

Notions avancées en scripting mIRC

par Erwan Guinier (eguinier.developpez.com)

Date de publication : 19/02/2008

Dernière mise à jour : 02/03/2008

Tutoriel sur les notions avancées en scripting mIRC.

Prélude

- I-A - Introduction
- I-B - Pré-requis
- I-C - Remerciements
- II - Les boîtes de dialogue
 - II-A - Les bases
 - II-B - Insérer des objets
 - II-C - Modifier les contrôles
 - II-D - Les événements
- III - Les expressions régulières
 - III-A - Les métacaractères
 - III-B - Le back-slash
 - III-C - Les ensembles
 - III-D - Les commandes
 - III-E - Les identifiants
- IV - La communication DDE
 - IV-A - Les sujets DDE
 - IV-B - Les commandes
- V - Le fichier mirc.ini
 - V-A - L'architecture du fichier
 - V-B - Charger vos scripts à l'ouverture
 - V-C - Commandes et identifiants associés au fichier [.ini]
- VI - Les variables binaires
 - VI-A - Les problèmes théoriques
 - VI-B - Les commandes
- VII - L'optimisation (1/2)
 - VII-A - Les optimisations basiques
 - VII-A-1 - Les alias
 - VII-A-2 - Les boucles
 - VII-A-3 - Les conditions
 - VII-A-4 - L'identifiants \$ticks
 - VII-B - Le file handling
 - VII-B-A - Le principe
 - VII-B-B - Les commandes
 - VII-B-C - Les identifiants
 - VII-C - Les événements
 - VII-C-A - La gestion des BDD
 - VII-C-B - Les préfixes
 - VII-C-C - Les raws
- VIII - L'optimisation (2/2)
 - VIII-A - Le submenu
 - VIII-A-1 - Une évolution des popups de base
 - VIII-A-2 - Construction d'un submenu
 - VIII-B - Commandes et techniques de scripting

Prélude

I-A - Introduction

Bonjour et bienvenue dans ce tutoriel sur les notions avancées en scripting mIRC.

Je vais tâcher, tout au long de ce cours, de vous éclairer sur des outils peu ou pas utilisés en scripting mIRC mais qui sont pourtant très importants dès lors que l'on souhaite progresser de manière significative dans ce domaine.

Je vous souhaite une bonne lecture,

Whether.

I-B - Pré-requis

Afin de comprendre ce tutoriel dans sa totalité, vous devez connaître **le cours sur les notions intermédiaires en scripting mIRC** et/ou être un scripteur déjà expérimenté.

I-C - Remerciements

Avant de commencer les développements relatifs à ce tutoriel, je tiens sincèrement à remercier Kleak, scripteur mIRC de talent, pour le travail appréciable et les critiques constructives dont il a su me faire part, le site developpez.com pour m'héberger et m'avoir fournis les outils et l'aide nécessaire pour mener à bien ce projet.

Notez bien que ce cours ne dresse pas une liste exhaustive des notions "avancées" en scripting mIRC et que les parties suivantes sont en cours de rédaction :

- Les boîtes de dialogue et les DLL (section n°2)
- Les fenêtres-images : bases et animations (1/2)
- Les fenêtres-images : la création de GUI (2/2)
- Les fenêtres-images : introduction à la 3D (si la demande se fait sentir)
- Les sockets

De plus, si vous jugez qu'un point important n'a pas été abordé ou qu'il n'est pas assez développé, n'hésitez pas à me contacter par message privé. Pour finir, sachez que tout avis, critique ou suggestion sont les bienvenus.

II - Les boîtes de dialogue

La maîtrise des boîtes de dialogue constitue souvent un tournant et marque une vraie différence entre les scripteurs sachant les utiliser et ceux qui ne le savent pas.

Je vous propose de découvrir à quoi elles peuvent bien servir dans cette première partie qui va traiter uniquement des boîtes de dialogue.

La seconde, quant à elle, va vous apprendre à utiliser les DLL et effectuer des actions que mIRC ne peut, en théorie, pas exécuter.


II-A - Les bases

Les boîtes de dialogue, que je vais appeler tout simplement dialogue ou BDD durant cette partie, sont un outil indispensable dès lors que vous commencez à progresser en scripting mIRC.

En effet, cet outil permet à l'utilisateur d'envoyer des requêtes directement à mIRC de manière claire et simple.

La création d'un dialogue, qui doit être réalisée dans l'onglet "remote" de l'éditeur de script, s'effectue à l'aide de la commande /dialog.

En voici les principales caractéristiques : **/dialog -maxdhlsrbponkcvie nom [table] [x y w h] [texte]**.

 *Note : je ne vais pas énumérer tous ces paramètres les uns après les autres. Ils sont tous détaillés dans l'aide de mIRC qui se trouve normalement dans le dossier racine de votre script.*

Maintenant, entrons dans le vif du sujet :

```
dialog test {  
  title "Boîte de dialogue"  
  size -1 -1 200 100  
  option dbu  
}
```

Ce script se décompose de la manière suivante :

dialog test { précise que l'on souhaite créer un dialogue dont le nom sera "test".

title "" permet de spécifier le titre du dialogue.

size -1 -1 200 100 indique les positions et la taille du dialogue.

option dbu prend une base différente de celle de mIRC pour afficher le dialogue (je vous conseille de l'utiliser).

```
/dialog -m test test
```

Va permettre de l'ouvrir. Question : pourquoi utiliser le paramètre "-m" ? Car ce paramètre permet de créer un dialogue "sans mode". C'est-à-dire qu'il n'arrête ni ne retourne une valeur au script appelant, et qu'il peut être affiché indéfiniment sans avoir quelque influence sur d'autres scripts.

Retournons à la commande /size qui permet de positionner et de modifier la taille du dialogue. Cette commande est suivie de quatre chiffres. Les deux premiers permettent de positionner le dialogue (ici, -1 et -1 précisent que le dialogue sera centré à l'intérieur de la fenêtre). Les deux derniers indiquent respectivement la longueur et la largeur de la boîte.

II-B - Insérer des objets

Il est possible d'insérer une grande quantité d'objets dans une boîte de dialogue. Comme je l'ai déjà dit précédemment, je ne vais pas lister tous les objets qui peuvent être inclus dans un dialogue, l'aide mIRC est faite pour ça. Mais je vais vous expliquer à quoi ces objets peuvent servir et comment faire pour les utiliser le mieux possible.

```
dialog test {  
  title "Boîte de dialogue"  
  size -1 -1 200 100  
  option dbu  
  text "Texte de base", 1, 1 5 50 7  
}
```

Nous avons ajouté le contrôle "text" qui permet d'afficher du texte dans une boîte de dialogue.


Intéressons-nous à cette ligne : text "Texte de base", 1, 1 5 50 7.

L'insertion d'un objet dans une BDD se fait, dans la majorité des cas (je vous expliquerai les exceptions plus tard), comme ceci :

objet "texte à inclure", n° objet (NDO), position de l'objet (toujours 4 chiffres), contrôle de l'objet.

 *Note : les paramètres "texte à inclure" et "contrôle de l'objet" sont facultatifs.*

Ici, rien de compliqué. Si vous souhaitez inclure une boîte d'édition, il suffit de remplacer l'objet "text", par "edit" et modifier le numéro de l'objet.

 *Note : modifier le numéro de l'objet est obligatoire. Vous ne pouvez, en aucun cas, avoir deux objets qui possèdent le même numéro. Si jamais cela se produit, vous ne pourrez tout simplement pas ouvrir votre dialogue et mIRC vous renverra un message de la sorte : "id duplicate at line 3".*

Pour l'instant, le NDO doit vous sembler tout à fait inutile et n'avoir aucune influence sur le dialogue. Et, dans l'absolu, vous avez raison. Car, si vous modifiez ce numéro, vous constatez qu'il ne se passe rien. Je vous expliquerai cela dans la troisième partie de ce tutoriel.

Passons maintenant à ce qui rebute un nombre impressionnant de personne en scripting mIRC : positionner un objet correctement dans une BDD.

Toujours dans cette même ligne : `text "Texte de base", 1, 1 5 50 7` la position de cet objet se fait avec les quatre derniers chiffres. C'est-à-dire `1 5 50 7`.

Le premier chiffre (1) permet de positionner l'objet selon l'axe horizontal. C'est-à-dire de droite à gauche dans le dialogue.

Le deuxième chiffre (5) permet de positionner l'objet selon l'axe vertical. C'est-à-dire de haut en bas dans le dialogue.

Le troisième chiffre (50) permet de donner une taille, en longueur, à l'objet.

Le quatrième chiffre (7) permet de donner une taille, en largeur, à l'objet.

Par exemple, si nous souhaitons centrer l'objet "text" dans le haut de la BDD, nous devons modifier le premier chiffre. Ce qui va donner :

```
dialog test {
  title "Boîte de dialogue"
  size -1 -1 200 100
  option dbu
  text "Texte de base", 1, 90 5 50 7
}
```

Les objets de base :

Nous allons maintenant étudier les objets de base que vous devez impérativement savoir utiliser si vous souhaitez réaliser des boîtes de dialogue un minimum présentables.

text : permet d'insérer du texte.

edit : permet d'insérer une barre d'édition.


button : insère un bouton dans la BDD.

list : insère une liste qui permet de stocker des valeurs.

radio : insère un radio dans le dialogue.


check : insère un check dans le dialogue.

combo : permet d'insérer un combo dans la BDD où des valeurs peuvent être stockées.

 *Note : la différence entre un check et un radio est qu'un seul radio peut être coché (si un radio est déjà coché, il est décoché automatiquement) alors qu'avec l'objet check, vous pouvez le cocher ou le décocher sans que cela n'ait d'influence sur les autres objets check.*

```
dialog test {
  title "Boîte de dialogue"
  size -1 -1 200 82
  option dbu
  text "Voici une liste non exhaustive des objets de base à connaître dans une BDD :", 1, 2 3 190 7
  edit "", 2, 2 13 70 10
  button "Bouton" 3, 2 25 70 10
  list 4, 2 37 70 40, size
  radio "Un radio" 5, 168 13 40 8
  check "Un check" 6, 168 25 40 8
  combo 8, 87 13 70 90, drop
}
```

Cet exemple illustre les objets dont je vous ai parlés.


 *Note : les paramètres spécifiés en fin de ligne servent à préciser des modes d'affichage différents. Comme ils sont très nombreux et très variés selon l'objet que vous souhaitez insérer, je vous conseille d'aller jeter un coup d'oeil à l'aide mIRC.*

Maintenant que vous savez comment créer une boîte de dialogue, nous allons passer à la modification des objets. C'est-à-dire leur apparence.

II-C - Modifier les contrôles

La modification des objets est une chose très importante lorsque vous utilisez des boîtes de dialogue car cela va permettre de modifier l'apparence dudit dialogue.

Ces modifications s'effectuent à l'aide de la commande /did. Cette commande, lorsqu'elle est suivie d'un ou plusieurs paramètres permet de modifier un objet.

 *Note : voici le format de la commande : /did -f**tebv**hnm**cukradiog**jsl **nom-du-dialogue** **numéro-id** [**n**] [**texte** | **nomdefichier**]*

Prenons un exemple. Ouvrez le dialogue que je vous ai montré dans la partie ci-dessus puis tapez ceci dans la fenêtre statut :

```
//did -h test 2
```

Vous allez voir que la boîte d'édition va disparaître. Et, si vous tapez ceci :

```
//did -v test 2
```

Qu'elle va réapparaître.

Ici, rien de vraiment très compliqué. Pour vous faciliter la tâche, je vous donne la liste complète des modifications que supporte la commande /did.

- a : ajoute la ligne de texte à la fin
- b : désactive l'id
- c : coche la boîte check/bouton radio, la ligne de la liste/combo
- d : efface la Nième ligne
- e : active l'id
- f : met le focus sur l'id
- g : met une nouvelle image icône/bmp dans un contrôle icon (did -g nom id [n] nomdefichier)
- h : cache l'id
- i : insère le texte à la Nième ligne
- j : efface les réglages édités dans une boîte d'édition
- k : fonctionne avec -cu, garde les autres sélections dans une liste
- l : décoche la case à cocher d'un objet dans un contrôle list checkbox
- m : désactive la boîte d'édition
- n : active la boîte d'édition
- o : écrase la Nième ligne par le texte
- r : efface tout le texte dans l'id

- s : coche la case à cocher d'un objet dans un contrôle list checkbox
- t : met l'id du bouton par défaut
- u : décoche la boîte check/bouton radio, la ligne de la liste/combo
- v : rend l'id visible
- z : efface la largeur de la barre de défilement (scrollbar) horizontale dans une liste

II-D - Les événements

Cette dernière sous-partie concernant les boîtes de dialogue est la plus importante. En effet, vous allez apprendre ici à faire réagir les objets que vous avez insérés dans la BDD, toujours à l'aide, de la commande /did mais cette fois-ci en utilisant les objets. Par exemple, nous allons ici voir comment effectuer une action lorsque vous allez cliquer sur un bouton.

L'événement qui permet de gérer les BDD est celui-ci :

```
on *:dialog:nom:événement:NDO: { }
```

Où nom est le nom de la boîte de dialogue, événement concerne l'action que vous avez effectuée (clic droit, gauche, double-clic, entrer du texte dans une boîte d'édition, etc.) et NDO le numéro de l'objet.

Normalement, si vous êtes un minimum logique, vous devriez commencer à comprendre pourquoi numéroté les objets que l'on positionne est très important et qu'il ne faut surtout pas numéroté deux objets de la même manière.

Reprenons notre précédent dialogue :

```
dialog test {
  title "Boîte de dialogue"
  size -1 -1 200 82
  option dbu
  text "Voici une liste non exhaustive des objets de base à connaître dans une BDD :", 1, 2 3 190 7
  edit "", 2, 2 13 70 10
  button "Bouton" 3, 2 25 70 10
  list 4, 2 37 70 40, size
  radio "Un radio" 5, 168 13 40 8
  check "Un check" 6, 168 25 40 8
  combo 8, 87 13 70 90, drop
}
```

Et ajoutons ensuite cette ligne :

```
on *:dialog:test:sclick:3: { echo 2 vous avez cliquez sur le bouton (objet n° 3) }
```

Maintenant, ouvrez le dialogue et cliquez sur le bouton. Normalement, vous devriez voir le message suivant apparaître "Vous avez cliquez sur le bouton "objet n° 3"

Avant de faire la même chose pour les autres objets, nous allons plutôt nous intéresser aux différentes actions qu'il est possible d'effectuer :

active : s'active sur le dialogue quand il est actif

init : juste avant que le dialogue soit affiché, les contrôles peuvent être initialisés dans cet événement. L'identité est nulle.

close : quand le dialogue est fermé.

edit : le texte dans la boîte d'édition ou la boîte combo a changé.

sclick : c'est l'action que nous avons utilisé précédemment. Un seul clic dans la liste/boîte combo, cocher/décocher des boutons radio/check ou cliquer sur un bouton.

dclick : double clic dans une liste/boîte combo.

menu : un item du menu a été sélectionné.

scroll : la position du contrôle scroll a changé.

Le cas particulier de l'événement init :

Cet événement, comme vous venez de le lire, correspond au moment où le dialogue s'ouvre. Par exemple, c'est ici que vous allez préciser si tel ou tel *check* doit être coché, etc.

La théorie n'étant ici pas très simple à comprendre, voici un exemple pratique :

```
; Tapez /dialog -m case case
dialog case {
  title "L'événement init"
  size -1 -1 87 23
  option dbu
  check "check n° 1" 1, 1 3 37 9
  check "check n° 2" 2, 1 13 37 9
  button "Ok" 3, 65 13 20 9, ok
}

on *:dialog:case:init:*: {
  ; On vérifie si les variables existent, si la condition est vérifiée, on coche les checks à
  l'aide de la commande /did.
```

```
if (%case.1) did -c $dname 1
if (%case.2) did -c $dname 2
}

on *:dialog:case:sclick:1: {
; Cette ligne s'active lorsque le check n°1 est décoché (où zéro est retourné).
if ($did($dname,1).state == 0) unset %case.1
; Cette ligne s'active lorsque le check n°1 est coché (où un est retourné).
elseif ($did($dname,1).state == 1) set %case.1 ON
}

; Cet événement fonctionne de la même manière que le précédent sauf qu'il concerne le check n°2.
on *:dialog:case:sclick:2: {
if ($did($dname,2).state == 0) unset %case.2
elseif ($did($dname,2).state == 1) set %case.2 ON
}
```

Comme vous avez pu le constater, lorsque vous cochez un check, que vous fermez le dialogue et que vous le relancez, le check en question est toujours coché. De plus, vous devez vous dire que ce code est un peu "lourd", voici donc une manière plus propre de réaliser exactement la même chose.

```
dialog case {
title "L'événement init"
size -1 -1 87 23
option dbu
check "check n° 1" 1, 1 3 37 9
check "check n° 2" 2, 1 13 37 9
button "Ok" 3, 65 13 20 9, ok
}

on *:dialog:case:init:*: {
if (%case.1) did -c $dname 1
if (%case.2) did -c $dname 2
}

on *:dialog:case:sclick:*: {
if ($did == 1) {
if ($did($dname,1).state == 0) unset %case.1
elseif ($did($dname,1).state == 1) set %case.1 ON
}
if ($did == 2) {
if ($did($dname,2).state == 0) unset %case.2
elseif ($did($dname,2).state == 1) set %case.2 ON
}
}
```


L'astuce est toute simple, il suffit de préciser de quel objet il s'agit lors d'un simple clic à l'aide de l'identifiant did.

III - Les expressions régulières

Pas facile de rédiger un chapitre sur les expressions régulières. Outil pourtant très puissant dans le langage mIRC. Effectivement, les expressions régulières, que nous allons ici appeler « regex » (*regular expression*), permettent de comparer, valider ou d'effectuer une recherche dans une chaîne de caractères (autrement dit, une phrase).

Les regex fonctionnent en utilisant des caractères et des métacaractères : elles permettent de préciser une position, une condition, etc.

Pour essayer de faire simple, les expressions régulières sont des systèmes de vérifications, validations ou comparaisons très poussés et très performants.

 *Note : une regex retourne 1 si la chaîne est vérifiée, sinon, elle retourne 0.*

III-A - Les métacaractères

Introduction :

```
//echo -s $regex(bonjour,b)
```

Retourne 1 (car il y a bien un « b » dans bonjour).

```
//echo -s $regex(bonjour,z)
```

Retourne 0 (car il n'y a pas de « z »).

Vous pourrez constater que si vous faites :

```
//echo -s $regex(bonjour,B)
```

la regex va retourner 0 car cet outil est sensible aux majuscules / minuscules.

Continuons : si vous faites :


```
//echo -s $regex(bonjour,bj)
```

0 sera retourné car l'expression se lit de gauche à droite, et que « b » est bien dans bonjour, « j » également, mais pas « bj ». Donc 0 est retourné.

Nous savons donc que les expressions régulières sont sensibles aux majuscules / minuscules, et que tous les paramètres doivent être validés pour que la valeur 1 soit retournée.

Définition des métacaractères :

Les métacaractères sont des symboles qui permettent de préciser des conditions à l'expression régulière. Grâce à eux, vous allez vite comprendre toute la puissance des regex.

 *Note : les métacaractères sont très sensibles. Soyez donc attentifs aux informations que vous souhaitez rechercher, sans quoi votre code va se retrouver faussé.*

Le métacaractère ^ :

Il précise le début de la chaîne.

```
//echo -s $regex(bonjour,^b)
```

va retourner 1, car « bonjour » commence bien par un « b ».

Mais :

```
//echo -s $regex(bonjour,^o)
```

va retourner 0, car « bonjour » ne commence pas par un « o ».

Vous pouvez également préciser plusieurs caractères comme :

```
//echo -s $regex(bonjour,^bon)
```

qui va retourner 1, car le début du mot est bien « bon ».

```
//echo -s $regex(bonjour,^bon)
```

qui va retourner 1, car le début du mot est bien « bon ».

Le métacaractère \$:

Ce métacaractère permet de spécifier « la fin de la chaîne ».

```
//echo -s $regex(bonjour,jour$)
```

car la partie « jour » est bien à la fin de « bonjour ».

```
//echo -s $regex(bonjour,u$) va retourner 0  
//echo -s $regex(bonjour,(^bonjour$)va retourner 1
```

Le métacaractère * :

Il permet de préciser « zéro ou plusieurs occurrences ».

Occurrence ? Qu'est-ce que c'est ?

Une occurrence est une répétition de caractères. Par exemple, si je dis « hello », il y a deux occurrences de « l » car cette lettre est deux fois dans la chaîne.

```
//echo -s $regex(bonjour,o*)
```

va retourner 1, car il y a bien un « o » dans la chaîne.

```
//echo -s $regex(bonjour,z*)
```

Va également retourner 1 ! Pourquoi ? Car le joker dit bien « zéro ou plusieurs occurrences » (je dois vous avouer que moi non plus, je ne comprends pas vraiment ce métacaractère, mais bon...).

Le métacaractère + :

Il permet de préciser « au moins une occurrence dans la chaîne ».

```
//echo -s $regex(bonjour,j+)
```

car il y a bien un « j » dans « bonjour ».

```
//echo -s $regex(bonjour,o+) va retourner 1  
//echo -s $regex(bonjour,z+) va retourner 0
```

Le métacaractère ? :

Offre une possibilité très intéressante qui est de vérifier « zéro ou une seule occurrence ».

```
//echo -s $regex(bonjour,j?)
```

car il y a bien un « j » et un seul dans « bonjour ».

```
//echo -s $regex(bonjour,o?)
```

car il y a deux « o », donc la condition n'est pas vérifiée !

Le métacaractère . :

Il permet de préciser n'importe quel caractère dans la chaîne.

```
//echo -s $regex(bonjour,b.n)
```

car il y a bien un caractère entre « b » et « n » (à savoir la lettre « o »).

```
//echo -s $regex(bonjour,b.o)
```

car il n'y a pas de caractère entre « b » et « o ».

III-B - Le back-slash

Maintenant que nous avons passé en revue les principaux métacaractères, nous allons passer au Back-Slash « \ », qui permet d'inclure premièrement un métacaractère dans la chaîne.

Déterminer des métacaractères :

```
//echo -s $regex(Bonjour+PuG,Bonjour+PuG)
```

Va retourner 1, car la présence du + nécessite l'utilisation du back-slash.

```
//echo -s $regex(Bonjour+PuG,Bonjour+PuG)
```

Va retourner 0.

```
//echo -s $regex(www.*,www+*)
```

Va retourner 1.

```
//echo -s $regex(www.*,www+*)
```

Va retourner 0, car il faut ici un back-slash.

Liste des caractères :

Les listes de caractère permettent de spécifier des ensembles de lettres, chiffres, caractères, etc. Retenez-les bien, car elles vont être très importantes pour la suite du chapitre. Et vont permettre de simplifier la structure des expressions régulières lorsque nous allons en créer des plus complètes.

b : début ou fin de mot.

B : non-début ou non-fin de mot.

d : un chiffre.

D : pas de chiffre.

s : une espace.

S : pas d'espace.

w : un mot.

W : pas de mot.

Exemple :

```
//echo -s $regex(bonjour,bon)
```

Va retourner 1, car « bon » est bien au début de « bonjour ».

```
//echo -s $regex(Bonjour,\d)
```

Va retourner 0, car il n'y a pas de chiffre.

```
//echo -s $regex(Bonjour2,\d)
```

va retourner 1, car il y a un chiffre.

```
//echo -s $regex(Bonjour à tous !,s)
```

va retourner 1, car il y a bien un ou plusieurs espaces dans la chaîne.

III-C - Les ensembles

Les ensembles de caractères vont vous expliquer comment spécifier des ensembles de caractères à rechercher dans une chaîne. Par exemple, si vous souhaitez rechercher tous les numéros de 0 à 6, vous allez devoir créer un ensemble. Cette sous-partie va vous expliquer comment procéder.

Les parenthèses

Les délimitations « () » permettent de spécifier quelle chaîne est concernée par un métacaractère.


Illustration :

```
//echo -s $regex(bonjour,(b|n)+jour)
```

va retourner 1, car il y a bien « b » ou « n » dans « bon ».

```
//echo -s $regex(bonjour,(b|n)+jour$)
```

va retourner 1 car il y a bien « b » ou « n » dans « bon » et que « bonjour » se termine par « jour ».

 *Note : vous pouvez bien sûr combiner les parenthèses avec tous les autres métacaractères vus plus haut.*

Les accolades :

Les « { } » permettent ici de vérifier le nombre de présences dans la chaîne. Où :

a{N} : N rencontre avec a.

a{N,P} : entre N et P rencontre avec a.

a{N,} : au minimum N rencontre de a.

Par exemple, dans « script mIRC », il y a **i{2}** (deux occurrences de la lettre « i »).

Mais vous auriez également pu mettre **i{1,3}** pour préciser une recherche pour un « i », deux ou trois.

Sinon **i{1,}** pour vérifier s'il y avait bien un « i » dans la chaîne.


Exemple :

```
alias testreg {  
  var %reg = o{1,2}  
  echo -s $regex(bonjour,%reg)  
}
```

va retourner 1, car il y a bien un « o » dans « bonjour ».

```
alias testreg {  
  var %reg = b{1}  
  echo -s $regex(bonjour,%reg)  
}
```

va retourner 1, car il y a bien un « b » dans « bonjour ».

 *Note : dans le cas des accolades, vous êtes obligés de créer une variable locale avant d'effectuer la comparaison avec les regex. Cette méthode va également vous permettre d'alléger votre expression.*

Les crochets


Les « [] » permettent de spécifier des ensembles de caractères.

[abc] les caractères a, b et c.

[a-z] de a à z en minuscules.

[A-Z] de A à Z en majuscules.

[0-9] de 0 à 9.

 *Note : vous pouvez bien sûr créer vous-mêmes vos ensembles de caractères. Comme par exemple [0-3] (tous les chiffres de 0 à 3).*

```
//echo -s $regex(bonjour,[a-z])
```

retourne 1, car il y a bien des caractères de la liste dans « bonjour ».

```
//echo -s $regex(10,[0-9])
```

retourne 1, car 1 et 0 sont bien dans entre 0 et 9

Les [:classes:]

Les « [:classe:] » permettent de spécifier un ensemble de caractères.

[:alpha:] : lettre.

[:alnum:] : chiffre ou lettre.

[:ascii:] : caractères ASCII.

[:blank:] : espace et tabulation.

[:cntrl:] : caractère de contrôle.

[:digit:] : chiffre.

[:graph:] : caractère imprimable sauf espace.

[:lower:] : minuscule.

[:print:] : caractère imprimable.

[:punct:] : ponctuation (? ! . ; ,).


[:space:] : n'importe quelle espace blanche.

[:upper:] : majuscule.

```
//echo -s $regex(bonjour,[[:alpha:]]) retourne 1
//echo -s $regex(bonjour,[[:punct:]]) retourne 0 car il n'y a pas de ponctuation
//echo -s $regex(bonjour !,[[:punct:]]) retourne 1 car ! marque la ponctuation
//echo -s $regex(Bonjour2,[[:digit:]]) retourne 1 car il y a bien un chiffre
//echo -s $regex(Bonjour,[[:space:]]) retourne 0 car il n'y a pas d'espace dans la chaîne.
//echo -s $regex(Bonjour PuG,[[:space:]]) retourne 1 car il y a un espace entre bonjour et PuG.
```

III-D - Les commandes

Les commandes, avec les expressions régulières, permettent de faire un paquet de choses. Elles sont également très pratiques car elles nous donnent la possibilité « d'élargir les champs de recherche » dans les chaînes.

 *Note : à partir de maintenant, une expression régulière peut retourner d'autres chiffres que 0 et 1. Et je vais vous expliquer pourquoi.*

La commande /g

/g retourne le nombre de présences dans la chaîne du caractère ou de la sous-chaîne donnée.

Exemple :


```
//echo -s $regex(Bonjour,/o/g)
```

va retourner 2, car il y a deux fois « o » dans la chaîne.

Vous comprenez maintenant pourquoi 2 peut être retourné ? Imaginez le contraire si une expression régulière était « obligée » de renvoyer soit 0 ou 1. Ici, elle renverrait 1, mais la commande /g serait alors devenue totalement inutile, puisqu'elle sert à retourner le nombre d'occurrences.

```
//echo -s $regex(bonjour,/b/g)
```

va retourner 1, car il y a une fois « b » dans « bonjour ».

 *Note : ici aussi, le traitement est sensible aux minuscules / majuscules.*

La commande /i

/i permet de traiter la chaîne sans tenir compte des majuscules ou des minuscules.

Exemple :

```
//echo -s $regex(bonjour,/b/i)
```

va retourner 1, car « b » est bien dans « bonjour ».

```
//echo -s $regex(Bonjour,/b/i)
```

va retourner 1, car le traitement n'est plus sensible aux majuscules / minuscules. Comme « b » ou « B » est bien dans la chaîne, 1 est retourné.

La commande /x

/x permet de ne pas tenir compte des espaces dans la chaîne.

Exemple :

```
//echo -s $regex(bonjour,b o)
```

va retourner 0, car il y a un espace.

```
//echo -s $regex(bonjour,/b o/x)
```


va retourner 1, car /x ignore les espaces.

III-E - Les identifiants

Rappeler une chaîne

Le `regm()` permet le rappel d'une chaîne en précisant un nom et / ou un chiffre. Par exemple, supposons que pour une raison X vous ayez besoin de réutiliser le « o » de « bonjour », ou encore si vous avez besoin d'obtenir le nombre d'occurrences d'un caractère dans une chaîne, vous allez avoir besoin de les rappeler. Les expressions régulières vous proposent pour cela une série de commandes très efficaces.

Format : `$regml([nom],N)`

 *Note : Le [nom] n'est pas obligatoire.*

Exemple :

```
$regex(bonjour,/o/g)
```

```
2
```

```
$regml(0)
```

```
2
```

retourne 2, car il y a deux « o » dans « bonjour ».

```
$regml(1) et $regml(2)
```

```
o o
```

vont chacun retourner o, car il y a deux « o ».


```
$regml(3)
```

ne va rien retourner, car il n'y a pas 3 « o » dans « bonjour ».

L'identifiant `$regml` est très intéressant car il permet d'isoler rapidement des caractères, sans avoir besoin d'effectuer plusieurs commandes.

Retirer ou remplacer des caractères dans une chaîne

\$regsub permet de retirer ou de remplacer certains caractères d'une chaîne. Format :
\$regsub([nom],chaîne,commande,caractère-à-remplacer,%variable)

 *Note : ici aussi, le [nom] est optionnel.*

Exemple :

```
alias ttrego {  
    var %txt = bonjour à tous !  
    echo -s $regsub(bonjour à tous,/o/g,O,%txt)  
    echo -s %txt  
}
```

```
3  
bOnjOur à tOus
```

va retourner 3, car il y a 3 « o » et le « bonjour à tous » va se transformer en « bOnjOur à tOus ». Car les « o » ont été remplacés par des « O ». Encore un identifiant très utile, non ? :)

IV - La communication DDE

La communication DDE permet à des applications externes de quérir des informations ou encore de contrôler mIRC. En plus simple, avec le contrôle DDE, vous pouvez obtenir des informations sur l'état de mIRC par rapport à un serveur, le fichier central [.ini] de votre client, ou encore de connaître les paramètres d'un autre mIRC lancé en parallèle.

IV-A - Les sujets DDE

Voici la liste des transactions et de sujets que mIRC supporte (les identifiants et commandes présents dans ce sous-chapitre sont expliqués dans le chapitre suivant).

Channels

Sujet : CHANNELS

Type de transaction : XTYP_REQUEST

Utilité : permet de lister les salons où vous vous trouvez.

Format : \$dde(mirc,channels)

Exemple :

```
alias dchan { echo -a Vous êtes sur les salons suivants $dde(mirc,channels) }
```

Command

Sujet : COMMAND


Type de transaction : XTYP_POKE

Utilité : ce contrôle dit à mIRC d'exécuter la commande que vous lui ordonnez.

Format : /dde mirc command "" /commande

Exemple :

```
alias dcom { dde mirc command "" /echo -s Vous avez demandé un écho dans le statut à $time }
```

 **Attention :** « *command* » est le sujet DDE et */commande* est la commande que vous donnez à mIRC. Le « / » n'est pas optionnel ! Vous devez le laisser pour que la requête s'exécute correctement !

Connect

Sujet : CONNECT

Type de transaction : XTYP_POKE

Utilité : permet la connexion à un serveur

Format : /dde mirc connect "" serveur:port,salon,numéro

Où numéro est optionnel, si = 0, la fenêtre du salon ne sera pas activée, si = 1, elle le sera.

Exemple :

```
alias dconnect { dde mirc connect "" irc.epiknet.org:6667,#scripts,1 }
```

Connected

Sujet : CONNECTED

Type de transaction : XTYP_REQUEST

Utilité : donne les informations sur votre connexion à un serveur (connecté, déconnecté, ou encore en cours de connexion).

Format : \$dde(mirc,connected)

Exemple :

```
alias dconnected { echo -s Vous êtes actuellement $dde(mirc,connected) }
```

Exename

Sujet : EXENAME

Type de transaction : XTYP_REQUEST

Utilité : retourne le chemin complet de mirc.exe

Format : \$dde(mirc,exename)

Exemple :

```
alias dexe { echo -s Chemin mIRC $dde(mirc,exename) }
```

Infile

Sujet : INFILE

Type de transaction : XTYP_REQUEST

Utilité : retourne le chemin du fichier [.ini] de mIRC.

Format : \$dde(mirc,infile)

Exemple :

```
alias dinfile { echo -s Le fichier [.ini] de mIRC est $dde(mirc,infile) }
```

Nickname

Sujet : NICKNAME

Type de transaction : XTYP_REQUEST

Utilité : retourne votre pseudonyme

Format : \$dde(mirc,pseudonyme)

Exemple :

```
alias dpseudo { echo -s Pseudonyme $dde(mirc,pseudonyme) }
```


Port

Sujet : PORT

Type de transaction : XTYP_REQUEST

Utilité : informe sur le port utilisé pour se connecter à IRC.

Format : \$dde(mirc,port)

 *Note : comme le port 6667 est utilisé par défaut, je ne vous illustre pas le cas présent.*

Server

Sujet : SERVER

Type de transaction : XTYP_REQUEST

Utilité : donne le serveur sous lequel vous êtes connectés ou, par défaut, le dernier serveur sur lequel vous avez été.

Format : \$dde(mirc,server)

Exemple :

```
alias dserv {
  if (!$server) echo -s Le dernier serveur sur lequel je me suis connecté a été $dde(mirc,server)
  else echo -s Je suis connecté sur $dde(mirc,server)
}
```

Users

Sujet : USERS


Type de transaction : XTYP_REQUEST

Utilité : permet d'avoir la liste des pseudonymes d'un salon.

Format : \$dde(mirc,users,salons)

Exemple :

```
alias duser {  
    if ($active == $chan) echo -t # Liste des pseudonymes du salon $dde(mirc,users $+ , $+ $active)  
    else return  
}
```

 **Attention** : plus le salon va être important, plus la liste des pseudonymes va être longue. Parfois même plusieurs milliers de caractères ; donc, si vous devez travailler sur cette liste, préparez des applications capables de traiter cela correctement.

Version

Sujet : VERSION

Type de transaction : XTYP_REQUEST

Utilité : retourne la version de mIRC

Format : \$dde(mirc,version)

Exemple :

```
alias dver { echo -s Version de mIRC $dde(mirc,version) }
```

IV-B - Les commandes

Les identifiants (et commandes) DDE vont vous permettre de communiquer avec d'autres applications DDE.

La commande /dde que vous avez vu dans les exemples ci-dessus permet d'envoyer différents « XTYP ».

/dde -e : un XTYP_EXECUTE est alors envoyé (les trois premiers arguments sont demandés)

/dde -r : XTYP_REQUEST est alors envoyé (les trois premiers arguments sont demandés, et le "" agit comme « système remplisseur »).

/dde : si aucun paramètre n'est spécifié, un XTYP_POKE est envoyé (les quatre arguments sont demandés).

L'identifiant \$dde peut également comprendre un délai avant de répondre, il faut donc préciser un chiffre. Ce chiffre sera compté en secondes par mIRC.

L'identifiant \$ddename permet d'obtenir le DDE en cours d'utilisation. Vous pouvez préciser \$dde(nom-du-service).

L'identifiant `$isdde` permet de vérifier si un DDE est bien en cours d'utilisation. (retourne `$true` si positif, sinon retourne `$false`).

V - Le fichier mirc.ini

Le fichier mirc.ini est considéré par certains scripteurs comme la "carte mémoire" de mIRC. C'est en effet ici que les réglages de mIRC sont enregistrés. Ce fichier, comme tous les fichiers [.ini], est structuré d'une manière bien spécifique et ce pour sauvegarder et réutiliser facilement différents réglages permettant de configurer le logiciel.

Je vous propose donc découvrir les fonctions les plus utilisées par les scripteurs confirmés qui permettent de configurer un script mIRC et comment faire pour tirer la meilleure partie de ce fichier trop souvent oublié des néophytes.

Note : le fichier [.ini] sur lequel je vous propose de travailler ici est issu de la version 6.21 de mIRC. Si vous utilisez une version différente de celle-ci, il se peut que le fichier mirc.ini comporte ou ne comporte pas certaines options que je vais détailler ci-dessous.

V-A - L'architecture du fichier

Une fois le fichier ouvert, vous constatez qu'il est structuré en deux parties :

- Les sections, représentées par des crochets
- Les items structurés selon une dichotomie paramètre-valeur/commande

Les sections permettent de gérer un ensemble de paramètres ayant une même fonction. Par exemple, la section *clicks* ([clicks]) est réservée à toutes les actions effectuées lors d'un double-clic ; que ce soit dans la fenêtre statut, un dialogue privé, sur un salon, etc.

Les items, quant à eux, permettent de spécifier quelle action va être effectuée lors de tel événement. Toujours dans la même section *clicks*, l'item *query* (*query=/whois \$\$1*) indique que, lors d'un double-clic dans une fenêtre de dialogue privé, la personne sera whoisée. Où *query* indique le paramètre et */whois \$\$1* la commande à effectuer. Le signe = permet à mIRC de faire la différence entre le paramètre et la valeur.

V-B - Charger vos scripts à l'ouverture

Les sections *rfiles* et *afiles* sont sûrement deux des plus importantes sections du fichier mirc.ini. En effet, elles permettent de préciser quels fichiers scripts (remotes et/ou aliases) doivent être chargés à l'ouverture de mIRC.


Théoriquement, si vous avez plusieurs fichiers scripts chargés, la section [rfiles] devrait ressembler à quelque chose de ce genre :

```
[rfiles]
n0=remote.ini
```

```
n1=remote.ini  
n2=system\scripts\init.mrc  
n3=system\scripts\affichage.mrc  
n4=system\scripts\configuration.mrc  
n5=system\scripts\modules.mrc
```

Il est ici indiqué que les fichiers scripts "init", "affichage", "configuration" et "modules" devront être chargés depuis le répertoire "system\scripts". Cette technique, massivement utilisée par tous les scripteurs respectueux du souhait de l'auteur de mIRC, Khaled Mardam-Bey, permet de distribuer son script mIRC personnel sans y inclure mirc.exe.

Il suffit alors simplement de copier-coller mirc.exe dans le dossier racine du script et de le lancer. Si le fichier mirc.ini contient bien les précédentes indications, le script s'exécutera correctement.

 *Note : je ne souhaite pas ouvrir un débat sans fin mais sachez que vous pouvez distribuer votre script mIRC en toute légalité si vous utilisez cette méthode. Distribuer votre script mIRC sous une installation [.exe] avec mirc.exe à l'intérieur n'est pas, en théorie, pas légal.*

V-C - Commandes et identifiants associés au fichier [.ini]

Cette section va être consacrée à la réalisation de commandes très simples qui vont pouvoir illustrer les possibilités offertes par le fichier mirc.ini.

Les commandes et identifiants `/writeini`, `/remini` et `$readini` constituent les principaux outils utilisés pour utiliser un fichier [.ini].

Format : `/writeini -n fichier INI section item valeur`

Cette commande permet d'écrire dans un fichier [.ini]. Si le paramètre -n est spécifié, mIRC va tenter d'écrire le fichier .ini même si sa taille est supérieure à 64ko.

La commande `/remini` permet d'effacer des sections ou des items dans un fichier [.ini]. Si vous ne précisez aucun item, la section entière sera supprimée.

Format : `/remini fichierini section [item]`

L'identifiant `$readini` permet de lire une ligne d'un fichier [.ini] ; cet identifiant fonctionne en conjonction de la commande `/writeini`.

Format : `$readini(nomdefichier, [np], section, item)`

```
//echo $readini(mirc.ini,mIRC,nick)
```

Si le paramètre *n* est spécifié alors la ligne est lue sans être évaluée et sera traitée comme du texte.

Si le paramètre *p* est spécifié, les séparateurs de commande | (*\$chr(124)* ou *ALTGR+6*) sont traités en tant que tel au lieu d'un texte normal.


Je vous propose, histoire de mieux comprendre l'intérêt pratique de ces commandes, un petit script qui permet de définir un fond d'écran pour les salons par défaut en tapant */fond fichier* (où votre image doit être placée dans le dossier racine du script).

```
alias fond {
  writeini mirc.ini background wchannel $+($$1,$chr(44),1)
  var %ic = 1 | while ($chan(%ic)) { .background -f $chan(%ic) $$1 | inc %ic }
}
```

Ici, nous utilisons la commande */writeini* pour écrire dans le fichier *mirc.ini* dans la section *background* et dans l'item *wchannel* et *\$+(\$\$1,\$chr(44),1)* retourne le nom de votre image, par exemple *fond-salon.png*. La boucle qui suit permet, si vous êtes connectés à l'IRC, de rafraîchir tous vos salons avec le nouveau fond d'écran que vous venez d'appliquer. Si vous fermez votre client, vous pourrez constater qu'à votre prochaine connexion, les salons auront le fond d'écran précisé dans le fichier *mirc.ini*.

VI - Les variables binaires

La notion de variable binaire, en scripting mIRC, est une chose assez complexe. Elles servent principalement à lire et à écrire des fichiers binaires à l'aide de commandes et d'identifiants spécifiques.

 *Note : ce chapitre, étant donné la complexité du sujet dont il traite, ne peut être que théorique. J'ai décidé de le publier plus pour les scripteurs qui cherchent plus à connaître cet outil que pour les "scripteurs-développeurs" qui cherchent avant tout à produire des scripts.*

VI-A - Les problèmes théoriques

Pour commencer en douceur, penchons-nous sur quelques problèmes dont notamment le cas de l'ascii 0.


En effet, si vous tapez :

```
//echo -a $len($chr(0))
```

Le chiffre zéro va être renvoyé. Pourquoi ? Car il s'agit ici de l'ascii 0 (si vous connaissez le langage C / C++, c'est la même chose). Le principe de cet "ascii zéro" est de renvoyer le premier caractère en partant de la fin de la chaîne. Autrement dit, ici présent la chaîne étant composée uniquement de \$chr(0), il va retourner zéro.

Question : et alors ? Les variables binaires dans tout ça ? L'intérêt des variables binaires commence ici : elles permettent d'éviter le problème du caractère final en mémorisant la longueur de la chaîne.

De plus, il subsiste le problème de "l'ascii treize", autrement dit, celui du retour à la ligne.

 *Note : l'identifiant \$cr et le \$chr(13) retournent le caractère appelé carriage return qui permet un retour à la ligne.*

Le problème est le suivant, lors d'un retour à la ligne, cet identifiant coupe purement et simplement les variables globales. Et donc, que ces dernières seront perdues à la fin du traitement d'un script donné.

Ces problèmes, que vous risquez de rencontrer si vous choisissez d'avancer très loin dans le scripting mIRC peuvent être résolus par l'utilisation des variables binaires.

Les variables binaires se trouvent, à l'exception des variables créées, dans les fichiers binaires (logique) et dans les sockets. Concernant le problème de notre "ascii zéro", la grande majorité, pour ne pas dire tous, des protocoles excluent tout bonnement cet ascii. Cependant, comprendre son fonctionnement peut toujours se révéler utile, notamment dès lors que vous allez vous attaquer à des protocoles qui l'acceptent, c'est ici que l'utilisation des variables binaires va entrer en jeu.

VI-B - Les commandes

Après avoir traité de problèmes théoriques, nous allons tenter de passer à une utilisation pratique des variables binaires.

VII - L'optimisation (1/2)

Posséder un script qui fonctionne est une chose, posséder un script optimisé en est une autre. Explication : mIRC n'est pas un langage compilé (c'est-à-dire que tout le monde peut avoir accès à votre code). Et donc qu'il est également « relativement » lent comparé aux langages compilés (C, Java, ...).


C'est pourquoi l'optimisation est une chose primordiale en scripting mIRC. D'une part, pour améliorer la vitesse de traitement de vos codes (si l'information est traitée plus rapidement, la vitesse d'action des commandes s'en verra accrue).

Et d'autre part, l'optimisation permet de gagner de la clarté dans les codes (chose assez pratique quand vos fichiers commencent à dépasser les 1 000 lignes).

VII-A - Les optimisations basiques

À ce stade du tutoriel, vous devez tous savoir à quoi correspond une alias, une boucle, une condition, etc. Cependant, je constate toujours sur les canaux d'aide en scripting mIRC, que beaucoup de scripteurs, même confirmés, oublient souvent d'optimiser les commandes et les opérations récurrentes.

Ici, nous allons voir comment faire en sorte que des commandes répétées un grand nombre de fois s'effectuent correctement et gagnent en clarté pour vous, comment faire augmenter la vitesse d'exécution de vos boucles, puis nous verrons comment tirer le meilleur profit des conditions et enfin, en bonus, une petite technique pour calculer la vitesse de traitement de vos scripts.

 *Note : je tiens à préciser une chose avant de continuer, les codes que vous allez trouver dans cette partie n'ont aucune "utilité pratique", c'est-à-dire qu'ils servent simplement à illustrer mes propos.*

VII-A-1 - Les aliases

Vous vous demandez peut-être comment les optimiser ? En réalité, ce n'est pas l'alias en particulier que nous allons optimiser mais votre "code général". C'est-à-dire tous les scripts où va se trouver ladite commande.

Par exemple, lorsque vous avez scripté l'affichage de votre script, vous avez normalement dû utiliser un grand nombre de fois le code couleur, les parenthèses, les crochets, etc. pour la mise en forme.

Si, plutôt que d'utiliser tous ces objets, vous utilisez :

```
alias b_ return $+($chr(2), $1, $chr(2))
alias k_ return $+($+($chr(3), $2), $1, $chr(3))
alias u_ return $+($chr(31), $1, $chr(31))
```

Pour les couleurs et la mise en forme, et :

```
alias c_ return $+($chr(91), $1-, $chr(93))  
alias p_ return $+($chr(40), $1-, $chr(41))
```

Pour les crochets et les parenthèses, votre code ne s'en retrouverait-il pas plus clair ? De plus, si vous décidez de faire appel à une DLL, dans une aliase donnant accès à une DLL par exemple, plutôt que de faire :


```
alias mdx dll { C:Programes filesMonScriptDllsmdx.dll $1- }
```

Utilisez l'identifiant \$mircdirdir (qui donne le chemin du programme mIRC) et des « "" » (pour que le chemin se fasse correctement, même si il y a des espaces).

```
alias mdx dll $+( "$mircdirdir, dllsmdx.dll, " ) $1-
```

Pour éviter de créer à chaque fois des « alias menu { dialog -m menu menu } ». Pensez à faire une aliase de dialogue, par exemple :

```
alias d { if (!$dialog($1)) dialog -m $1 $1 }
```

 *Note : je vous conseille, quand une commande se répète plusieurs fois dans un script, d'en faire une aliase. Cela permet de gagner de la lisibilité et un peu de vitesse.*

L'utilisation des alias locales peut se révéler d'une grande aide. Ces alias, à la différence des alias classiques (ou globales) ne s'exécutent que dans le fichier script dans lequel elles sont créées. C'est-à-dire que si vous essayez de l'appeler vous-même : vous ne pourrez pas.

Par exemple, si vous faites :


```
alias -l bonjour { echo -s Bonjour à tous }
```

dans le dossier apprentissage.txt et, dans le fichier test.txt :

```
alias -l bonjour { window -a @hi }
```

Dans le fichier apprentissage.txt, /bonjour fera l'écho, et dans test.txt /bonjour va créer une window.

Et, si vous tapez /bonjour dans l'editbox, vous verrez un message « Commande inconnue » ou « Unknow command ».

 **Attention** : ne créez pas d'alias générale et locale du même nom. Sinon votre code va se retrouver faussé.

VII-A-2 - Les boucles

Une boucle est un système assez lourd, qui peut vite devenir "problématique" si vous ne savez pas comment bien les utiliser.

```
alias boucle {
  var %i = 1
  var %x = 10
  while (%i <= %x) {
    echo -s valeur du chiffre : %i
    inc %i
  }
}
```

Vous pouvez créer des variables locales sur une seule ligne, en faisant :

```
var %i = 1, %x = 10
```

De même, plutôt que d'annuler une variable par ligne, vous pouvez faire :

```
/unset %autojoin %awaymessage %b
```


Note : attention à l'espace avant et / ou après la virgule. Il se peut que par moment mIRC ne comprenne pas bien. Si votre code ne marche pas, pensez à mettre des espaces.

Prenez un autre exemple : si vous souhaitez calculer le nombre de lignes d'un fichier. Disons, le fichier mirc.ini. Plutôt que d'utiliser des conditions et des « goto » lents et inutiles, utilisez une boucle.

```
alias calculer {
  var %x = 1, %a = $ticks, %max = $lines(mirc.ini)
  while (%x <= %max) { inc %x }
  echo -a %x lignes calculées en $calc($ticks - %a) millisecondes
}
```

J'explique. Première ligne, on crée les variables locales. %x sera la variable incrémentée. %a les \$ticks (pour calculer la vitesse) et %max le nombre de lignes du fichier de mirc.ini.

Deuxième ligne, vous créez la boucle qui dit « tant que %x est inférieur au nombre de lignes de mirc.ini, on continue d'augmenter la valeur de %x. » La troisième ligne donne le nombre de lignes (%x) et la vitesse d'exécution du code.

 *Note : les identifiants avec des \$ font ralentir les boucles. Comme les %variables sont traitées plus vite, il est souvent judicieux de créer une variable locale (/var) pour que le code soit plus optimisé.*


VII-A-3 - Les conditions

Vous avez utilisé beaucoup de conditions et d'opérateurs en tous genres. Je vais vous donner quelques astuces pour améliorer leur vitesse de traitement.

Dans le cas des « if / elseif / else », si vous n'avez que deux conditions (c'est-à-dire if et else), préférez l'utilisation de \$iif(condition,commande-si-condition-respectée,commande-si-condition-fausse)

Exemple :

```
alias fene { echo -s $1 est entre 0 et 5 : $iif($1 isnum 0-5,Vrai,Faux) }
```


 *Note : évitez l'utiliser le \$iif dans les grosses boucles car il y a une perte de vitesse dans ce cas-là.*

Vitesse d'exécution :

if (condition) commande

if condition { commande }

if (condition) { commande }

 *Note : le « if (condition) commande » n'est valable que si votre code tient sur une ligne. Sinon, restez avec les { }.*

VII-A-4 - L'identifiants \$ticks

Cet identifiant n'a aucune utilité en lui-même. Par contre, il peut se révéler très utile pour connaître la vitesse de traitement de vos scripts.

Ce n'est pas compliqué, si vous tapez //echo -s \$ticks vous allez apercevoir un chiffre élevé.

C'est le nombre de centaines de millisecondes écoulées depuis que votre système d'exploitation a démarré.

Pour calculer la vitesse d'exécution de votre script, il suffit donc de :

1. Au début du script : mettre en variable locale le nombre de \$ticks
2. Exécuter votre script
3. A la fin du script : comparer le nombre de ticks actuel à celui du début que vous avez stocké, d'en faire la soustraction, et vous aurez le temps de traitement de votre script.

Si vous n'avez pas compris, supposons que le nombre de ticks au début du script soit de 200. Vous allez donc mettre le nombre 200 en variable, exécuter votre script (pendant ce temps le nombre de \$ticks a augmenté) puis comparer le nombre du début (200) au nombre de fin (par exemple 210). il suffit ensuite, à l'aide de l'identifiant \$calc, de faire 210 - 200, ce qui va donner 10 : vous en déduisez donc que votre script a mis dix millisecondes à être exécuté.

L'intérêt de cette astuce est de comparer le temps d'exécution de différents scripts ayant la même utilité et de connaître lequel est le plus performant, et donc celui le plus approprié à vos besoins.

VII-B - Le file handling

Le file handling est une autre manière de gérer les fichiers au format [.txt]. Cette option de mIRC permet, en effet, d'avoir plus de contrôle sur l'écriture de données dans un fichier.txt ou dans sa lecture.

VII-B-A - Le principe

Le file handling est « monumentalement » plus rapide que la commande /write.

Prenez un exemple :

```
alias ecrire {
  write test.txt T'as déjà téléchargé des programmes ?
  write test.txt Non : mIRC est arrivé pas la poste.
  write test.txt C'est ce que je me disais !
}
```


Avec la commande /write, mIRC va effectuer les actions suivantes :

- Ouverture du fichier test.txt
- Écriture de « T'as déjà téléchargé des programmes ? »
- Fermeture de test.txt
- Ouverture du fichier test.txt - Écriture de « Non : mIRC est arrivé par la poste. »
- Fermeture de test.txt

- Ouverture du fichier test.txt
- Écriture de « C'est ce que je me disais ! »
- Fermeture de test.txt

Voici ce que cela donnerait avec l'utilisation du file handling :

```
alias ecrire {  
    .fopen -n test test.txt  
    .fwrite -n test T'as déjà téléchargé des programmes ?  
    .fwrite -n test Non : mIRC est arrivé par la poste.  
    .fwrite -n test C'est ce que je me disais !  
    .fclose test  
}
```

 *Note : le « -n » permet la création du fichier test.txt s'il n'existe pas. « -o » servirait à l'écraser s'il existait déjà.*

mIRC va donc effectuer, avec le file handling, les actions suivantes :

- Ouverture du fichier test.txt
- Écriture de « T'as déjà téléchargé des programmes ? »
- Écriture de « Non : mIRC est arrivé par la poste. »
- Écriture de « C'est ce que je me disais ! »
- Fermeture de test.txt

Vous comprenez pourquoi ce système est bien plus pratique.

Bien sûr, si vous n'avez qu'une seule ligne à écrire, préférez encore le /write.

VII-B-B - Les commandes

La commande /flist, permet de faire la liste des fichiers spécifiés. Cette commande peut être utilisée sans paramètre pour afficher toute la liste des fichiers ouverts.

Format : /flist nom

La commande /fseek permet de faire varier le pointeur de lecture dans le fichier.txt.

Format : /fseek nom position

Vous pouvez également précisez les paramètres suivants :

-l nom nuLa commande /flist, permet de faire la liste des fichiers spécifiés. Cette commande peut être utilisée sans paramètre pour afficher toute la liste des fichiers ouverts.

Format : /flist nom

La commande /fseek, permet de faire varier le pointeur de lecture dans le fichier.txt.

Format : /fseek nom position

Vous pouvez également précisez les paramètres suivants :

-l : nom numéro de ligne

-n : nom prochaine ligne

-r : nom regex

-w : nom wildcard.

VII-B-C - Les identifiants

L'identifiant \$fopen, permet de donner des informations sur le fichier ouvert.

Format : \$fopen(nom).extension Les différentes extensions :

.eof : indique la fin du fichier


.err : indique qu'il y a eu une erreur

.fname : retourne le nom du fichier ouvert

.pos : retourne la position du pointeur dans le fichier.

Exemple (remplacer l'ancienne aliase) :

```
alias ecrire {
  .fopen -on test test.txt
  echo -s Le fichier ouvert est : $fopen(test).fname
  echo -s Position : $fopen(test).pos
  .fwrite -n test For a fresher World !
  echo -s Position : $fopen(test).pos
  .fwrite -n test Just do it
  echo -s Position : $fopen(test).pos
  .fwrite -n test Nike et Heineken
  echo -s Position : $fopen(test).pos
  .fclose test
}
```

 **Note** : comme à peu près partout dans mIRC, vous pouvez aussi dans le file handling, utiliser le joker *. Par exemple, pour fermer plusieurs fichiers simultanément, ou alors faire la liste de plusieurs fichiers avec /flist et gagner encore de la vitesse !

VII-C - Les événements

VII-C-A - La gestion des BDD

Supposons que vous ayez réalisé un code comme celui-ci :

```
dialog test {
  title "Optimisation des dialogues"
  size -1 -1 117 35
  option dbu
  check "Un check coché" 1, 1 2 50 7
  button "Un bouton" 2, 1 12 50 10
  button "Cliquez !" 3, 1 22 50 10
  button "Clique encore !" 4, 65 2 50 10
  button "Clique droit !" 5, 65 12 50 10
}

on *:DIALOG:test:init:*: {
  did -c $dname 1
  did -b $dname 2
}

on *:DIALOG:test:sclick:3: { echo -a Vous avez bien cliqué sur le bouton n° 3 ! }
on *:DIALOG:test:sclick:4: { echo -a Vous avez bien cliqué sur le bouton n° 4 ! }
on *:DIALOG:test:rclick:5: { echo -a Clic droit sur le bouton n° 5 ! }
```


Vous allez pouvoir optimiser ce code avec le if (\$devent == init), if (\$devent == sclick), etc. Et tous les autres événements du dialogue !

Comme ceci :

```
on *:DIALOG:test:*:*: {
  if ($devent == init) {
    did -c $dname 1
    did -b $dname 2
  }
  if ($devent == sclick) {
```

```
if ($did == 3) echo -a Vous avez bien cliqué sur le bouton n° 3 !
if ($did == 4) echo -a Vous avez bien cliqué sur le bouton n° 4 !
}
if ($devent == rclick) && ($did== 5) echo -a Clic droit sur le bouton n° 5 !
if ($devent == close) echo -s Vous venez de fermer le dialogue !
}
```

VII-C-B - Les préfixes

 **Note** : cette sous-partie ne s'adresse qu'aux personnes utilisant beaucoup d'événements. Si vous regroupez toutes les commandes dans un seul événement, vous pouvez passer votre chemin.

Beaucoup de gens utilisent souvent des conditions inutiles, prenons l'événement PART.

Si vous faites :

L'événement part

```
on *:PART:#{
    if ($nick == $me) { commande }
}
```

Préférez le préfixe « me »

```
on me*:PART:#{ commande }
```


De même, si vous voulez que mIRC prenne en compte « tous les pseudonymes sauf moi », utilisez le préfixe « ! » .

```
on !*:PART:#{ commande }
```

Si vous souhaitez que, seuls les @ soient pris en compte, utilisez le préfixe « @ » .

```
on @*:PART:#{ mode # +l $calc($nick(#,0) + $r(3,6)) }
```

Permet de changer le mode +l sur un salon, après le départ d'une personne.

 **Note** : vous pouvez également regrouper ces préfixes. Faites également attention à l'ordre de vos événements (pour que mIRC puisse traiter l'information de façon complète). Placez donc l'événement le plus « général » à la fin de votre fichier script.

VII-C-C - Les raws

Les raws sont des informations renvoyées par un serveur IRC. On peut assimiler le fonctionnement des raws à celui des événements. Supposons, par exemple, que votre système de whois ressemble à ceci :

```
raw 301:** { set %whois.away $3- | halt }

raw 311:** {
  unset %whois.*
  set %whois.fullname $6-
  set %whois.adresse $+(**,$3,$chr(64),$4)
  halt
}

raw 312:** { set %whois.serveur $3 | halt }

raw 313:** { set %whois.statut $5- | halt }

raw 317:** {
  set %whois.idle $duration($3)
  set %whois.connexion $gettok($asctime($4),4,32)
  halt
}

raw 318:** {
  echo -a $+($chr(2),$chr(3),4) Whois $2
  echo -a $+($chr(3),2) Fullname : %whois.fullname
  echo -a $+($chr(3),2) Adresse : %whois.adresse
  if (%whois.salon) echo -a $+($chr(3),2) $2 est présent sur %whois.salon
  echo -a $+($chr(3),2) Serveur utilisé : %whois.serveur
  echo -a $+($chr(3),2) Connexion à : %whois.connexion
  if (%whois.statut) echo -a $+($chr(3),2) Statut sur ce serveur : %whois.statut
  if (%whois.away) echo -a $+($chr(3),2) $2 est away : %whois.away
  unset %whois.*
  halt
}

raw 319:** { set %whois.salon $3- | halt }
```

Si vous prenez le début de cette ligne :

```
raw 318:** {
```

Le nombre 318 permet de dire à mIRC que le raw à traiter est le 318. Ce chiffre est retourné par l'identifiant *\$numeric*.

Vous pouvez donc créer le code suivant :

```
raw *:** { if ($numeric)
```

Qui va fonctionner de la même manière que le précédent. Cette petite astuce va vous permettre de regrouper vos raws dans un seul événement. Pour mieux vous faire comprendre cette méthode, voici le système de whois une fois optimisé :

```
raw *:*: {
  if ($numeric == 310) { set %whois.away $3- | halt }
  elseif ($numeric == 311) {
    unset %whois.*
    set %whois.fullname $6-
    set %whois.adresse $+(!*,$3,$chr(64),$4)
    halt
  }
  elseif ($numeric == 312) { set %whois.serveur $3 | halt }
  elseif ($numeric == 313) { set %whois.statut $5- | halt }
  elseif ($numeric == 317) {
    set %whois.idle $duration($3)
    set %whois.connexion $gettok($asctime($4),4,32)
    halt
  }
  elseif ($numeric == 318) {
    echo -a $+($chr(2),$chr(3),4) Whois $2
    echo -a $+($chr(3),2) Fullname : %whois.fullname
    echo -a $+($chr(3),2) Adresse : %whois.adresse
    if (%whois.salon) echo -a $+($chr(3),2) $2 est présent sur %whois.salon
    echo -a $+($chr(3),2) Serveur utilisé : %whois.serveur
    echo -a $+($chr(3),2) Connexion à : %whois.connexion
    if (%whois.statut) echo -a $+($chr(3),2) Statut sur ce serveur : %whois.statut
    if (%whois.away) echo -a $+($chr(3),2) $2 est away : %whois.away
    unset %whois.*
    halt
  }
  elseif ($numeric == 319) { set %whois.salon $3- | halt }
}
```

VIII - L'optimisation (2/2)

VIII-A - Le submenu

VIII-A-1 - Une évolution des popups de base

Un submenu est une version des popups très optimisée. Vous avez en effet sûrement déjà utilisé des popups.

Les popups ont une longueur déterminée, cela veut dire que lorsque vous créez un popup, vous connaissez son début et sa fin. Alors qu'avec le submenu, vous pouvez créer des popups d'une longueur indéterminée ! Comme par exemple : lister les adresses bannies d'un salon, lister les salons sur lesquels vous êtes, etc.

VIII-A-2 - Construction d'un submenu

Réaliser un submenu n'est pas très compliqué dès lors que l'on connaît son format et que l'on sait utiliser les boucles et les tokens. Le format du submenu est :

```
menu type {
  Item
  .$submenu($alias($1))
}
```

Illustration :

```
alias sub.partchan {
  if ($1 == begin || $1 == end) { return - }
  if ($chan($1)) { return $chan($1) : part $chan($1) }
}

menu channel {
  Part
  .$submenu($sub.partchan($1))
}
```

Comme vous pouvez le constater, ce submenu permet de lister tous vos salons et offre la possibilité de partir du salon souhaité.

L'alias `sub.partchan` permet de construire le submenu.

```
if ($1 == begin || $1 == end) { return - }
```

Begin précise le début et end la fin du menu. Un tiret est retourné pour que le format du menu soit respecté.

i *Note : les accolades sont optionnelles quand la commande à effectuer ne comporte qu'une seule ligne. Mais, depuis mIRC v6.17, les submenus sont devenus plus stricts et renvoient un message d'erreur lorsque vous omettez de les mettre. Je vous conseille donc de le faire.*

Le prochain submenu, plus complexe, va permettre de lister les adresses bannies d'un salon et la possibilité de retirer l'adresse souhaitée.

```
alias sub.banlist {
    ;début du submenu
    if ($1 == begin) || ($1 == end) { return - }
    ;s'il n'y a aucune adresse bannie
    if ($1 == 1) && (!$ibl($active,0)) { return Vide : halt }
    ;si des adresses sont inscrites dans la liste
    else {
        ;on crée des variables locales qui vont être utilisées plus tard
        var %i = 1, %ban, %a = $active
        ;une boucle qui va permettre de lister toutes les adresses bannies
        while ($ialchan($ibl(%a,$1),%a,%i).nick) {
            ;on ajoute à la variable %ban toutes les adresses bannies à l'aide du $addtok
            %ban = $addtok(%ban,$ifmatch,32)
            ;on incrémente la variable %i et on recommence l'opération jusqu'à la sortie de la boucle
            inc %i
        }
        ;après être sorti de la boucle, toutes les adresses sont retournées avec la possibilité de
        retirer le ban
        return $+(($chr(46),$ibl(%a,$1)) %ban : mode %a -b $ibl(%a,$1)
    }
}

menu channel {
    Ban-liste
    .$submenu($sub.banlist($1))
}
```

Le principe des submenus est souvent le même. Il suffit de créer une boucle qui permet de lister tous les éléments de ce qui est recherché puis d'ajouter un à un chaque élément dans une variable à l'aide du \$addtok et puis d'afficher le résultat.

Utiliser des submenus permet de rendre votre script plus performant. Mais, il est malheureusement impossible d'imbriquer les submenus les uns dans les autres. C'est-à-dire que réaliser un script de la manière suivante :

```
menu channel {
    Menu
    .$submenu($sub.menu($1))
    ..$submenu($sub.sous-menu1($1))
    ..$submenu($sub.sous-menu2($1))
}
```

N'est, pour le moment, pas possible.

VIII-B - Commandes et techniques de scripting

/did : vous savez normalement déjà à quoi elle correspond, mais saviez-vous que (depuis mIRC 6.17), au lieu de faire :

```
did -b $dname 1,2,3,4,5
```

vous pouviez faire :

```
did -b $dname 1-5
```

Utilisation similaire au « isnum ».

/didtok : permet de réécrire une liste d'items dans un combo ou une liste (d'un dialogue).

Format : /didtok nom \$dname n°-du-caractère-ascii texte

Exemple :

Si vous avez, dans l'init d'un dialogue,

```
did -a $dname 1 Blanc  
did -a $dname 1 Noir  
did -a $dname 1 Bleu
```

faites plutôt :

```
/didtok $dname 1 59 Blanc;Noir;Bleu
```

/loadbuf : commande permettant de charger le nombre de lignes spécifiées d'un fichier dans la fenêtre spécifiée. Cette commande est énormément plus rapide que de réaliser un système de boucle qui va réécrire chaque ligne dans la fenêtre spécifiée.

Illustration (avec une boucle) :

```
alias bouc {  
  var %t = $ticks , %i = 1, %x = $lines(mirc.ini)  
  window -a @k  
  while (%i <= %x) {  
    echo @k $read(mirc.ini,n,%i)  
    inc %i  
  }  
  echo @k Exécution en $calc($ticks - %t) millisecondes  
}
```

```
}
```

Avec la commande /loadbuf :

```
alias buf {
  var %t = $ticks
  window -a @k
  loadbuf @k mirc.ini
  echo @k Exécution en $calc($ticks - %t) millisecondes
}
```

On peut constater avec le système de boucle : 1984 millisecondes.

Et avec le /loadbuf : 46 millisecondes.

Inutile de commenter... Ou alors, simplement de dire que le gain de temps est plus de 40 fois supérieur avec la seconde méthode !

\$istok : permet de chercher un token dans un texte.

Format : **\$istok(texte,token-à-chercher,n°-du-caractère-ascii)**

Par exemple :


```
alias testor {
  var %a = $ticks, %i = 1
  while (%i <= 1000) {
    if (%i == 100 || %i == 500 || %i == 999) { inc %i }
    else inc %i
  }
  echo -s Vitesse traitement classique : $calc($ticks - %a) millisecondes
}

alias testis {
  var %a = $ticks, %i = 1, %num = 100;500;999
  while (%i <= 1000) {
    if ($istok(%num,%i,59)) inc %i
    else inc %i
  }
  echo -s Vitesse traitement istok : $calc($ticks - %a) millisecondes
}
```

Tapez /testor et /testis.

L'écart n'est pas forcément des plus significatifs, (j'obtiens 141 millisecondes dans le premier cas, et 93 dans le second), mais plus le traitement devient lourd, plus l'écart se creuse. Mais là n'est pas le plus important ; dans


l'exemple précédent, mIRC a évalué une seule condition (dans le cas du \$istok), contrairement à trois dans le cas du « if ». Donc, le traitement sera logiquement plus rapide (et préférable) avec le \$istok.

 *Note : un autre identificateur, \$istokcs, fonctionne sur le même format que \$istok, mais lui est sensible à la casse des caractères (majuscules / minuscules).*

\$+() : cette commande est dérivée de la commande \$+ (permettant de coller plusieurs choses les unes aux autres).

Exemple :

```
m $+ I $+ R $+ C devient $+(m,I,R,C)
```

 *Note : l'utilisation des expressions régulières est aussi fortement recommandée dès lors qu'il s'agit d'effectuer des validations ou des tris quelconques ou bien de manipuler des chaînes. Lisez donc la partie sur les regex !*

