les profits ; 3 : ne prend en compte que les pertes
AccountStopMode	int AccountStopoutMode()	Renvoie la méthode de calcul du niveau Stopout (0 : ratio entre marge et équité ; 1 : comparaison entre la marge libre et la valeur absolue du compte)

Tableau 22 : Fonctions pour obtenir des informations sur le compte

Dans le calcul du risque maximal, nous avons utilisé *AccountBalance()* et non pas *AccountEquity()* car ce dernier ne prend en compte que les profits/pertes non réalisés et peut donc fausser les calculs à la fin.

La deuxième étape consiste à déterminer la valeur d'un pip et adapter celle-ci en fonction du fait que le courtier utilise une décimale supplémentaire ou non.

```
double dValeurPip = MarketInfo(Symbol(), MODE_TICKVALUE);

if(Point = 0.001 || Point = 0.00001)
  {
  dValeurPip = dValeur * 10;
  }
```

Une fois le risque maximal calculé et la valeur du pip, il ne nous reste plus qu'à calculer le volume de la position.

```
double dLots = (dRisqueMax / iStop) / dValeurPip;
```

La dernière étape consiste à s'assurer de la compatibilité de notre volume avec les paramètres imposés par le courtier.

```
if(dLots < MarketInfo(Symbol(),MODE_MINLOT))
  dLots = MarketInfo(Symbol(),MODE_MINLOT);

if(dLots > MarketInfo(Symbol(),MODE_MAXLOT))
```

```
  dLots = MarketInfo(Symbol(),MODE_MAXLOT);

if(MarketInfo(Symbol(),MODE_LOTSTEP) == 0.01)
  dLots = NormalizeDouble(dLots, 2);
else
  dLots = NormalizeDouble(dLots, 1);
```

Dans le code ci-dessus, nous vérifions que le volume calculé ne soit pas inférieur au minimum requis, supérieur au maximum requis et enfin que le nombre de décimales correspond à celui autorisé par le courtier (le programme arrondira automatiquement le volume dépendant du nombre de décimales requis par le courtier grâce à la fonction *NormalizeDouble()*).

Nous allons maintenant créer une fonction qui contiendra tous les éléments de la gestion du risque.

```
double fLots(double dPourcentageRisque, int iStop)
  {
  double dRisqueMax = (dPourcentageRisque/100) * AccountBalance();
  double dValeurPip = MarketInfo(Symbol(), MODE_TICKVALUE);
  if(Point == 0.001 || Point == 0.00001)
    {
    dValeurPip = dValeurPip * 10;
    }
  double dLots = (dRisqueMax / iStop) / dValeurPip;
  if(dLots < MarketInfo(Symbol(),MODE_MINLOT))
    dLots = MarketInfo(Symbol(),MODE_MINLOT);
  if(dLots > MarketInfo(Symbol(),MODE_MAXLOT))
    dLots = MarketInfo(Symbol(),MODE_MAXLOT);
  if(MarketInfo(Symbol(),MODE_LOTSTEP) == 0.01)
    dLots = NormalizeDouble(dLots, 2);
  else
    dLots = NormalizeDouble(dLots, 1);
  return(dLots);
  }
```

Vous pouvez maintenant copier-coller cette fonction dans n'importe quel programme sans nécessité de modifier quoique ce soit comme nous le ferons pour notre programme dans un prochain chapitre.

23
Fonctions de conversion

Nous avons utilisé dans le chapitre précédent la fonction de conversion *NormalizeDouble()* pour arrondir notre volume. Nous allons maintenant étudier de façon plus approfondie ces fonctions. Elles sont au nombre de 7.

Fonction CharToStr()

Cette fonction permet la conversion d'un code caractère ASCII dans son caractère correspondant. Si vous désirez connaître les codes caractères, nous vous invitons à consulter le site internet suivant : http://www.asciitable.com/

La syntaxe de la fonction est :

```
string CharToStr(int iCodeCaractere)
```

Il faut donc remplacer *int iCodeCaractere* par le code dont vous désirez obtenir le caractère correspondant.

Fonction DoubleToStr()

DoubleToStr() permet de convertir un nombre réel en chaîne de caractères en précisant le nombre de décimales souhaité. La syntaxe de la fonction est la suivante :

```
string DoubleToStr(double dReel, int iPrecision)
```

double dReel correspond au nombre réel que vous souhaitez convertir.

int iPrecision sert à préciser le nombre de décimales (entre 0 et 8) souhaité après le point.

Un exemple pourrait être :

```
string sPrix=DoubleToStr(Bid, 2);
```

Si nous appliquons le code ci-dessus sur un graphique EURUSD où le prix *BID* est 1.3345, la chaîne de caractère assignée à *sPrix* sera « 1.33 ».

Fonction NormalizeDouble()

Comme nous l'avons vu précédemment, *NormalizeDouble()* permet d'arrondir un nombre réel en précisant le nombre de décimales (entre 0 et 8) souhaité après le point. La syntaxe de la fonction est :

```
double NormalizeDouble(double dReel, int iPrecision)
```

double dReel correspond au nombre réel que vous souhaitez arrondir.

int iPrecision sert à préciser le nombre de décimales (entre 0 et 8) souhaité après le point.

```
double  dArrondi = NormalizeDouble(Bid, 1);
```

Si nous reprenons l'exemple précédent avec EURUSD et un prix *BID* de 1.3345, la valeur assignée à *dArrondi* sera 1.3.

Fonction StrToInteger()

Cette fonction permet de convertir une chaîne de caractères en nombre entier. La syntaxe de la fonction est :

```
int StrToInteger(string sChaineDeCaractere)
```

string sChaineDeCaractere est à remplacer par la chaîne de caractères composée de caractères numériques et sans point.

```
int iEntier = StrToInteger("2345");
```

Le code ci-dessus va assigner la valeur 2345 à la variable *int iEntier*. Vous remarquerez les guillemets encadrant 2345 dans le code pour signifier qu'il s'agit d'une chaîne de caractères.

Fonction StrToDouble()

Similaire à la fonction précédente, celle-ci convertit une chaîne de caractères en nombre réel. La syntaxe de la fonction est :

```
int StrToDouble(string sChaineDeCaractere)
```

string sChaineDeCaractere est à remplacer par la chaîne de caractères composée de caractères numériques avec ou sans point.

```
double dReel = StrToInteger("2345.1234");
```

Le code ci-dessus va assigner la valeur 2345.1234 à la variable *double dReel*.

Fonction StrToTime()

Cette fonction permet de convertir une chaîne de caractères de la forme "YYYY.MM.DD HH:MI" (ANNE.MOIS.JOUR HEURE:MINUTES) en secondes pour respecter le type temporel de MetaTrader. La syntaxe de la fonction est la suivante :

```
datetime StrToTime( string  sDate)
```

string sDate est à remplacer par la date que vous souhaitez convertir. Attention à bien respecter le format "YYYY.MM.DD HH:MI".

Voici quelques exemples :

```
datetime dtDate = StrToTime("2011.1.23 14 :16" );

datetime dtDate = StrToTime("2011.1.23" );

datetime dtDate = StrToTime("14 :16" );
```

Le premier exemple montre la conversion d'une date et d'une heure. Dans le deuxième exemple, seule la date est écrite, le programme convertira donc cette date en utilisant minuit (00:00) comme heure. Dans le dernier exemple, nous n'avons écrit que l'heure, le programme convertira l'heure en utilisant la date du jour.

Fonction TimeToStr()

Cette fonction est en fait l'inverse de la fonction précédente. Dans ce cas-ci, nous fournissons une date au format temporel de MetaTrader et souhaitons obtenir sa conversion en chaîne de caractères plus facile à lire. La syntaxe de la fonction est :

```
string TimeToStr(datetime dtDate, int iMode)
```

datetime dtDate est à remplacer par la variable temporel de MetaTrader.

int Mode est optionnel et permet de choisir le format de la conversion.

Mode	Conversion	Exemple
TIME_DATE	"YYYY.MM.DD"	"2010.1.23"
TIME_MINUTES	"HH.MI"	"14:23"
TIME_SECONDS	"HH:MI:SS"	"14:23:45"
TIME_DATE|TIME_MINUTES	"YYYY.MM.DD HH:MI"	"2010.1.23 14:23"
TIME_DATE|TIME_SECONDS	« YYYY.MM.DD HH :MI :SS »	"2010.1.23 14:23:45"

Tableau 23 : Format de conversion pour la fonction TimeToStr()

Par défaut, le mode est *TIME_DATE | TIME_MINUTES*.

Un exemple :

```
string  sDate = TimeToStr(Time[2],TIME_DATE|TIME_MINUTES);
```

Le code ci-dessus convertira l'heure d'ouverture de la troisième barre en partant de l'actuelle barre sous la forme "YYYY.MM.DD HH:MI" et assignera la chaîne de caractères à la variable *string sDate*.

24
La gestion des erreurs

Nous allons maintenant traiter de la détection et de la résolution des erreurs qui pourraient survenir au cours de l'exécution du programme.

Un premier point important à retenir concerne le fonctionnement interne de MetaTrader. En effet, MetaTrader ne dispose que d'un seul canal pour l'exécution d'ordres provenant des experts consultants. Cela signifie que MetaTrader ne peut traiter qu'un ordre à la fois. Si vous n'utilisez qu'un expert sur votre plateforme, ce problème sera inexistant mais dans l'éventualité que vous ayez plus d'un expert fonctionnant en parallèle, la probabilité d'avoir des erreurs d'exécution augmente.

La première étape avant d'exécuter un ordre est donc de vérifier si le canal est disponible et s'il ne l'est pas, de retarder l'exécution (ce délai ne sera que de quelques secondes voire millisecondes et ne risque donc pas d'altérer les conditions du trade).

La fonction que nous allons utiliser est *IsTradeContextBusy()*. Il s'Agit d'une fonction booléenne qui renvoie *True* si le canal est occupé et *False* lorsqu'il est libre. La syntaxe de la fonction est la suivante :

```
bool IsTradeContextBusy()
```

Afin d'ordonner au programme de patienter lorsque le canal est occupé, nous devons insérer la fonction suivante dans une boucle de type *while*. De cette façon, le programme ne permettra pas l'exécution d'un ordre tant et aussi longtemps que le canal est occupé.

```
while(IsTradeContextBusy() == True)
  Sleep(1000);
```

Dans le code ci-dessus, le programme va vérifier en premier lieu la disponibilité du canal. Si celui-ci est libre, le programme continuera d'exécuter le code présent à la suite et en dehors de la boucle. Si le canal est occupé, le programme exécutera la fonction *Sleep()* qui permet de suspendre l'exécution du programme pendant une période précisée. Dans ce cas-ci, nous avons indiqué 1 seconde soit 1000 millisecondes. Ce code est à insérer avant tout type de passage d'ordre par le programme.

Un autre problème assez récurrent est lié à l'utilisation de variables prédéfinies (*Ask* et *Bid* par exemple). En effet, la valeur associée à chacune de ces valeurs, lorsqu'elles sont utilisées dans un programme, est définie au début de chaque itération de votre programme. À chaque fois que votre programme est exécuté dans sa totalité, la valeur de ces variables est réinitialisée. Le temps nécessaire à l'exécution d'un programme est en général de quelques millisecondes mais lorsque la volatilité est élevée sur les marchés, ce laps de temps peut être suffisant pour constater une variation des prix plus grande que votre tolérance au slippage. Pour remédier à ce problème éventuel, il existe la fonction *RefreshRates()* que nous avons vu dans les premiers chapitres et qui permet de rafraîchir la valeur des variables.

La syntaxe de la fonction est la suivante :

```
bool RefreshRates()
```

Pour être utile, cette fonction doit donc être placée avant une exécution de commande dans laquelle il y aurait une ou des variables prédéfinies.

```
RefreshRates();
OrderSend(Symbol(), OP_BUY, Ask, 3, 0, 0, "Achat", 0, 0, Blue);
```

Pour éviter ce problème, vous pouvez, au lieu de la variable prédéfinie *Ask*/*Bid*, utiliser la fonction équivalente avec *MarketInfo()*.

Enfin, nous allons maintenant voir comment identifier les erreurs lors de l'exécution d'ordres via les fonctions *OrderSend()*, *OrderModify(), OderClose()* et *OrderDelete()*. Nous nous concentrons sur ces fonctions car ce sont les erreurs sur ces fonctions qui risquent d'affecter vos profits ou vos pertes.

Prenons par exemple une instruction utilisant la fonction *OrderSend()*.

```
OrderSend(Symbol(), OP_BUY, Ask, 3, 0, 0, "Achat", 0, 0, Blue);
```

Comme nous l'avons vu précédemment, la fonction *OrderSend()* est de type *int*. En cas de succès, elle renvoie le numéro de ticket de l'ordre et -1 en cas d'échec.

La première étape pour détecter une erreur est donc de vérifier la valeur renvoyée par la fonction. Nous allons donc créer une variable int *iTicket* pour stocker la valeur.

```
int iTicket = OrderSend(Symbol(), OP_BUY, Ask, 3, 0, 0, "Achat", 0, 0, Blue);
```

Nous devons maintenant vérifier si la valeur de la variable nouvellement créée est égale à -1. Pour cela, nous allons utiliser une instruction conditionnelle de type *if*. S'il s'avère que la valeur est -1, il nous faut alors identifier l'erreur en utilisant la fonction *int GetLastError()*. Nous stockerons la valeur renvoyée par cette fonction dans une nouvelle variable *int iErreur*.

Nous devons maintenant afficher l'erreur à l'utilisateur. Nous pourrions afficher la valeur de la variable *iErreur* mais ce ne serait pas très instructif étant donné qu'il nous faudrait aller chercher la traduction du code numérique de l'erreur dans l'aide de MetaTrader. Heureusement pour nous, MetaTrader est très bien conçu et dispose d'une fonction chargée de traduire en chaîne de caractères les codes numériques des différentes erreurs.

Pour traduire la valeur numérique de l'erreur en chaîne de caractères, nous allons utiliser la fonction *string ErrorDescription()*. Nous stockerons la valeur de cette erreur dans une variable de type string : *string sErreur*.

Comme les descriptions des erreurs ne sont pas essentielles au bon fonctionnement de MetaTrader, elles ne sont pas stockées directement dans le code source de MetaTrader. Afin d'utiliser la fonction *string ErrorDescription()*, il faudra ajouter en début de programme la directive *#include* appelant la librairie *stdlib.mqh*.

```
#include <stdlib.mqh>

int iTicket = OrderSend(Symbol(), OP_BUY, Ask, 3, 0, 0, "Achat", 0, 0, Blue);

if(iTicket == -1)
  {
  int iErreur = GetLastError();
  string sErreur = ErrorDescription(iErreur);
  }
```

La fonction ci-dessus fonctionne uniquement pour la fonction de passage d'ordre *int OrderSend()* car elle fait appel au numéro de ticket. Pour être en mesure de repérer les erreurs sur les autres fonctions comme *bool OrderClose()*, *bool OrderModify()*, *bool OrderDelete()*, etc., vous devez tester la valeur booléenne renvoyée. Pour ce faire, le code serait le suivant :

```
#include <stdlib.mqh>

bool bErreur = OrderModify(OrderTicket(), OrderOpenPrice(), dStopSuiveurSell, OrderTakeProfit(), 0) ;

if(bErreur == 0)
  {
  int iErreur = GetLastError();
  string sErreur = ErrorDescription(iErreur);
  }
```

Vous trouverez la liste complète des codes d'identification d'erreurs et la description correspondante à l'annexe A à la fin de ce livre.

Pour certaines erreurs, le problème n'est parfois que temporaire et il suffit de retenter de placer un ordre quelques secondes après pour que le problème soit arrangé.

Entre autres, il peut s'agir du fait que le canal de passage d'ordre est occupé, que les prix ont changé ou que le serveur a mis trop de temps à répondre.

Pour retenter automatiquement de placer un ordre, vous pouvez utiliser une boucle de type *while* comme l'illustre l'exemple suivant.

```
int iMaxTentatives = 3;
int iTentatives;
int iTicket;


while(iTicket <= 0)
  {
  iTicket = OrderSend(Symbol(), OP_BUY, Ask, 3, 0, 0, "Achat", 0, 0, Blue);

  if(iTentatives <= iMaxTentatives)
    iTentatives = iTentatives + 1;
  else
    break;
  }
```

Un possible déroulement de l'exécution du programme est résumé dans le tableau suivant :

	Avant la boucle	Après la boucle
1ère exécution	iTicket = 0 iTentatives = 0	iTicket = -1 (supposition) iTentatives = 1
2ème exécution	iTicket = -1 iTentatives = 1	iTicket = -1 (supposition) iTentatives = 2
3ème exécution	iTicket = -1 iTentatives = 2	iTicket = 324324 (supposition) iTentatives = 3

Tableau 24 : Boucle while pour passage d'ordre automatique

Le programme tentera de placer l'ordre 3 fois de suite tant et aussi longtemps que le numéro de ticket est égal ou inférieur à 3.

Ce code n'étant qu'une portion, il ne sert qu'à illustrer une possible utilisation de la fonction *while*. Si vous utilisiez cette méthode dans un programme, il serait important de réinitialiser la variable *iTicket* à 0 après la boucle ou votre programme ne pourrait plus placer d'ordre car lorsque le programme sort de la boucle, il est fort probable que la variable *iTicket* stocke le numéro de ticket de la position.

25
Fonctions communes

Les fonctions communes sont des fonctions permettant d'effectuer une action spécifique mais qui ne sont pas directement nécessaires au bon fonctionnement de vos programmes. Nous avons déjà vu dans les chapitres précédents certaines de ces fonctions telles que *MarketInfo()* et *Sleep()*.

Fonction void Alert()

La fonction *Alert()* permet d'afficher une boîte de dialogue contenant les données de votre choix comme dans la figure ci-dessous.

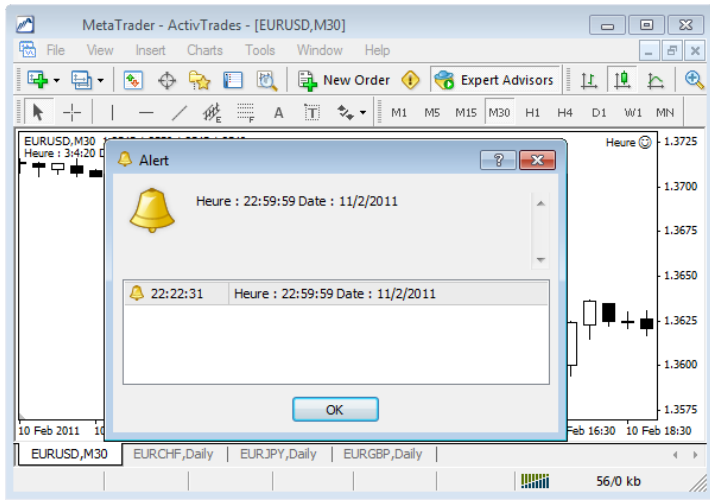

Figure 30 : Boîte de dialogue avec la fonction Alert()

Les données à afficher peuvent être de tout type mais retenez néanmoins les indications suivantes :

1. La boîte de dialogue ne peut pas contenir les données provenant d'un tableau (*array*). En MQL4, un tableau est un objet permettant de stocker une collection de données. Contrairement aux variables et constantes qui ne contiendront qu'une donnée, les tableaux en contiendront plusieurs.
2. Les données de type *double* ne seront affichées qu'avec une précision de 4 décimales après le point. Si vous désirez afficher plus de décimales, il vous faudra convertir la donnée de type *double* en chaîne de caractères à l'aide de la fonction *DoubleToStr()*.
3. Les données de type *datetime* seront affichées sous la forme du nombre de secondes écoulées depuis le 1er janvier 1970. Pour éviter cela et afficher ces données dans un format plus lisible, utilisez la fonction *TimeToStr()*.
4. Les données de type *bool* et *color* seront affichées sous leur forme numérique.

Vous pouvez afficher un maximum de 64 variables ou constantes avec cette fonction. Autrement dit, votre fonction ne pourra avoir que 64 paramètres. Chaque paramètre est séparé par une virgule ou un signe +.

Voici le code qui a permis d'afficher la boîte de dialogue de la figure 30 :

```
Alert("Heure : "+Hour()+":"+Minute()+":"+Seconds()+" "+"Date : "+Day()+"/"+Month()+"/"+Year());
```

Fonction void Comment()

La fonction *Comment()* vous permet d'afficher des données dans le coin gauche du graphique sur lequel l'expert est placé comme dans la figure ci-dessous.

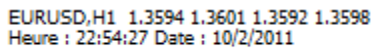

Figure 31 : Exemple de la fonction Comment()

Les mêmes restrictions ou astuces existantes pour la fonction *Alert()* le sont également pour cette fonction-ci. Le code ci-dessous permet d'afficher les informations comme sur la figure 31.

```
Comment("Heure : "+Hour()+":"+Minute()+":"+Seconds()+" "+"Date : "+Day()+"/"+Month()+"/"+Year());
```

Si vous désirez afficher les informations sur plusieurs lignes, il suffit d'insérer le caractère spécial *n*. N'oubliez pas les guillemets car il s'agit d'un caractère.

Pour afficher l'information précédente sur deux lignes, le code serait donc (*n* en gras):

```
Comment("Heure : "+Hour()+":"+Minute()+":"+Seconds()+"\n"+"Date : "+Day()+"/"+Month()+"/"+Year());
```

Fonction int GetTickCount()

Cette fonction permet d'obtenir le nombre de millisecondes écoulées depuis le démarrage de l'ordinateur jusqu'à un maximum de 49.7 jours. Vous pouvez utiliser cette fonction pour savoir depuis combien de temps l'expert est en train de fonctionner grâce au code ci-dessous.

```
int iLancement = GetTickCount();

Print("Temps de fonctionnement : " + GetTickCount() – iLancement + " millisecondes.");
```

Supposons que vous placiez le code *int iLancement = GetTickCount()* dans la fonction *init()* et le reste du code dans la fonction *start()*, le calcul effectué permettra de savoir depuis combien de millisecondes l'expert fonctionne.

Fonction double MarketInfo()

La fonction *MarketInfo()* ayant déjà été traitée dans un chapitre précédent, référez-vous à la page 114 pour plus d'informations.

Fonction int MessageBox()

La fonction *Alert()* permet d'afficher une boîte de dialogue pour vous informer simplement tandis que la fonction *MessageBox()* affiche une boîte de dialogue permettant d'interagir avec l'utilisateur. Voici ci-dessous une capture d'écran pour vous donner une idée de ce que la fonction permet de produire.

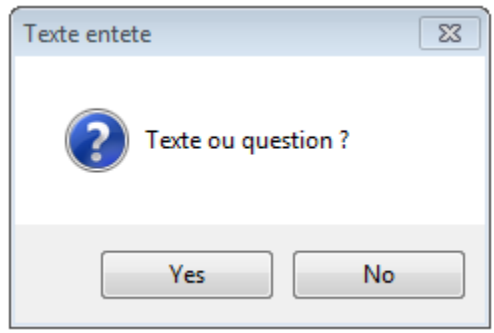

Figure 32 : Exemple de boîte de dialogue

La syntaxe de la fonction est la suivante :

```
int MessageBox(string sTexte = NULL, string sLegende = NULL, int  iAction = EMPTY)

// string sTexte : texte que vous désirez afficher dans la boîte de dialogue.

// string sLegende : texte que vous désirez afficher dans l'entête de la boîte – si le paramètre est laissé vide,
// le nom de l'expert sera affiché.

// int iAction : permet de définir les boutons et paramètres de la boîte de dialogue
```

Les options disponibles pour les boutons sont résumées dans le tableau suivant :

Constante	Description
MB_OK	Affiche un unique bouton : OK
MN_OKCANCEL	Affiche deux boutons : OK et CANCEL
MB_ABORTRETRYIGNORE	Affiche trois boutons : ABORT, RETRY et CANCEL
MB_YESNOCANCEL	Affiche trois boutons : YES, NO et CANCEL
MB_YESNO	Affiche deux boutons : YES et NO

MB_RETRYCANCEL	Affiche deux boutons : RETRY et CANCEL
MB_CANCELTRYCONTINUE	Affiche trois boutons : CANCEL, TRY AGAIN et CONTINUE

Tableau 25 : Choix de boutons pour la fonction MessageBox()

Comme on peut le voir dans la figure 32, il est également possible d'ajouter une icône dans la boîte de dialogue (un point d'interrogation dans notre exemple). Les différentes options sont résumées dans le tableau suivant :

Constante	Description
MB_ICONSTOP	
MN_ICONQUESTION	
MB_ICONEXCLAMATION	
MB_ICONINFORMATION	

Tableau 26 : Choix d'icônes pour la fonction MessageBox()

Finalement, vous pouvez également paramétrer certaines options de la boîte de dialogue comme choisir lequel des boutons disponibles devra être sélectionné par défaut ou si la boîte de dialogue doit rester au premier plan. La liste non exhaustive de ces paramètres est résumée ci-dessous. Comme la fonction fait appel à des fonctionnalités de windows (affichage d'une boîte de dialogue), vous devrez faire appel à la librairie de windows *WinUser32.mqh* via la directive *#include*. Par conséquent, pour connaître tous les paramètres disponibles, il vous faudra regarder l'aide de windows.

Constante	Description
MB_DEFBUTTON1	Le bouton 1 est le bouton par défaut
MB_DEFBUTTON2	Le bouton 2 est le bouton par défaut
MB_DEFBUTTON3	Le bouton 3 est le bouton par défaut
MB_DEFBUTTON4	Le bouton 4 est le bouton par défaut
MB_TOPMOST	La boîte de dialogue apparaîtra en premier plan de l'écran

Tableau 27 : Paramètres disponibles pour la boîte de dialogue

Analysons maintenant le code qui a permis de créer la boîte dialogue de la figure 32.

```
#include <WinUser32.mqh>  // Insérer en début de programme

if(bCondition == False) // Condition qui fera s'afficher notre boîte de dialogue
  {
 int iReponse  = MessageBox("Texte ou question ?", "Texte entete", MB_YESNO|MB_ICONQUESTION);
 if(iReponse == IDYES)
   // Action à effectuer si l'utilisateur clique sur le bouton YES
 }
```

La variable *iReponse* est en charge de récupérer la réponse ou choix de l'utilisateur. Afin d'être en mesure d'interpréter la réponse, MetaTrader propose les différentes possibilités suivantes :

Constante	Valeur numérique	Description
IDOK	1	L'utilisateur a sélectionné le bouton OK
IDCANCEL	2	L'utilisateur a sélectionné le bouton Cancel
IDABORT	3	L'utilisateur a sélectionné le bouton Abort
IDRETRY	4	L'utilisateur a sélectionné le bouton Retry
IDIGNORE	5	L'utilisateur a sélectionné le bouton Ignore
IDYES	6	L'utilisateur a sélectionné le bouton Yes
IDNO	7	L'utilisateur a sélectionné le bouton No
IDTRYAGAIN	10	L'utilisateur a sélectionné le bouton Try Again
IDCONTINUE	11	L'utilisateur a sélectionné le bouton Continue

Tableau 28 : Réponses possibles d'une boîte de dialogue

Si une boîte de dialogue comprend un bouton *Cancel*, l'utilisateur peut soit cliquer sur le bouton, soit faire usage de la touche *Échap* de son clavier. Le programme interprétera les deux actions comme une réponse de type *Cancel*.

Dans le cas du code précédent, notre programme est censé analyser si la réponse de l'utilisateur est *Yes*. Si l'utilisateur choisit *No* comme réponse, le programme va suivre son cours normal sans effectuer d'action particulière.

Un exemple concret d'utilisation de cette fonction pourrait être de demander à l'utilisateur, lorsque son choix de volume de positions est incorrect (trop grand ou trop petit), s'il désire que le programme change ces valeurs ou s'il désire arrêter l'exécution du programme. Le code serait donc :

```
#include <WinUser32.mqh>  // Insérer en début de programme

extern double dLots = 0.0001;

inti nit()
 {
 if(dLots < MarketInfo(Symbol(),MODE_MINLOT) || dLots > MarketInfo(Symbol(),MODE_MAXLOT))
   {
   int iReponse  = MessageBox("Taille de lots incorrecte – souhaitez-vous que le programme modifie le
   volume automatiquement ? Si vous répondez NON, l\'exécution du programme sera stoppée.", "Volume
   incorrect", MB_YESNO|MB_ICONSTOP);

   if(iReponse == IDYES)
    {
    if(dLots < MarketInfo(Symbol(),MODE_MINLOT))
      dLots = MarketInfo(Symbol(),MODE_MINLOT);
    if(dLots > MarketInfo(Symbol(),MODE_MAXLOT))
      dLots = MarketInfo(Symbol(),MODE_MAXLOT);
    }
   if(iReponse == IDNO)
```

```
    PostMessageA(WindowHandle( Symbol(), Period()), WM_COMMAND, 33050, 0);
  }
}
```

Le code ci-dessus s'exécutera au lancement d'un expert. Dans notre exemple, l'utilisateur a paramétré le volume à 0.0001 Lot, ce qui est bien évidemment impossible et déclenchera l'affichage de la boîte de dialogue. Si l'utilisateur clique sur le bouton *Yes*, le programme vérifiera si le volume est trop petit ou trop grand et modifiera celui-ci en conséquence en utilisant les paramètres du courtier. Si l'utilisateur clique sur le bouton *No*, l'expert sera arrêté et retiré du graphique. Le code permettant le retrait de l'expert fait appel à des fonctionnalités de Windows et nécessite l'autorisation d'utiliser des Dlls par l'expert. Ce code ne faisant pas directement partie du langage MQL4, nous ne l'expliquerons donc pas ici mais retenez simplement qu'il fonctionne et peut donc être inséré dans vos codes tel quel.

Vous trouvez ci-dessous la capture d'écran de la boîte de dialogue provenant de notre exemple :

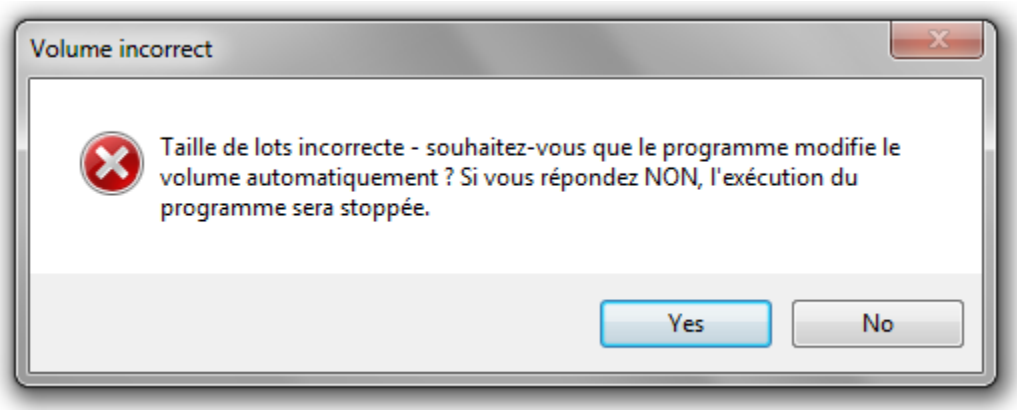

Figure 33: Boîte de dialogue pour modifier le volume des positions

Fonction void PlaySound()

La fonction *PlaySound()* permet de lancer la lecture d'un fichier son disponible dans le dossier *sounds* de l'installation de votre plateforme (par exemple C:\MetaTrader 4\sounds\).

La syntaxe de la fonction est :

```
void PlaySound(string sFichierSon)

// string sFichierSon : nom du fichier son incluant son extension.
```

Supposons que vous désiriez faire en sorte que le fichier alerte.wav soit lu, le code serait donc si le fichier se trouve dans le dossier *sounds* :

```
PlaySound("alerte.wav ");
```

Fonction void Print()

La fonction *Print()* permet d'afficher des informations dans l'onglet journal de la fenêtre Terminal de votre plateforme. La syntaxe de la fonction est la suivante :

```
void Print()
```

Les données que vous souhaitez afficher sont à insérer entre parenthèses.

Vous pouvez afficher un maximum de 64 variables ou constantes avec cette fonction. Autrement dit, votre fonction ne pourra avoir que 64 paramètres. Chaque paramètre est séparé par une virgule ou par un signe +.

Les remarques de la page 134 sont également applicables pour cette fonction-ci.

Voici un exemple de code suivi du résultat dans l'onglet journal.

```
Print("Bienvenue! Votre expert est maintenant opérationnel");
```

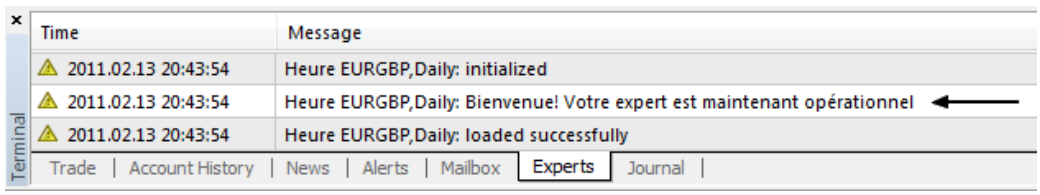

Figure 34 : Exemple d'utilisation de la fonction Print()

Fonction bool SendFTP()

La fonction *SendFTP()* comme son nom l'indique permet d'envoyer des fichiers en utilisant le protocole FTP. La syntaxe de la fonction est la suivante :

```
bool SendFTP(string sNomFichier, string sCheminFTP = NULL)

// string sNomFichier : nom du fichier à envoyer en incluant l'extension.

// string sChemingFTP : adresse ftp où vous désirez envoyer les fichiers.
```

Le fichier à envoyer doit se situer dans le dossier */experts/files/* de votre installation de MetaTrader.

L'adresse du serveur FTP est par défaut laissée en blanc dans cette fonction car la fonction va aller chercher ces informations (adresse IP du serveur, chemin d'accès, nom d'utilisateur et mot de passe) dans les paramètres de la plateforme. Pour accéder à ces paramètres, il vous suffit d'aller, dans la plateforme, dans *Tools > Options > Publisher*.

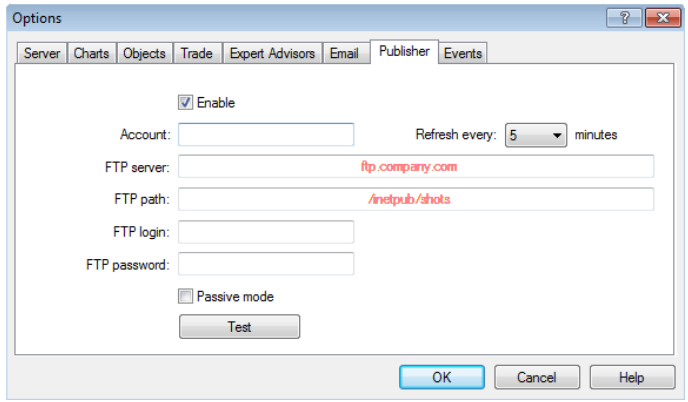

Figure 35 : Paramétrage du serveur FTP

Un exemple d'envoi de fichier pourrait être :

```
SendFTP("fichier.txt");

// le programme utilisera les paramètres du serveur FTP présent dans l'onglet Publisher.
```

Nous verrons dans un prochain chapitre comment faire pour créer, modifier, effacer des fichiers mais également comment se connecter à un serveur pour récupérer un fichier ou envoyer un fichier sans passer par la fonction *SendFTP()* qui reste très limitée.

Fonction void SendMail()

La fonction *SendMail()* permet d'envoyer un email à l'adresse spécifiée dans l'onglet *Email* disponible en allant dans *Tools > Options*.

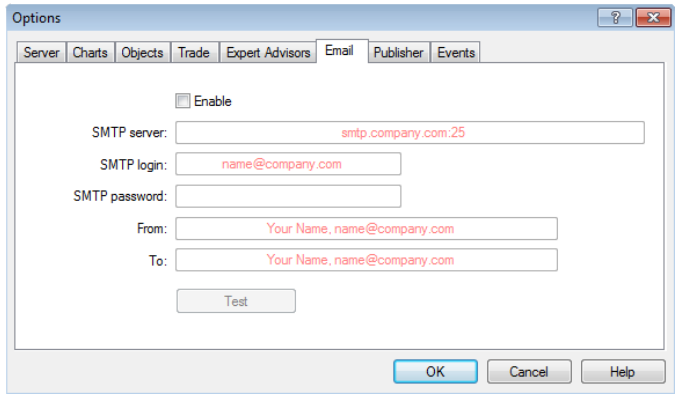

Figure 36 : Onglet Email des options de MetaTrader

La syntaxe de la fonction est :

```
void SendMail(string sSujet, string sTexte)

// string sSujet : sujet du courrier électronique.

// string sTexte : texte du message.
```

Voici un exemple d'utilisation possible :

```
int iTicket = OrderSend("EURUSD", OP_BUY, 1, Ask, 3, 0, 0, " ", 0, 0, Blue);


if(iTicket != -1)
  {
  OrderSelect(iTicket, SELECT_BY_TICKET);
  SendMail("Position confirmée", "Une position de "+OderLots()+" de type "+OrderType()+" a été ouverte au
  prix de "+OrderOpenPrice());
  }
```

Le code ci-dessus envoi par email les informations concernant une prise de position en précisant le type de position, son volume et le prix d'entrée.

Fonction void Sleep()

La fonction *Sleep()* a déjà été vu dans un chapitre précédent, nous allons donc simplement résumer sa fonction, syntaxe et voir un exemple.

Cette fonction permet de suspendre temporairement l'exécution de votre programme pendant un délai paramétré par l'utilisateur. La syntaxe de la fonction est la suivante :

```
void Sleep()

// Le délai de suspension doit être précisé entre les parenthèses en millisecondes.
```

Un exemple d'utilisation que nous avions déjà vu auparavant :

```
while(IsTradeContextBusy() == True)
   Sleep(1000);
```

Le code ci-dessus suspend l'exécution du programme pendant 1 seconde (1000 millisecondes = 1 seconde) lorsque le canal de trading est occupé.

26
Fonctions de vérification

Nous avons déjà vu dans les chapitres précédents les fonctions *GetLastError()* et *IsTradeContextBusy()*. Ces deux fonctions font partie du groupe des fonctions de vérification. Comme le nom l'indique, ce type de fonctions vérifie un statut et renvoie son état actuel. Nous allons maintenant voir la syntaxe et l'usage de chacune des fonctions appartenant à ce groupe.

Fonction int GetLastError()

Nous ne reviendrons pas sur cette fonction dans ce chapitre. Veuillez vous référer au chapitre 24 pour plus d'informations au sujet de *GetLastError()*.

Fonction bool IsConnected()

Cette fonction permet de vérifier le statut de la connexion entre le serveur du courtier et la plateforme. La fonction renvoie *True* lorsque la connexion est établie et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool IsConnected()
```

Vous pouvez utiliser cette fonction au début de votre programme pour vérifier que la connexion est bien établie et éviter ainsi une éventuelle inutile exécution de votre programme.

```
if( IsConnected() == False)
  {
  Print("Connexion avec le serveur non établie ");
  return(0);
  }
```

Fonction bool IsDemo()

Cette fonction permet de vérifier si le compte de la plateforme sur laquelle le programme est exécuté est un compte de démonstration (argent fictif) ou un compte réel. La fonction renvoie *True* lorsque le compte est un compte de démonstration et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool IsDemo()
```

Vous pouvez utiliser cette fonction au début de votre programme pour vérifier le type de compte et avertir l'utilisateur sur les risques liées à la spéculation.

```
if(IsDemo() == False)
  {
  int iReponse  = MessageBox("Attention, la spéculation sur le marché des changes peut présenter des
  risques de pertes….. Souhaitez-vous vraiment continuer ?", "Avertissement",
  MB_YESNO|MB_ICONSTOP);

  if(iReponse == IDNO)
    {
    Print("Arrêt de l'exécution du programme ");
    PostMessageA( WindowHandle( Symbol(), Period()), WM_COMMAND, 33050, 0);
    }
  }
```

Le code ci-dessus vérifie le type de compte en premier lieu. Si le compte est un compte réel, une fenêtre s'affichera avec un avertissement sur les risques et demande à l'utilisateur s'il désire tout de même utiliser le programme. Si la réponse est oui, le programme continuera son cours comme si de rien n'était. Si la réponse est non, un message s'affichera indiquant l'arrêt de l'exécution du programme et l'expert sera arrêté et retiré du graphique.

Fonction bool IsDllsAllowed()

Cette fonction permet de vérifier si l'utilisation de DLL est autorisé ou non. La fonction renvoie *True* lorsque le programme a le droit d'appeler des DLLs et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool IsDllsAllowed()
```

L'unique moyen pour autoriser l'appel de DLL est manuellement via la fenêtre de propriétés de votre programme si vous désirez activer la fonctionnalité uniquement pour un programme ou via le menu Tools => Options => Expert Advisors et où vous devrez cocher la case « *Allow DLL Imports* ».

```
if( IsDllsAllowed() == False)
  {
  Alert("L'expert n'a pas l'autorisation d'appeler des DLLs! ");
  }
```

Fonction bool IsExpertEnabled()

Cette fonction permet de vérifier si les experts consultants sont activés ou non sur la plateforme. Autrement dit si le bouton Expert Advisors de la barre d'outils est activé ou non.

Figure 37 : Bouton Expert Advisors (désactivé à gauche et activé à droite)

La fonction renvoie *True* lorsque les experts sont activés et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool IsExpertEnabled()
```

Vous pouvez utiliser cette fonction au début de votre programme pour vérifier que l'utilisation d'experts consultants est bien activée. Dans le cas contraire, une fenêtre s'affichera pour informer l'utilisateur.

```
if(IsExpertEnabled () == False)
 {
 Alert("Vous devez cliquer sur le bouton Expert Advisors pour activer l'utilisation d'experts consultants ");
 return(0);
 }
```

Fonction bool IsLibrariesAllowed()

Cette fonction permet de vérifier si les experts consultants ont le droit d'appeler des librairies. La fonction renvoie *True* lorsque les experts ont le droit et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool IsLibrariesAllowed()
```

Vous pouvez utiliser cette fonction au début de votre programme lorsque celui fait appel ;a des librairies pour s'assurer que la plateforme sur laquelle tourne le programme autorise l'expert à faire appel à ces librairies.

```
if(IsLibrariesAllowed () == False)
 {
 Alert("Votre expert n'a pas le droit d'appeler des librairies. ");
 return(0);
 }
```

Fonction bool IsTesting()

Cette fonction permet de vérifier si votre expert est en train d'être exécuté dans le testeur de stratégies de MetaTrader. La fonction renvoie *True* lorsque l'expert est exécuté dans le testeur de stratégies et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool IsTesting ()
```

Cette fonction peut être très utile lorsque votre programme est complexe et contient des fonctions qui ne sont utiles que lorsque l'expert est exécuté en temps réel.

```
if(IsTesting () == False)
 {
 // Inclure ici les fonctions que vous ne voulez pas voir exécuter lors du test de la stratégie dans le testeur.
 }
```

Fonction bool IsOptimization()

Cette fonction permet de vérifier si votre expert est en train d'être exécuté dans le testeur de stratégies de MetaTrader et si vous avez choisi d'optimiser la stratégie. La fonction renvoie *True* lorsque l'expert est exécuté dans le testeur de stratégies et en train d'être optimisé et *False* dans le cas contraire.

Vous pouvez tester une stratégie sans l'optimiser mais vous ne pouvez pas optimiser une stratégie sans la tester.

La syntaxe de la fonction est la suivante :

```
bool IsOptimization()
```

De la même façon que pour *IsTesting()*, cette fonction permet de trier les fonctions ou commandes que vous désirez voir ou non exécuter lorsque l'expert est en train d'être testé et optimisé.

```
if(IsOptimization () == True)
  {
  // Inclure ici les fonctions que vous voulez voir exécuter lors de l'optimisation.
  }
```

Fonction bool IsVisualMode()

Cette fonction permet de vérifier si votre expert est en train d'être exécuté dans le testeur de stratégie avec l'option de visualisation activée ou non. La fonction renvoie *True* lorsque l'option est activée et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool IsVisualMode()
```

Un exemple pourrait être :

```
if(IsVisualMode() == True)
  {
  Print("Option de visualisation activée.");
  }
```

Fonction bool IsStopped()

Cette fonction permet de vérifier si votre expert est stoppé suite à une instruction lui indiquant de le faire. La fonction renvoie *True* lorsque l'expert est stoppé et *False* dans le cas contraire.

La syntaxe de la fonction est la suivante :

```
bool IsStopped()
```

Cette fonction peut être utilisée pour limiter les risques liés à une boucle infinie.

```
while(!IsStopped())
  {
  // Inclure ici les instructions pour la boucle infinie.
  }
```

Fonction bool IsTradeAllowed()

Cette fonction permet de vérifier si votre expert a le droit de placer des ordres ou non. La fonction renvoie *True* lorsque l'expert a le droit et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool IsTradeAllowed()
```

L'unique moyen pour autoriser les experts à placer des ordres est manuellement via la fenêtre de propriétés de votre programme si vous désirez activer la fonctionnalité uniquement pour un programme ou via le menu *Tools* => *Options* => *Expert Advisors* et où vous devrez cocher la case *Allow live trading*.

```
if(isTradeAllowed() == False)
  {
  Comment("L'expert  n'a pas le droit de placer des ordres!");
  }
```

bool IsTradeContextBusy()

Nous ne reviendrons pas sur cette fonction dans ce chapitre. Veuillez vous référer au chapitre 24 pour plus d'informations au sujet de *IsTradeContextBusy()*.

27
Votre premier expert consultant #6

Nous allons maintenant insérer les différents codes que nous avons élaboré au cours des derniers chapitres dans notre programme principal.

En premier lieu, nous allons ajouter la fonction de gestion du risque pour que notre expert prenne des positions en calculant automatiquement le volume de la position en fonction du risque maximal que l'utilisateur définira.

La fonction que nous avions créée et que nous pouvons copier-coller directement dans notre code est la suivante :

```
double fLots(double dPourcentageRisque, int iStop)
 {
 double dRisqueMax = (dPourcentageRisque/100) * AccountBalance();
 double dValeurPip = MarketInfo(Symbol(), MODE_TICKVALUE);
 if(Point == 0.001 || Point == 0.00001)
  {
  dValeurPip = dValeurPip * 10;
  }
 double dLots = (dRisqueMax / iStop) / dValeurPip;
 if(dLots < MarketInfo(Symbol(),MODE_MINLOT))
  dLots = MarketInfo(Symbol(),MODE_MINLOT);
 if(dLots > MarketInfo(Symbol(),MODE_MAXLOT))
  dLots = MarketInfo(Symbol(),MODE_MAXLOT);
 if(MarketInfo(Symbol(),MODE_LOTSTEP) == 0.01)
  dLots = NormalizeDouble(dLots, 2);
 else
  dLots = NormalizeDouble(dLots, 1);
 return(dLots);
 }
```

En plus d'ajouter la fonction à la fin du programme, nous devons définir la variable *double dPourcentageRisque* en début de programme.

```
extern double dPourcentageRisque = 5;
```

Il ne reste plus qu'à insérer la fonction dans nos deux fonctions de passage d'ordre.

```
iTicket = OrderSend(Symbol(), OP_BUYSTOP, fLots(dPourcentageRisque, iStop), dPlusHaut + iDistance
* fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la veille",
iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0])) * 3600) + ((iMinutesFin - TimeMinute(Time[0])) * 60)),
Blue);

iTicket = OrderSend(Symbol(), OP_SELLSTOP, fLots(dPourcentageRisque, iStop), dPlusBas - iDistance *
fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la veille",
iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0])) * 3600)+((iMinutesFin - TimeMinute(Time[0])) * 60)),
Red);
```

La fonction calculera donc le volume de chaque position en utilisant le pourcentage de risque défini par la variable *extern double dPourcentageRisque* et la variable *extern int iStop*.

Le deuxième ajout est en rapport avec la gestion des erreurs et plus précisément le code que nous avions vu pour vérifier si le canal de trading est occupé ou non.

```
while(IsTradeContextBusy() == True)
  Sleep(1000);
```

Nous devons ajouter le code ci-dessus avant chacune de nos fonctions de passage d'ordre.

```
while(IsTradeContextBusy() == True)
  Sleep(1000);
iTicket = OrderSend(Symbol(), OP_BUYSTOP, fLots(dPourcentageRisque, iStop), dPlusHaut + iDistance *
fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la veille",
iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0])) * 3600) + ((iMinutesFin - TimeMinute(Time[0])) * 60)),
Blue);

while(IsTradeContextBusy() == True)
  Sleep(1000);
iTicket = OrderSend(Symbol(), OP_SELLSTOP, fLots(dPourcentageRisque, iStop), dPlusBas - iDistance *
fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la veille",
iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0])) * 3600)+((iMinutesFin - TimeMinute(Time[0])) * 60)),
Red);
```

Le dernier ajout concerne la détection et l'affichage des éventuelles erreurs qui pourraient survenir lors du passage d'un ordre. Pour ce faire, nous allons utiliser le code que nous avions étudié auparavant.

```
if(iTicket == -1)
 {
 int iErreur = GetLastError();
 string sErreur = ErrorDescription(iErreur);
 }
```

Comme nous l'avions également vu, il est nécessaire, afin de pouvoir utiliser la fonction *ErrorDescription(),* d'appeler la librairie *stdlib.mqh* à l'aide d'une directive *#include*. Nous allons donc ajouter le code ci-dessous dans l'entête de notre programme.

```
#include <stdlib.mqh>
```

Le code de détection n'est pour l'instant capable que de détecter les erreurs, nous devons ajouter une instruction permettant d'afficher l'erreur et sa description afin que nous puissions en prendre connaissance. Nous allons pour cela utiliser la fonction *Print()* que nous avons vue dans le chapitre précédent qui nous permet d'afficher les données de notre choix dans le journal de la fenêtre Terminal de notre plateforme. Pour simplifier l'utilisation de la fonction, nous allons en même temps créer une fonction qui stockera notre code.

```
void fErreur(string sPosition)
 {
 if(iTicket == -1)
   {
   int iErreur = GetLastError();
   string sErreur = ErrorDescription(iErreur);
   Print("Erreur position "+sPosition+" : "+iErreur+" | "+sErreur);
   }
 }
```

La fonction *fErreur()* fonctionne comme ceci : si le numéro de ticket est égal à -1, signifiant qu'il y a eu une erreur lors du passage d'ordre, le programme va d'abord stocker le numéro de l'erreur obtenu à l'aide de la fonction *GetLastError()* dans la variable *iErreur*. Ensuite, le programme ira chercher la description de l'erreur à l'aide de la fonction *ErrorDescription()* en utilisant le numéro obtenu précédemment et stockera le tout dans la variable *sErreur*.

La dernière étape consiste à afficher les informations recueillies grâce à la fonction *Print()*. La variable *sPosition* permet de définir un texte pour identifier de quel type d'opération il s'agit.

Si la fonction est appelée, le résultat obtenu serait similaire à la capture ci-dessous. Chaque élément a été identifié afin que vous puissiez voir sa correspondance avec le code.

Figure 38 : Description d'erreur dans le journal

Il ne reste plus qu'à ajouter un appel de fonction après chaque fonction de passage d'ordre en remplaçant le paramètre de la fonction (la variable *string sPosition*) par un mot clé pour que l'utilisateur puisse identifier de quelle position il s'agit. Dans le code ci-dessous, nous avons choisi *Achat* et *Vente*.

```
while(IsTradeContextBusy() == True)
   Sleep(1000);
iTicket = OrderSend(Symbol(), OP_BUYSTOP, fLots(dPourcentageRisque, iStop), dPlusHaut + iDistance *
fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la veille",
iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0])) * 3600) + ((iMinutesFin - TimeMinute(Time[0])) * 60)),
Blue);
fErreur("Achat");

while(IsTradeContextBusy() == True)
   Sleep(1000);
iTicket = OrderSend(Symbol(), OP_SELLSTOP, fLots(dPourcentageRisque, iStop), dPlusBas - iDistance *
fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la veille",
iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0])) * 3600)+((iMinutesFin - TimeMinute(Time[0])) * 60)),
Red);
fErreur("Vente");
```

Voici maintenant la totalité du code avec les modifications et ajouts (identifiés en gras).

```
//+------------------------------------------------------------------+
//|                          Votre Premier Expert.mq4                |
//|                          Copyright © 2011,                       |
//|                        http://www. eole-trading.com              |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www. eole-trading.com"

#include <stdlib.mqh>


extern int iHeureDebut = 9;
extern int iMinutesDebut = 30;
extern int iHeureFin = 17;
extern int iMinutesFin = 30;
extern int iDistance = 5;
```

```
extern int iMagic = 123456;
extern int iStop = 50;
extern int iLimite = 50;
extern double dPourcentageRisque = 5;

int iDate;
int iTimeDebut;
int iTimeFin;
int iTimeSeconds;
int iTicket;
double dPlusHaut;
double dPlusBas;
bool bRecuperation = False;
bool bTrade = True;


//+------------------------------------------------------+
//| expert initialization fonction                   |
//+------------------------------------------------------+
int init()
  {

  iTimeDebut = iHeureDebut * 3600 + iMinutesDebut * 60;
  iTimeFin = iHeureFin * 3600 + iMinutesFin * 60;

  return(0);
  }
//+------------------------------------------------------+
//| expert deinitialization fonction                 |
//+------------------------------------------------------+
int deinit()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------+
//| expert start fonction                            |
//+------------------------------------------------------+
int start()
  {

  iTimeSeconds = Hour()*3600 + Minute()*60 + Seconds();


  if(iTimeSeconds >= iTimeDebut && iTimeSeconds < iTimeFin)
    {
    if(bRecuperation == False)
      {
      iDate = Day();
      dPlusHaut = iHigh(NULL, PERIOD_D1, 1);
      dPlusBas = iLow(NULL, PERIOD_D1, 1);
      bRecuperation = True;
      fTracageLignes("Point plus haut", dPlusHaut, Blue);
      fTracageLignes("Point plus bas", dPlusBas, Red);
      }


    if(bTrade == True)
      {
      while(IsTradeContextBusy() == True)
        Sleep(1000);
      iTicket = OrderSend(Symbol(), OP_BUYSTOP, fLots(dPourcentageRisque, iStop), dPlusHaut +
      iDistance* fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la
      veille", iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600) + ((iMinutesFin -
      TimeMinute(Time[0]))*60)), Blue);
      fTicket(iTicket);
      fErreur("Achat");
```

```
      while(IsTradeContextBusy() == True)
       Sleep(1000);
      iTicket = OrderSend(Symbol(), OP_SELLSTOP, fLots(dPourcentageRisque, iStop),  dPlusBas -
      iDistance* fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la
      veille", iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600)+((iMinutesFin -
      TimeMinute(Time[0]))*60)), Red);
      fTicket(iTicket);
      fErreur("Vente");


      bTrade = False;
      }

    if(iDate != Day())
      {
      bRecuperation = False;
      bTrade = True;
      }

    }

  return(0);
  }
//+------------------------------------------------------+


void fTracageLignes(string sNomLigne, double dPrix, color cCouleur)
  {
  ObjectDelete(sNomLigne);
  ObjectCreate(sNomLigne,OBJ_HLINE, 0, 0, dPrix);
  ObjectSet(sNomLigne, OBJPROP_COLOR, cCouleur);
  }


double fPoint(string sPaire)
  {
  if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 3)
    {
    double dPoint = 0.01;
    }
  if((MarketInfo(sPaire, MODE_DIGITS)) == 4 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
    {
    dPoint = 0.0001;
    }
  return(dPoint);
  }


double fSlippage(string sPaire, int iPipsSlippage)
  {
  if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 4)
    {
    double dSlippage = iPipsSlippage;
    }
  if((MarketInfo(sPaire, MODE_DIGITS)) == 3 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
    {
    dSlippage = iPipsSlippage * 10;
    }
  return(dSlippage);
  }


void fTicket(int iTicket)
  {
  if(iTicket != -1)
    {
    OrderSelect(iTicket, SELECT_BY_TICKET);
    if(OrderType() == 4)
      OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() - iStop * fPoint(Symbol()),
      OrderOpenPrice() + iLimite * fPoint(Symbol()), OrderExpiration());
```

```
    if(OrderType() == 5)
      OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() + iStop * fPoint(Symbol()),
      OrderOpenPrice() - iLimite * fPoint(Symbol()), OrderExpiration());
      }
  }

double fLots(double dPourcentageRisque, int iStop)
  {
  double dRisqueMax = (dPourcentageRisque/100) * AccountBalance();
  double dValeurPip = MarketInfo(Symbol(), MODE_TICKVALUE);
  if(Point == 0.001 || Point == 0.00001)
    {
    dValeurPip = dValeurPip * 10;
    }
  double dLots = (dRisqueMax / iStop) / dValeurPip;
  if(dLots < MarketInfo(Symbol(),MODE_MINLOT))
    dLots = MarketInfo(Symbol(),MODE_MINLOT);
  if(dLots > MarketInfo(Symbol(),MODE_MAXLOT))
    dLots = MarketInfo(Symbol(),MODE_MAXLOT);
  if(MarketInfo(Symbol(),MODE_LOTSTEP) == 0.01)
    dLots = NormalizeDouble(dLots, 2);
  else
    dLots = NormalizeDouble(dLots, 1);
  return(dLots);
  }

void fErreur(string sPosition)
  {
  if(iTicket == -1)
    {
    int iErreur = GetLastError();
    string sErreur = ErrorDescription(iErreur);
    Print("Erreur position "+sPosition+" : "+iErreur+" | "+sErreur);
    }
  }
```

28
Stop suiveur et seuil de rentabilité

Parfois mieux connu sous leurs appellations anglophones, le « trailing stop » et « breakeven » sont primordiales dans une bonne gestion des risques. En effet, ils évitent de transformer des positions gagnantes en perdantes.

Stop suiveur

Un stop suiveur permet de réduire votre risque et protéger vos profits automatiquement lorsque le marché évolue dans le sens de votre position. Le niveau de stop s'ajustera automatiquement en fonction de critères prédéfinis par vos soins. Cette gestion des stops permet de réduire progressivement le risque de votre exposition au fur et à mesure que vos profits augmentent. La figure suivante illustre le fonctionnement d'un stop suiveur.

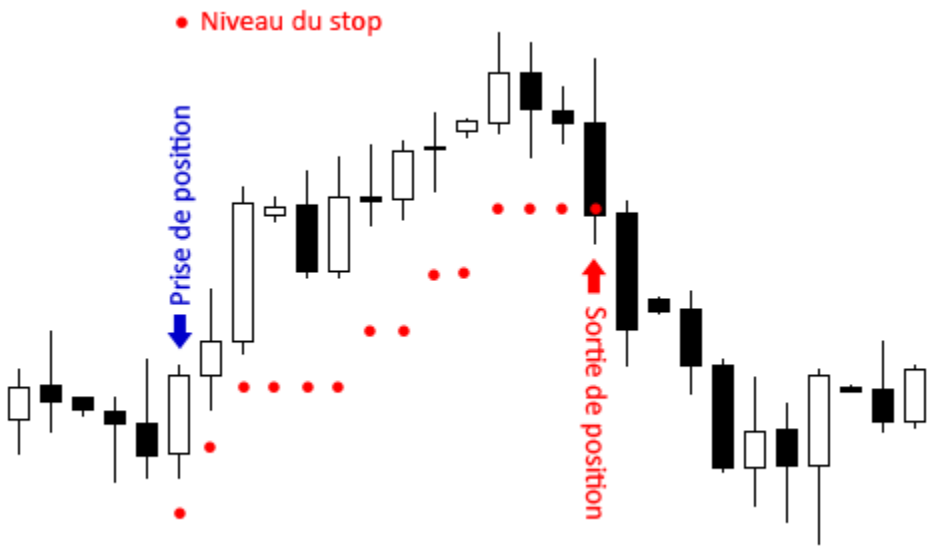

Figure 39 : Fonctionnement d'un stop suiveur

Chaque point rouge représente le niveau du stop au cours du temps. À chaque fois que le cours augmente d'une quantité de pips prédéfinis, le stop est ajusté vers le haut. Si le cours n'augmente pas suffisamment ou pas du tout, le stop est maintenu à son niveau antérieur. Vous remarquerez que, grâce au stop suiveur, cette position est clôturée avec profit ; ce qui ne serait pas le cas sans stop suiveur et sans aucune intervention humaine.

Nous allons maintenant voir comment créer une fonction pour automatiser un stop suiveur sur un expert consultant.

La première étape consiste à ajouter une variable externe permettant de stocker le nombre de pips dans le sens de la position nécessaire pour qu'il y ait une modification du stop.

```
extern int iStopSuiveur = 25;
```

Nous devons maintenant créer une fonction qui sélectionnera les positions ouvertes, vérifiera si elles ont bien été ouvertes par notre expert et enfin modifiera le stop lorsque nécessaire.

```
void fStopSuiveur (string sSymbol, int iStopSuiveur, int iMagic)
 {
 for (int iCompteur = 0; iCompteur <= OrdersTotal() – 1; iCompteur++)
  {
  OrderSelect(iCompteur, SELECT_BY_POS);

  double dStopSuiveurBuy = MarketInfo(sSymbol, MODE_BID) – (iStopSuiveur * fPoint(sSymbol));
  double dStopSuiveurSell = MarketInfo(sSymbol, MODE_ASK) + (iStopSuiveur * fPoint(sSymbol));

  if(OrderMagicNumber() == iMagic && OrderSymbol() == sSymbol  && OrderType() == OP_BUY &&
  OrderStopLoss() < dStopSuiveurBuy)
   {
   bool bErreur = OrderModify(OrderTicket(), OrderOpenPrice(), dStopSuiveurBuy, OrderTakeProfit(), 0) ;
   if(bErreur == 0)
    {
    int iErreur = GetLastError();
    string sErreur = ErrorDescription(iErreur);
    Print("Erreur modification position achat : "+iErreur+" | "+sErreur);
    }
   }

  if(OrderMagicNumber() == iMagic && OrderSymbol() == sSymbol && OrderType() == OP_SELL &&
  OrderStopLoss() > dStopSuiveurSell)
   {
   bErreur = OrderModify(OrderTicket(), OrderOpenPrice(), dStopSuiveurSell, OrderTakeProfit(), 0) ;
   if(bErreur == 0)
    {
    iErreur = GetLastError();
    sErreur = ErrorDescription(iErreur);
    Print("Erreur modification position achat : "+iErreur+" | "+sErreur);
    }
   }

  }
 }
```

Lorsque vous ouvrez une position à l'achat, l'ordre s'effectuera au prix *Ask* et au prix *Bid* dans le cas d'une position à la vente. Lorsque vous désirez clôturer ces positions, l'ordre s'effectuera au prix *Bid* pour une position à l'achat et *Ask* pour une position à la vente. C'est pour cela que notre variable *double dStopSuiveurBuy* est calculée avec le prix *Bid* et non *Ask*.

Le code ci-dessus fonctionne comme ceci : en premier lieu, le programme vérifiera combien de positions sont en cours (ouvertes ou en attente) puis les sélectionnera une à une. À l'intérieur de la boucle, nous calculerons quels seraient les nouveaux stops en fonction du niveau de stop suiveur paramétré et du prix du marché pour les deux types de positions. À partir de là, il ne reste plus qu'à vérifier si il y a effectivement une position ouverte par notre expert grâce à son numéro magique, si la paire de la position correspond bien au graphique sur lequel est placé notre expert, si la position est un achat ou une vente et enfin si le nouveau stop calculé plus haut est en notre faveur ou non. Si toutes les conditions sont réunies, le programme modifiera le stop de la position en remplaçant celui-ci par le stop calculé précédemment. En cas d'erreur

lors de la modification, le programme affichera le numéro d'identification de l'erreur et sa description dans le journal de la fenêtre Terminal de la plateforme.

Seuil de rentabilité

Si au lieu d'un stop suiveur, vous désirez simplement que le stop soit ramené au prix d'entrée (ou à un niveau paramétré) lorsqu'un profit minimum est atteint, il suffirait de modifier légèrement le code précédent comme ci-dessous. Au lieu d'indiquer le nombre de pips pour le stop suiveur, vous devrez indiquer le nombre de pips minimum de profit avant que le stop ne soit modifié.

```
extern int iSeuil = 25;
```

Nous devons maintenant créer une fonction qui sélectionnera les positions ouvertes, vérifiera si elles ont bien été ouvertes par notre expert et enfin modifiera le stop lorsque nécessaire.

```
void fSeuil (string sSymbol, int iStopSuiveur, int iMagic)
 {
 for (int iCompteur = 0; iCompteur <= OrdersTotal() – 1; iCompteur++)
   {
   OrderSelect(iCompteur, SELECT_BY_POS);

   double dSeuil = iSeuil * fPoint(sSymbol) ;

   double dProfitPipsBuy = MarketInfo(sSymbol, MODE_BID) – OrderOpenPrice() ;
   double dProfitPipsSell = OrderOpenPrice()  - MarketInfo(sSymbol, MODE_ASK);

   if(OrderMagicNumber() == iMagic && OrderSymbol() == sSymbol  && OrderType() == OP_BUY &&
   dProfitPipsBuy > dSeuil)
    {
    bool bErreur = OrderModify(OrderTicket(), OrderOpenPrice(),OrderOpenPrice(), OrderTakeProfit(), 0) ;
    if(bErreur == 0)
     {
     int iErreur = GetLastError();
     string sErreur = ErrorDescription(iErreur);
     Print("Erreur modification position achat : "+iErreur+" | "+sErreur);
     }
    }

   if(OrderMagicNumber() == iMagic && OrderSymbol() == sSymbol && OrderType() == OP_SELL &&
   dProfitPipsSell  > dSeuil)
    {
    bErreur = OrderModify(OrderTicket(), OrderOpenPrice(),OrderOpenPrice(), OrderTakeProfit(), 0) ;
    if(bErreur == 0)
     {
     iErreur = GetLastError();
     sErreur = ErrorDescription(iErreur);
     Print("Erreur modification position achat : "+iErreur+" | "+sErreur);
     }
    }

   }
 }
```

La principale différence entre le stop suiveur et le seuil de rentabilité est que dans le cas de ce dernier, il faut vérifier que le nombre de pips en faveur de la position entre le prix du marché et le prix d'entrée soit supérieur au nombre minimum de pips de profit requis par l'utilisateur.

Ces calculs sont effectués par les instructions suivantes :

```
double dSeuil = iSeuil * fPoint(sSymbol) ;

double dProfitPipsBuy = MarketInfo(sSymbol, MODE_BID) – OrderOpenPrice() ;
double dProfitPipsSell = OrderOpenPrice()  - MarketInfo(sSymbol, MODE_ASK);
```

29
Numéro magique

Dans le dernier chapitre, nous avions besoin d'identifier si une position avait bien été ouverte par notre expert ou non. C'est là le principal voire l'unique avantage du numéro magique. Compte tenu du nombre éventuellement important de transactions, l'attribution d'un numéro de ticket qui ne peut pas être connu à l'avance et de la possibilité qu'il y ait plusieurs experts exécutant des ordres sur la même paire, il est primordial de pouvoir identifier l'origine d'une position.

Lorsqu'une position est ouverte directement par l'utilisateur, MetaTrader n'attribuera pas de numéro magique à cette position.

Comme vous avez pu voir dans les exemples de code du dernier chapitre, une position peut être identifiée selon plusieurs critères :

- Numéro de ticket
- Paire de la position
- Sens de la position (Achat ou Vente)
- Volume de la position, prix d'entrée, prix limite, prix stop

Tous les critères énumérés ci-dessus à l'exception du premier qui est unique pour chaque position, peuvent être communs à plusieurs positions ouvertes par différents experts. Afin de savoir sur quelles positions un expert est censé apporter des modifications (stop suiveur, clôture ou autre), nous utilisons le numéro magique qui devra être unique à chaque expert et permet donc de grouper des positions répondant aux critères de votre choix.

Le numéro magique peut également permettre de séparer des positions au sein du même expert. De cette façon, vous pourrez contrôler chacune des positions indépendamment de l'autre.

Le numéro magique est un entier assigné à une position au moment de son ouverture. Il n'est donc pas possible de le modifier après que la position a été ouverte.

Le numéro magique peut être renvoyé par la fonction *OrderMagicNumber()* et est attribué à une position au travers de la fonction *OrderSend().*

30
Votre premier expert consultant #7

Nous allons ajouter la fonction de stop suiveur créée précédemment à notre programme. Il nous faut donc ajouter la définition de la variable permettant à l'utilisateur de définir le nombre de pips minimum pour déplacer le stop, ajouter la fonction puis un appel de fonction en début de programme.

```
//+------------------------------------------------------------------+
//|                              Votre Premier Expert.mq4            |
//|                               Copyright © 2011,                  |
//|                            http://www. eole-trading.com          |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www. eole-trading.com"

#include <stdlib.mqh>


extern int iHeureDebut = 9;
extern int iMinutesDebut = 30;
extern int iHeureFin = 17;
extern int iMinutesFin = 30;
extern int iDistance = 5;
extern int iMagic = 123456;
extern int iStop = 50;
extern int iLimite = 50;
extern double dPourcentageRisque = 5;
extern int iStopSuiveur = 25;

int iDate;
int iTimeDebut;
int iTimeFin;
int iTimeSeconds;
int iTicket;
double dPlusHaut;
double dPlusBas;
bool bRecuperation = False;
bool bTrade = True;


//+------------------------------------------------------+
//| expert initialization fonction                       |
//+------------------------------------------------------+
int init()
  {

  iTimeDebut = iHeureDebut * 3600 + iMinutesDebut * 60;
  iTimeFin = iHeureFin * 3600 + iMinutesFin * 60;

  return(0);
  }
//+------------------------------------------------------+
//| expert deinitialization fonction                     |
//+------------------------------------------------------+
int deinit()
  {
//----
```

```
//----
  return(0);
  }
//+------------------------------------------------------+
//| expert start fonction                                |
//+------------------------------------------------------+
int start()
  {

  iTimeSeconds = Hour()*3600 + Minute()*60 + Seconds();

  fStopSuiveur(Symbol(), iStopSuiveur, iMagic);


  if(iTimeSeconds >= iTimeDebut && iTimeSeconds < iTimeFin)
    {
    if(bRecuperation == False)
      {
      iDate = Day();
      dPlusHaut = iHigh(NULL, PERIOD_D1, 1);
      dPlusBas = iLow(NULL, PERIOD_D1, 1);
      bRecuperation = True;
      fTracageLignes("Point plus haut", dPlusHaut, Blue);
      fTracageLignes("Point plus bas", dPlusBas, Red);
      }


    if(bTrade == True)
      {
      while(IsTradeContextBusy() == True)
        Sleep(1000);
      iTicket = OrderSend(Symbol(), OP_BUYSTOP, fLots(dPourcentageRisque, iStop), dPlusHaut +
      iDistance* fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la
      veille", iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600) + ((iMinutesFin -
      TimeMinute(Time[0]))*60)), Blue);
      fTicket(iTicket);
      fErreur("Achat");


      while(IsTradeContextBusy() == True)
       Sleep(1000);
      iTicket = OrderSend(Symbol(), OP_SELLSTOP, fLots(dPourcentageRisque, iStop),  dPlusBas -
      iDistance* fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la
      veille", iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600)+((iMinutesFin -
      TimeMinute(Time[0]))*60)), Red);
      fTicket(iTicket);
      fErreur("Vente");


      bTrade = False;
      }

    if(iDate != Day())
      {
      bRecuperation = False;
      bTrade = True;
      }

    }

  return(0);
  }
//+------------------------------------------------------+


void fTracageLignes(string sNomLigne, double dPrix, color cCouleur)
  {
  ObjectDelete(sNomLigne);
  ObjectCreate(sNomLigne,OBJ_HLINE, 0, 0, dPrix);
```

```
 ObjectSet(sNomLigne, OBJPROP_COLOR, cCouleur);
 }

double fPoint(string sPaire)
 {
 if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 3)
  {
  double dPoint = 0.01;
  }
 if((MarketInfo(sPaire, MODE_DIGITS)) == 4 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
  {
  dPoint = 0.0001;
  }
 return(dPoint);
 }

double fSlippage(string sPaire, int iPipsSlippage)
 {
 if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 4)
  {
  double dSlippage = iPipsSlippage;
  }
 if((MarketInfo(sPaire, MODE_DIGITS)) == 3 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
  {
  dSlippage = iPipsSlippage * 10;
  }
 return(dSlippage);
 }

void fTicket(int iTicket)
 {
 if(iTicket != -1)
  {
  OrderSelect(iTicket, SELECT_BY_TICKET);
  if(OrderType() == 4)
    OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() - iStop * fPoint(Symbol()),
    OrderOpenPrice() + iLimite * fPoint(Symbol()), OrderExpiration());

  if(OrderType() == 5)
    OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() + iStop * fPoint(Symbol()),
    OrderOpenPrice() - iLimite * fPoint(Symbol()), OrderExpiration());
    }
 }

double fLots(double dPourcentageRisque, int iStop)
 {
 double dRisqueMax = (dPourcentageRisque/100) * AccountBalance();
 double dValeurPip = MarketInfo(Symbol(), MODE_TICKVALUE);
 if(Point == 0.001 || Point == 0.00001)
  {
  dValeurPip = dValeurPip * 10;
  }
 double dLots = (dRisqueMax / iStop) / dValeurPip;
 if(dLots < MarketInfo(Symbol(),MODE_MINLOT))
  dLots = MarketInfo(Symbol(),MODE_MINLOT);
 if(dLots > MarketInfo(Symbol(),MODE_MAXLOT))
  dLots = MarketInfo(Symbol(),MODE_MAXLOT);
 if(MarketInfo(Symbol(),MODE_LOTSTEP) == 0.01)
  dLots = NormalizeDouble(dLots, 2);
 else
  dLots = NormalizeDouble(dLots, 1);
 return(dLots);
 }

void fErreur(string sPosition)
 {
```

```
  if(iTicket == -1)
    {
    int iErreur = GetLastError();
    string sErreur = ErrorDescription(iErreur);
    Print("Erreur position "+sPosition+" : "+iErreur+" | "+sErreur);
    }
  }

void fStopSuiveur (string sSymbol, int iStopSuiveur, int iMagic)
  {
  for (int iCompteur = 0; iCompteur <= OrdersTotal() – 1; iCompteur++)
    {
    OrderSelect(iCompteur, SELECT_BY_POS);

    double dStopSuiveurBuy = MarketInfo(sSymbol, MODE_BID) – (iStopSuiveur * fPoint(sSymbol));
    double dStopSuiveurSell = MarketInfo(sSymbol, MODE_ASK) + (iStopSuiveur * fPoint(sSymbol));

    if(OrderMagicNumber() == iMagic && OrderSymbol() == sSymbol  && OrderType() == OP_BUY &&
    OrderStopLoss() < dStopSuiveurBuy)
      {
      bool bErreur = OrderModify(OrderTicket(), OrderOpenPrice(), dStopSuiveurBuy, OrderTakeProfit(),
      0) ;
      if(bErreur == 0)
        {
        int iErreur = GetLastError();
        string sErreur = ErrorDescription(iErreur);
        Print("Erreur modification position achat : "+iErreur+" | "+sErreur);
        }
      }

    if(OrderMagicNumber() == iMagic && OrderSymbol() == sSymbol && OrderType() == OP_SELL &&
    OrderStopLoss() > dStopSuiveurSell)
      {
      bErreur = OrderModify(OrderTicket(), OrderOpenPrice(), dStopSuiveurSell, OrderTakeProfit(), 0) ;
      if(bErreur == 0)
        {
        iErreur = GetLastError();
        sErreur = ErrorDescription(iErreur);
        Print("Erreur modification position achat : "+iErreur+" | "+sErreur);
        }
      }

    }
  }
```

Si vous observez attentivement le code, vous remarquerez que si la plateforme est relancée, deux nouvelles positions risquent d'être placées si les conditions de marché le permettent. Nous allons voir dans le chapitre suivant les variables globales qui peuvent nous aider à résoudre cette faille.

31
Variables globales au niveau de la plateforme

Nous avons vu dans le chapitre 3 les variables globales au niveau d'un programme. Ces variables que vous déclariez au début d'un programme, en dehors des fonctions spéciales ou personnalisées et qui par conséquent peuvent être utilisées par toutes les fonctions de ce programme spécifique. Nous allons voir maintenant les fonctions globales au niveau de la plateforme. Ces variables peuvent être créées, modifiées et effacées par n'importe quel programme tournant sur la plateforme. Le nom et la valeur associée à la variable sont en fait sauvegardés sur le disque dur.

Les propriétés de ces variables sont les suivantes :

- Ces variables ne peuvent être que du type *double*.
- La durée de vie d'une variable globale au niveau de la plateforme est de 4 semaines si aucun programme n'a appelé cette variable pendant ce laps de temps depuis sa création.

Pour manipuler ces variables, nous disposons de 8 fonctions spécifiques que nous allons détailler ci-après.

Fonction datetime GlobalVariableSet()

La fonction *GlobalVariableSet()* permet de modifier la valeur associée à une variable ou, lorsque la variable n'existe pas, de créer une variable et de lui associer une variable. La fonction est de type *datetime* car elle renvoie la date et heure du dernier accès à la variable lorsqu'elle a été exécutée avec succès et 0 dans le cas contraire. La syntaxe de la fonction est :

```
datetime GlobalVariableSet(string sNom, double dValeur)

// string sNom : nom de la variable existante ou à créer

// double dValeur : valeur à associer à la variable (uniquement de type double)
```

Fonction bool GlobalVariableCheck()

La fonction *GlobalVariableCheck()* renvoie *True* si la variable existe et *False* dans le cas contraire. La syntaxe de la fonction est :

```
bool GlobalVariableCheck(string sNom)

// string sNom : nom de la variable
```

Fonction double GlobalVariableGet()

La fonction *GlobalVariableGet()* renvoie la valeur associée à une variable. Si la variable n'existe pas, la fonction renvoie 0. La syntaxe de la fonction est :

```
double GlobalVariableGet(string sNom)

// string sNom : nom de la variable
```

Fonction bool GlobalVariableDel()

La fonction *GlobalVariableDel()* permet de supprimer la variable indiquée. Renvoie *True* si la suppression est effectuée avec succès et *False* dans le cas contraire. La syntaxe de la fonction est :

```
bool GlobalVariableDel(string sNom)

// string sNom : nom de la variable
```

Fonction string GlobalVariableName()

La fonction *GlobalVariableNamel()* renvoie le nom d'une variable par rapport à sa position dans la liste des variables globales. La syntaxe de la fonction est :

```
String  GlobalVariableName(int iPosition)

// int iPosition  : position de la variable dans la liste des variables globales
```

La liste dont il est question ci-dessus est la liste des variables globales que vous pouvez retrouver sur votre plateforme dans le menu *Tools* > *Global Variables* ou en appuyant sur la touche *F3* lorsque vous êtes sur MetaTrader.

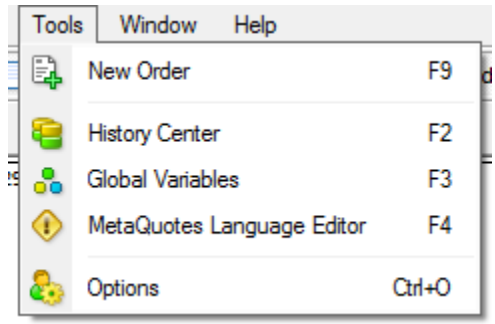

Figure 40 : Menu pour accéder à la liste des variables globales dans MetaTrader

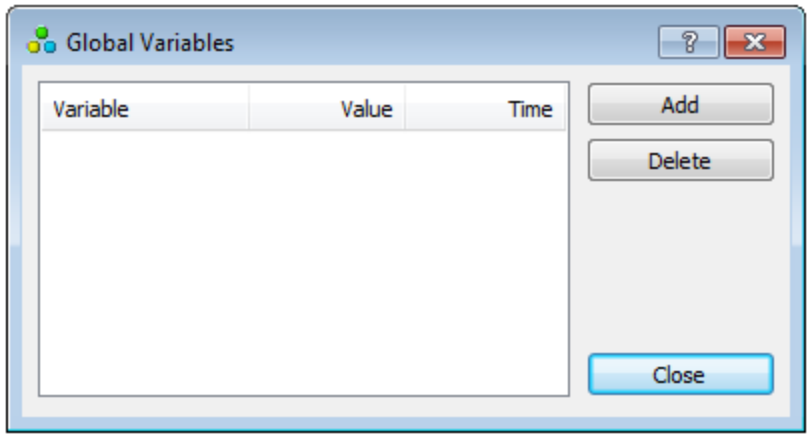

Figure 41 : Fenêtre permettant de visualiser la liste des variables globales dans MetaTrader

Fonction bool GlobalVariableSetOnCondition()

La fonction *GlobalVariableSetOnCondition()* permet de modifier la valeur associée à la variable lorsque la valeur actuelle de la variable est égale à une valeur spécifiée. Renvoie *True* si la modification est effectuée avec succès et *False* dans le cas contraire ou lorsque la valeur actuelle de la variable est différente de la valeur spécifiée pour la comparaison. La syntaxe de la fonction est :

```
bool GlobalVariableSetOnCondition(string sNom, double dValeur, double dVerifValeur)

// string sNom : nom de la variable

// double dValeur : valeur à associé à la variable

// double dVerifValeur : valeur à comparer à la valeur actuelle de la variable
```

Fonction int GlobalVariablesDeleteAll()

La fonction *GlobalVariablesDeleteAll()* permet de supprimer les variables globales contenant un préfixe spécifique ou la totalité des variables globales de la plateforme lorsqu'aucun préfixe n'est spécifié. La fonction renvoie le nombre de variables supprimées. La syntaxe est la suivante :

```
int GlobalVariableDeleteAll(string sPrefixe)

// string sPrefixe : prefixe du nom des variables à supprimer
```

Fonction int GlobalVariablesTotal ()

La fonction *GlobalVariablesTotal()* renvoie le nombre total de variables globales. La syntaxe est la suivante :

```
int GlobalVariableTotal()
```

Exemples d'application des variables globales

Dans cette section, nous allons détailler deux exemples possibles d'application pour des variables globales et étudier le code qui serait utilisé.

Premier exemple

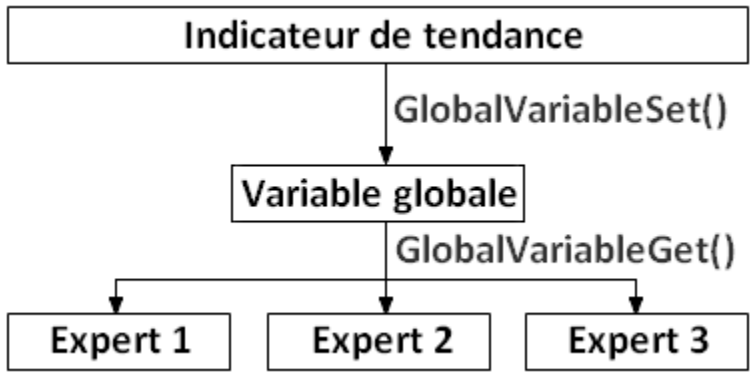

Figure 42 : Exemple d'application #1 des variables globales

Dans notre premier exemple illustré par la figure 42, nous avons 3 experts consultants fonctionnant en parallèle sur une même paire mais avec des paramètres différents. Néanmoins, nous désirons que tous les experts prennent position dans le sens de la tendance générale du marché. Cette tendance générale est fournie par un indicateur de tendance externe à nos 3 experts.

Notre indicateur de tendance va donc créer une variable globale au niveau de la plateforme qui pourra prendre une des deux valeurs suivantes : 1 ou 0 (respectivement tendance haussière et baissière). Chacun de nos experts n'a ensuite qu'à aller chercher la valeur de la variable et utiliser l'information pour filtrer sa prise de position.

Voici donc le code qui devrait figurer dans le code de l'indicateur de tendance.

```
// supposons que l'indicateur donne la valeur Haussiere ou Baissiere à la variable sTendance

if(string sTendance == "Haussiere")
  {
  bool bErreur = GlobalVariableSet("gvTendance", 1) ;
  if(bErreur == 0)
    {
    int iErreur = GetLastError();
    string sErreur = ErrorDescription(iErreur);
    Print("Erreur lors de la creation/modification de la variable globale gvTendance : " + sErreur);
    }
  }

else if(sTendance == "Baissiere")
  {
  GlobalVariableSet("gvTendance", 0)
    {
    int iErreur = GetLastError();
```

```
   string sErreur = ErrorDescription(iErreur);
   Print("Erreur lors de la creation/modification de la variable globale gvTendance : " + sErreur);
   }
  }
```

Et voici le code qui devrait figurer dans chacun des experts.

```
if(GlobalVariableCheck("gvTendance") == True) // test de l'existence de la variable
  {
  if(GlobalVariableGet("gvTendance") == 1)
    {
    bool bAcheter = True; // autoriser à acheter
    bool bVendre = False; // interdire de vendre
    }
  if(GlobalVariableGet("gvTendance") == 0)
    {
    bool bAcheter = False; // interdire d'acheter
    bool bVendre = True; // autoriser à vendre
    }
  }
```

Deuxième exemple

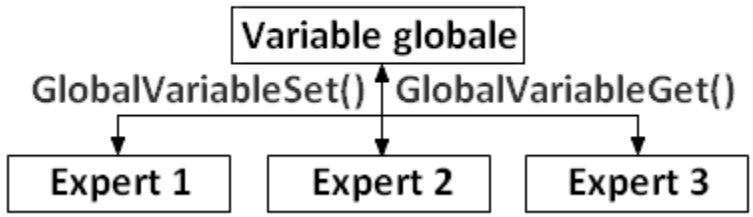

Figure 43 : Exemple d'application #2 des variables globales

Dans ce deuxième exemple, nous avons 3 experts qui vont communiquer avec la variable globale aussi bien pour la modifier que pour lire la valeur qui lui est associée. Supposons que vous désiriez limiter le nombre de positions ouvertes à 10 sur votre plateforme (si un expert a deux positions ouvertes, il ne reste donc que 8 positions pour les autres experts).

Voici donc le code qui permettra aux experts de transférer et d'obtenir l'information nécessaire. Différentes approches sont possibles pour résoudre cet exemple – celle qui suit n'est qu'une façon parmi tant d'autres.

En premier lieu, nous avons besoin d'une fonction qui comptera les positions pour chacun des experts.

```
int fNbrPositions( int iMagic)
  {
   int iDecompte = 0;
   for (int iDecompte = OrdersTotal() - 1 ; iCount >= 0 ; iCount --)
     {
     if (OrderSelect(iCount,SELECT_BY_POS))
       {
       if (OrderMagicNumber() == iMagic)
         {
          iDecompte ++;
         }
```

```
    }
   }
   return(iDecompte);
 }
```

En deuxième étape, chacun des experts devra créer une variable globale contenant le décompte de ses positions afin que les autres experts puissent en prendre connaissance.

```
// Variables à définir pour l'expert

int iDecompteExpert = 0;
int iMagic = 34324324;  // Chaque expert devra avoir un numéro magique unique

// Code à insérer dans la fonction init();

   bool bErreur = GlobalVariableSet("dDecompteExpertN", 0) ; // Changer N par 1,2,3,...
   if(bErreur == 0)
    {
    int iErreur = GetLastError();
    string sErreur = ErrorDescription(iErreur);
    Print("Erreur lors de la creation/modification de la variable globale dDecompteExpertN : " + sErreur);
    }

// code à insérer dans la fonction int start()

if(iDecompteExpert != fNbrPositions(iMagic))
  {
  iDecompteExpert = fNbrPositions(iMagic));
   bool bErreur = GlobalVariableSet("dDecompteExpertN", iDecompteExpert) // Changer N par 1,2,3,...
   if(bErreur == 0)
    {
    int iErreur = GetLastError();
    string sErreur = ErrorDescription(iErreur);
    Print("Erreur lors de la creation/modification de la variable globale dDecompteExpertN : " + sErreur);
    }
  }
```

Finalement chaque expert récupère l'information envoyée par les autres experts et l'interprète.

```
// Variables à définir pour l'expert

int iMaximumPositions = 10;
int iDecomptePlateforme = 0;
intiPositionsPossibles = 0;
bool bTradingAutorise = False;

// Dans le cas de l'expert 1


iDecomptePlateforme = iDecomptePlateforme + GlobalVariableGet("dDecompteExpert2")
iDecomptePlateforme = iDecomptePlateforme + GlobalVariableGet("dDecompteExpert3")

iPositionsPossibles = iMaximumPositions – iDecomptePlateforme;

if(fNbrPositions(iMagic) < iPositionsPossibles)
  bTradingAutorise = True;
else
  bTradingAutorise = False;

if(bTradingAutorise = True)
  {
  // Inclure ici les fonctions de passage d'ordres.
  }
```

32
Votre premier expert consultant #8

Supposons que l'expert fonctionne normalement et que soudainement, l'ordinateur ou le serveur redémarre. Lors du redémarrage de la plateforme et si les conditions sont encore correctes, l'expert placera deux nouveaux ordres identiques aux précédents.

Pour résoudre ce problème, plusieurs solutions sont possibles mais étant donné que le but de l'élaboration de cet expert est l'apprentissage, nous allons utiliser des variables globales au niveau de la plateforme.

Nous allons donc stocker le numéro de ticket de chaque ordre dans une variable globale. Si la plateforme est relancée, l'expert vérifiera si il existe une position (en attente, ouverte ou clôturée) avec ce même numéro de ticket, si c'est le cas, l'expert n'autorisera pas l'expert à prendre de nouvelles positions avant la prochaine nouvelle journée.

La première étape est donc de créer deux variables globales qui contiendront nos numéros de ticket à chaque nouvelle prise de position.

Nous allons créer une fonction qui créer et modifier les variables globales contenant les numéros de ticket.

```
void fTicketVariableGloable(string sNomVariableGlobale, int iNumeroTicket)
 {
 bool bErreur = GlobalVariableSet(sNomVariableGlobale, iNumeroTicket) ;
 if(bErreur == 0)
   {
   int iErreur = GetLastError();
   string sErreur = ErrorDescription(iErreur);
   Print("Erreur lors de la creation/modification de la variable globale "+ sNomVariableGlobale +" : " +
   sErreur);
   }
 }
```

Nous ajouterons un appel de fonction après chacune de nos fonctions de passage d'ordres.

```
  if(bTrade == True)
    {
    while(IsTradeContextBusy() == True)
      Sleep(1000);
    iTicket = OrderSend(Symbol(), OP_BUYSTOP, fLots(dPourcentageRisque, iStop), dPlusHaut +
    iDistance* fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la
    veille", iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600) + ((iMinutesFin -
    TimeMinute(Time[0]))*60)), Blue);
    fTicket(iTicket);
    fTicketVariableGloable("gvNumeroTicketAchat", iTicket);
    fErreur("Achat");

    while(IsTradeContextBusy() == True)
     Sleep(1000);
    iTicket = OrderSend(Symbol(), OP_SELLSTOP, fLots(dPourcentageRisque, iStop),  dPlusBas -
    iDistance* fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la
    veille", iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600)+((iMinutesFin -
    TimeMinute(Time[0]))*60)), Red);
    fTicket(iTicket);
```

```
    fTicketVariableGloable("gvNumeroTicketVente", iTicket);
    fErreur("Vente");

    bTrade = False;
    }
```

Il ne reste plus qu'à rajouter des instructions pour que l'expert vérifie avant de passer un ordre si des ordres ont déjà été ouverts. Nous allons donc créer une autre fonction pour effectuer la vérification.

```
void fVerifPositions()
 {
  if(OrderSelect(GlobalVariableGet("gvNumeroTicketAchat"),SELECT_BY_TICKET) == True ||
  OrderSelect(GlobalVariableGet("gvNumeroTicketVente"),SELECT_BY_TICKET) == True)
   {
   bTrade = False;
   }
  }
```

La fonction ci-dessus vérifie s'il existe un ordre avec le même numéro de ticket que ceux associés aux variables globales. Si c'est le cas, l'expert empêchera l'expert de placer de nouvelles positions. Nous avons choisi de placer un « ou » (||) au lieu d'un « et » au cas où un des ordres n'aurait pas été placé correctement et fausserait donc le résultat de la vérification.

Il ne reste plus qu'à placer un appel pour cette fonction dans notre fonction spéciale *init()* car nous ne désirons effectuer cette vérification que lorsque l'expert démarre.

```
fVerifPositions();
```

Nous allons également ajouter une instruction conditionnelle avant chaque passage d'ordre permettant de vérifier si les conditions minimales requises par le courtier sont remplies (distance par rapport au cours actuel) afin d'éviter que le programme ne tente de placer un ordre si les conditions ne sont pas remplies. L'instruction vérifie donc si la distance entre le cours actuel (*Ask* pour achat et *Bid* pour Vente) et notre point d'entrée est supérieur à la valeur minimale requise par le courtier que nous obtenons à l'aide de la fonction *MarketInfo()*.

```
if((dPlusHaut + iDistance * fPoint(Symbol())) - Ask > (MarketInfo(Symbol(), MODE_STOPLEVEL)) *
fPoint(Symbol()))
 {
 iTicket = OrderSend(Symbol(), OP_BUYSTOP, fLots(dPourcentageRisque, iStop), dPlusHaut + iDistance*
 fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la veille",
 iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600) + ((iMinutesFin - TimeMinute(Time[0]))*60)),
 Blue);
 fTicket(iTicket);
 fTicketVariableGloable("gvNumeroTicketAchat", iTicket);
 fErreur("Achat");
 }


if(Bid - (dPlusBas - iDistance* fPoint(Symbol())) > (MarketInfo(Symbol(), MODE_STOPLEVEL)) *
fPoint(Symbol()))
 {
 iTicket = OrderSend(Symbol(), OP_SELLSTOP, fLots(dPourcentageRisque, iStop),  dPlusBas - iDistance*
 fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la veille",
 iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600)+((iMinutesFin - TimeMinute(Time[0]))*60)), Red);
 fTicket(iTicket);
 fTicketVariableGloable("gvNumeroTicketVente", iTicket);
 fErreur("Vente");
 }
```

Voici à quoi ressemble le code avec ces nouveaux ajouts (ajouts et modifications en gras).

```
//+------------------------------------------------------------------+
//|                                Votre Premier Expert.mq4          |
//|                                  Copyright © 2011,               |
//|                               http://www. eole-trading.com       |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www. eole-trading.com"

#include <stdlib.mqh>


extern int iHeureDebut = 9;
extern int iMinutesDebut = 30;
extern int iHeureFin = 17;
extern int iMinutesFin = 30;
extern int iDistance = 5;
extern int iMagic = 123456;
extern int iStop = 50;
extern int iLimite = 50;
extern double dPourcentageRisque = 5;
extern int iStopSuiveur = 25;

int iDate;
int iTimeDebut;
int iTimeFin;
int iTimeSeconds;
int iTicket;
double dPlusHaut;
double dPlusBas;
bool bRecuperation = False;
bool bTrade = True;


//+------------------------------------------------------+
//| expert initialization fonction                       |
//+------------------------------------------------------+
int init()
  {

  iTimeDebut = iHeureDebut * 3600 + iMinutesDebut * 60;
  iTimeFin = iHeureFin * 3600 + iMinutesFin * 60;

  fVerifPositions();

  return(0);
  }
//+------------------------------------------------------+
//| expert deinitialization fonction                     |
//+------------------------------------------------------+
int deinit()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------+
//| expert start fonction                                |
//+------------------------------------------------------+
int start()
  {

  iTimeSeconds = Hour()*3600 + Minute()*60 + Seconds();


  fStopSuiveur(Symbol(), iStopSuiveur, iMagic);
```

```
  if(iTimeSeconds >= iTimeDebut && iTimeSeconds < iTimeFin)
    {
    if(bRecuperation == False)
      {
      iDate = Day();
      dPlusHaut = iHigh(NULL, PERIOD_D1, 1);
      dPlusBas = iLow(NULL, PERIOD_D1, 1);
      bRecuperation = True;
      fTracageLignes("Point plus haut", dPlusHaut, Blue);
      fTracageLignes("Point plus bas", dPlusBas, Red);
      }


    if(bTrade == True)
      {
      while(IsTradeContextBusy() == True)
        Sleep(1000);
      if((dPlusHaut + iDistance * fPoint(Symbol())) - Ask > (MarketInfo(Symbol(), MODE_STOPLEVEL)) *
      fPoint(Symbol()))
        {
        iTicket = OrderSend(Symbol(), OP_BUYSTOP, fLots(dPourcentageRisque, iStop), dPlusHaut +
        iDistance* fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la
        veille", iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600) + ((iMinutesFin -
        TimeMinute(Time[0]))*60)), Blue);
        fTicket(iTicket);
        fTicketVariableGloable("gvNumeroTicketAchat", iTicket);
        fErreur("Achat");
        }

      while(IsTradeContextBusy() == True)
        Sleep(1000);
      if(Bid - (dPlusBas - iDistance* fPoint(Symbol())) > (MarketInfo(Symbol(), MODE_STOPLEVEL)) *
      fPoint(Symbol()))
        {
        iTicket = OrderSend(Symbol(), OP_SELLSTOP, fLots(dPourcentageRisque, iStop),  dPlusBas -
        iDistance* fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la
        veille", iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600)+((iMinutesFin -
        TimeMinute(Time[0]))*60)), Red);
        fTicket(iTicket);
        fTicketVariableGloable("gvNumeroTicketVente", iTicket);
        fErreur("Vente");
        }

      bTrade = False;
      }

    if(iDate != Day())
      {
      bRecuperation = False;
      bTrade = True;
      }

    }

  return(0);
  }
//+------------------------------------------------------+


void fTracageLignes(string sNomLigne, double dPrix, color cCouleur)
  {
  ObjectDelete(sNomLigne);
  ObjectCreate(sNomLigne,OBJ_HLINE, 0, 0, dPrix);
  ObjectSet(sNomLigne, OBJPROP_COLOR, cCouleur);
  }


double fPoint(string sPaire)
  {
  if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 3)
```

```
   {
   double dPoint = 0.01;
   }
  if((MarketInfo(sPaire, MODE_DIGITS)) == 4 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
   {
   dPoint = 0.0001;
   }
  return(dPoint);
  }

double fSlippage(string sPaire, int iPipsSlippage)
  {
  if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 4)
   {
   double dSlippage = iPipsSlippage;
   }
  if((MarketInfo(sPaire, MODE_DIGITS)) == 3 || (MarketInfo(sPaire, MODE_DIGITS)) == 5)
   {
   dSlippage = iPipsSlippage * 10;
   }
  return(dSlippage);
  }

void fTicket(int iTicket)
  {
  if(iTicket != -1)
   {
   OrderSelect(iTicket, SELECT_BY_TICKET);
   if(OrderType() == 4)
     OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() - iStop * fPoint(Symbol()),
     OrderOpenPrice() + iLimite * fPoint(Symbol()), OrderExpiration());

   if(OrderType() == 5)
     OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() + iStop * fPoint(Symbol()),
     OrderOpenPrice() - iLimite * fPoint(Symbol()), OrderExpiration());
     }
  }

double fLots(double dPourcentageRisque, int iStop)
  {
  double dRisqueMax = (dPourcentageRisque/100) * AccountBalance();
  double dValeurPip = MarketInfo(Symbol(), MODE_TICKVALUE);
  if(Point == 0.001 || Point == 0.00001)
   {
   dValeurPip = dValeurPip * 10;
   }
  double dLots = (dRisqueMax / iStop) / dValeurPip;
  if(dLots < MarketInfo(Symbol(),MODE_MINLOT))
   dLots = MarketInfo(Symbol(),MODE_MINLOT);
  if(dLots > MarketInfo(Symbol(),MODE_MAXLOT))
   dLots = MarketInfo(Symbol(),MODE_MAXLOT);
  if(MarketInfo(Symbol(),MODE_LOTSTEP) == 0.01)
   dLots = NormalizeDouble(dLots, 2);
  else
   dLots = NormalizeDouble(dLots, 1);
  return(dLots);
  }

void fErreur(string sPosition)
  {
  if(iTicket == -1)
   {
   int iErreur = GetLastError();
   string sErreur = ErrorDescription(iErreur);
   Print("Erreur position "+sPosition+" : "+iErreur+" | "+sErreur);
   }
  }
```

```
void fStopSuiveur (string sSymbol, int iStopSuiveur, int iMagic)
 {
 for (int iCompteur = 0; iCompteur <= OrdersTotal() – 1; iCompteur++)
  {
  OrderSelect(iCompteur, SELECT_BY_POS);

  double dStopSuiveurBuy = MarketInfo(sSymbol, MODE_BID) – (iStopSuiveur * fPoint(sSymbol));
  double dStopSuiveurSell = MarketInfo(sSymbol, MODE_ASK) + (iStopSuiveur * fPoint(sSymbol));

  if(OrderMagicNumber() == iMagic && OrderSymbol() == sSymbol  && OrderType() == OP_BUY &&
  OrderStopLoss() < dStopSuiveurBuy)
   {
   bool bErreur = OrderModify(OrderTicket(), OrderOpenPrice(), dStopSuiveurBuy, OrderTakeProfit(),
   0) ;
   if(bErreur == 0)
    {
    int iErreur = GetLastError();
    string sErreur = ErrorDescription(iErreur);
    Print("Erreur modification position achat : "+iErreur+" | "+sErreur);
    }
   }

  if(OrderMagicNumber() == iMagic && OrderSymbol() == sSymbol && OrderType() == OP_SELL &&
  OrderStopLoss() > dStopSuiveurSell)
   {
   bErreur = OrderModify(OrderTicket(), OrderOpenPrice(), dStopSuiveurSell, OrderTakeProfit(), 0) ;
   if(bErreur == 0)
    {
    iErreur = GetLastError();
    sErreur = ErrorDescription(iErreur);
    Print("Erreur modification position achat : "+iErreur+" | "+sErreur);
    }
   }

  }
 }

void fVerifPositions()
 {
  if(OrderSelect(GlobalVariableGet("gvNumeroTicketAchat"),SELECT_BY_TICKET) == True ||
  OrderSelect(GlobalVariableGet("gvNumeroTicketVente"),SELECT_BY_TICKET) == True)
   {
   bTrade = False;
   }
 }

void fTicketVariableGloable(string sNomVariableGlobale, int iNumeroTicket)
 {
 bool bErreur = GlobalVariableSet(sNomVariableGlobale, iNumeroTicket);
 if(bErreur == 0)
   {
   int iErreur = GetLastError();
   string sErreur = ErrorDescription(iErreur);
   Print("Erreur lors de la creation/modification de la variable globale "+ sNomVariableGlobale +" : "
  + sErreur);
   }
 }
```

33
Le traitement des tableaux en MQL4

Les tableaux sont des variables –structurées- que les différentes fonctions vont utiliser pour traiter des informations. Nous avons ajouté l'adjectif « structurées » précédemment car les tableaux sont en fait un ensemble de valeurs de même type et qui ont la même dénomination. Nous avons déjà vu des tableaux précédemment lors de l'étude des fonctions sur les barres (*Open[]*, *Close[]*, *High[]*, *Low[]*, *Volume[]*, *Time[]*). Nous allons maintenant voir comment créer des tableaux personnalisés ou traiter l'information contenue dans les tableaux.

Les tableaux peuvent être de tous les types de données disponibles en MQL4 et peuvent être de plusieurs dimensions comme vous pouvez le voir ci-dessous.

En MQL4, les tableaux sont indexés en commençant la numérotation à 0 comme vous pouvez le voir sur les figures. Le premier élément du tableau sera donc indexé à 0 et le dernier à n-1 (n étant le nombre d'éléments composant le tableau).

0	1	2	3	4	5	6	7	8	n
-47	-63	84	65	76	-6	-23	57	2	23

Figure 44 : Tableau à une dimension

	0	1	2	3	4	5	6	7	8	n
0	61.92	37.72	56.60	19.83	82.24	60.26	6.01	69.88	35.23	17.56
1	11.75	75.37	51.68	87.81	41.88	17.79	58.10	27.38	32.69	1.33
2	90.75	-2.98	29.77	8.13	30.44	66.66	-1.23	23.83	66.44	96.64
3	88.89	47.11	98.68	83.45	99.33	41.62	73.61	57.58	94.28	65.91
4	78.87	4.36	47.04	62.03	-5.04	14.38	54.91	95.23	97.56	49.05
5	97.88	97.50	56.40	96.49	60.81	85.38	29.96	72.20	75.21	89.39
6	-6.69	98.89	43.20	54.65	8.07	1.72	89.57	56.96	42.46	-6.23
7	62.11	3.14	77.50	82.53	7.28	29.18	65.01	1.46	92.97	61.37
8	7.21	46.27	3.06	47.47	-8.43	83.09	86.70	-7.49	92.92	89.28
n	11.36	19.92	50.74	56.00	15.20	77.99	70.90	81.42	1.12	4.12

Figure 45 : Tableau à 2 dimensions

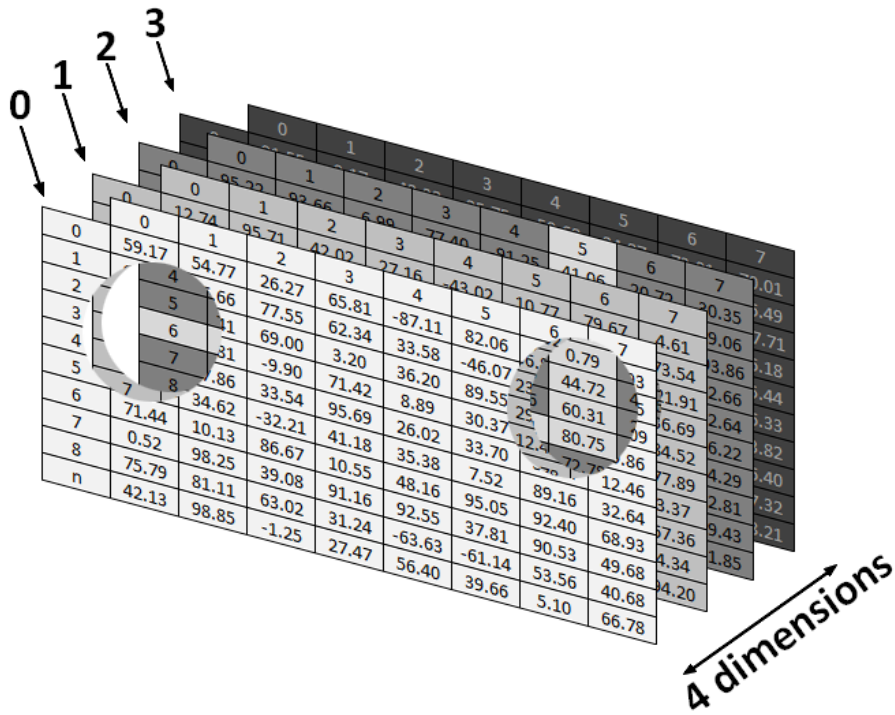

Figure 46 : Tableau à 4 dimensions

Le nombre maximal de dimensions est 4 pour les tableaux.

Accès aux données d'un tableau

Prenons par exemple notre tableau de la figure 45. Avec le code suivant et en supposant que le nom du tableau est *tExemple* :

```
Print(tExemple[4, 5]);

// Ou encore

Print(tExemple[4] [5]);
```

L'information que le programme afficherait serait 60.81.

```
Print(tExemple[6]); // Pour la figure 44, le programme renverrait -23

Print(tExemple[5, 6, 2]); // Pour la figure 46, le programme renverrait 80.75
```

Attention à l'affichage des données d'un tableau avec les fonctions *Print()* ou *Comment()*. En effet, les codes précédents sont corrects mais il n'est pas possible d'utiliser une boucle for comme ceci.

```
for(int i = 0, i < 20, i++)
  Print(tExemple[i]);
```

Le code ci-dessus est incorrect. Vous devez impérativement toujours préciser la case exacte que vous désirez afficher comme dans les exemples précédents.

Initialisation d'un tableau

Pour initialiser un tableau à une dimension, la syntaxe est la suivante :

```
type tTbableau [n] = {x1, x2, x3, xn};

// type : correspond au type des données que le tableau contiendra

// n : nombre d'élément pour un tableau à 1 dimension

// x1, x2, x3, xn : données du tableau
```

Voici quelques exemples d'initialisation avec des tableaux à différentes dimensions.

```
int tTableau [5] = {3, 5, -2, 89, 23};

int tTableau [2] [3] = {3, 5, -2,    12, 45, -34};

int tTableau [2] [3] [2] = {3, 5, -2,    12, 45, -34,    98, -67, 2,    123, -86, 1};
```

Si au cours de l'initialisation du tableau, le nombre de données déclarées est inférieur à la taille du tableau, les cases vides du tableau seront remplies avec un 0. Par exemple :

```
int tTableau [5] = {3, 5, -2, 89};
```

Dans le cas ci-dessus, nous obtiendrons le tableau suivant (le paramètre manquant est remplacé par 0) :

0	1	2	3	4
3	5	-2	89	0

Figure 47 : Initialisation d'un tableau avec moins de paramètres que requis

Fonctions de traitement de tableaux

Vous trouverez ci-dessous la liste de toutes les fonctions permettant de traiter/modifier les informations contenues dans un tableau.

Fonction int ArrayBsearch()

Cette fonction permet de chercher la position d'une case d'un tableau contenant une valeur précise. Si la valeur n'existe pas, la fonction renverra la position de l'élément le plus proche avec la plus petite valeur.

Cette fonction ne peut effectuer une recherche que dans un tableau ordonné. Nous verrons la fonction *ArraySort()* qui permet de trier et ranger un tableau par la suite.

La syntaxe de la fonction est la suivante :

```
int ArrayBsearch(double tTableau[], double dValeur, int iDecompte = WHOLE_ARRAY, int iDebut = 0, int iDirection = MODE_ASCEND)

// double tTableau[] : nom du tableau sur lequel la recherché doit être effectuée.

// double dValeur : valeur à rechercher.

// inti Count : nombre d'élément sur lesquels effectuer la recherche (par défaut, tout le tableau).

// int iDebut : Position de la case du tableau qui servira de point de départ pour débuter la recherche.

// int iDirection : Sens de la recherche (MODE_ASCEND ou MODE_DESCEND). Par défaut ascendant.
```

Un exemple :

```
double tTableau[5] = {1,2,3,4,5};

ArraySort(tTableau);

int iPosition = ArrayBsearch(tTableau, 2);

Print ("Position : "+iPosition); // le programme renverra « Position : 1 »
```

Fonction int ArrayCopy()

Cette fonction permet de copier le contenu d'un tableau dans un autre. Il est important que les deux tableaux soient du même type. La fonction renverra le nombre d'éléments qui ont été copiés. La syntaxe de la fonction est la suivante :

```
int ArrayCopy(void tTableauDest[], object tTableauSource[], int iDebutDest = 0, int iDebutSource = 0, int iDecompte = WHOLE_ARRAY)

// void tTableauDest[] : nom du tableau de destination.

// object tTableauSource[] : nom du tableau source.

// int iDebutDest : position de la case de début pour le tableau de destination. Par défaut 0.

// int iDebutSource : position de la case de début pour le tableau source. Par défaut 0.

// int iDecompte: nombre d'élément à copier (par défaut, tout le tableau).
```

Un exemple :

```
double tTableauDest[5];
double tTableauSource[5] = {1,2,3,4,5};

ArrayCopy(tTableauDest, tTableauSource, 0, 0, WHOLE_ARRAY);
```

Fonction int ArrayCopyRates()

Cette fonction permet de copier les six informations suivantes : *time*, *open*, *low*, *high*, *close* et *volume* dans un tableau de type *double* à deux dimensions. La fonction renverra le nombre de barres dont l'information a été copiée ou -1 si la copie a échoué. La syntaxe de la fonction est la suivante :

```
int ArrayCopyRates(double tTableauDest[], string sSymbol = NULL, int iUniteTemps = 0)

// double tTableauDest[] :  nom du tableau à deux dimensions de destination.

// string sSymbol : paire des barres dont vous désirez copier les informations. Par défaut la paire du
graphique.

// int iUniteTemps : unité de temps des barres dont vous voulez copier l'information. Par défaut celle du
// graphique.
```

Un exemple :

```
double tTableauDest[] [6]; // La deuxième dimension doit être 6

ArrayCopyRates(tTableauDest, Symbol(), PERIOD_D1);
```

Fonction int ArrayCopySeries()

Cette fonction permet de copier une information sous la forme d'une série temporelle (l'information la plus récente jusqu'à la plus ancienne) dans un tableau et renvoie le nombre d'éléments copiés. La syntaxe de la fonction est la suivante :

```
int ArrayCopySeries(double tTableauDest[], int iInformation, string sSymbol = NULL, int iUniteTemps = 0)

// double tTableauDest[] :  nom du tableau à une dimension de destination.

// int iInformation : information que vous désirez copier dans le tableau. Voir tableau 14.

// string sSymbol : paire des barres dont vous désirez copier les informations. Par défaut la paire du
graphique.

// int iUniteTemps : unité de temps des barres dont vous voulez copier l'information. Par défaut celle du
// graphique.
```

Un exemple :

```
double tTableauDest[];

ArrayCopySeries(tTableauDest, MODE_OPEN, Symbol(), 0);
```

Lorsque vous désirez copier l'heure d'ouverture des barres, *MODE_TIME*, le tableau de destination doit être impérativement de type *datetime*.

Fonction int ArrayDimension()

Cette fonction permet de renvoyer la dimension d'un tableau. La syntaxe de la fonction est la suivante :

```
int ArrayDimension(object tTableau[])
// object tTableau [] : nom du tableau dont vous désirez obtenir la dimension
```

Un exemple :

```
double tTableauDest[4][5]; // Tableau de deux dimensions
ArrayDimension(tTableau); // La fonction renverra 2.
```

Fonction int ArrayGetAsSeries()

Cette fonction permet de savoir si un tableau est organisé comme une série temporelle (du dernier au premier). La fonction renverra *True* si c'est le cas et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool ArrayGetAsSeries( object array[])
// object tTableau[] : nom du tableau duquel vous désirez obtenir l'information
```

Un exemple :

```
ArrayGetAsSeries(tTableau);
```

Fonction int ArrayInitialize()

Cette fonction permet d'initialiser toutes les cases d'un tableau avec la même valeur et renvoie le nombre de cases qui ont été initialisées. La syntaxe de la fonction est la suivante :

```
int ArrayInitialize(double tTableau[], double dValeur)
// double tTableau[] : nom du tableau dont vous désirez initialiser toutes les cases.
// double dValeur : valeur à utiliser pour initialiser toutes les cases.
```

Un exemple :

```
ArrayInitialize(tTableau, 100);
```

Fonction int ArrayIsSeries()

Cette fonction permet de savoir si le tableau est une série temporelle, c'est-à-dire un tableau contenant une des informations suivantes : *Time[]*, *Open[]*, *Close[]*, *High[]*, *Low[]* ou *Volume[]*. La fonction renverra *True* si c'est le cas et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool ArrayIsSeries(object tTableau[])
// object  tTableau[] : nom du tableau duquel vous désirez obtenir l'information.
```

Un exemple :

```
ArrayIsSeries (tTableau);
```

Fonction int ArrayMaximum()

Cette fonction permet de chercher la position de la case contenant la valeur la plus élevée dans le tableau. La syntaxe de la fonction est la suivante :

```
int ArrayMaximum(double tTableau[], int iDecompte = WHOLE_ARRAY, int iDebut = 0)
// double  tTableau[] : nom du tableau dans lequel vous désirez effectuer la recherche.
// int iDecompte : nombre de cases où vous désirez effectuer la recherche. Par défaut tout le tableau.
// int iDebut : case de départ pour débuter la recherche. Par défaut la première case.
```

Un exemple :

```
int tTableau[5]={43,1,23,-4,9};
int iPositionValeurMaxTableau = ArrayMaximum(tTableau);
// ou encore

int iPositionValeurMaxTableau = ArrayMaximum(tTableau, 5, 0);
// iPositionValeurMaxTableau = 0
Print("Valeur Maximale dans tTableau : "+ tTableau[iPositionValeurMaxTableau]);
// affichera « Valeur Maximale dans tTableau : 43 »
```

Fonction int ArrayMinimum()

Cette fonction permet de chercher la position de la case contenant la valeur la plus basse dans le tableau. La syntaxe de la fonction est la suivante :

```
int ArrayMinimum(double tTableau[], int iDecompte = WHOLE_ARRAY, int iDebut = 0)
// double  tTableau[] : nom du tableau dans lequel vous désirez effectuer la recherche.
// int iDecompte : nombre de cases où vous désirez effectuer la recherche. Par défaut tout le tableau.
```

```
// int iDebut : case de départ pour débuter la recherche. Par défaut la première case.
```

Un exemple :

```
int tTableau[5]={43,1,23,-4,9};
int iPositionValeurMinTableau = ArrayMinimum(tTableau);
// ou encore
int iPositionValeurMinTableau = ArrayMinimum(tTableau, 5, 0);
// iPositionValeurMinTableau = 3
Print("Valeur Minimale dans tTableau : "+ tTableau[iPositionValeurMinTableau]);
// affichera « Valeur Maximale dans tTableau : -4 »
```

Fonction int ArrayRange()

Cette fonction permet de renvoyer le nombre d'éléments dans une dimension particulière d'un tableau. La syntaxe de la fonction est la suivante :

```
int ArrayRange(object tTableau[], int iDimension)
// object tTableau[] : nom du tableau à analyser.
// int iDimension : dimension à analyser.
```

Un exemple :

```
int tTableau [30][2][3];
int iNbrElementDimension = ArrayRange(tTableau, 2);
```

Fonction int ArrayResize()

Cette fonction permet de modifier la taille de la première dimension d'un tableau. La fonction renverra le nombre d'éléments dans le tableau redimensionné lorsque la modification s'est effectuée avec succès et -1 lorsque celle-ci a échoué. La syntaxe de la fonction est la suivante :

```
int ArrayResize(object tTableau[], int iNouvelleTaille)
// object tTableau[] : nom du tableau à redimensionner.
// int iNouvelleTaille: nouvelle taille de la première dimension.
```

Un exemple :

```
int tTableau [10][2];
int ArrayResize(tTableau, 20);
```

```
// Nombre d'éléments avant modification : 20.
// Nombre d'éléments après modification : 40.
```

Fonction bool ArraySetAsSeries()

Cette fonction permet de modifier la direction de la numérotation du tableau. Soit le dernier élément est placé en position 0, soit le premier élément est en position 0. La fonction renverra la direction préalable au changement. La syntaxe de la fonction est la suivante :

```
bool ArraySetAsSeries (double  tTableau[], bool bDirection)

// double tTableau[] : nom du tableau dont vous désirez modifier la direction de la numérotation .

// bool bDirection: nouvelle direction (True : dernier élément est 0, False : premier élément est 0).
```

Un exemple :

```
ArraySetAsSeries(tTableau, True);
```

Fonction int ArraySize()

Cette fonction permet de renvoyer le nombre d'éléments contenus dans un tableau. La syntaxe de la fonction est la suivante :

```
int ArraySize(object tTableau[])

// object tTableau[] : nom du tableau dont vous désirez obtenir le nombre d'éléments.
```

Un exemple :

```
int tTableau [10][2];

ArraySize(tTableau); // La fonction renverra 20.
```

Fonction int ArraySort()

Cette fonction permet de ranger les éléments d'un tableau par ordre croissant ou décroissant.

Il n'est pas possible de ranger les éléments d'un tableau contenant une série temporelle.

La syntaxe de la fonction est la suivante :

```
int ArraySort(double tTableau[], int iDecompte = WHOLE_ARRAY, int iDebut = 0, int iSens =
MODE_ASCEND)
```

```
// double tTableau[] : nom du tableau dont vous désirez classer les éléments.

// int iDecompte : nombre de cases que vous désirez classer. Par défaut tout le tableau.

// int iDebut : case de départ pour débuter le classement. Par défaut la première case.

// int iSens : Sens du classement (MODE_ASCEND pour croissant et MODE_DESCEND pour décroissant).
// Par défaut, MODE_ASCEND.
```

Un exemple :

```
int tTableau[10]={34,11,-6,323,-679, 1, 23, -6, 9, 15}; // (1)

ArraySort(tTableau); // (2)

ArraySort(tTableau,WHOLE_ARRAY,0,MODE_DESCEND); // (3)

// après (1) : 34,11,-6,323,-679, 1, 23, -6, 9, 15

// après (2) : -679, -6, -6, 1, 9, 11, 15, 23, 34, 323

// après (3) : 323, 34, 23, 15, 11, 9, 1, -6, -6, -679
```

Calcul de fréquences relatives

Pour démontrer l'utilisation des tableaux, nous allons créer un petit programme permettant d'obtenir la fréquence relative des changements en pourcentage de prix d'une paire.

La première étape consiste à obtenir les prix de clôtures des barres. Nous utiliserons 60 barres pour cet exemple.

```
double tPrixCloture[60];
ArrayCopySeries(tPrixCloture, MODE_CLOSE, Symbol(), PERIOD_D1);
```

Le code ci-dessus va donc stocker le prix de clôture des 60 dernières barres quotidiennes. Il faut ensuite obtenir le nombre de pips entre chaque barre afin de pouvoir calculer la variation en termes de pourcentage.

```
int iPeriode = 260;
double tDifferencePrix[259];
int i, y, z, w;

for(i = 0 ; I < iPeriode ; i++)
  for(y = 0, int z = 1; y < iPeriode – 1; y++, z++)
    tDifferencePrix[y] = tPrixCloture[z-1] - tPrixCloture[z];
```

Nous sommes maintenant en mesure de calculer la variation en pourcentage entre chaque barre.

```
double tPercentageChange[259];


for(y = 0; y < iPeriode – 1; y++)
  tPercentageChange[y] = (tDifferencePrix[y] / tPrixCloture[y+1])*100;
```

Finalement, nous pouvons calculer la fréquence relative de chaque variation. Pour cela, il nous faut définir des intervalles pour les pourcentages.

Nous allons diviser les pourcentages obtenus en 12 intervalles. Pour calculer l'intervalle, nous nous baserons sur la plus grande et la plus basse variation afin d'avoir des intervalles de même amplitude.

```
double dNbrFrequence = 12.0;
double dValeurMax = tPercentageChange[ArrayMaximum(tPercentageChange)];
double dValeurMin = tPercentageChange[ArrayMinimum(tPercentageChange)];
double dFrequence = NormalizeDouble((dValeurMax - dValeurMin) / dNbrFrequence, 1);
```

Voici maintenant le code permettant de stocker les fréquences dans l'intervalle correct. Le tableau est initialisé à 0 pour éviter qu'entre deux itérations, le programme ajoute une valeur à une fréquence déjà présente dans le tableau d'une itération précédente et fausse ainsi le résultat.

```
ArraySort(tPercentageChange, WHOLE_ARRAY, 0, MODE_ASCEND);

double tFrequence[11];

ArrayInitialize(tFrequence, 0);


for(z = 0 ; z < iPeriode ; z++)
 {
 for(y = -(dNbrFrequence / 2), w = 0; y <= dNbrFrequence;  y++, w++)
   if(tPercentageChange[z] >= y*dFrequence && tPercentageChange[z] < (y+1)*dFrequence)
     tFrequence[w]++;
 }
```

Finalement, il ne reste plus qu'à diviser par le nombre de période pour obtenir la fréquence.

```
for(z=0;z<11;z++)
  tFrequence[z] = (tFrequence[z] / (iPeriode - 1));
```

Le code dans sa totalité ressemble à ceci.

```
double tPrixCloture[260];
ArrayCopySeries(tPrixCloture, MODE_CLOSE, Symbol(), PERIOD_D1);


int iPeriode = 260;
double tDifferencePrix[259];
int i, y, z, w;

for(I = 0; I < iPeriode; i++)
  for(y = 0, z = 1; y < iPeriode - 1; y++, z++)
    tDifferencePrix[y] = tPrixCloture[z-1] - tPrixCloture[z];


double tPercentageChange[259];


for(y = 0; y < iPeriode - 1; y++)
  tPercentageChange[y] = (tDifferencePrix[y] / tPrixCloture[y+1])*100;
```

```
double dNbrFrequence = 12.0;
double dValeurMax = tPercentageChange[ArrayMaximum(tPercentageChange)];
double dValeurMin = tPercentageChange[ArrayMinimum(tPercentageChange)];
double dFrequence = NormalizeDouble((dValeurMax - dValeurMin) / dNbrFrequence, 1);

ArraySort(tPercentageChange, WHOLE_ARRAY, 0, MODE_ASCEND);

double tFrequence[11];
ArrayInitialize(tFrequence, 0);


for(z = 0; z < iPeriode; z++)
 {
 for(y = -(dNbrFrequence / 2), w = 0; y <= dNbrFrequence; y++, w++)
   if(tPercentageChange[z] >= y*dFrequence && tPercentageChange[z] < (y+1)*dFrequence)
     tFrequence[w]++;
 }


for(z = 0; z < 11; z++)
 tFrequence[z] = (tFrequence[z] / (iPeriode - 1));
```

Si vous placez le code ci-dessus dans une fonction *int start(),* le programme calculera donc la fréquence relative de la variation du prix d'une paire pour une période donnée.

Voilà le résultat obtenu pour la paire GBPUSD. Si vous faitez la somme des pourcentage, vous n'obtiendrez pas 100% car la fonction *DoubleToStr()* a été utilisé pour n'afficher que 2 décimales.

```
GBPUSD,H1: >= 2.0 et < 2.4 = 0.00000000%
GBPUSD,H1: >= 1.6 et < 2.0 = 0.00000000%
GBPUSD,H1: >= 1.2 et < 1.6 = 0.00000000%
GBPUSD,H1: >= 0.8 et < 1.2 = 6.94980695%
GBPUSD,H1: >= 0.4 et < 0.8 = 18.53281853%
GBPUSD,H1: >= 0.0 et < 0.4 = 23.16602317%
GBPUSD,H1: >= -0.4 et < 0.0 = 29.34362934%
GBPUSD,H1: >= -0.8 et < -0.4 = 15.05791506%
GBPUSD,H1: >= -1.2 et < -0.8 = 5.01930502%
GBPUSD,H1: >= -1.6 et < -1.2 = 1.15830116%
GBPUSD,H1: >= -2.0 et < -1.6 = 0.00000000%
GBPUSD,H1: >= -2.4 et < -2.0 = 0.38610039%
```

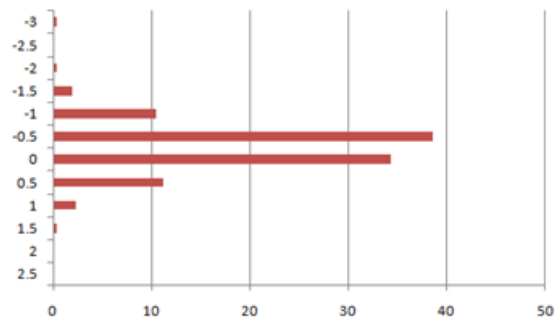

Figure 48 : Fréquences relatives

Si l'on se base sur les fréquences obtenues et le graphique ci-dessus, on remarque que les variations de prix de la paire GBPUSD suivent apparement une loi normale.

34
Les indicateurs

Nous allons maintenant voir plus en détails les particularités et comment faire pour créer vos propres indicateurs personnalisés.

Les indicateurs personnalisés

Les indicateurs sont des programmes permettant d'afficher graphiquement le résultat de calculs effectués par ce même indicateur. Vous trouverez deux catégories d'indicateurs dans MetaTrader : les indicateurs techniques (fournis par défaut avec la plateforme) et les indicateurs personnalisés (créés par vous-même ou d'autres programmeurs). Nous allons pour l'instant nous concentrer sur la deuxième catégorie.

La première notion à retenir lorsque vous voulez créer vos propres indicateurs est celle de mémoire tampon (buffer en anglais). Cette mémoire est en fait une zone de mémoire vive que le code de l'indicateur utilisera pour stocker temporairement des données et les transmettre à une autre entité (dans le cas qui nous occupe ici, le transfert se fait de l'indicateur vers la plateforme). Un indicateur personnalisé peut avoir au maximum 8 mémoires tampons. Référez-vous à la figure ci-dessous pour mieux comprendre l'utilité et le fonctionnement d'une mémoire tampon.

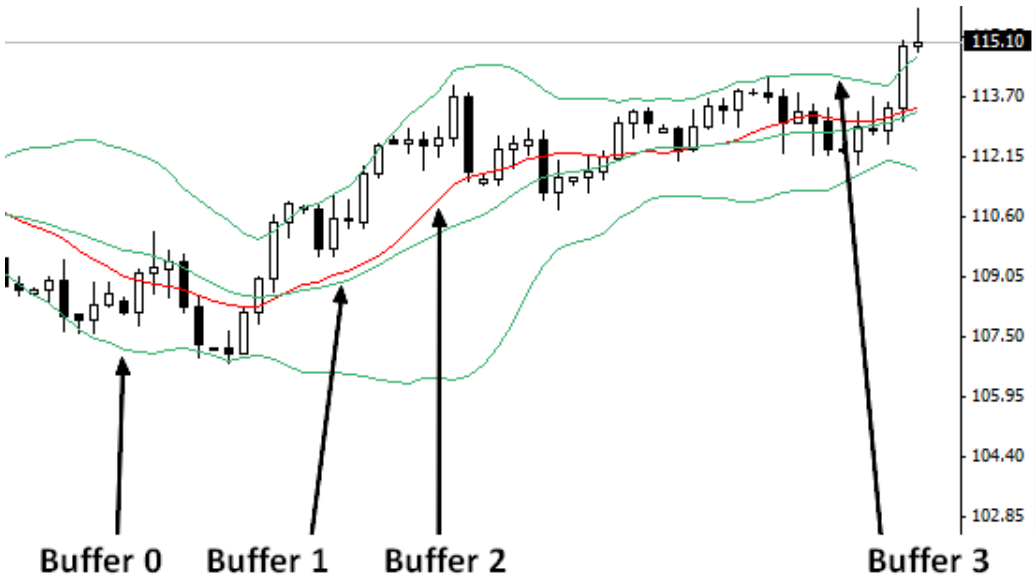

Figure 49 : Mémoires tampons

Comme vous pouvez le voir sur la figure 48, chacune des lignes affichées sur le graphique correspond à une mémoire tampon. L'indexation des mémoires tampons commence à 0 et

termine par conséquent à 7 (8 – 1). L'indicateur va effectuer les calculs requis puis enverra le résultat des calculs dans les différentes mémoires tampons. La plateforme prendra ensuite le relais en récupérant ces valeurs et en affichant les valeurs sur le graphique sous forme de lignes, points ou autres.

Nous allons maintenant créer un indicateur qui nous servira pour notre premier expert consultant afin que vous puissiez voir comment intégrer les deux types de programmes en MQL4.

Cet indicateur affichera dans la fenêtre graphique principale une ligne représentant le niveau de point pivot pour chaque période de 4 heures (soit six points pivots pour une journée) selon la formule suivante :

(Plus haut prix + Plus bas prix + Clôture) / 3 = Point Pivot

Si vous créez un nouvel indicateur en utilisant le menu *New => Custom Indicator* de MetaTrader, vous devriez obtenir un code similaire à celui-ci-dessous.

```
//+------------------------------------------------------------------+
//|                                          Premier Indicateur.mq4  |
//|                                        Copyright © 2011,         |
//|                                     http://www. eole-trading.com |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www. eole-trading.com"

#property indicator_chart_window
//+------------------------------------------------------------------+
//| Custom indicator initialization fonction                         |
//+------------------------------------------------------------------+
int init()
  {
//---- indicators
//----
   return(0);
  }
//+------------------------------------------------------------------+
//| Custom indicator deinitialization fonction                       |
//+------------------------------------------------------------------+
int deinit()
  {
//----


//----

  return(0);
  }
//+------------------------------------------------------------------+
//| Custom indicator iteration fonction                              |
//+------------------------------------------------------------------+
int start()
  {
   int    counted_bars=IndicatorCounted();
//----

//----
   return(0);
  }
//+------------------------------------------------------------------+
```

En premier lieu, nous allons définir si nous désirons afficher les éléments graphiques dans la fenêtre principale ou dans des fenêtres séparées.

Figure 50 : Affichage d'un indicateur personnalisé (0 = fenêtre principale; 1,2 = fenêtre séparée)

Le code déjà présent définit que l'affichage se fera dans la fenêtre principale.

```
#property indicator_chart_window
```

Pour que l'affichage se fasse dans une fenêtre séparée, il aurait fallu indiquer le code ci-dessous :

```
#property indicator_separate_window
```

Nous allons maintenant ajouter au début du programme quelques lignes de code. Nous avons besoin d'une mémoire tampon et la ligne représentant le point pivot sera affichée en bleu avec une épaisseur de 2. Enfin, nous déclarons le tableau qui nous servira à stocker les différentes valeurs au cours du temps du point pivot.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_width1 2

extern color cCouleur = Blue;
double tPointPivot[];
```

Vous pouvez consulter les différentes options pour la directive *#property* pour les indicateurs dans le chapitre 8.

La deuxième étape pour la creation de l'indicateur consiste à indiquer le format de l'affichage dans la fonction *inti nit()*.

```
SetIndexStyle(0, DRAW_LINE, EMPTY, EMPTY, cCouleur);
SetIndexBuffer(0, tPointPivot);
```

Dans le code ci-dessus, nous avons indiqués que nous désirons que les valeurs s'affichent sous la forme d'une ligne et que les valeurs utilisées soient celle correspondante à la mémoire tampon indexée au numéro 0 qui prendra ses valeurs du tableau *tPointPivot[]* déclaré précédemment.

Avant d'aller plus loin, voici les différentes fonctions pour les indicateurs personnalisés.

Fonction void IndicatorBuffers()

Cette fonction permet d'allouer de la mémoire pour les mémoires tampons utilisées par l'indicateur. En fait, cette fonction vous permet de spécifier le nombre total de mémoire tampon que l'indicateur devra utiliser (il est parfois nécessaire d'avoir plus de mémoire tampon que de lignes affichées car certaines d'entre elles sont uniquement utilisées pour les calculs). La syntaxe de la fonction est la suivante :

```
void IndicatorBuffers(int iDecompte)

// int iDecompte permet d'indiquer le nombre de mémoires tampons à allouer
```

Fonction int IndicatorCounted()

Cette fonction renvoie le nombre de barres qui n'ont pas changé depuis la dernière exécution de l'indicateur. De cette façon, l'indicateur ne perd pas de temps à recalculer des barres pour lesquelles le calcul a déjà été effectué. La syntaxe de la fonction est la suivante :

```
int IndicatorBuffers ()
```

Fonction void IndicatorDigits()

Cette fonction permet de paramétrer le niveau de précision en termes de décimales pour les valeurs de l'indicateur. La syntaxe de la fonction est la suivante :

```
void IndicatorDigits(int iDecimales)

// iDecimales est à remplacer par le nombre de décimales que vous souhaitez voir apparaître après le point
```

Fonction void IndicatorShortName ()

Cette fonction permet de définir un nom pour votre indicateur. Le nom sera affiché en haut à gauche de la fenêtre lorsque l'indicateur est situé dans une fenêtre séparée ou dans la fenêtre de données lorsque l'indicateur est placé dans la fenêtre principale. La syntaxe de la fonction est la suivante :

```
void IndicatorShortName(string sNom)

// sNom est à remplacer par le nom que vous souhaitez donner à l'indicateur
```

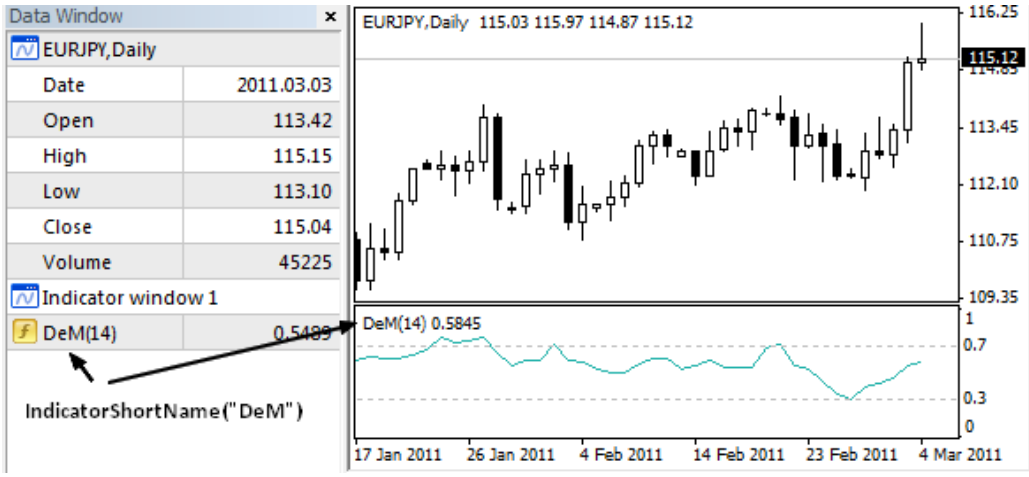

Figure 51 : Localisation de l'affichage pour la fonction IndicatorShortname()

Fonction void SetIndexStyle()

Cette fonction vous permet de choisir le type de ligne et le style associé pour l'affichage de l'indicateur. La syntaxe de la fonction est la suivante :

```
void SetIndexStyle( int iIndex, int iType, int iStyle = EMPTY, int iLargeur = EMPTY, color cCouleur = CLR_NONE)
```

Vous devez indiquer en premier lieu le numéro d'indexation de la mémoire tampon. Le deuxième paramètre correspond au type d'affichage désiré. Les options disponibles sont résumées dans le tableau suivant :

Constante	Valeur numérique	Description
DRAW_LINE	0	Affichage sous forme d'une ligne
DRAW_SECTION	1	Affichage sous forme d'un segment
DRAW_HISTOGRAM	2	Affichage sous forme d'histogramme
DRAW_ARROW	3	Affichage sous forme de flèche
DRAW_ZIGZAG	4	Affichage sous forme de segment entre une mémoire tampon pair et impair
DRAW_NONE	12	Aucun affichage

Tableau 29 : Options disponibles pour l'affichage des indicateurs personnalisés

Les autres paramètres ne sont pas obligatoires et sont par défaut paramétrés à 0. Le paramètre *int iStyle* permet de préciser un style d'affichage pour le type choisi précédemment. Les options disponibles sont les suivantes :

Constante	Valeur numérique	Description
STYLE_SOLID	0	Ligne continue
STYLE_DASH	1	Ligne en tiret
STYLE_DOT	2	Ligne en pointillés
STYLE_DASHDOT	3	Ligne alterne entre tirets et pointillés
STYLE_DASHDOTDOT	4	Ligne alterne entre tirets et double pointillés

Tableau 30 : Options disponibles pour le style de l'affichage des indicateurs personnalisés

Le paramètre suivant, *int iLargeur*, par défaut à 1 permet de choisir la largeur de la ligne à afficher.

Si vous paramétrez *int iLargeur* à 0, cela revient à choisir *DRAW_NONE* et aucun affichage ne sera effectué.

Le dernier paramètre vous permet de choisir la couleur au cas où vous n'auriez pas choisi celle-ci lors de la déclaration des propriétés au début du programme.

Fonction void SetIndexArrow()

Cette fonction permet de choisir un type de flèche lorsque *DRAW_ARROW* est choisi dans les paramètres de l'affichage. La syntaxe de la fonction est la suivante :

```
void SetIndexArrow(int iIndex, int iCode)
```

iIndex sert à indiquer l'indexation de la mémoire tampon correspondante et *iCode* correspond au code du type de flèche que vous souhaitez utiliser. Les différentes options sont les suivantes :

Constante	Valeur numérique	Description
	1	↱
	2	↳
	3	◀
	4	–
	5	Étiquette de prix orientée à gauche
	6	Étiquette de prix orientée à droite
SYMBOL_THUMBSUP	67	👍
SYMBOL_THUMBSDOWN	68	👎
SYMBOL_ARROWUP	241	⇧
SYMBOL_ARROWDOWN	242	⇩

SYMBOL_STOPSIGN	251	✗
SYMBOL_CHECKSIGN	252	✓

Tableau 31 : Options disponibles pour le style des flèches

Fonction bool SetIndexBuffer()

Cette fonction permet de paramétrer quel numéro d'indexation de mémoire tampon correspondra à quel tableau. La syntaxe de la fonction est la suivante :

```
bool SetIndexBuffer( int iIndex, double tTableau[])
```

Il suffit d'indiquer le numéro d'indexation de la mémoire tampon à utiliser (de 0 à 7) et lui assigner le tableau qui devra stocker les valeurs.

Fonction void SetIndexDrawBegin()

Cette fonction permet d'indiquer à la plateforme à partir de quelle barre vous souhaitez commencer à afficher l'indicateur. Par défaut, l'indicateur utilise toutes les barres présentes sur le graphique. La syntaxe de la fonction est la suivante :

```
void SetIndexDrawBegin(int iIndex, int iDebut)
```

iIndex vous permet d'indiquer le numéro d'indexation de la mémoire tampon à utiliser (de 0 à 7) et *idebut* le numéro de la barre à partir de laquelle commencer l'affichage (toutes les barres à gauches de celle-ci ne seront pas prises en compte pour l'affichage).

N'oubliez pas la numérotation à l'envers : la barre en cours porte le numéro 0.

Fonction void SetIndexEmptyValue()

Cette fonction permet d'initialiser tous les paramètres de l'affichage à 0. Cette fonction est généralement utlisée dans la fonction spéciale *init()*. La syntaxe de la fonction est la suivante :

```
void SetIndexEmptyValue(int iIndex, 0.0)
```

Il suffit d'indiquer le numéro d'indexation de la mémoire tampon à utiliser (de 0 à 7).

Fonction void SetIndexLabel()

Cette fonction vous permet d'afficher une description pour votre indicateur. Deux cas de figures sont possibles : soit votre indicateur est dans une fenêtre séparée et la description sera affichée dans le coin supérieur gauche de la fenêtre. Soit votre indicateur est dans la fenêtre

principale et dans ce cas la description s'affichera lorsque le curseur de la souris est placé sur une des lignes de l'indicateur. La syntaxe de la fonction est la suivante :

```
void SetIndexLabel int iIndex, string sTexte)
```

Il suffit d'indiquer le numéro d'indexation de la mémoire tampon à utiliser (de 0 à 7) et de choisir un nom ou une description sous forme d'une chaîne de caractères.

Fonction void SetIndexShift()

Cette fonction permet de décaler l'affichage de l'indicateur d'un certain nombre de barres. La syntaxe de la fonction est la suivante :

```
void SetIndexShift(int iIndex, int iDecalage)
```

Il suffit d'indiquer le numéro d'indexation de la mémoire tampon à utiliser (de 0 à 7) et de préciser le nombre de barres à partir de la barre actuelle pour le décalage. Si vous indiquez un nombre négatif, le décalage se fera dans le passé.

Fonction void SetLevelValue()

Si vous choisissez d'afficher l'indicateur dans une fenêtre séparée, cette fonction permet d'afficher des lignes constantes horizontales supplémentaires (très utile pour visualiser des niveaux clés). La syntaxe de la fonction est la suivante :

```
void SetLevelValue(int iNiveau, double dValeur)
```

iNiveau vous permet de définir dans quelle fenêtre séparée le niveau devra être affiché (le nombre maximum de fenêtres possible dans MetaTrader est 32). *dvaleur* est à remplacer par la position sur l'axe vertical de la ligne (il peut s'agir d'un prix ou d'une autre échelle dépendant de l'indicateur).

Fonction void SetLevelStyle()

Cette fonction permet de choisir le style des lignes constantes horizontales. La syntaxe de la fonction est la suivante :

```
void SetLevelStyle(int iStyle, int iLargeur, color cCouleur = CLR_NONE)
```

Pour choisir le style de la ligne, vous pouvez utiliser les options du tableau 49. La largeur de la ligne devra être comprise entre 1 et 5 et le dernier paramètre vous permet de choisir la couleur de la ligne.

Revenons maintenant au code de notre indicateur. La première ligne présente dans la fonction spéciale *start()* par défaut lorsque vous demandez à MetaEditor de créer un indicateur personnalisé est la suivante :

```
int counted_bars=IndicatorCounted();
```

Pour plus de facilité, nous transformerons le nom de la variable comme ci-dessous mais comme nous l'avons vu précédemment, la fonction *IndicatorCounted()* renvoie le nombre de barres inchangées depuis la dernière exécution de l'indicateur et évite ainsi que ce dernier n'effectue à nouveau les mêmes calculs. La ligne de code modifiée donnera donc :

```
int iBarresComptabilisees = IndicatorCounted();
```

Une fois que nous savons le nombre de barre inchangées, nous ajoutons le code suivant pour refaire les calculs pour la dernière barre par sécurité.

```
if(iBarresComptabilisees > 0)
  iBarresComptabilisees--;
```

L'étape suivante consiste à savoir combien de barres n'ont pas encore été traitées par notre indicateur. Pour cela, nous allons demander au programme de calculer la différence entre le nombre de barres présente sur le graphique à l'aide de la fonction *Bars* et notre décompte obtenu antérieurement contenu dans la variable *iBarresComptabilisees*.

```
int iBarresCalculees = Bars - iBarresComptabilisees;
```

Il ne reste plus qu'à demander à notre indicateur d'effectuer les calculs nécessaires pour connaître le point pivot correspondant à chaque session. Le code ci-dessous permettra d'effectuer ces calculs.

```
for(int iCompteur = iBarresCalculees; iCompteur >= 0; iCompteur--)
  {
  if(Time[iCompteur] % 14400 == 0)
    tPointPivot[iCompteur] = (High[iHighest(Symbol(), 0, MODE_HIGH, 4, iCompteur)] +
    Low[iLowest(Symbol(), 0, MODE_HIGH, 4, iCompteur)] + Close[iCompteur]) / 3;
  else
    tPointPivot[iCompteur] = tPointPivot[iCompteur + 1];
  }
```

La boucle *for* est en charge d'effectuer les calculs pour les barres du passé, actuelles et celles à venir. L'instruction conditionnelle *if* permet de s'assurer que le point pivot ne sera calculé que toutes les quatre heures grâce à l'opérateur *%* qui permet de vérifier que le reste de la division est 0. En effet, comme nous voulons effectuer le calcul que toutes les quatre heures, il faut s'assurer que l'indicateur n'effectue les calculs qu'aux heures 0, 4, 8, 12, 16 et 20. En divisant l'équivalent en secondes de ces heures (fourni par la fonction *Time[]*) par 14400 (équivalent de 4 heures en secondes) et en vérifiant que le reste de la division soit bien 0, nous serons sûr qu'il s'agit bien d'une chandelle correspondant aux heures souhaitées.

Cette indicateur assume que le graphique est paramétré sur H1 comme unité temporelle. L'indicateur n'est pas utilisable sur des unités de temps supérieures à H4 car les calculs seraient faussés.

Finalement, nous attribuons la valeur trouvée pour le point pivot à la case correspondante du tableau *tPointPivot*. C'est ce tableau qui sera en fin de compte affiché sur notre graphique comme vous pouvez le voir sur la figure suivante.

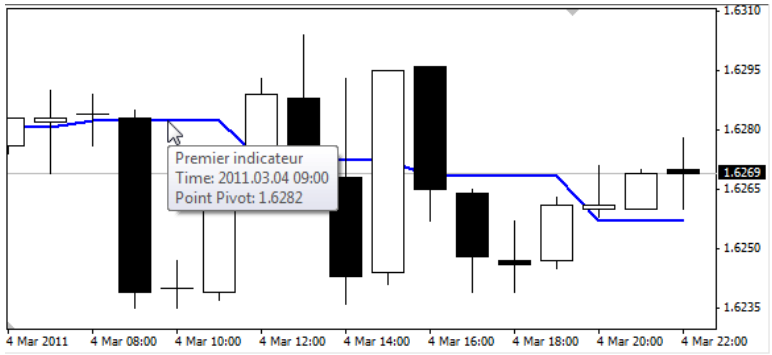

Figure 52 : Apparence de l'indicateur « Premier indicateur »

Le code complet de cet indicateur est disponible ci-dessous.

```
//+------------------------------------------------------------------+
//|                                  Premier Indicateur.mq4          |
//|                                  Copyright © 2011,               |
//|                                http://www. eole-trading.com      |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www. eole-trading.com"


#property indicator_chart_window
#property indicator_buffers 1
#property indicator_width1 2

extern color cCouleur = Blue;
double tPointPivot[];

//+------------------------------------------------------------------+
//| Custom indicator initialization fonction                         |
//+------------------------------------------------------------------+
int init()
  {
//---- indicators


SetIndexStyle(0, DRAW_LINE, EMPTY, EMPTY, cCouleur);
SetIndexBuffer(0, tPointPivot);


//----
   return(0);
  }
//+------------------------------------------------------------------+
//| Custom indicator deinitialization fonction                       |
//+------------------------------------------------------------------+
int deinit()
  {
//----


//---
   return(0);
  }

//+------------------------------------------------------------------+
//| Custom indicator iteration fonction                              |
//+------------------------------------------------------------------+
int start()
  {
   int iBarresComptabilisees = IndicatorCounted();
```

```
//----
   if(iBarresComptabilisees > 0)
     iBarresComptabilisees--;

    int iBarresCalculees = Bars - iBarresComptabilisees;


   for(int iCompteur = iBarresCalculees; iCompteur >= 0; iCompteur--)
     {
     if(Time[iCompteur] % 14400 == 0)
       tPointPivot[iCompteur] = (High[iHighest(Symbol(), 0, MODE_HIGH, 4, iCompteur)] +
       Low[iLowest(Symbol(), 0, MODE_HIGH, 4, iCompteur)] + Close[iCompteur]) / 3;
     else tPointPivot[iCompteur] = tPointPivot[iCompteur + 1];
     }

   return(0);
  }
//+------------------------------------------------------------------+
```

Intégration indicateur-expert

Maintenant que nous avons vu les concepts de bases et comment créer un indicateur, nous allons voir comment faire pour intégrer et utiliser un indicateur au sein d'un expert consultant. De nouveau, plusieurs options s'offrent à nous, soit utiliser un indicateur technique présent par défaut dans MetaTrader ou utiliser un indicateur personnalisé. Dans le premier cas, chaque indicateur possède sa propre fonction. Nous n'allons détailler que la fonction correspondante aux moyennes mobiles afin que vous puissiez comprendre la syntaxe de ces fonctions qui sont toutes plus ou moins identiques (vous pouvez vous référer à l'aide de MetaTrader pour la liste de toutes les autres fonctions).

Supposons que dans votre expert, vous ayez besoin de connaître la valeur d'une moyenne mobile, la syntaxe de la fonction permettant de renvoyer cette valeur est la suivante :

```
double iMA(string sPaire, int iUniteTemps,  int iPeriode, int iDecalageMa, int iMethodeMa, int iTypePrix, int
iDecalage)


// sPaire est à remplacer par la paire sur laquelle vous souhaitez obtenir l'information
// iUniteTemps est à remplacer par l'unité de temps désirée (voir tableau 13)
// iPeriode est à remplacer par le nombre de barres à prendre en compte pour le calcul
// iDecalageMa permet de choisir un décalage horizontale de la ligne représentant la moyenne mobile
// iMethodeMa permet de définir la méthode de calcul pour la moyenne mobile (voir tableau 29)
// iTypePrix permet de choisir le prix à prendre en compte pour le calcul (voir tableau 30 )
// iDecalage permet de définir le décalage à partir de la barre actuelle pour le calcul de la moyenne mobile
```

Constante	Valeur numérique	Description
MODE_SMA	0	Moyenne mobile simple
MODE_EMA	1	Moyenne mobile exponentielle
MODE_SMMA	2	Moyenne mobile lissée
MODE_LWMA	3	Moyenne mobile linéaire pondérée

Tableau 32 : Liste des options pour le calcul de l'indicateur moyenne mobile (MM)

Constante	Valeur numérique	Description
PRICE_CLOSE	0	Prix de fermeture
PRICE_OPEN	1	Prix d'ouverture
PRICE_HIGH	2	Plus haut prix
PRICE_LOW	3	Plus bas prix
PRICE_MEDIAN	4	(Plus haut + Plus bas) / 2
PRICE_TYPICAL	5	(Plus haut + Plus bas + Prix fermeture) / 3
PRICE_WEIGHTED	6	(Plus haut + Plus bas + 2*Prix fermeture) / 4

Tableau 33 : Liste des options pour le prix utilisé pour le calcul de l'indicateur MM

Un exemple d'utilisation de la fonction pourrait être :

```
double dValeurMa = iMA(Symbol(), 0, 14, 8, MODE_SMA, PRICE_CLOSE, 1);
```

Dans le cas des indicateurs personnalisés, la syntaxe de la fonction pour l'intégration est :

```
double iCustom(string sPaire, int iUniteTemps, string sNom, ..., int iIndex, int iDecalage)

// sPaire est à remplacer par la paire sur laquelle vous souhaitez obtenir l'information
// iUniteTemps est à remplacer par l'unité de temps désirée (voir tableau 13)
// sNom est à remplacer par le nom exact de l'indicateur
// iIndex est à remplacer par le numéro d'indexation de la mémoire tampon de l'indicateur (de 0 à 7)
// iDecalage permet de définir le décalage à partir de la barre actuelle pour le calcul de la moyenne mobile
```

À la place des points (…), vous devez inclure les paramètres externes de l'indicateur en séparant chacun des paramètres par une virgule.

Voici comment se ferait l'intégration de l'indicateur que nous avons créé précédemment :

```
double dValeurPointPivot = iCustom(Symbol(), "Premier indicateur", Blue, 0, 0);
```

Pour pouvoir obtenir la valeur d'un indicateur via la fonction *iCustom*, il est nécessaire que l'indicateur soit compilé et présent dans le dossier *experts/indicators* de l'installation de la plateforme.

35
Les fonctions mathématiques

Étant donné que les indicateurs servent le plus souvent à effectuer des calculs, nous avons jugé nécessaire de placer à la suite du chapitre sur les indicateurs la liste de toutes les fonctions mathématiques disponibles dans MetaTrader.

Syntaxe	Résultat
double MathAbs(x)	Valeur absolue de x
double MathArccos(x)	Arc cosinus de x
double MathArcsin(x)	Arc sinus de x
double MathArctan(x)	Arc tangente de x
double MathCeil(x)	Renvoie le plus petit entier qui est égal ou plus grand que x
double MathCos(x)	Cosinus de l'angle x
double MathExp(x)	Renvoie le résultat du calcul e^x
double MathFloor(x)	Renvoie le plus grand entier qui est égal ou plus petit que x
double MathLog(x)	Renvoie le logarithme naturel de x
double MathMax(x, y)	Renvoie la valeur la plus grande entre x et y
double MathMin(x, y)	Renvoie la valeur la plus petite entre x et y
double MathMod(x, y)	Renvoie le reste z de la division x / y tel que x = i * y + f où i est un entier
double MathPow(x, y)	Renvoie le résultat de x^y
int MathRand()	Renvoie un entier aléatoire entre 0 et 32676 (nécessite MathSrand() au préalable pour fonctionner)
void MathSrand(x)	Fonction de base pour générer un entier aléatoire avec la fonction MathRand(). x est à remplacer par n'importe quel nombre.
double MathRound(x)	Renvoie x arrondi au plus proche entier
double MathSin(x)	Sinus de l'angle x
double MathSqrt(x)	Renvoie la racine carrée de x
double MathTan(x)	Tangente de l'angle x

Tableau 34 : Liste des fonctions mathématiques

Pour toutes les fonctions du tableau 34, si le calcul est impossible, MetaTrader renverra la valeur *NaN* correspondante à valeur indéterminée. Dans certains cas, la valeur sera *INF* correspondante à infinie.

36
La gestion des fichiers

Nous avons vu dans un chapitre précédent qu'il était possible d'envoyer des fichiers sur un serveur FTP grâce à la fonction *SendFTP()*. Il est également possible de créer, modifier, importer ou extraire des données d'un fichier grâce à une série de fonctions que vous allez découvrir dans ce chapitre.

Localisation des fichiers

Pour qu'un expert puisse travailler (le verbe travailler est ici utilisé au sens large : créer, ouvrir, modifier, lire, etc.…) avec un fichier, ce fichier doit être placé dans des dossiers spécifiques.

Nous résumons ci-dessous les différents dossiers présents dans le dossier d'installation de la plateforme MetaTrader et la fonction des fichiers contenus dans le dossier. Pour le courtier ActivTrades, le chemin d'accès du dossier sera :

C:\Program Files\MetaTrader – ActivTrades

À l'intérieur de ce dossier, nous aurons :

...**\experts** : contient les fichiers sources ou exécutables des experts consultants.

...**\experts\indicators** : contient les fichiers sources ou exécutables des indicateurs.

...**\experts\include** : contient les fichiers avec l'extension .mqh.

...**\experts\libraries** : contient les librairies et fichiers DLLs.

...**\experts\scripts** : contient les fichiers sources ou exécutables des scripts.

...**\experts\templates** : contient les modèles pour les graphiques MetaTrader.

...**\experts\logs** : contient les journaux d'activité des experts consultants.

...**\experts\presets** : contient les fichiers de préréglages pour les experts consultants.

...**\experts\files** : contient les fichiers utilisés par les experts consultants.

...**\history\Nom du Courtier** : contient les historiques.

...**\tester\files** : contient les fichiers utilisés par les experts consultants lorsqu'ils sont exécutés dans le testeur de stratégie.

Si le fichier n'est pas dans le dossier adéquat, MetaTrader sera incapable de travailler avec le dit fichier.

Fonction de gestion des fichiers

En MQL4, ouvrir un fichier lorsque celui n'existe pas revient au même que créer ce dernier. Voilà pourquoi il n'existe pas de fonction de création mais uniquement une fonction pour ouvrir un fichier.

Fonction int FileOpen()

Cette fonction permet d'ouvrir un fichier que ce soit pour en extraire des informations ou pour écrire dans le fichier. Si le fichier est ouvert ou créé avec succès, la fonction renverra un numéro entier unique correspondant au fichier. Dans le cas contraire, la fonction renverra -1. La fonction *FileOpen()* ne peut ouvrir ou créer des fichiers que dans le dossier *\experts\files* ou *\tester\files* lorsque l'expert est exécuté dans le testeur de stratégie. La syntaxe de la fonction est la suivante :

```
int FileOpen(string sNomFichier, int iMode, int iSeparateur = ';')

// sNomFichier est à remplacer par le non complet du fichier incluant son extension
// iMode permet de choisir la raison pour laquelle vous ouvrez ou créez le fichier (voir tableau 32)
// iSeparateur indique au programme le séparateur pour les fichiers de type *.csv (par défaut « ; »)
```

Mode	Description
FILE_BIN|FILE_READ	Ouvre un fichier binaire
FILE_CSV|FILE_READ	Ouvre un fichier csv
FILE_BIN|FILE_WRITE	Efface le contenu et écrit dans un fichier binaire
FILE_CSV|FILE_WRITE	Efface le contenu et écrit dans un fichier csv
FILE_BIN|FILE_READ|FILE_WRITE	Ouvre et écrit dans un fichier binaire sans effacer le contenu déjà présent
FILE_CSV|FILE_READ|FILE_WRITE	Ouvre et écrit dans un fichier csv sans effacer le contenu déjà présent

Tableau 35 : Modes disponibles pour la fonction FileOpen()

Vous ne pouvez pas ouvrir simultanément plus de 32 fichiers dans MetaTrader avec le même programme (avec deux experts, vous pourrez donc ouvrir 64 fichiers et ainsi de suite).

Pour séparer les données dans un fichier, MetaTrader utilise un séparateur dans les fichiers *.csv. Par défaut, ce séparateur est le point virgule « ; » mais vous pouvez également choisir le caractère de votre choix. Pour aller à la ligne, le caractère est *\r\n*.

Fonction int FileOpenHistory()

Cette fonction permet d'ouvrir les fichiers de type *.HST présents dans le dossier ou un sous-dossier de *\history\Nom du Courtier*. En cas d'ouverture avec succès, la fonction renvoie un

numéro entier unique correspondant au fichier ou -1 dans le cas contraire. La syntaxe de la fonction est la suivante :

```
int FileOpenHistory(string sNomFichier, int iMode, int iSeparateur = ';')

// sNomFichier est à remplacer par le non complet du fichier incluant son extension
// iMode permet de choisir la raison pour laquelle vous ouvrez ou créez le fichier (voir tableau 32)
// iSeparateur indique au programme le séparateur pour les fichiers de type *.csv (par défaut « ; »)
```

Fonction int FileReadArray(), double FileReadDouble(), int FileReadInteger(), double FileReadNumber() et string FileReadString()

Toutes ces fonctions permettent de lire les informations d'un type spécifique contenues dans un fichier. Les syntaxes de ces fonctions sont les suivantes :

```
int FileReadArray(int iNumeroFichier, tTableau[], int iDebut, int iDecompte)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
// tTableau est le nom du tableau dans lequel seront écrites les données pour pouvoir les lire
// iDebut est la case à partir de laquelle seront écrites les données dans MetaTrader
// iDecompte est le nombre d'éléments qui devront être lus et copiés dans le tableau

double FileReadDouble(int iNumeroFichier, int itaille = DOUBLE_VALUE)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
// iTaille vous permet de choisir le format des réels (simple : FLOAT_VALUE ou double : DOUBLE_VALUE)

int FileReadInteger(int iNumeroFichier, int iTaille = LONG_VALUE)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
// iTaille vous permet de choisir le format des entiers (CHAR_VALUE, SHORT_VALUE ou LONG_VALUE)

double FileReadNumber(int iNumeroFichier)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie

string FileReadString(int iNumeroFichier, int iLongueur = 0)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
// iLongueur vous permet d'indiquer le nombre de caractères que la fonction devra lire (par défaut 0 qui
// correspond à la totalité du fichier)
```

FLOAT_VALUE : précision de 7 décimales
DOUBLE_VALUE : précision de 15 décimales

CHAR_VALUE : entiers de 0 à 65535 inclu
SHORT_VALUE : entiers de -32768 à 32767 inclu
LONG_VALUE : 9223372036854775808 à 9223372036854775807 inclu

Fonction int FileWrite(), int FileWriteArray(), int FileWriteDouble(), int FileWriteInteger() et int FileWriteString()

Toutes ces fonctions permettent d'écrire un certain type d'information dans un fichier. Les syntaxes de ces fonctions sont les suivantes :

Fonction pour écrire dans un fichier CSV :

```
int FileWrite(int iNumeroFichier, ....)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
// …. est à remplacer par les données que vous voulez écrire en séparant ces dernières par des virgules
```

Lorsque vous écrivez dans un fichier CSV, les données de type *int* et *double* sont automatiquement converties en chaînes de caractères. Les données de type *color, datetime* et *bool* ne seront pas converties et écrites telles quelles dans le fichier.

Fonction pour écrire les données contenues dans un tableau dans un fichier :

```
int FileWriteArray(int iNumeroFichier, tTbableau[], int iDebut, int iDecompte)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
// tTableau est le nom du tableau dont vous voulez écrire les données dans le fichier
// iDebut est la case du tableau à partir de laquelle les données seront copiées et écrites dans le fichier
// iDecompte est le nombre d'éléments qui devront être écrits
```

Fonction pour écrire une valeur réelle dans un fichier :

```
int FileWriteDouble(int iNumeroFichier, double dValeur, int iTaille = DOUBLE_VALUE)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
// dValeur est à remplacer par la valeur réel que vous voulez écrire dans le fichier
// iTaille vous permet de choisir le format des réels (simple : FLOAT_VALUE ou double : DOUBLE_VALUE)
```

Fonction pour écrire une valeur entière dans un fichier :

```
int FileWriteInteger(int iNumeroFichier, int iValeur, int iTaille = LONG_VALUE)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
// dValeur est à remplacer par la valeur entière que vous voulez écrire dans le fichier
// iTaille vous permet de choisir le format des entiers (CHAR_VALUE, SHORT_VALUE ou LONG_VALUE)
```

Fonction pour écrire une chaîne de caractères dans un fichier :

```
int FileWriteString(int iNumeroFichier, string sChaine, in iLongueur)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
// sChaine est à remplacer par la châine de caractère que vous voulez écrire dans le fichier
// iLongueur vous permet d'indiquer le nombre de caractères que la fonction devra écrire
```

Metatrader compte également les espaces et caractères spéciaux pour déterminer la longueur de la chaîne de caractères.

Fonction void FileClose()

Cette fonction permet de fermer un fichier ouvert à l'aide de la fonction *FileOpen()*. La syntaxe de la fonction est la suivante :

```
void FileClose(int iNumeroFichier)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
```

Fonction void FileDelete()

Cette fonction permet de supprimer un fichier présent dans le dossier **...***experts**files* ou un de ses sous-dossiers. La syntaxe de la fonction est la suivante :

```
void FileDelete(string sNomFichier)

// sNomFichier est à remplacer par le nom complet du fichier avec son extension que vous désirez supprimer
```

Fonction void FileFlush()

Cette fonction permet de supprimer toutes les données présentes dans la mémoire tampon du disque dur. Pour être effective, cette fonction devra être appelée entre les instructions de lecture et d'écriture de fichiers.

Les données présentes dans la mémoire tampon sont automatiquement supprimées lorsque la fonction *FileClose()* est appelée.

La syntaxe de la fonction est la suivante :

```
void FileFlush(int iNumeroFichier)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
```

Fonction bool FileIsEnding()

Cette fonction renvoie la valeur *True* lorsque le pointeur du fichier se trouve à la fin de ce dernier (le pointeur indique en fait la position précise à l'intérieur du fichier lors de la lecture ou écriture de celui-ci). Dans le cas contraire, la fonction renverra *False*. La syntaxe de la fonction est la suivante :

```
bool FileIsEnding(int iNumeroFichier)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
```

Fonction bool FileIsLineEnding()

Dans le cas de fichier de type CSV, cette fonction renvoie la valeur *True* lorsque le pointeur du fichier se trouve à la fin d'une ligne. Dans le cas contraire, la fonction renverra *False*. La syntaxe de la fonction est la suivante :

```
bool FileIsLineEnding(int iNumeroFichier)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
```

Fonction bool FileSeek()

Cette fonction permet de déplacer le pointeur du fichier à l'intérieur de ce dernier. De cette façon, la lecture ou écriture qui suivra ce déplacement se fera à partir de la nouvelle position. La syntaxe de la fonction est la suivante :

```
bool FileSeek(int iNumeroFichier, int iDecalage, int iOrigine)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
//  iDecalage correspond au décalage en octets que vous désirez appliquer
// iOrigine permet de définir le point d'origine pour le décalage (SEEK_CUR : position actuelle ; SEEK_SET :
// début du fichier ; SEEK_END : fin du fichier)
```

Fonction int FileSize()

Cette fonction permet d'obtenir la taille du fichier en octets. La syntaxe de la fonction est la suivante :

```
int FileSize(int iNumeroFichier)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
```

Fonction int FileTell()

Cette fonction permet d'obtenir la position actuelle du pointeur de fichier. La syntaxe de la fonction est la suivante :

```
int FileTell(int iNumeroFichier)

// iNumeroFichier correspond au numéro unique que la fonction FileOpen() renvoie
```

Exemples

Pour illustrer l'utilisation des fonctions que nous venons de voir, nous allons créer deux codes sources.

Si vous utilisez Windows 7, il vous faudra exécuter MetaTrader en mode administrateur sinon le programme ne pourra pas créer de fichiers.

Écriture de données diverses dans un fichier

Notre premier exemple consistera en une fonction illustrant comment vous pourriez écrire dans un fichier à chaque passage d'ordre pour conserver une série d'information.

```
void EcrireFichier(int iTicket)
 {
 OrderSelect(iTicket, SELECT_BY_TICKET);

 string sTypeOrdre;
 switch(OrderType())
  {
  case 0: sTypeOrdre = "Buy";
    break;
  case 1: sTypeOrdre = "Sell";
    break;
  case 2: sTypeOrdre = "Buy Limit";
    break;
  case 3: sTypeOrdre = "Sell Limit";
    break;
  case 4: sTypeOrdre = "Buy Stop";
    break;
  case 5: sTypeOrdre = "Sell Stop";
    break;
  }

 int iNumeroFichier = FileOpen("Transactions.txt", FILE_BIN|FILE_READ|FILE_WRITE);
 FileSeek(iNumeroFichier, 0, SEEK_END);

 FileWriteString(iNumeroFichier,"\r\n"+TimeDay(OrderOpenTime())+"/"+TimeMonth(OrderOpenTime())+"/"+
 TimeYear(OrderOpenTime())+" "+TimeHour(OrderOpenTime())+":"+TimeMinute(OrderOpenTime())+":"+
 TimeSeconds(OrderOpenTime())+" : "+sTypeOrdre+" @ "+ DoubleToStr(OrderOpenPrice(), 4), 40);

 FileClose(iNumeroFichier);
 }
```

Dans l'exemple ci-dessus, la fonction va écrire dans un fichier texte. Il suffit de placer un appel de cette fonction à la suite des instructions de passage d'ordre.

La première partie du code est composée de deux instructions : la première permettant d'ouvrir ou de créer le fichier. Pour ne pas effacer les données antérieures si le fichier existait déjà, l'ouverture du fichier s'effectue avec le mode suivant *FILE_BIN*|*FILE_READ*|*FILE_WRITE*.

```
int iNumeroFichier = FileOpen("Transactions.txt", FILE_BIN|FILE_READ|FILE_WRITE);
FileSeek(iNumeroFichier, 0, SEEK_END);
```

La deuxième instruction indique au programme que vous désirez écrire les données à la suite et à la ligne de la précédente et ce afin d'éviter de voir vos données se superposer.

L'instruction suivante sert à écrire les données choisies dans le fichier.

```
FileWriteString(iNumeroFichier,"\r\n"+TimeDay(OrderOpenTime())+"/"+TimeMonth(OrderOpenTime())+"/"+
TimeYear(OrderOpenTime())+" "+TimeHour(OrderOpenTime())+":"+TimeMinute(OrderOpenTime())+":"+
TimeSeconds(OrderOpenTime())+" : "+sTypeOrdre+" @ "+ DoubleToStr(OrderOpenPrice(), 4), 40);
```

Il aurait été possible de remplacer la fonction *FileWriteString()* par *FileWrite()* afin d'écrire dans un fichier CSV pour effectuer des analyses dans Excel par la suite par exemple. Dans son état actuel, la fonction permet d'obtenir un fichier similaire à la figure ci-dessous.

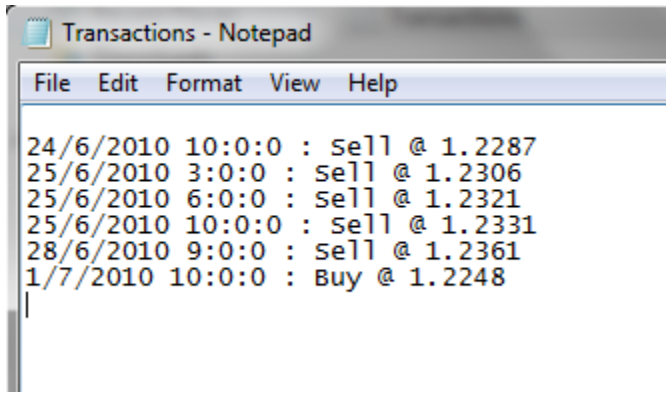

Figure 53 : Exemple d'écriture dans un fichier

Obtenir et utiliser un fichier depuis Internet

Dans cet exemple, nous allons illustrer comment il est possible de télécharger un fichier depuis MetaTrader et interpréter les données contenues dans ce dernier. À des fins utiles, nous allons utiliser un exemple souvent utilisé, celui d'un calendrier de nouvelles économiques.

Il existe différentes façons de télécharger un fichier depuis MetaTrader. Nous allons montrer une des plus simples en utilisant une DLL de windows. Par soucis de simplicité et comme elles ne font pas partie du langage MQL4, nous ne détaillerons pas ces fonctions. Sachez simplement que vous pouvez les utiliser telles quelles.

Le code ci-dessous est à inclure dans les directives au début du programme. Il s'agit de la fonction que MetaTrader utilisera pour télécharger le fichier.

```
#import "urlmon.dll"
  int URLDownloadToFileA(int pCaller, string szURL, string szFileName, int dwReserved, int lpfnCB);
#import
```

Il ne reste plus qu'à ajouter dans la fonction *start()* les instructions pour télécharger le fichier et définir les variables au début du programme.

```
string sAdresseUrlFichier;
string sDossierDestination;

int start()
  {
  sAdresseUrlFichier = "http://www.dailyfx.com/files/Calendar-03-27-2011.csv";
  sDossierDestination = "TerminalPath()+"\experts\files\Calendrier.csv";

  URLDownloadToFileA(0, sAdresseUrlFichier, sDossierDestination, 0, 0);
  return(0);
  }
```

Si nous exécutons le programme, le fichier « Calendrier.csv » sera créé dans le dossier indiqué. Si nous ouvrons le fichier obtenu, nous obtenons les données comme affichées dans la figure ci-dessous.

	A	B	C	D	E	F	G	H	I
1	Date	Time	Time Zone	Currency	Descriptic	Importanc	Actual	Forecast	Previous
2	Mon Mar 21	0:01	GMT	gbp	GBP Right	Medium	0.90%		0.30%
3	Mon Mar 21	0:01	GMT	gbp	GBP Right	Medium	0.80%		3.10%
4	Mon Mar 21	2:00	GMT	nzd	NZD Credi	Low	-0.30%		3.40%
5	Mon Mar 21	2:00	GMT	nzd	NZD Credi	Low	5.30%		5.50%
6	Mon Mar 21	8:00	GMT	chf	CHF Mone	Low	7.70%		6.80%
7	Mon Mar 21	12:30	GMT	usd	USD Chica	Medium	-0.04	-0.2	-0.01
8	Mon Mar 21	14:00	GMT	usd	USD Existi	High	4.88M	5.12M	5.40M
9	Mon Mar 21	14:00	GMT	usd	USD Existi	Medium	-9.60%	-4.50%	3.40%
10	Tue Mar 22	1:30	GMT	JPY	JPY BOJ Bc	LOW			
11	Tue Mar 22	4:00	GMT	jpy	JPY Tokyo	Low	24.90%		-13.50%
12	Tue Mar 22	4:30	GMT	jpy	JPY All Ind	Medium	2.90%	2.80%	-0.30%
13	Tue Mar 22	7:00	GMT	jpy	JPY Conve	Low	6.50%		5.10%

Figure 54 : Apparence du calendrier économique de DailyFX

Pour être parfait, le code précédent devrait inclure deux éléments. En premier lieu, des instructions pour éviter que le fichier ne soit téléchargé en permanence et en second lieu, des instructions permettant de paramétrer automatiquement le nom du fichier pour la semaine. En effet, si vous observez l'exemple, le calendrier utilisé provenant du site dailyfx.com inclut dans le nom du fichier la date de début de semaine pour identifier le fichier. Il suffirait donc d'extraire cette date pour la semaine en cours et déterminer ainsi le nom correct du fichier à télécharger pour chaque semaine.

L'étape suivante consiste à ouvrir le fichier.

```
int iNumeroFichier = FileOpen("Calendrier.csv",FILE_READ|FILE_CSV,',');
if(iNumeroFichier == -1)
  {
  Print("Le fichier n\'existe pas");
  return(0);
  }
```

Si le fichier n'existait pas, le programme renverrait au début de la fonction *start()* pour réessayer de télécharger ce dernier. Une fois le fichier ouvert, nous devons créer une boucle capable de lire toutes les données une à une, transformer la date sous forme de secondes afin que MetaTrader puisse l'interpréter et enfin créer des objets graphiques pour signaler les nouvelles économiques sur le graphique.

Pour ce faire, nous allons initialiser la boucle puis lire les données dans l'ordre figurant sur la figure 53.

```
int iCompteur = 0;
while(!FileIsEnding(iNumeroFichier))
  {
  string sDate = FileReadString(iNumeroFichier);
  string sTime = FileReadString(iNumeroFichier);
  string sTimeZone = FileReadString(iNumeroFichier);
  string sCurrency = FileReadString(iNumeroFichier);
  string sDescription = FileReadString(iNumeroFichier);
  string sImportance = FileReadString(iNumeroFichier);
  string sActual = FileReadString(iNumeroFichier);
```

```
 string sForecast = FileReadString(iNumeroFichier);
 string sPrevious = FileReadString(iNumeroFichier);

 // Le code que nous allons voir à la suite devra être inséré ici

 iCompteur++;
 }
```

Nous devons maintenant transformer la date sous forme de secondes.

```
string sSecondesEcoulees = Year()+"."+Month()+"."+StringSubstr(sDate, 8, 2)+" "+ sTime;

datetime dtDate = StrToTime(sSecondesEcoulees);
```

Nous allons créer une date sous la forme *AAAA.MM.JJ HH:MM* afin de pouvoir utiliser la fonction *StrToTime()* qui transformera cette dernière en secondes. La fonction *StringSubStr()* permet d'obtenir une portion d'une chaîne de caractères. Dans notre cas, nous désirions extraire la date en chiffres de la chaîne de caractères contenant le jour de la semaine, le mois et la date. Nous verrons ces fonctions dans un prochain chapitre.
Le reste du code servira à afficher les données sur notre graphique.

```
double dMoyenneBarresVisibles = (WindowPriceMax(0) + WindowPriceMin(0)) / 2;

color cImportance = White;
if(sImportance == "Low")
  cImportance = Yellow;
if(sImportance == "Medium")
  cImportance = Orange;
if(sImportance == "High")
  cImportance = Red;

ObjectCreate("TxtNouvelleEco"+iCompteur, OBJ_TEXT, 0, dtDate, dMoyenneBarresVisibles);
ObjectSet("TxtNouvelleEco "+iCompteur, OBJPROP_COLOR, cImportance);
ObjectSetText("TxtNouvelleEco "+iCompteur, sDate + " : "+ sDescription, 8);
ObjectSet("TxtNouvelleEco "+iCompteur, OBJPROP_ANGLE, 90);

ObjectCreate("LigneNouvelleEco "+iCompteur, OBJ_VLINE, 0, dtDate, dMoyenneBarresVisibles);
ObjectSet("LigneNouvelleEco "+iCompteur, OBJPROP_COLOR, cImportance);
ObjectSet("LigneNouvelleEco "+iCompteur, OBJPROP_STYLE, STYLE_DASH);
ObjectSet("LigneNouvelleEco "+iCompteur, OBJPROP_BACK, True);
ObjectSetText("LigneNouvelleEco "+iCompteur, sDescription, 8);
```

Le premier bloc sert à définir le milieu de l'écran en se basant uniquement sur les barres visibles. Le deuxième bloc sert à définir une couleur différente en fonction de l'importance de la nouvelle (la couleur blanche a été choisie pour les nouvelles neutres). Les deux derniers blocs servent à créer la ligne et le texte.

Le code au complet sera donc :

```
#import "urlmon.dll"
  int URLDownloadToFileA(int pCaller,string szURL,string szFileName,int dwReserved,int Callback);
#import

string sAdresseUrlFichier;
string sDossierDestination;
```

```
int start()
 {
 sAdresseUrlFichier="http://www.dailyfx.com/files/Calendar-03-27-2011.csv";
 sDossierDestination=TerminalPath()+"\experts\files\Calendrier.csv";

 URLDownloadToFileA(0, sAdresseUrlFichier, sDossierDestination, 0, 0);

 int iNumeroFichier = FileOpen("Calendrier.csv",FILE_READ|FILE_CSV,',');
 if(iNumeroFichier == -1)
  {
  Print("Le fichier n\'existe pas");
  return(0);
  }

 int iCompteur = 0;
 while(!FileIsEnding(iNumeroFichier))
  {
  string sDate = FileReadString(iNumeroFichier);
  string sTime = FileReadString(iNumeroFichier);
  string sTimeZone = FileReadString(iNumeroFichier);
  string sCurrency = FileReadString(iNumeroFichier);
  string sDescription = FileReadString(iNumeroFichier);
  string sImportance = FileReadString(iNumeroFichier);
  string sActual = FileReadString(iNumeroFichier);
  string sForecast = FileReadString(iNumeroFichier);
  string sPrevious = FileReadString(iNumeroFichier);

  string sSecondesEcoulees = Year()+"."+Month()+"."+StringSubstr(sDate, 8, 2)+" "+sTime;
  datetime dtDate = StrToTime(sSecondesEcoulees);

  double dMoyenneBarresVisibles = (WindowPriceMax(0) + WindowPriceMin(0)) / 2;

  color cImportance = White;
  if(sImportance == "Low")
    cImportance = Yellow;
  if(sImportance == "Medium")
    cImportance = Orange;
  if(sImportance == "High")
    cImportance = Red;

  ObjectCreate("TxtNouvelleEco "+iCompteur, OBJ_TEXT, 0, dtDate, dMoyenneBarresVisibles);
  ObjectSet("TxtNouvelleEco "+iCompteur, OBJPROP_COLOR, cImportance);
  ObjectSetText("TxtNouvelleEco "+iCompteur, sDescription, 8);
  ObjectSet("TxtNouvelleEco "+iCompteur, OBJPROP_ANGLE, 90);

  ObjectCreate("LigneNouvelleEco "+iCompteur, OBJ_VLINE, 0, dtDate, dMoyenneBarresVisibles);
  ObjectSet("LigneNouvelleEco "+iCompteur, OBJPROP_COLOR, cImportance);
  ObjectSet("LigneNouvelleEco "+iCompteur, OBJPROP_STYLE, STYLE_DASH);
  ObjectSet("LigneNouvelleEco "+iCompteur, OBJPROP_BACK, True);
  ObjectSetText("LigneNouvelleEco "+iCompteur, sDescription, 8);

  iCompteur++;
  }

 return(0);
 }
```

Le code suivant fonctionne parfaitement dans son état actuel, il est néanmoins important de noter qu'il est très loin d'être parfait et ne devrait par conséquent être utilisé qu'à des fins éducatives. Le résultat obtenu ressemblera à la figure ci-dessous :

Figure 55 : Affichage des nouvelles économiques à partir d'un fichier

37
Les fonctions sur les chaînes de caractères

Nous avons vu dans le chapitre précédent la fonction *StringSubstr()* permettant d'obtenir une portion spécifique d'une chaîne de caractères. Il existe sept autres fonctions spécifiques aux chaînes de caractères.

Fonction string StringConcatenate()

Dans le dernier exemple du chapitre sur la gestion des fichiers, nous avons utilisé le code suivant :

```
string sSecondesEcoulees = Year()+"."+Month()+"."+StringSubstr(sDate, 8, 2)+" "+sTime;
```

Bien que correct, le fait d'utiliser l'opérateur « + » peut parfois être très coûteux en termes de mémoire et/ou temps. Pour simplifier la transformation d'une suite de variables (un maximum de 64 variables) en chaînes de caractères, le langage MQL4 dispose de la fonction *StringConcatenate()*. La syntaxe de la fonction est la suivante :

```
string StringConcatenate(…)

// …. Est à remplacer par la suite de variables que vous souhaitez transformer en chaîne de caractères en
// séparant ces dernières par des virgules
```

Par exemple, si nous avions utilisé cette fonction pour notre première instruction, nous aurions donc :

```
string sSecondesEcoulees = StringConcatenate(Year(),".",Month(),".",StringSubstr(sDate, 8, 2)," ",sTime);
```

Fonction int StringFind()

Cette fonction permet de chercher une portion de chaîne de caractères (lettre, mot ou phrase) dans une chaîne de caractères. La fonction renverra 1 si la portion est trouvée ou -1 dans le cas contraire. La syntaxe de la fonction est la suivante :

```
int StringFind(string sChaineDeCaracteres, string sTexteCherche, int iDebut)

// sChaineDeCaracteres correspond à la chaîne de caractères dans laquelle vous désirez effectuer la
// recherche
// sTexteCherche correspond au caractère, mot ou phrase que vous désirez chercher
// iDebut indique la position dans la chaîne de caractères à partir de laquelle commencer la recherche
```

Cette fonction peut être utile pour identifier une des lignes dans un fichier. Par exemple, vérifier à chaque lecture de ligne dans un fichier CSV si le mois est bien le mois courant. Si nous reprenons notre dernier exemple du chapitre sur la gestion des fichiers, nous pourrions avoir :

```
string sDate = FileReadString(iNumeroFichier);

if(Month() == 3)
  string sMoisActuel = "MAR";

if(StringFind(sDate, sMoisActuel, 0) == 1)
  Print("Le mois sur le calendrier est le même que le mois actuel");
else
  Print("Les mois sont différents");
```

Fonction int StringGetChar()

Cette fonction permet d'obtenir le code décimal correspondant à un caractère spécifique d'une chaîne de caractères. La syntaxe de la fonction est la suivante :

```
int StringGetChar(string sChaineDeCaracteres, int iPosition)

// sChaineDeCaracteres correspond à la chaîne de caractères dans laquelle vous désirez effectuer la
// recherche
// iPosition correspond à la position du caractère dont vous désirez obtenir le code (le décompte commence à
// 0)
```

Si nous reprenons de nouveau l'exemple précédent, nous aurions donc, en utilisant le format de la date sur le calendrier (Mon Mar 21), à extraire et vérifier si le quatrième caractère de la date correspond au code associé au caractère M (77) :

```
string sDate = FileReadString(iNumeroFichier);

if(Month() == 3)
  string sMoisActuel = "MAR";

if(StringGetChar(sDate, 5) == 77 )
  Print("Le mois sur le calendrier est le même que le mois actuel");
else
  Print("Les mois sont différents");
```

Vous pouvez consulter la liste complète des codes pour chaque caractère à l'adresse suivante : http://www.asciitable.com

Fonction int StringLen()

Cette fonction permet d'obtenir le nombre de caractères d'une chaîne de caractères spécifique. Le décompte commence à 0 et inclut les espaces et caractères spéciaux. La syntaxe de la fonction est la suivante :

```
int StringLen(string sChaineDeCaracteres)

// sChaineDeCaracteres correspond à la chaîne de caractères dont vous désirez obtenir le nombre de
// caractères
```

Fonction string StringSetChar()

Cette fonction permet de modifier un caractère spécifique par un autre dans une chaîne de caractères. La syntaxe de la fonction est la suivante :

```
string StringSetChar(string sChaineDeCaracteres, int iPosition, int iNouveauCaractere)

// sChaineDeCaracteres correspond à la chaîne de caractères dans laquelle vous désirez changer un
// caractère
// iPosition est la position du caractère que vous désirez remplacer
// iNouveauCaractere est le code correspond au nouveau caractère
```

Un exemple pourrait être :

```
string sPhraseFaute = "Une louvelle position a été ouverte";

string sPhraseCorrecte =StringSetChar(sPhraseFaute, 4, 'n');
```

Fonction string StringSubstr()

Cette fonction permet d'obtenir une portion d'une chaîne de caractères en spécifiant la longueur désirée de la portion ainsi que la position de départ pour l'extraction. La syntaxe de la fonction est la suivante :

```
string StringSubstr(string sChaineDeCaracteres, int iDebut, int iLongueur)

// sChaineDeCaracteres correspond à la chaîne de caractères de laquelle vous désirez extraire une portion
// iDebut correspond à la position à partir de laquelle vous désirez commencer l'extraction de la portion
// iLongueur est le nombre de caractères que vous désirez extraire
```

Dans notre dernier exemple de gestion des fichiers, nous désirions extraire la date d'une chaîne de caractères similaire à la suivante : Mon Mar 21. Pour ce faire, le code serait donc :

```
string sDate = "Mon Mar 21";

string sJour = StringSubstr(sDate, 8, 2);
```

Fonction string StringTrimLeft() et string StringTrimRight()

Ces deux fonctions permettent de supprimer les passages à la ligne, espaces et tabulations dans une chaîne de caractères. La première fonction supprime ces derniers lorsqu'ils se présentent du côté gauche tandis que la deuxième fonction fait de même pour le côté droit. La syntaxe de la fonction est la suivante :

```
string StringTrimLeft(string sChaineDeCaracteres)
string StringTrimRight(string sChaineDeCaracteres)

// sChaineDeCaracteres correspond à la chaîne de caractères dont vous désirez effacer les passages à la
// ligne, espaces et tabulations
```

Un exemple pourrait être :

```
string sDate = "              Mon Mar 21                  ";

sDate = StringTrimLeft(sDate);
// sDate = "Mon Mar 21                  "

sDate = StringTrimRight(sDate);
// sDate = "Mon Mar 21"
```

38
Fonctions sur les fenêtres

Nous avons déjà vu certaines de ces fonctions mais nous allons maintenant les revoir et en ajouter de nouvelles. Référez-vous à la figure 49 pour mieux comprendre comment fonctionne la numérotation des fenêtres dans MetaTrader.

Fonction int Period()

Cette fonction permet de renvoyer l'unité de temps du graphique en minutes comme affichées dans la colonne « Forme numérique » du tableau 13. La syntaxe de la fonction est la suivante :

```
int Period()
```

Fonction bool RefreshRates()

Cette fonction permet de rafraîchir les données du graphique afin de s'assurer que le programme utilise bien les dernières données. La fonction renvoie *True* si les données sont rafraîchies avec succès et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool Refreshrates()
```

Fonction string sPaire()

La fonction *Symbol()* renvoie une chaîne de caractères correspondant au nom de la paire du graphique sur lequel est placé l'expert. Le nom de la paire sera toujours sous la même forme que celle affichée par la plateforme (par exemple : EURUSD ou EURUSDm dans le cas de compte micro). La syntaxe de la fonction est la suivante :

```
string sPaire()
```

Fonction int iFenetresTotal()

Cette fonction renvoie le nombre total de fenêtres figurant dans un graphique. La syntaxe de la fonction est la suivante :

```
int iFenetresTotal()
```

Fonction void HideTestIndicators()

Lorsque vous effectuez un test sur un expert consultant en mode visuel le ou les indicateurs sont généralement affichés à la fin sur le graphique. Cette fonction permet d'éviter de voir les indicateurs affichés. La syntaxe de la fonction est la suivante :

```
void HideTestIndicators(bool bChoix)

// bChoix est à remplacer par True si vous désirez cacher les indicateurs et False dans le cas contraire
```

Fonction int iFenetreBarsPerChart()

Cette fonction renvoie le nombre de barres visibles à l'écran. La syntaxe de la fonction est la suivante :

```
int iFenetreBarsPerChart()
```

Fonction string WindowExpertName()

Cette fonction renvoie le nom de l'expert consultant, indicateur ou script à partir duquel la fonction a été appelée. La syntaxe de la fonction est la suivante :

```
string WindowExpertName()
```

Fonction int iFenetreFind()

Cette fonction permet de savoir si un indicateur portant le nom *sNomIndicateur* est présent sur le graphique. Si c'est le cas, la fonction renverra le numéro d'index de la fenêtre contenant cet indicateur. Dans le cas contraire, la fonction renverra -1. La syntaxe de la fonction est la suivante :

```
int iFenetreFind(string sNomIndicateur)

// sNomIndicateur correspond au nom de l'indicateur que vous recherchez
```

Fonction int iFenetreFirstVisibleBar()

Cette fonction renvoie le numéro de position de la première barre visible du graphique. La syntaxe de la fonction est la suivante :

```
int iFenetreFirstVisibleBar()
```

Dans la figure ci-dessous, le cadre noir représente les barres visibles à l'écran. Si nous utilisions la fonction *WindowFirstVisibleBar()* dans ce cas, la fonction renverrait la valeur 9.

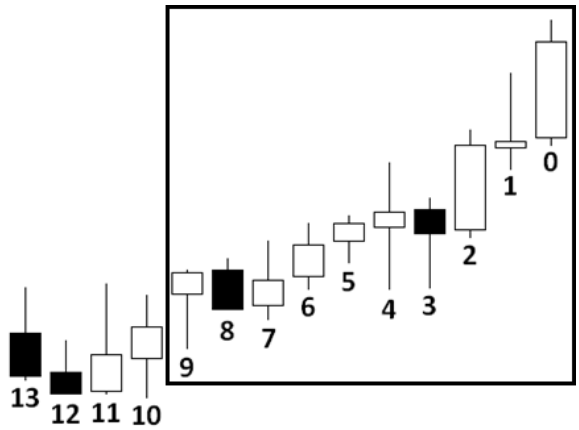

Figure 56 : Illustration de la fonction WindowFirstVisibleBar()

Fonction int iFenetreHandle()

Cette fonction permet de connaître le numéro d'identification de la fenêtre correspondante aux paramètres spécifiés (paire et unité de temps). Si aucune fenêtre ne correspond aux paramètres, la fonction renverra 0. La syntaxe de la fonction est la suivante :

```
int iFenetreHandle(string sPaire, int iUniteTemps)

// sPaire correspondant à la paire du graphique
// sUniteTemps correspond à l'unité de temps du graphique (voir tableau 13)
```

Fonction bool WindowIsVisible()

Cette fonction permet de savoir si la sous-fenêtre spécifiée est visible ou non. Si la sous-fenêtre est visible, la fonction renverra *True* et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool WindowIsVisible(int iIndex)

// iIndex correspond au numéro d'index de la sous-fenêtre
```

Fonction int iFenetreOnDropped()

Cette fonction renvoie le numéro d'index de la fenêtre sur laquelle l'expert consultant, l'indicateur ou le script a été placé. Cette fonction ne fonctionne que lorsque le programme a été déposé sur le graphique avec la souris. La syntaxe de la fonction est la suivante :

```
int iFenetreOnDropped()
```

Fonction double WindowPriceOnDropped(), datetime WindowTimeOnDropped(), int iFenetreXOnDropped() et int WindowYOnDropped()

Toutes ces fonctions permettent de savoir où sur le graphique un programme a été déposé. Ces fonctions ne renverront une valeur que lorsque le programme a été déposé sur le graphique avec la souris via une manœuvre de glisser-déposer. Les deux premières fonctions, *WindowPriceOnDropped()* et *WindowTimeOnDropped()* permettent d'obtenir le positionnement en fonction d'un prix et d'une date tandis que les deux autres fonctions, *WindowXOnDropped()* et *WindowYOnDropped()*, donnent la position en pixels en fonction de la fenêtre.

Notez que *WindowPriceOnDropped()* et *WindowTimeOnDropped()* ne fonctionnent pas dans le cas d'indicateurs personnalisés

Les syntaxes de ces fonctions sont les suivantes :

```
double WindowPriceOnDropped()
double WindowTimeOnDropped()

double WindowXOnDropped()
double WindowYOnDropped()
```

Fonction double WindowPriceMax() et double WindowPriceMin()

Ces deux fonctions permettent d'obtenir la valeur maximale et minimale de l'axe vertical d'une fenêtre spécifiée. Les syntaxes de ces fonctions sont les suivantes :

```
double WindowPriceMax(inti Index = 0)

double WindowPriceMin(inti Index = 0)

// iIndex correspond au numéro d'index de la fenêtre dont vous voulez connaître la valeur maximale de l'axe
// verticale (0 par défaut)
```

Fonction void WindowRedraw()

Cette fonction est utilisée pour forcer la réinitialisation d'un graphique. La syntaxe de la fonction est la suivante :

```
void WindowRedraw()
```

Fonction bool WindowScreenShot()

Cette fonction permet de sauvegarder une capture d'écran au format *.gif dans le répertoire …*experts**files* ou …*tester**files* (en cas de test). La fonction renvoie *True* lorsque la capture est effectuée avec succès et *False* dans le cas contraire. La syntaxe de la fonction est la suivante :

```
bool WindowScreenShot(string sNomFichier, int iTailleX, int iTailleY, int iBarreDebut = -1, int iEchelle = -1,
inti Mode = -1)

// sNomFichier est à remplacer par le nom que vous désirez donner au fichier (inclure l'extension *.gif)
// iTailleX est la hauteur en pixels de la capture
// iTailleY est la largeur en pixels de la capture
// iBarreDebut définit à partir de quelle barre le programme doit effectuer la capture (0 = première barre
// visible et valeur négative = tout le graphique (par défaut))
// iEchelle définit l'échelle horizontale du graphique pour la capture (entre 0 et 5). Si une valeur négative est
// spécifiée, l'échelle actuelle du graphique sera utilisée
// iMode permet de choisir entre 1 : CHART_BAR, 2 : CHART_CANDLE et 3 : CHART_LINE. Si une valeur
// négative est spécifiée, le mode actuelle du graphique sera utilisé
```

Par exemple :

```
WindowScreenShot("capture.gif ", 1024, 768, -1, -1, CHART_CANDLE);
```

39
Votre premier expert consultant #9

Nous allons maintenant compléter notre expert avec quelques unes des fonctionnalités que nous venons de voir. En premier lieu, nous allons ajouter un filtre basé sur l'indicateur personnalisé que nous avons créé.

Nous voulons néanmoins laisser le choix à l'utilisateur d'activer ou non le filtre. Pour ce faire nous allons donc ajouter une variable externe permettant de paramétrer cela (*True* = utiliser le filtre; *False* = ne pas utiliser le filtre)

```
extern bool bFiltrePointPivot = True;
```

Nous devons maintenant créer la fonction qui permettra d'appeler l'indicateur et d'interpréter les valeurs de ce dernier.

```
bool fFiltrePointPivot(bool bFiltrePointPivot, string sOperation)
 {
 double dValeurPointPivot = iCustom(Symbol(), 0, "Premier indicateur", Blue, 0, 0);

 if(bFiltrePointPivot == False)
   bool bFiltreAutorise = True;
 if(sOperation == "Achat" && bFiltrePointPivot == True)
   {
   if(Ask >= dValeurPointPivot)
     bFiltreAutorise = True;
   if(Ask < dValeurPointPivot)
     bFiltreAutorise = False;
   }


   if(sOperation == "Vente" && bFiltrePointPivot == True)
   {
   if(Bid <= dValeurPointPivot)
     bFiltreAutorise = True;
   if(Bid > dValeurPointPivot)
     bFiltreAutorise = False;
   }

 return(bFiltreAutorise);
 }
```

Les deux paramètres de la fonction sont la valeur associée à notre variable externe *bFiltrePointPivot* et une variable permettant de préciser le type d'opération dont il s'agit (achat ou vente) car la vérification à effectuer sera différente pour chacun des cas.

La première instruction de la fonction est l'appel de notre indicateur suivi par une boucle conditionnelle pour chacun de nos cas : pas d'utilisation du filtre, utilisation du filtre sur une position de type achat et finalement utilisation du filtre sur une position vente. À la fin de la fonction, la fonction *return()* renvoie une valeur qui autorise ou non le passage de l'ordre.

Maintenant que la fonction est créée, il ne reste plus qu'à incorporer un appel de fonction avant chaque passage d'ordre.

```
if((dPlusHaut + iDistance * fPoint(Symbol())) - Ask > (MarketInfo(Symbol(), MODE_STOPLEVEL)) *
fPoint(Symbol())  && fFiltrePointPivot(bFiltrePointPivot, "Achat") == True)
 {
 iTicket = OrderSend(Symbol(), OP_BUYSTOP, fLots(dPourcentageRisque, iStop), dPlusHaut + iDistance*
 fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la veille",
 iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600) + ((iMinutesFin - TimeMinute(Time[0]))*60)),
 Blue);
 fTicket(iTicket);
 fTicketVariableGloable("gvNumeroTicketAchat", iTicket);
 fErreur("Achat");
 }


if(Bid - (dPlusBas - iDistance* fPoint(Symbol())) > (MarketInfo(Symbol(), MODE_STOPLEVEL)) *
fPoint(Symbol()) && fFiltrePointPivot(bFiltrePointPivot, "Vente") == True)
 {
 iTicket = OrderSend(Symbol(), OP_SELLSTOP, fLots(dPourcentageRisque, iStop),  dPlusBas - iDistance*
 fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la veille",
 iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600)+((iMinutesFin - TimeMinute(Time[0]))*60)), Red);
 fTicket(iTicket);
 fTicketVariableGloable("gvNumeroTicketVente", iTicket);
 fErreur("Vente");
 }
```

Le deuxième ajout que nous ferons à notre expert sera un calendrier qui s'affiche sur le côté gauche du graphique et qui se met à jour automatiquement. Nous allons donc utiliser une partie du code que nous avons vu dans le deuxième exemple de la gestion des fichiers.

La première étape consiste à créer les variables que nous allons utiliser ainsi que la directive permettant de télécharger un fichier. Une variable externe sera créée pour permettre à l'utilisateur de choisir s'il désire afficher ou non les nouvelles.

```
#import "urlmon.dll"
  int URLDownloadToFileA(int pCaller,string szURL,string szFileName,int dwReserved,int Callback);
#import

extern bool bAffichageNouvelles = True;


int iDay = 0;
string sAdresseUrlFichier;
string sDossierDestination;
string sNomFichier;
```

Pour simplifier la lecture, nous allons créer une fonction qui contiendra tout le processus depuis le téléchargement du fichier jusqu'à l'affichage sur le graphique.

Le calendrier sera téléchargé depuis le site DailyFX. Le format du nom du fichier *.CSV est le suivant : Calendar-MM-JJ-AAAA.csv (MM = mois, JJ = jour et AAAA = année). La date figurant sur le nom du calendrier correspond à la date du dimanche avant la semaine dont nous voulons obtenir les nouvelles économiques. La première étape pour notre programme sera donc de déterminer la date correspondante à ce dimanche et ce peu importe à quel moment de la semaine nous nous trouvons. Nous allons donc créer une boucle *switch* qui vérifie en premier lieu le jour de la semaine où nous nous trouvons. S'il s'agit d'un dimanche, nous conservons dans les variables *iJour*, *iMois* et *iAnnee* les données correspondant à la journée actuelle.

S'il s'agit d'un jour postérieur au dimanche, nous allons utiliser les fonctions *TimeDay()*, *TimeMonth()* et *TimeYear()* pour récupérer les informations sur le dimanche en question.

Étant donné que le format de la date dans le nom du fichier est toujours à deux chiffres pour le jour et le mois, nous ajoutons des instructions conditionnelles pour modifier le jour et le mois

renvoyés par la boucle *switch* si ceux-ci ne comportent qu'un chiffre (pour le 1er juin, iDay et iMonth renvoie 1 et 6 respectivement et non pas 01 et 06). Étant donné qu'il est impossible dans MetaTrader de rajouter un 0 devant un entier, nous transformons donc *iJour* et *iMonth* en une chaîne de caractères où il est possible d'ajouter un 0 en préfixe.

```
int iJour, iMois, iAnnee, iNumeroFichier;
string sJour, sMois, sJourSemaine;


switch(DayOfWeek())
  {
  case 0:
    iJour = Day();
    iMois = Month();
    iAnnee = Year();
    sJourSemaine = "Sun";
    break;
  case 1:
    iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 1));
    iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 1));
    iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 1));
    sJourSemaine = "Mon";
    break;
  case 2:
    iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 2));
    iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 2));
    iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 2));
    sJourSemaine = "Tue";
    break;
  case 3:
    iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 3));
    iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 3));
    iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 3));
    sJourSemaine = "Wed";
    break;
  case 4:
    iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 4));
    iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 4));
    iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 4));
    sJourSemaine = "Thu";
    break;
  case 5:
    iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 5));
    iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 5));
    iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 5));
    sJourSemaine = "Fri";
    break;
  }

  if(iJour < 10)
    sJour = "0" + iJour;
  else
    sJour = iJour;
  if(iMois < 10)
    sMois = "0" + iMois;
  else
    sMois = iMois;
```

Maintenant que nous avons les informations permettant de récupérer le fichier, nous devons vérifier si celui-ci n'a pas déjà été téléchargé.

```
  sNomFichier = "Calendrier"+"-"+sMois+"-"+sJour+".csv";
  iNumeroFichier = FileOpen(sNomFichier,FILE_READ|FILE_CSV,',');
  if(iNumeroFichier == -1)
    {
```

```
  sAdresseUrlFichier = "http://www.dailyfx.com/files/Calendar-"+sMois+"-"+sJour+"-"+iAnnee+".csv";
  sNomFichier = "\Calendrier"+"-"+sMois+"-"+sJour+".csv";
  sDossierDestination = StringConcatenate(TerminalPath(),"\experts\files",sNomFichier);
  URLDownloadToFileA(0, sAdresseUrlFichier, sDossierDestination, 0, 0);
  }
 else
  FileClose(iNumeroFichier);
```

Nous attribuons donc à la variable sNomFichier le nom que le fichier de la semaine devrait avoir sur le disque dur et essayons d'ouvrir le fichier mentionné précédemment. Si le fichier n'est pas trouvé (i.e. *iNuméroFichier* égal à -1), le programme essayera de télécharger le fichier. Dans le cas contraire, c'est-à-dire si le fichier est trouvé, il sera simplement fermé.

Nous avons redéfini deux fois la variable sNomFichier avec le nom sauf que dans le deuxième cas, nous ajoutons « \ » devant le nom afin que *sDossierDestination* contienne le bon chemin. En effet, si nous essayons d'écrire *files*\", le système pense que nous essayons d'écrire le caractère « " » et non pas qu'il s'agit de la fin de la chaîne de caractères. Voir tableau 2 pour plus de détails.

Il ne reste plus maintenant qu'à afficher les informations. Afin de ne pas surcharger le graphique, nous désirons que ne soient affichées que les nouvelles économiques pour le jour même ou le futur. Nous devons donc ajouter des instructions permettant d'effacer les objets graphiques lorsque nous changeons de journée. L'instruction conditionnelle vérifie donc si la journée actuelle est la même que celle stockée dans la variable *iDay* (lors de la première exécution, cette variable est égale à 0, ce qui fait que les instructions figurant à l'intérieur de l'instruction conditionnelle seront exécutées). Si c'est le cas, le programme n'a pas besoin d'effacer ou redessiner les objets graphiques et la boucle est sautée.

La suite du code permet la lecture de chaque variable du fichier *.CSV ainsi que l'attribution d'une couleur en fonction de l'importance de la couleur.

```
if(Day() != iDay)
  {
  ObjectsDeleteAll(0, OBJ_LABEL);
  iDay = Day();

  iNumeroFichier = FileOpen(sNomFichier,FILE_READ|FILE_CSV,',');
  int iCompteur = 0;
  int iPixels = 0;


  while(!FileIsEnding(iNumeroFichier))
   {
   string sDate = FileReadString(iNumeroFichier);
   string sTime = FileReadString(iNumeroFichier);
   string sTimeZone = FileReadString(iNumeroFichier);
   string sCurrency = FileReadString(iNumeroFichier);
   string sDescription = FileReadString(iNumeroFichier);
   string sImportance = FileReadString(iNumeroFichier);
   string sActual = FileReadString(iNumeroFichier);
   string sForecast = FileReadString(iNumeroFichier);
   string sPrevious = FileReadString(iNumeroFichier);

   color cImportance = White;
   if(sImportance == "Low")
    cImportance = Yellow;
   if(sImportance == "Medium")
    cImportance = Orange;
```

```
      if(sImportance == "High")
        cImportance = Red;
      }
    }
```

Nous devons maintenant filtrer les nouvelles. Nous allons pour ce faire utiliser l'information que nous avons recueillie au début de la fonction et stockée dans la variable *sJourSemaine*. Il s'agit du jour de la semaine. Enfin, nous devons extraire la journée à laquelle chaque nouvelle est associée pour que notre programme sache s'il faut afficher ou ignorer la nouvelle (lorsqu'on sera mardi, le programme n'affichera plus les nouvelles associées à dimanche et lundi et ainsi de suite).

```
      if(StringSubstr(sDate, 0, 3) == "Sun")
        int iJourSemaine = 0;
      if(StringSubstr(sDate, 0, 3) == "Mon")
        iJourSemaine = 1;
      if(StringSubstr(sDate, 0, 3) == "Tue")
        iJourSemaine = 2;
      if(StringSubstr(sDate, 0, 3) == "Wed")
        iJourSemaine = 3;
      if(StringSubstr(sDate, 0, 3) == "Thu")
        iJourSemaine = 4;
      if(StringSubstr(sDate, 0, 3) == "Fri")
        iJourSemaine = 5;


      if(iJourSemaine >= DayOfWeek())
        {
        ObjectCreate("TxtNouvelleEco"+iCompteur, OBJ_LABEL, 0, 0, 0);
        ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_CORNER, 0);
        ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_XDISTANCE, 10);
        ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_YDISTANCE, 10+iPixels);
        ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_COLOR, cImportance);
        ObjectSetText("TxtNouvelleEco"+iCompteur, sDate + " : "+ sDescription, 8);
        iPixels = iPixels + 10;
        }
      iCompteur++;
      }
  FileClose(iNumeroFichier);
  }
```

Notre fonction au complet sera donc:

```
void fCalendrier()
 {
 int iJour, iMois, iAnnee, iNumeroFichier;
 string sJour, sMois, sJourSemaine;


 switch(DayOfWeek())
  {
  case 0:
    iJour = Day();
    iMois = Month();
    iAnnee = Year();
    sJourSemaine = "Sun";
    break;
  case 1:
    iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 1));
    iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 1));
    iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 1));
    sJourSemaine = "Mon";
    break;
  case 2:
```

```
   iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 2));
   iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 2));
   iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 2));
   sJourSemaine = "Tue";
   break;
  case 3:
   iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 3));
   iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 3));
   iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 3));
   sJourSemaine = "Wed";
   break;
  case 4:
   iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 4));
   iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 4));
   iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 4));
   sJourSemaine = "Thu";
   break;
  case 5:
   iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 5));
   iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 5));
   iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 5));
   sJourSemaine = "Fri";
   break;
  }

  if(iJour < 10)
   sJour = "0" + iJour;
  else
   sJour = iJour;
  if(iMois < 10)
   sMois = "0" + iMois;
  else
   sMois = iMois;


  sNomFichier = "Calendrier"+"-"+sMois+"-"+sJour+".csv";


  iNumeroFichier = FileOpen(sNomFichier,FILE_READ|FILE_CSV,',');
  if(iNumeroFichier == -1)
   {
   sAdresseUrlFichier = "http://www.dailyfx.com/files/Calendar-"+sMois+"-"+sJour+"-"+iAnnee+".csv";
   sNomFichier = "\Calendrier"+"-"+sMois+"-"+sJour+".csv";
   sDossierDestination = StringConcatenate(TerminalPath(),"\experts\files",sNomFichier);
   URLDownloadToFileA(0, sAdresseUrlFichier, sDossierDestination, 0, 0);
   bAffichageNouvelles = False;
   }
  else
   FileClose(iNumeroFichier);


  if(Day() != iDay)
   {
   ObjectsDeleteAll(0, OBJ_LABEL);
   iDay = Day();

   iNumeroFichier = FileOpen(sNomFichier,FILE_READ|FILE_CSV,',');
   int iCompteur = 0;
   int iPixels = 0;


   while(!FileIsEnding(iNumeroFichier))
    {
    string sDate = FileReadString(iNumeroFichier);
    string sTime = FileReadString(iNumeroFichier);
    string sTimeZone = FileReadString(iNumeroFichier);
    string sCurrency = FileReadString(iNumeroFichier);
    string sDescription = FileReadString(iNumeroFichier);
    string sImportance = FileReadString(iNumeroFichier);
    string sActual = FileReadString(iNumeroFichier);
    string sForecast = FileReadString(iNumeroFichier);
```

```
      string sPrevious = FileReadString(iNumeroFichier);

      color cImportance = White;
      if(sImportance == "Low")
        cImportance = Yellow;
      if(sImportance == "Medium")
        cImportance = Orange;
      if(sImportance == "High")
        cImportance = Red;


      if(StringSubstr(sDate, 0, 3) == "Sun")
        int iJourSemaine = 0;
      if(StringSubstr(sDate, 0, 3) == "Mon")
        iJourSemaine = 1;
      if(StringSubstr(sDate, 0, 3) == "Tue")
        iJourSemaine = 2;
      if(StringSubstr(sDate, 0, 3) == "Wed")
        iJourSemaine = 3;
      if(StringSubstr(sDate, 0, 3) == "Thu")
        iJourSemaine = 4;
      if(StringSubstr(sDate, 0, 3) == "Fri")
        iJourSemaine = 5;


      if(iJourSemaine >= DayOfWeek())
        {
        ObjectCreate("TxtNouvelleEco"+iCompteur, OBJ_LABEL, 0, 0, 0);
        ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_CORNER, 0);
        ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_XDISTANCE, 10);
        ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_YDISTANCE, 10+iPixels);
        ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_COLOR, cImportance);
        ObjectSetText("TxtNouvelleEco"+iCompteur, sDate + " : "+ sDescription, 8);
        iPixels = iPixels + 10;
        }
      iCompteur++;
      }
    FileClose(iNumeroFichier);
    }
```

Il ne reste plus qu'à ajouter cette fonction à la fin de notre programme ainsi qu'un appel de fonction au début de celui-ci dans la fonction *start()*.

```
if(bAffichageNouvelles == True)
  fCalendrier();
```

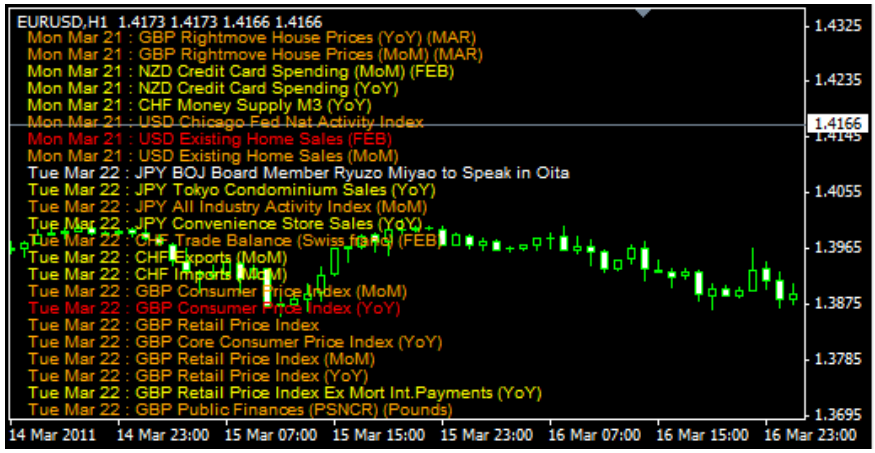

Figure 57 : Affichage d'un calendrier économique sur le graphique de l'expert

40
Protéger son code

Dans certains cas, il peut être utile de protéger son code afin d'éviter que les utilisateurs aient accès au code source ou limiter le temps d'utilisation de l'expert dans le cadre d'une période d'essai.

Même si distribuer le fichier compilé demeure la meilleure méthode pour protéger son code, notez néanmoins qu'il existe déjà à la vente des décompilateurs permettant d'obtenir le code source de n'importe quel fichier *.ex4. Il s'agit donc d'une protection de base. Il est possible de protéger son code efficacement en codant les instructions dans un fichier *.DLL et un expert qui contiendra des appels pour votre *.DLL. Il s'agit là de programmes plus complexes que nous ne traiterons pas dans cet ouvrage.

Le principe de toutes les méthodes détaillées ci-dessous est le même : vérifier au début de la boucle *start()* si certaines conditions sont remplies ou non. Si les conditions ne sont pas remplies, le programme exécutera une instruction *return(0)* qui renverra ainsi au début de la boucle *start()* sans exécuter le reste du code comme illustré dans la figure ci-dessous.

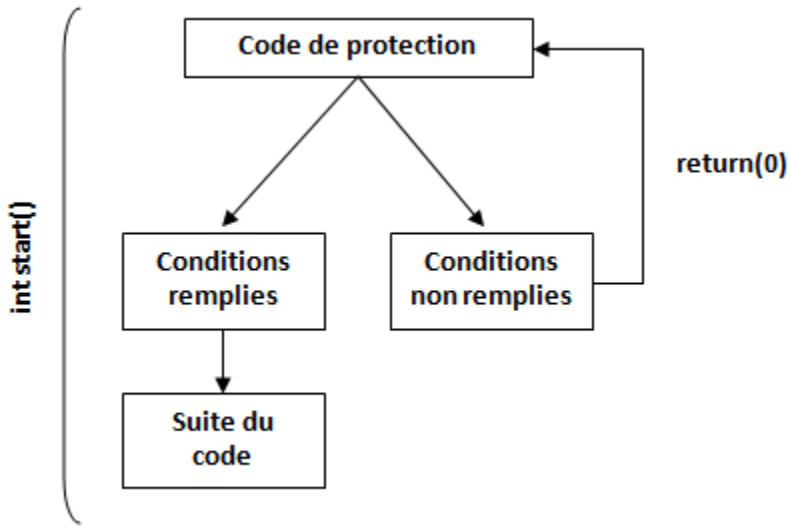

Figure 58 : Principe de base des codes de protection

Protection par mot de passe

Pour ajouter un mot de passe, il suffirait de créer une variable externe que l'utilisateur puisse modifier et de vérifier au début de la boucle *start()* que le mot de passe correspond bien à celui que le programme a en mémoire. Une version plus complexe impliquerait de faire en sorte que le programme communique avec un serveur pour vérifier si le mot de passe indiqué est bien celui présent sur le serveur (l'avantage dans ce cas serait de pouvoir périodiquement modifier le mot de passe).

Le code pour la première version serait donc le suivant :

```
extern string sMotDePasse = " ";

int start()
  {
  if(sMotdePasse != "Mot de passe choisi par le programmeur ")
    return(0);

  // suite du programme
  }
```

Protection temporelle

Si vous fournissez une version d'essai, vous souhaitez parfois limiter celle-ci à une période précise. Le code serait donc :

```
int start()
  {
  datetime dtDateExpiration = StrToTime("2011.15.04");

  if (TimeCurrent() > dtDateExpiration)
    return(0);

  // suite du programme
  }
```

Protection par numéro et type de compte

Vous pouvez également limiter l'utilisation de votre programme à un numéro de compte spécifique ou à un type de compte (par exemple un compte de démonstration uniquement). Le code serait :

```
int start()
  {

  if(123456  != AccountNumber())
    return(0);

  if (IsDemo() == False)
    return(0);

  // suite du programme
  }
```

41
Conclusion

MQL5?

MetaTrader 5 est actuellement dans une de ses ultimes versions béta avant son déploiement à plus grande échelle. Au même titre que MetaTrader 4 et le MQL4, la nouvelle version de MetaTrader sera accompagnée de son propre langage de programmation : le MQL5.

Une des grandes nouveautés du langage MQL5 est qu'il s'agit d'un langage orienté objet –méthode facilitant l'intégration et l'utilisation de différentes structures au sein du même programme-. Cette caractéristique devrait grandement simplifier l'écriture, la lecture et la correction des codes sources.

Les concepteurs de MetaTrader ont également ajouté le concept d'événement permettant de coder des instructions qui reconnaîtront certains événements tels que frappes au clavier, des changements sur les graphiques ou clôtures d'ordres par exemple. Actuellement, il était possible d'identifier certains de ces événements au travers de boucles testant plusieurs itérations. Dorénavant, le programme sera à même d'identifier immédiatement l'événement.

Plusieurs nouveaux types de données, fonctions, objets graphiques sont également inclus dans le nouveau langage. La syntaxe pour écrire les paramètres ou obtenir des données issues d'une série temporelle (*iHigh()* par exemple) a été modifiée.

En termes de fonctionnalités, MetaTrader a été muni de nouvelles variables permettant de simplifier le débogage des programmes ainsi que de la possibilité de tester un expert sur un portefeuille d'instruments (MetaTrader 4 ne pouvant tester les experts consultants que sur un seul instrument à la fois).

Qu'est-ce que le futur nous réserve?

Au niveau du trading automatisé accessible aux particuliers, plusieurs nouveaux concepts ont fait leur apparition ces dernières années. Le premier est l'utilisation de réseaux de neurones artificiels (RNA). Ces réseaux tentent d'imiter la structure d'un cerveau humain afin de pouvoir identifier des configurations, faire des prévisions ou prendre des décisions en se basant sur l'expérience passée. Un cerveau humain reçoit les informations qu'il utilise au travers des différents sens tandis qu'un RNA utilisera les informations issues de séries de données telles que des séries temporelles financières.

Ces informations transiteront à travers une ou plusieurs couches de neurones afin de les analyser pour obtenir à la fin la prévision ou décision. Les réseaux de neurones sont de plus en populaires grâce à leur capacité d'apprentissage. En effet, une des étapes dans la conception d'un réseau de neurone est « l'entraînement » de ce dernier sur une période précise afin qu'il puisse apprendre. Ce processus sera recommencé tant et aussi longtemps que le RNA peut encore améliorer l'exactitude de ses prévisions. Ci-dessous vous trouverez une figure illustrant un simple RNA.

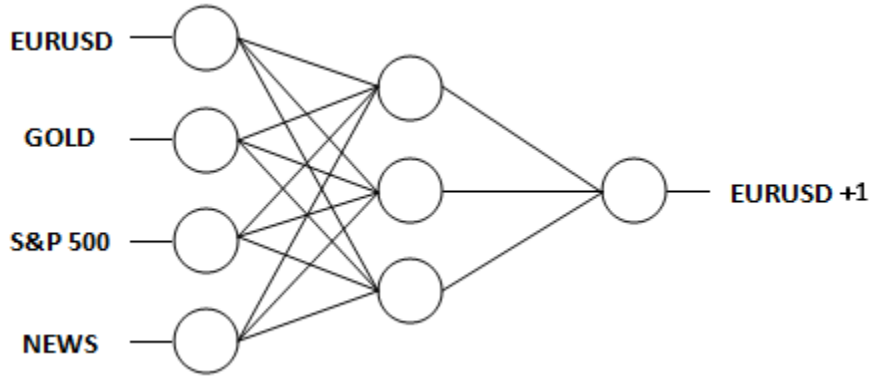

Figure 59 : Réseau de neurones artificiels

En utilisant des données telles que le cours historique de la paire EURUSD, de l'or, de l'indice S&P 500 ainsi que le fait que des nouvelles économiques majeures ont été publiées pendant la journée afin de filtrer les journées à haute volatilité, le réseau prédira un cours pour la paire EURUSD un jour dans le futur. La première couche de neurones sert à recevoir les données (1 neurone par type de donnée). La deuxième couche sert à interpréter les données et déterminer l'importance de chacune d'entre elles. Le dernier neurone est celui qui, en se basant sur l'information fournie par la deuxième couche, prédira le cours de la paire EURUSD un jour dans le futur.

Il serait relativement complexe de coder un RNA de a à z en MQL4. Pour vous simplifier la tâche, de nombreuses librairies, libres de droit, sont maintenant disponibles sur les forums spécialisés tels que celui du site mql4.com. Une des librairies les plus courantes est FANN[1] qui permet de créer un RNA de trois couches.

Le deuxième concept de plus en plus populaire est l'application de théories économiques telles que la théorie des jeux au trading automatisé. En effet, rien de nouveau au fait d'utiliser la théorie économique pour essayer de « battre » le marché mais avec le perfectionnement constant des langages de programmation et les nouvelles opportunités que cette innovation représente, il est maintenant possible d'automatiser tous les calculs nécessaires.

Pour continuer avec l'exemple de la théorie des jeux, l'interaction entre le marché et les spéculateurs peut être résumée par la matrice de gains suivante :

		Spéculateur	
		Agressif	Averse
Marché	Faveur	Profit	Profit
	Défaveur légère	Profit	Perte mineure
	Défaveur complète	Perte majeure	Perte mineure

Figure 60 : Matrice des gains de l'interaction entre le marché et un spéculateur

[1] L'article présentant la librairie et comment l'utiliser est disponible ici : http://articles.mql4.com/777

La matrice précédente se base sur les suppositions suivantes :

- Trois conditions de marché sont possibles. La première est que le marché va en faveur du spéculateur. La deuxième est que le marché va légèrement en défaveur du spéculateur avant de se retourner. La troisième est que le marché va complètement en défaveur du spéculateur.
- Le spéculateur est soit agressif, soit adverse. Dans le premier cas, le stop sera placé beaucoup plus loin que dans le deuxième cas.
- Le spéculateur ne ferme pas sa position tant que le stop ou la limite ne sont pas atteints.

En se basant sur la logique de la matrice, il est possible de transformer le raisonnement en équations qui une fois résolues permettent de savoir quelle stratégie, agressive ou adverse, le spéculateur devrait suivre afin de maximiser ses chances. Si vous désirez en savoir plus, un des premiers ouvrages écrits sur le sujet et très accessible au niveau mathématique, est « Gaming the market » par Ronald B. Shelton aux éditions Wiley Trading Advantage. Un expert basé sur ce concept est actuellement en préparation. Visitez le forum de Eole Trading pour en savoir plus.

Le mot de la fin

D'aucuns pourraient être déçus par le peu d'information sur certains concepts (notamment les RNAs) mais ce livre étant avant tout une introduction, il était malheureusement impossible de couvrir la totalité des sujets compte tenu de l'amplitude du sujet. La meilleure façon d'apprendre reste d'essayer de programmer par vous-même et contrairement à d'autres domaines, c'est ici sans risques tant et aussi longtemps que vous n'utilisez pas vos programmes sur un compte réel.

Quelques sites internet

En français :

http://eole-trading.com
http://www.activtrades.fr/index.aspx?page=events_webinararchiv
http://www.trader-forex.fr/forum/systemes-de-trading-auto/
http://www.trading-automatique.fr/

En anglais :

http://mql4.com
http://forex-tsd.com

Pour contacter l'auteur :

henri@eole-trading.com

Annexe A

Constante	Code numérique	Description
ERR_NO_ERROR	0	Aucune erreur
ERR_NO_RESULT	1	Aucune erreur mais résultat inconnu
ERR_COMMON_ERROR	2	Erreur commune
ERR_INVALID_TRADE_PARAMETERS	3	Paramètres du passage d'ordre incorrects
ERR_SERVER_BUSY	4	Serveur de trading occupé
ERR_OLD_VERSION	5	Version du terminal obsolète
ERR_NO_CONNECTION	6	Pas de connexion avec le serveur
ERR_NOT_ENOUGH_RIGHTS	7	Droits insuffisants
ERR_TOO_FREQUENT_REQUESTS	8	Requêtes trop fréquentes
ERR_MALFUNCTIONAL_TRADE	9	Malfonctionnement du passage d'ordre
ERR_ACCOUNT_DISABLED	64	Compte désactivé
ERR_INVALID_ACCOUNT	65	Compte incorrect
ERR_TRADE_TIMEOUT	128	Délai pour le passage d'ordre dépassé
ERR_INVALID_PRICE	129	Le prix indiqué pour passer ou clôturer l'ordre n'est pas valide
ERR_INVALID_STOPS	130	Distance du prix limite, stop avec le prix d'entrée incorrecte ou distance du prix d'entrée avec le prix du marché incorrecte (ordres en attentes)
ERR_INVALID_TRADE_VOLUME	131	Volume incorrect
ERR_MARKET_CLOSED	132	Marchés fermés
ERR_TRADE_DISABLED	133	Passage d'ordre désactivé
ERR_NOT_ENOUGH_MONEY	134	Balance insuffisante
ERR_PRICE_CHANGED	135	Changement de prix entre le passage d'ordre et son exécution
ERR_OFF_QUOTES	136	Prix hors cotation
ERR_BROKER_BUSY	137	Courtier occupé
ERR_REQUOTE	138	Nouvelle cotation
ERR_ORDER_LOCKED	139	Ordre verrouillé

ERR_LONG_POSITIONS_ONLY_ALLOWED	140	Seules les positions achat sont autorisées
ERR_TOO_MANY_REQUESTS	141	Requêtes trop nombreuses
ERR_TRADE_MODIFY_DENIED	145	Modification refusée car le prix du marché est trop proche
ERR_TRADE_CONTEXT_BUSY	146	Canal de passage d'ordres occupé
ERR_TRADE_EXPIRATION_DENIED	147	Expiration refusée par le courtier
ERR_TRADE_TOO_MANY_ORDERS	148	Limite du nombre de positions ouvertes ou en attentes atteintes
ERR_TRADE_HEDGE_PROHIBITED	149	Hedging non autorisé
ERR_TRADE_PROHIBITED_BY_FIFO	150	Impossible de clôture la position car va à l'encontre de la règlementation FIFO.
ERR_NO_MQLERROR	4000	Pas d'erreur d'exécution
ERR_WRONG_FUNCTION_POINTER	4001	Appel de fonction incorrect
ERR_ARRAY_INDEX_OUT_OF_RANGE	4002	Indexation de tableau hors de portée
ERR_NO_MEMORY_FOR_CALL_STACK	4003	Mémoire insuffisante pour la pile d'appel de la fonction
ERR_RECURSIVE_STACK_OVERFLOW	4004	Débordement de la pile récurrent
ERR_NOT_ENOUGH_STACK_FOR_PARAM	4005	Pile insuffisante pour le paramètre
ERR_NO_MEMORY_FOR_PARAM_STRING	4006	Mémoire insuffisante pour le paramètre chaîne de caractères
ERR_NO_MEMORY_FOR_TEMP_STRING	4007	Mémoire insuffisante pour la chaîne de caractère temporaire
ERR_NOT_INITIALIZED_STRING	4008	Pas d'initialisation de chaîne de caractères
ERR_NOT_INITIALIZED_ARRAYSTRING	4009	Pas d'initialisation de chaîne de caractères dans le tableau
ERR_NO_MEMORY_FOR_ARRAYSTRING	4010	Mémoire insuffisante pour le tableau de chaîne de caractères
ERR_TOO_LONG_STRING	4011	Chaîne de caractères trop longue
ERR_REMAINDER_FROM_ZERO_DIVIDE	4012	Reste de la division par zéro
ERR_ZERO_DIVIDE	4013	Division par zéro
ERR_UNKNOWN_COMMAND	4014	Commande inconnue
ERR_WRONG_JUMP	4015	Saut incorrect
ERR_NOT_INITIALIZED_ARRAY	4016	Pas d'initialisation de tableau
ERR_DLL_CALLS_NOT_ALLOWED	4017	Appel de DLL non autorisé
ERR_CANNOT_LOAD_LIBRARY	4018	Impossible de charger la librairie
ERR_CANNOT_CALL_FUNCTION	4019	Impossible d'appeler la fonction
ERR_EXTERNAL_CALLS_NOT_ALLOWED	4020	Appel de fonction depuis un expert externe non autorisé
ERR_NO_MEMORY_FOR_RETURNED_STR	4021	Mémoire insuffisante pour la chaîne de caractères temporaire renvoyée

		par la fonction
ERR_SYSTEM_BUSY	4022	Système occupée
ERR_INVALID_FUNCTION_PARAMSCNT	4050	Nombre de paramètre de la fonction incorrect
ERR_INVALID_FUNCTION_PARAMVALUE	4051	Valeur des paramètres de la fonction incorrecte
ERR_STRING_FUNCTION_INTERNAL	4052	Erreur interne de la fonction sur la chaîne de caractères
ERR_SOME_ARRAY_ERROR	4053	Erreur de tableau
ERR_INCORRECT_SERIESARRAY_USING	4054	Utilisation incorrecte de la série de tableau
ERR_CUSTOM_INDICATOR_ERROR	4055	Erreur de l'indicateur personnalisé
ERR_INCOMPATIBLE_ARRAYS	4056	Les tableaux sont incompatibles
ERR_GLOBAL_VARIABLES_PROCESSING	4057	Erreur de traitement des variables globales
ERR_GLOBAL_VARIABLE_NOT_FOUND	4058	Variable globale introuvable
ERR_FUNC_NOT_ALLOWED_IN_TESTING	4059	Fonction non autorisée en mode backtest
ERR_FUNCTION_NOT_CONFIRMED	4060	Fonction non confirmée
ERR_SEND_MAIL_ERROR	4061	Erreur d'envoi d'email
ERR_STRING_PARAMETER_EXPECTED	4062	Paramètre de type chaîne de caractères attendu
ERR_INTEGER_PARAMETER_EXPECTED	4063	Paramètre de type entier attendu
ERR_DOUBLE_PARAMETER_EXPECTED	4064	Paramètre de type réel attendu
ERR_ARRAY_AS_PARAMETER_EXPECTED	4065	Paramètre de type tableau attendu
ERR_HISTORY_WILL_UPDATED	4066	Historique en cours de chargement
ERR_TRADE_ERROR	4067	Erreur dans la fonction de trading
ERR_END_OF_FILE	4099	Fin de fichier
ERR_SOME_FILE_ERROR	4100	Erreur dans le fichier
ERR_WRONG_FILE_NAME	4101	Nom de fichier incorrect
ERR_TOO_MANY_OPENED_FILES	4102	Trop de fichiers ouverts
ERR_CANNOT_OPEN_FILE	4103	Impossible d'ouvrir le fichier
ERR_INCOMPATIBLE_FILEACCESS	4104	Impossible d'accéder le fichier
ERR_NO_ORDER_SELECTED	4105	Aucun ordre sélectionné
ERR_UNKNOWN_SYMBOL	4106	Symbole inconnu
ERR_INVALID_PRICE_PARAM	4107	Prix invalide
ERR_INVALID_TICKET	4108	Numéro de ticket invalide
ERR_TRADE_NOT_ALLOWED	4109	Trading interdit. Cocher la case "Allow live trading" dans les propriétés de l'expert
ERR_LONGS_NOT_ALLOWED	4110	Positions de type achat non autorisées.

ERR_SHORTS_NOT_ALLOWED	4111	Positions de type vente non autorisées
ERR_OBJECT_ALREADY_EXISTS	4200	Objets déjà existants
ERR_UNKNOWN_OBJECT_PROPERTY	4201	Propriété de l'objet inconnu
ERR_OBJECT_DOES_NOT_EXIST	4202	Objet inexistant
ERR_UNKNOWN_OBJECT_TYPE	4203	Type d'objet inconnu
ERR_NO_OBJECT_NAME	4204	Pas de nom d'objet
ERR_OBJECT_COORDINATES_ERROR	4205	Erreur de coordonnées d'objet
ERR_NO_SPECIFIED_SUBWINDOW	4206	Pas de numéro de fenêtre spécifié
ERR_SOME_OBJECT_ERROR	4207	Erreur dans la fonction objet

Tableau 36 : Liste des codes d'erreurs et leurs significations

Annexe B

Voici le code source complet avec commentaires de l'expert consultant (téléchargeable sur http://eole-trading.com)

```
//+------------------------------------------------------------------+
//|                          Votre Premier Expert.mq4                |
//|                           Copyright © 2011,                      |
//|                        http://www. eole-trading.com              |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2011,"
#property link      "http://www.eole-trading.com"


#include <stdlib.mqh> // Appel de la librairie stdlib pour pouvoir afficher le descriptif des erreurs

// Importation de la fonction Windows permettant de télécharger un fichier
#import "urlmon.dll"
  int URLDownloadToFileA(int pCaller,string szURL,string szFileName,int dwReserved,int Callback);
#import

extern int iHeureDebut = 9; // Heure du début de la plage horaire du filtre horaire
extern int iMinutesDebut = 30; // Minutes du début de la plage horaire du filtre horaire
extern int iHeureFin = 17; // Heure de fin de la plage horaire du filtre horaire
extern int iMinutesFin = 30; // Minutes de fin de la plage horaire du filtre horaire
extern int iDistance = 5; // Distance en pips entre le plus haut/bas et le prix d'entrée
extern int iMagic = 123456; // Numéro magique
extern int iStop = 50; // Stop en pips
extern int iLimite = 50; // Limite en pips
extern double dPourcentageRisque = 5; // Pourcentage de la balance du compte à risquer à chaque trade
extern int iStopSuiveur = 25; // Stop suiveur en pips
extern bool bFiltrePointPivot = True; // True : utiliser l'indicateur / False : ne pas utiliser l'indicateur

int iDate; // Variable permettant de réinitialiser l'expert lors du passage à une nouvelle journée
int iDay = 0; // Variable permettant de réinitialiser le téléchargement du fichier int iTimeDebut; // Variable
stockant la valeur de l'heure du début du filtre horaire en secondes
int iTimeFin; // Variable stockant la valeur de l'heure de fin du filtre horaire en secondes
int iTimeSeconds; // Variable stockant la valeur de l'heure actuelle en secondes
int iTicket; // Variable stockant le numéro de ticket lorsqu'une position est placée
double dPlusHaut; // Variable stockant le prix plus haut de la veille
double dPlusBas; // Variable stockant le prix plus bas de la veille
bool bRecuperation = False; // Variable permettant de limiter l'obtention des niveaux plus haut/bas de la veille
bool bTrade = True; // Variable permettant de limiter le passage d'ordre à une fois par journée
string sAdresseUrlFichier; // Variable stockant l'adresse URL du fichier à télécharger
string sDossierDestination; // Variable stockant le chemin du dossier du fichier téléchargé
string sNomFichier; // Variable stockant le nom du fichier (nom différent à chaque semaine)

//+------------------------------------------------------+
//| expert initialization fonction                       |
//+------------------------------------------------------+
int init()
  {
  iTimeDebut = iHeureDebut * 3600 + iMinutesDebut * 60; // Conversion de l'heure du début en secondes
  iTimeFin = iHeureFin * 3600 + iMinutesFin * 60; // Conversion de l'heure de fin en secondes

  fVerifPositions(); // Appel de fonction pour s'assurer qu'il n'y pas déjà des positions ouvertes

  return(0);
```

```
  }
//+------------------------------------------------------+
//| expert deinitialization fonction                     |
//+------------------------------------------------------+
int deinit()
  {
//----

//----
  return(0);
  }
//+------------------------------------------------------+
//| expert start fonction                                |
//+------------------------------------------------------+
int start()
  {
  fCalendrier(); // Appel de fonction pour télécharger et afficher les nouvelles économiques
  iTimeSeconds = Hour()*3600 + Minute()*60 + Seconds(); // Conversion de l'heure actuelle en secondes
  fStopSuiveur(Symbol(), iStopSuiveur, iMagic); // Appel de fonction pour modifier les stops si besoin

  if(iTimeSeconds >= iTimeDebut && iTimeSeconds < iTimeFin) // Filtre horaire
    {
    if(bRecuperation == False) // Vérification si les niveaux n'ont pas déjà été récupérés
      {
      iDate = Day(); // Stockage de la date du jour dans la variable iDate
      dPlusHaut = iHigh(NULL, PERIOD_D1, 1); // Récupération du niveau plus haut de la veille
      dPlusBas = iLow(NULL, PERIOD_D1, 1);  // Récupération du niveau plus bas de la veille
      bRecuperation = True; // bRecuperation = True pour éviter une nouvelle exécution de ce bloc pendant la
journée
      fTracageLignes("Point plus haut", dPlusHaut, Blue); // Appel de fonction pour tracer le niveau plus haut
      fTracageLignes("Point plus bas", dPlusBas, Red); // Appel de fonction pour afficher le niveau plus bas
      }
    if(bTrade == True)  // Vérification si des positions n'ont pas déjà été placées
      {
      while(IsTradeContextBusy() == True) // Vérification si le canal de trading est occupé
        Sleep(1000); // Pause de 1 seconde si le canal est occupé
      // Vérification des conditions du courtier pour placer des ordres et la valeur de l'indicateur Point Pivot
      if((dPlusHaut + iDistance * fPoint(Symbol())) - Ask > (MarketInfo(Symbol(), MODE_STOPLEVEL)) *
      fPoint(Symbol())  && fFiltrePointPivot(bFiltrePointPivot, "Achat") == True)
        {
        iTicket = OrderSend(Symbol(), OP_BUYSTOP, fLots(dPourcentageRisque, iStop), dPlusHaut +
        iDistance* fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Achat sur rupture du niveau plus haut de la
        veille", iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600) + ((iMinutesFin -
        TimeMinute(Time[0]))*60)), Blue); // Passage de l'ordre achat stop
        fTicket(iTicket); // Ajout du stop et de la limite ;a la position
        fTicketVariableGloable("gvNumeroTicketAchat", iTicket); // Stockage du numéro de ticket
        fErreur("Achat"); // Affichage du descriptif de l'erreur si il y a lieu
        }

      while(IsTradeContextBusy() == True) // Vérification si le canal de trading est occupé
       Sleep(1000); // Pause de 1 seconde si le canal est occupé
      if(Bid - (dPlusBas - iDistance* fPoint(Symbol())) > (MarketInfo(Symbol(), MODE_STOPLEVEL)) *
      fPoint(Symbol()) && fFiltrePointPivot(bFiltrePointPivot, "Vente") == True) // Vérification des conditions du
      courtier pour placer des ordres et la valeur de l'indicateur Point Pivot
        {
        iTicket = OrderSend(Symbol(), OP_SELLSTOP, fLots(dPourcentageRisque, iStop),  dPlusBas -
        iDistance* fPoint(Symbol()), fSlippage(Symbol(), 3), 0, 0, "Vente sur rupture du niveau plus bas de la
        veille", iMagic,Time[0] + (((iHeureFin - TimeHour(Time[0]))*3600)+((iMinutesFin -
       TimeMinute(Time[0]))*60)), Red); // Passage de l'ordre vente stop
        fTicket(iTicket); // Ajout du stop et de la limite ;a la position
        fTicketVariableGloable("gvNumeroTicketVente", iTicket); // Stockage du numéro de ticket
        fErreur("Vente"); // Affichage du descriptif de l'erreur si il y a lieu
        }

      bTrade = False; // bTrade = False pour éviter de placer d'autres ordres dans la même journée
      }

    if(iDate != Day()) // Vérification si une nouvelle journée est en cours
      {
```

```
      bRecuperation = False; // bRecuperation = False pour autoriser la récupération des nouveaux niveaux
      bTrade = True; // bTrade = True pour autoriser le passage de nouvelles positions
      }
    }
  return(0); // Opérateur return pour retourner au début de la fonction int start()
  }
//+------------------------------------------------------+
void fTracageLignes(string sNomLigne, double dPrix, color cCouleur) // Fonction pour tracer les niveaux
  {
  ObjectDelete(sNomLigne); // Effacement de la ligne de la veille
  ObjectCreate(sNomLigne,OBJ_HLINE, 0, 0, dPrix); // Création de la nouvelle ligne
  ObjectSet(sNomLigne, OBJPROP_COLOR, cCouleur); // Paramétrage de la nouvelle ligne
  }

double fPoint(string sPaire) // Fonction pour déterminer la valeur correcte de Point
  {
  if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 3) // 2 - 3 décimales
    {
    double dPoint = 0.01; // Point = 0.01
    }
  if((MarketInfo(sPaire, MODE_DIGITS)) == 4 || (MarketInfo(sPaire, MODE_DIGITS)) == 5) // 4 - 5 décimales
    {
    dPoint = 0.0001; // Point = 0.0001
    }
  return(dPoint); // Renvoi de la valeur correcte
  }

double fSlippage(string sPaire, int iPipsSlippage) // Fonction pour calculer la valeur correcte du slippage
  {
  if((MarketInfo(sPaire, MODE_DIGITS)) == 2 || (MarketInfo(sPaire, MODE_DIGITS)) == 4) // 2 - 4 décimales
    {
    double dSlippage = iPipsSlippage; // Pas besoin de modifier la valeur
    }
  if((MarketInfo(sPaire, MODE_DIGITS)) == 3 || (MarketInfo(sPaire, MODE_DIGITS)) == 5) // 3 - 5 décimales
    {
    dSlippage = iPipsSlippage * 10; // Nécessité de multiplier la valeur par 10
    }
  return(dSlippage); // Renvoi de la valeur correcte
  }

void fTicket(int iTicket) // Fonction pour ajouter un stop et une limite
  {
  if(iTicket != -1) // Vérification si une position a bien été placée
    {
    OrderSelect(iTicket, SELECT_BY_TICKET); // Sélection de la position qui vient d'être placée
    if(OrderType() == 4) // Vérification si la position est de type achat stop
      OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() - iStop * fPoint(Symbol()),
      OrderOpenPrice() + iLimite * fPoint(Symbol()), OrderExpiration()); // Ajout d'un stop et d'une limite

    if(OrderType() == 5) // Vérification si la position est de type vente stop
      OrderModify(OrderTicket(), OrderOpenPrice(), OrderOpenPrice() + iStop * fPoint(Symbol()),
      OrderOpenPrice() - iLimite * fPoint(Symbol()), OrderExpiration()); // Ajout d'un stop et d'une limite
      }
  }

double fLots(double dPourcentageRisque, int iStop) // Fonction pour calculer le volume de chaque position
  {
  double dRisqueMax = (dPourcentageRisque/100) * AccountBalance(); // Calcul du risque maximale autorisé
  double dValeurPip = MarketInfo(Symbol(), MODE_TICKVALUE); // Détermination de la valeur d'un pip
  if(Point == 0.001 || Point == 0.00001) // Ajustement de la valeur du pip si nécessaire
    {
    dValeurPip = dValeurPip * 10; // Multiplication par 10 de la valeur du pip lorsque la paire a 3 ou 5
décimales
    }
  double dLots = (dRisqueMax / iStop) / dValeurPip; // Calcul du volume en lots
  if(dLots < MarketInfo(Symbol(),MODE_MINLOT)) // Vérification si le volume n'est pas trop petit
    dLots = MarketInfo(Symbol(),MODE_MINLOT); // Ajustement si le volume est trop petit
  if(dLots > MarketInfo(Symbol(),MODE_MAXLOT)) // Vérification si le volume n'est pas trop grand
    dLots = MarketInfo(Symbol(),MODE_MAXLOT); // Ajustement si le volume est trop grand
```

```
  if(MarketInfo(Symbol(),MODE_LOTSTEP) == 0.01) // Vérification de l'incrémentation autorisé par le courtier
    dLots = NormalizeDouble(dLots, 2); // Si les micro-lots sont permis, le volume aura 2 décimales
  else
    dLots = NormalizeDouble(dLots, 1); // Si les micro-lors ne sont pas permis, le volume aura 1 décimale
  return(dLots); // Renvoi du volume
  }

void fErreur(string sPosition) // Fonction pour afficher le descriptif de l'erreur
  {
  if(iTicket == -1) // Vérification si une erreur est survenue
    {
    int iErreur = GetLastError(); // Récupération du numéro de l'erreur
    string sErreur = ErrorDescription(iErreur); // Affectation du descriptif de l'erreur à la variable sErreur
    Print("Erreur position "+sPosition+" : "+iErreur+" | "+sErreur); // Affichage du descriptif de l'erreur
    }
  }

void fStopSuiveur (string sSymbol, int iStopSuiveur, int iMagic) // Fonction de stop suiveur
  {
  // Boucle pour vérifier toutes les positions ouvertes ou en attentes
  for(int iCompteur = 0; iCompteur <= OrdersTotal() - 1; iCompteur++)
    {
    OrderSelect(iCompteur, SELECT_BY_POS); // Sélection de chaque position une par une

    // Détermination du prix du stop suiveur pour une position achat
    double dStopSuiveurBuy = MarketInfo(sSymbol, MODE_BID) - (iStopSuiveur * fPoint(sSymbol));
    // Détermination du prix du stop suiveur pour une position vente
    double dStopSuiveurSell = MarketInfo(sSymbol, MODE_ASK) + (iStopSuiveur * fPoint(sSymbol));

    if(OrderMagicNumber() == iMagic && OrderSymbol() == sSymbol  && OrderType() == OP_BUY &&
    OrderStopLoss() < dStopSuiveurBuy) // Vérification des conditions pour modifier le stop
      {
      // Modification du stop
      bool bErreur = OrderModify(OrderTicket(), OrderOpenPrice(), dStopSuiveurBuy, OrderTakeProfit(), 0);
      if(bErreur == 0) // Vérification si une erreur est survenue
        {
        int iErreur = GetLastError(); // Récupération du numéro de l'erreur
        string sErreur = ErrorDescription(iErreur); // Affectation du descriptif de l'erreur à la variable sErreur
        Print("Erreur modification position achat : "+iErreur+" | "+sErreur); // Affichage du descriptif de l'erreur
        }
      }

    if(OrderMagicNumber() == iMagic && OrderSymbol() == sSymbol && OrderType() == OP_SELL &&
OrderStopLoss() > dStopSuiveurSell) // Vérification des conditions pour modifier le stop de la position vente
      {
      // Modification du stop
      bErreur = OrderModify(OrderTicket(), OrderOpenPrice(), dStopSuiveurSell, OrderTakeProfit(), 0);
      if(bErreur == 0) // Vérification si une erreur est survenue
        {
        iErreur = GetLastError(); // Récupération du numéro de l'erreur
        sErreur = ErrorDescription(iErreur); // Affectation du descriptif de l'erreur à la variable sErreur
        Print("Erreur modification position achat : "+iErreur+" | "+sErreur); // Affichage du descriptif de l'erreur
        }
      }
    }
  }

void fVerifPositions() // Fonction pour vérifier si des positions ont déjà été ouvertes
  {
  // Vérification si des positions ont déjà été placées
  if(OrderSelect(GlobalVariableGet("gvNumeroTicketAchat"),SELECT_BY_TICKET) == True ||
  OrderSelect(GlobalVariableGet("gvNumeroTicketVente"),SELECT_BY_TICKET) == True)
    {
    bTrade = False; // Si des positions ont déjà été placées, l'expert ne pourra pas placer d'autres positions
    }
  }
```

```
// Fonction pour stocker le numéro de ticket des positions dans une variable globale
void fTicketVariableGloable(string sNomVariableGlobale, int iNumeroTicket)
 {
 bool bErreur = GlobalVariableSet(sNomVariableGlobale, iNumeroTicket); // Stockage du numéro de ticket
 if(bErreur == 0) // Vérification si une erreur est survenue
   {
   int iErreur = GetLastError(); // Récupération du numéro de l'erreur
   string sErreur = ErrorDescription(iErreur); // Affectation du descriptif de l'erreur à la variable sErreur
   Print("Erreur lors de la creation/modification de la variable globale "+ sNomVariableGlobale +" : "+
   sErreur); // Affichage du descriptif de l'erreur
   }
 }

// Fonction pour vérifier la valeur de l'indicateur Point Pivot
bool fFiltrePointPivot(bool bFiltrePointPivot, string sOperation)
 {
 // Récupération de la valeur de l'indicateur
 double dValeurPointPivot = iCustom(Symbol(), 0, "Premier indicateur", Blue, 0, 0);

 if(bFiltrePointPivot == False) // Vérification si l'utilisateur veut utiliser ou non l'indicateur
   bool bFiltreAutorise = True; // Si non, la variable bFiltreAutorise est paramétré sur autoriser par défaut
 if(sOperation == "Achat") // Vérification si la position est de type achat
   {
   if(Ask >= dValeurPointPivot && bFiltrePointPivot == True) // Prix Ask est supérieur ou égal au point pivot?
     bFiltreAutorise = True; // Autoriser la prise de position à l'achat
   else
     bFiltreAutorise = False; // Ne pas autoriser la prise de position à la vente
   }

 if(sOperation == "Vente") // Vérification si la position est de type vente
   {
   if(Bid <= dValeurPointPivot && bFiltrePointPivot == True) // Prix Bid est inférieur ou égal au point pivot?
     bFiltreAutorise = True; // Autoriser la prise de position à la vente
   else
     bFiltreAutorise = False; // Ne pas autoriser la prise de position à la vente
   }
 return(bFiltreAutorise); // Renvoi l'autorisation ou l'interdiction de prendre la position
 }

void fCalendrier() // Fonction pour afficher les nouvelles économiques sur le côté gauche du graphique
 {
 int iJour, iMois, iAnnee, iNumeroFichier; // Déclaration des variables int qui seront utilisées dans la fonction
 string sJour, sMois, sJourSemaine; // // Déclaration des variables string qui seront utilisées dans la fonction

 switch(DayOfWeek()) // Boucle switch pour déterminer quel jour correspond au début de la semaine
   {
   case 0: // Dimanche - Pas besoin de modifier la date car dimanche correspond au début de la semaine
     iJour = Day();
     iMois = Month();
     iAnnee = Year();
     sJourSemaine = "Sun";
     break;
   case 1: // Lundi - Récupérer la date de la veille soir 1 jour avant
     iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 1));
     iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 1));
     iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 1));
     sJourSemaine = "Mon";
     break;
   case 2: // Mardi - Récupérer la date de 2 jours auparavant
     iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 2));
     iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 2));
     iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 2));
     sJourSemaine = "Tue";
     break;
   case 3: // Mercredi - Récupérer la date de 3 jours auparavant
     iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 3));
     iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 3));
     iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 3));
     sJourSemaine = "Wed";
```

```
      break;
    case 4: // Jeudi - Récupérer la date de 4 jours auparavant
      iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 4));
      iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 4));
      iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 4));
      sJourSemaine = "Thu";
      break;
    case 5: // Vendredi - Récupérer la date de 5 jours auparavant
      iJour = TimeDay(iTime(Symbol(), PERIOD_D1, 5));
      iMois = TimeMonth(iTime(Symbol(), PERIOD_D1, 5));
      iAnnee = TimeYear(iTime(Symbol(), PERIOD_D1, 5));
      sJourSemaine = "Fri";
      break;
    }

    if(iJour < 10) // Si le jour est inférieur à 10, nécessité de rajouter un 0 devant le chiffre
      sJour = "0" + iJour;
    else
      sJour = iJour; // Aucun changement si le jour est supérieur ou égal à 10
    if(iMois < 10) // Si le mois est inférieur à 10, nécessité de rajouter un 0 devant le chiffre
      sMois = "0" + iMois;
    else
      sMois = iMois; // Aucun changement si le mois est supérieur ou égal à 10

    sNomFichier = "Calendrier"+"-"+sMois+"-"+sJour+".csv"; // Affectation du nom correcte du fichier de la
semaine

    iNumeroFichier = FileOpen(sNomFichier,FILE_READ|FILE_CSV,','); // Tentative d'ouverture du fichier
    if(iNumeroFichier == -1) // Vérification si le fichier n'est pas disponible
      {
      // Adresse URL de téléchargement
      sAdresseUrlFichier = "http://www.dailyfx.com/files/Calendar-"+sMois+"-"+sJour+"-"+iAnnee+".csv";
      sNomFichier = "\Calendrier"+"-"+sMois+"-"+sJour+".csv"; // Nom du fichier sur le disque dur
      // Chemin du fichier sur le disque dur
      sDossierDestination = StringConcatenate(TerminalPath(),"\experts\files",sNomFichier);
      URLDownloadToFileA(0, sAdresseUrlFichier, sDossierDestination, 0, 0); // Téléchargement du fichier
      }
    else
      FileClose(iNumeroFichier); // Fermeture du fichier si le fichier est déjà présent sur le disque dur

    if(Day() != iDay) // Vérification si la date du jour a changé
      {
      ObjectsDeleteAll(0, OBJ_LABEL); // Effacement de toutes les nouvelles
      iDay = Day(); // Affectation de la date du jour à la variable iDay
      iNumeroFichier = FileOpen(sNomFichier,FILE_READ|FILE_CSV,','); // Ouverture du fichier
      int iCompteur = 0; // Initialisation du compteur
      int iPixels = 0; // Initialisation de l'espace en pixels entre les nouvelles

      while(!FileIsEnding(iNumeroFichier)) // Vérification si le pointeur se situe à la fin du fichier
        {
        string sDate = FileReadString(iNumeroFichier); // Lecture de la colonne Date du fichier
        string sTime = FileReadString(iNumeroFichier); // Lecture de la colonne Time du fichier
        string sTimeZone = FileReadString(iNumeroFichier); // Lecture de la colonne TimeZone du fichier
        string sCurrency = FileReadString(iNumeroFichier); // Lecture de la colonne Currency du fichier
        string sDescription = FileReadString(iNumeroFichier); // Lecture de la colonne Description du fichier
        string sImportance = FileReadString(iNumeroFichier); // Lecture de la colonne Importance du fichier
        string sActual = FileReadString(iNumeroFichier); // Lecture de la colonne Actual du fichier
        string sForecast = FileReadString(iNumeroFichier); // Lecture de la colonne Forecast du fichier
        string sPrevious = FileReadString(iNumeroFichier); // Lecture de la colonne Previous du fichier

        color cImportance = White; // Si la nouvelle est neutre, la couleur sera blanc
        if(sImportance == "Low")
          cImportance = Yellow; // Si la nouvelle est "Low", la couleur sera jaune
        if(sImportance == "Medium")
          cImportance = Orange; // Si la nouvelle est "Medium", la couleur sera orange
        if(sImportance == "High")
          cImportance = Red; // Si la nouvelle est "High", la couleur sera rouge

        if(StringSubstr(sDate, 0, 3) == "Sun") // Détermine si l'on est dimanche
```

```
      int iJourSemaine = 0;
    if(StringSubstr(sDate, 0, 3) == "Mon") // Détermine si l'on est lundi
      iJourSemaine = 1;
    if(StringSubstr(sDate, 0, 3) == "Tue") // Détermine si l'on est mardi
      iJourSemaine = 2;
    if(StringSubstr(sDate, 0, 3) == "Wed") // Détermine si l'on est mercredi
      iJourSemaine = 3;
    if(StringSubstr(sDate, 0, 3) == "Thu") // Détermine si l'on est jeudi
      iJourSemaine = 4;
    if(StringSubstr(sDate, 0, 3) == "Fri") // Détermine si l'on est vendredi
      iJourSemaine = 5;

    if(iJourSemaine >= DayOfWeek()) // Vérification si le jour de la nouvelle n'est pas déjà passé
      {
      ObjectCreate("TxtNouvelleEco"+iCompteur, OBJ_LABEL, 0, 0, 0); // Création du texte de la nouvelle
      ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_CORNER, 0); // Affichage à partir du coin gauche
      ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_XDISTANCE, 10); // Distance du côté gauche
      ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_YDISTANCE, 10+iPixels); // Distance verticale
      ObjectSet("TxtNouvelleEco"+iCompteur, OBJPROP_COLOR, cImportance); // Détermine la couleur
      ObjectSetText("TxtNouvelleEco"+iCompteur, sDate + " : "+ sDescription, 8); // Texte à afficher
      iPixels = iPixels + 10; // Détermine la distance entre chaque nouvelle
      }
    iCompteur++; // Permet d'avoir un nom différent pour chaque objet
    }
  FileClose(iNumeroFichier); // Fermeture du fichier de nouvelles
  }
}
```

Index des tableaux

Tableau 1 : Les types de données 14
Tableau 2 : Caractères spéciaux 16
Tableau 3 : Tableau des couleurs internet 18
Tableau 4 : Comment nommer les variables 21
Tableau 5 : Mots réservés 21
Tableau 6 : Opérateurs arithmétiques 26
Tableau 7 : Opérateurs d'incrémentation et de décrémentation 27
Tableau 8 : Opérateurs d'affectation 28
Tableau 9 : Opérateurs de comparaison 30
Tableau 10 : Opérateurs logiques 31
Tableau 11 : Opérateurs au niveau du bit 32
Tableau 12: Directives #property 52
Tableau 13 : Unités de temps 72
Tableau 14: Identificateurs de variables des séries de temps 79
Tableau 15 : Valeurs possibles pour les fonctions temporelles 83
Tableau 16: Objets disponibles dans MetaTrader 92
Tableau 17 : Paramètres modifiables pour chaque objet dans MetaTrader 94
Tableau 18 : Différents types d'ordre 104
Tableau 19 : Fonctions pour obtenir de l'information sur les positions 110
Tableau 20 : Variables prédéfinies 112
Tableau 21 : Informations disponibles avec la fonction MarketInfo() 115
Tableau 22 : Fonctions pour obtenir des informations sur le compte 123
Tableau 23 : Format de conversion pour la fonction TimeToStr() 127
Tableau 24 : Boucle while pour passage d'ordre automatique 132
Tableau 25 : Choix de boutons pour la fonction MessageBox() 136
Tableau 26 : Choix d'icônes pour la fonction MessageBox() 136
Tableau 27 : Paramètres disponibles pour la boîte de dialogue 136
Tableau 28 : Réponses possibles d'une boîte de dialogue 137
Tableau 29 : Options disponibles pour l'affichage des indicateurs personnalisés 190
Tableau 30 : Options disponibles pour le style de l'affichage des indicateurs personnalisés 191
Tableau 31 : Options disponibles pour le style des flèches 192

TABLEAU 32 : LISTE DES OPTIONS POUR LE CALCUL DE L'INDICATEUR MOYENNE MOBILE (MM) 196
TABLEAU 33 : LISTE DES OPTIONS POUR LE PRIX UTILISÉ POUR LE CALCUL DE L'INDICATEUR MM 197
TABLEAU 34 : LISTE DES FONCTIONS MATHÉMATIQUES 198
TABLEAU 35 : MODES DISPONIBLES POUR LA FONCTION FILEOPEN() 200
TABLEAU 36 : LISTE DES CODES D'ERREURS ET LEURS SIGNIFICATIONS.....235

Index des figures

FIGURE 1 : IDENTIFICATION DES ZONES DE LA PLATEFORME METATRADER 4 12
FIGURE 2 : VARIABLES EXTERNES DE L'INDICATEUR ICHIMOKU KINKO HYO 23
FIGURE 3 : PARAMÈTRES DE L'INDICATEUR ICHIMOKU KINKO HYO 23
FIGURE 4: DIFFÉRENTES SECTIONS DU CODE SOURCE 42
FIGURE 5 : ALGORITHME DE L'EXPERT "VOTRE PREMIER EXPERT" 60
FIGURE 6 : ICÔNE METAEDITOR DANS LA BARRE D'OUTILS METATRADER 61
FIGURE 7 : METAEDITOR 61
FIGURE 8 : MENU FICHIER 62
FIGURE 9 : FENÊTRE DE CRÉATION D'UN NOUVEAU PROGRAMME 62
FIGURE 10 : ASSISTANT CRÉATION EXPERT CONSULTANT 63
FIGURE 11 : BOUTON COMPILER DANS LA BARRE D'OUTILS DE METAEDITOR 63
FIGURE 12 : COMPILATION DE PREMIER EXPERT RÉUSSIE 64
FIGURE 13 : PREMIER EXPERT DANS LA FENÊTRE NAVIGATEUR DE METATRADER 64
FIGURE 14 : PLUS HAUT/PLUS BAS JOURNÉE PRÉCÉDENTE 66
FIGURE 15 : DÉCOMPTE DES BARRES EN MQL4 67
FIGURE 16: ANALYSE DES 2 BARRES PRÉCÉDENTES 73
FIGURE 17 : OMBRE D'UNE CHANDELLE 76
FIGURE 18 : ILLUSTRATION DU FONCTIONNEMENT DE LA FONCTION IHIGHEST() 79
FIGURE 19 : ILLUSTRATION DU CODE PERMETTANT D'AFFICHER L'HEURE ET LA DATE 83
FIGURE 20 : ILLUSTRATION DU FILTRE HORAIRE 89
FIGURE 21 : LIGNES INDIQUANT PLUS HAUT/BAS DU GRAPHIQUE 96
FIGURE 22 : EXEMPLE DE RECTANGLE ENGLOBANT UNE JOURNÉE 97
FIGURE 23 : RECTANGLE INDIQUANT LA JOURNÉE ÉCOULÉE 97
FIGURE 24 : RECTANGLE INDIQUANT LA JOURNÉE EN COURS 98
FIGURE 25 : TEXTE AU-DESSUS DU RECTANGLE (POSITION INCORRECTE) 99
FIGURE 26 : POINT DE POSITIONNEMENT DU TEXTE 99
FIGURE 27 : TEXTE AU-DESSUS DU RECTANGLE (POSITION CORRECTE) 99
FIGURE 28 : POSITIONNEMENT D'UNE ÉTIQUETTE DE TEXTE 100
FIGURE 29: COMMENTAIRE DE POSITION OUVERTE 105
FIGURE 30 : BOÎTE DE DIALOGUE AVEC LA FONCTION ALERT() 133
FIGURE 31 : EXEMPLE DE LA FONCTION COMMENT() 134

FIGURE 32 : EXEMPLE DE BOÎTE DE DIALOGUE 135
FIGURE 33: BOÎTE DE DIALOGUE POUR MODIFIER LE VOLUME DES POSITIONS 138
FIGURE 34 : EXEMPLE D'UTILISATION DE LA FONCTION PRINT() 139
FIGURE 35 : PARAMÉTRAGE DU SERVEUR FTP 140
FIGURE 36 : ONGLET EMAIL DES OPTIONS DE METATRADER 140
FIGURE 37 : BOUTON EXPERT ADVISORS (DÉSACTIVÉ À GAUCHE ET ACTIVÉ À DROITE) 143
FIGURE 38 : DESCRIPTION D'ERREUR DANS LE JOURNAL 149
FIGURE 39 : FONCTIONNEMENT D'UN STOP SUIVEUR 153
FIGURE 40 : MENU POUR ACCÉDER À LA LISTE DES VARIABLES GLOBALES DANS METATRADER 163
FIGURE 41 : FENÊTRE PERMETTANT DE VISUALISER LA LISTE DES VARIABLES GLOBALES DANS METATRADER 164
FIGURE 42 : EXEMPLE D'APPLICATION #1 DES VARIABLES GLOBALES 165
FIGURE 43 : EXEMPLE D'APPLICATION #2 DES VARIABLES GLOBALES 166
FIGURE 44 : TABLEAU À UNE DIMENSION 174
FIGURE 45 : TABLEAU À 2 DIMENSIONS 174
FIGURE 46 : TABLEAU À 4 DIMENSIONS 175
FIGURE 47 : INITIALISATION D'UN TABLEAU AVEC MOINS DE PARAMÈTRES QUE REQUIS 176
FIGURE 48 : FRÉQUENCES RELATIVES 185
FIGURE 49 : MÉMOIRES TAMPONS 186
FIGURE 50 : AFFICHAGE D'UN INDICATEUR PERSONNALISÉ (0 = FENÊTRE PRINCIPALE; 1,2 = FENÊTRE SÉPARÉE) 188
FIGURE 51 : LOCALISATION DE L'AFFICHAGE POUR LA FONCTION INDICATORSHORTNAME() 190
FIGURE 52 : APPARENCE DE L'INDICATEUR « PREMIER INDICATEUR » 195
FIGURE 53 : EXEMPLE D'ÉCRITURE DANS UN FICHIER 206
FIGURE 54 : APPARENCE DU CALENDRIER ÉCONOMIQUE DE DAILYFX 207
FIGURE 55 : AFFICHAGE DES NOUVELLES ÉCONOMIQUES À PARTIR D'UN FICHIER 210
FIGURE 56 : ILLUSTRATION DE LA FONCTION WINDOWFIRSTVISIBLEBAR() 217
FIGURE 57 : AFFICHAGE D'UN CALENDRIER ÉCONOMIQUE SUR LE GRAPHIQUE DE L'EXPERT 226
FIGURE 58 : PRINCIPE DE BASE DES CODES DE PROTECTION 227
FIGURE 59 : RÉSEAU DE NEURONES ARTIFICIELS 230
FIGURE 60 : MATRICE DES GAINS DE L'INTERACTION ENTRE LE MARCHÉ ET UN SPÉCULATEUR 230