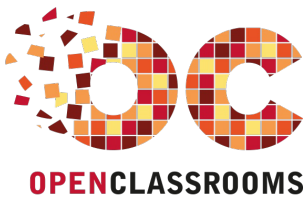
Le préprocesseur

Par Pouet_forever



www.openclassrooms.com

Dernière mise à jour le 17/01/2012

Sommaire

Le préprocesseur

Par  Pouet_forever

Mise à jour : 17/01/2012
Difficulté : Intermédiaire

Bonjour à tous,

M@teo21 dans son tutoriel sur le langage C nous fait un petit aperçu de ce qu'est le préprocesseur mais il y a encore beaucoup de choses à apprendre !
Toutefois, ce que je vais vous montrer n'est pas indispensable, mais c'est toujours mieux de savoir ce que c'est quand on en rencontre.
Vous allez voir que pour créer des macros il faut réfléchir un peu avant de faire n'importe quoi, parce que pour débugger ce n'est pas aussi facile qu'avec des fonctions.

Pour bien comprendre ce tutoriel il faut avoir compris et être un minimum à l'aise avec le préprocesseur.

Je ne reparlerai pas des `#include #ifdef #else #elif #endif`. Si vous avez un doute là-dessus retourner voir le tuto de M@teo21.

** Bonus GCC **

Si vous utilisez GCC vous pouvez voir le code après le passage du préprocesseur et avant la compilation. Il faut utiliser `-E`.
C'est option est très utile pour débugger les parties qui contiennent le préprocesseur.

Sommaire du tutoriel :

#define, defined, #undef

#define

Je ne vais pas trop m'étaler sur ce paramètre car il est expliqué dans le tuto de M@teo21.
Pour résumer, cette directive sert à définir un nom ou une macro.

La macro suivante définit un nom sans aucun paramètre :

Code : C

```
#define DEBUG
```

La macro suivante définit un nom et associe une valeur à ce nom :

Code : C

```
#define NOIR 0x00000000
```

Une macro peut être redéfinie autre part dans le programme à la seule condition que celle-ci soit exactement pareille, seul quelques espaces peuvent être rajoutés. Exemple :

Code : C

```
/* Incorrect */
#define N 1
#define N 2
```

Code : C

```
/* Correct */
#define N 1
#define N 1
```

La macro suivante contient 1 paramètre et retourne la valeur absolue de ce paramètre :

Code : C

```
#define MY_ABS(x) (((x) < 0) ? -(x) : (x))
```

Notez que les parenthèses sont très importantes dans les macros dues à la priorité des opérateurs. Certaines peuvent être omises, d'autres non. Pour être sûr de ne pas avoir de mauvaise surprise je vous conseille d'abuser des parenthèses pour ne laisser aucune ambiguïté possible !

Il faut faire très attention aux effets de bords. Une macro appelée normalement ne pose pas de problèmes, mais on aurait aussi pu appeler notre macro comme ça `MY_ABS(a++)`. Cette expression évalue 2 fois (a++), ce qui est gênant. Mettre des parenthèses limite déjà certains effets de bords, mais c'est à l'utilisateur de faire attention à ce qu'il fait !

Vous pouvez créer des macros qui portent le même nom qu'une fonction déjà existante. Cependant, il faut bien différencier l'appel de la fonction de celle de la macro, sinon c'est uniquement la macro qui est prise en compte et pas la fonction. Pour utiliser la macro il faut l'appeler comme on le ferait normalement, par contre pour la fonction il faut entourer le nom de parenthèses. Certains puristes diront qu'il faut aussi déclarer votre fonction avec des parenthèses pour ne pas interférer avec les macros.

Code : C

```
#include <stdio.h>
#include <stdlib.h>

#define puts(s) printf("Macro : %s\n", (s))

int main(void) {
  char s[] = "Salut :-)";

  puts(s);   /* Macro */
  (puts)(s); /* Fonction */
  return EXIT_SUCCESS;
}
```

Les macros sur plusieurs lignes

Les macros peuvent être définies sur plusieurs lignes, pour ça il faut terminer la ligne par un antislash (\) et avoir un retour à la ligne juste après. Un antislash non suivit d'un retour la la ligne termine la macro et tout ce qui suit n'est pas compris dedans.
Il est important de noter qu'aucune macro ne peut contenir de directives commençant par # (y compris la directive nulle). Dès que le préprocesseur rencontre un antislash suivit d'un retour à la ligne il supprime ces 2 là pour en faire une seule ligne. Voilà pourquoi l'utilisation d'autres directives à l'intérieur d'une macro est incorrect. 
On peut parfois retrouver cette utilisation pour les chaînes de caractères. Une chaîne de caractère longue peut être 'coupée' par l'antislash + retour à la ligne et être terminée sur la ligne d'après.
Voilà un exemple de ce qu'il faut pas faire :

Code : C

```c
#define MAJEUR(age) \
#if (age >= 18) \
...
```

Un petit exemple pour montrer comment agit le préprocesseur :

Code : C

```c
#define M() 3 + \
2 + 1;

int main(void) {
  char s[] = "Bonjour a tous ! Vous etes bien sur le site du zero\
mais comme cette chaine est trop longue je la met sur plusieurs\
lignes. Evitez tout de meme d'utiliser ca, c'est pas tres propre !";
  M()
  return 0;
}
```

Secret (cliquez pour afficher)

Code : Console

```
int main(void) {
  char s[] = "Bonjour a tous ! Vous etes bien sur le site du zero  mais comme ce


  3 + 2 + 1;
  return 0;
}
```

Il faut faire attention tout de même aux macros faites sur plusieurs lignes. Le piège étant de mettre une suite d'instruction et d'appeler cette macro dans un if sans accolades, par exemple. Un exemple vaut mieux qu'un long discours :

Code : C

```c
#include <stdio.h>
#include <stdlib.h>

#define M(a) \
if ((a) < 0) \
printf("Inferieur a 0\n"); \
if ((a) > 0) \
printf("Superieur a 0\n"); \
if ((a) == 0) \
```

```
printf("Egal a 0");

int main(void) {
  int a = 5;

  if (a < 0)
    M(a)
  else
    printf("Autre instruction");

  return EXIT_SUCCESS;
}
```

Ce code bidon et anodin n'affichera pas `Autre instruction` mais `Superieur a 0 Autre instruction`.
Pour palier ce problème, une astuce assez courante est d'utiliser la boucle `do { ... } while(0);` qui permet de regrouper les instructions et éventuellement, d'obliger le programmeur à mettre un point virgule après l'appel de la macro. Pour que le code du dessus soit correct, on aurait dû écrire :

Secret (cliquez pour afficher)

Code : C

```
#include <stdio.h>
#include <stdlib.h>

#define M(a) \
do { \
if ((a) < 0) \
printf("Inferieur a 0\n"); \
if ((a) > 0) \
printf("Superieur a 0\n"); \
if ((a) == 0) \
printf("Egal a 0"); \
} while(0)

int main(void) {
  int a = 5;

  if (a < 0)
    M(a);
  else
    printf("Autre instruction");

  return EXIT_SUCCESS;
}
```

On peut aussi n'utiliser que les accolades, mais l'utilisation de la boucle sert à bien montrer au programmeur qu'il s'agit d'un bloc d'instructions.

defined

L'opérateur `defined` agit au même titre que `#ifdef` : il vérifie si la macro existe ; il est remplacé par 1 si elle existe, par 0 le cas échéant. Avec `#if` ou `#ifdef` on ne peut tester qu'un seul paramètre à la fois, avec l'utilisation de `defined` on va pouvoir en tester plusieurs. 
L'utilisation se fait avec ou sans parenthèses.

Exemple pour l'implémentation sur différents systèmes d'exploitation :

Code : C

```
#if defined __APPLE__ || defined linux
# include <unistd.h>
```

```
#elif defined ( WIN32 ) || defined ( WIN64 )
# include <windows.h>
#endif
```

Vous trouverez une liste des différentes constantes définies en fonction de l'implémentation en allant sur ce lien.

* Bonus GCC *

Plutôt que de mettre des `#define` un peu partout pour faire des tests (par exemple le `#ifdef DEBUG` qu'on retrouve souvent) on peut définir ce paramètre avec l'option `-D` de GCC. Avec l'exemple du DEBUG cité précédemment, on a `$ gcc -DDEBUG main.c`. Cette commande aura la même action que `#define DEBUG` et agira sur l'ensemble du fichier.

Cette astuce fonctionne avec Code::Blocks, il suffit d'aller dans les options de compilation.

#undef

Le `#undef` fait exactement l'inverse de ce que fait `#define`. 
Il supprime ce qui a été défini auparavant !
Utilisation très simple :

Code : C

```
#undef X
```

Où X est le nom à supprimer.

Le # et le

L'opérateur

Vous avez déjà dû le rencontrer plus d'une fois celui-là dans les déclarations telles que `#define #if #else` etc.
Maintenant je vais vous montrer une autre manière de l'utiliser. 

On va commencer par la plus simple : la directive nulle !
C'est tout simplement le # sans rien derrière (éventuellement des espaces/tabulations ou commentaires).
On s'en sert généralement pour 'lier' différentes directives. Ce que j'entends par 'lier' c'est en quelque sorte faire un 'bloc' de directives.
Un exemple vaut mieux qu'un long discours :

Code : C

```
#if defined ( __APPLE__ )
# /* Si on est sur du matériel APPLE on inclut */
# /* <unistd.h> et tout le reste pour les sockets */
# include <unistd.h>
# include <sys/socket.h>
# include <sys/types.h>
# include <arpa/inet.h>
#elif defined ( linux )
# /* Si on est sur Linux on inclut ... la même chose */
# include <unistd.h>
# include <sys/socket.h>
# include <sys/types.h>
# include <arpa/inet.h>
#else
# /* Sinon on est sous Windows */
# include <windows.h>
# include <winsock2.h>
```

```
#endif
```

Il ne peut en aucun cas être placé dans une macro.

Maintenant le plus intéressant !
L'opérateur # suivi d'un nom remplace automatiquement ce nom en chaîne de caractères.
Petit exemple :

Code : C

```
#define AFFICHE_INT(x) printf( #x " = %d\n", (x) );
```

Comme vous pouvez le remarquer j'ai placé #x au début du printf, ce qui aura pour effet de transformer l'argument x en chaîne de caractère. Les espaces contenus entre le # et le paramètre sont ignorés (#a est équivalent à # a).
Un petit exemple pour illustrer tout ça :

Code : C

```
#include <stdio.h>
#include <stdlib.h>

#define AFFICHE_INT(x) printf( #x " = %d\n", (x) );

int main(void) {
  int a = 5, b = 6;
  AFFICHE_INT(a)
  AFFICHE_INT(b)
  AFFICHE_INT(a + b)
  return EXIT_SUCCESS;
}
```

Ce code sera 'transformé' en sortie en :

Code : C

```
#include <stdio.h>
#include <stdlib.h>



int main(void) {
  int a = 5, b = 6;
  printf( "a" " = %d\n", a );
  printf( "b" " = %d\n", b );
  printf( "a + b" " = %d\n", a + b );
  return EXIT_SUCCESS;
}
```

Ce qui nous donne en sortie :

Code : Console

```
a = 5
b = 6
a + b = 11
```

L'opérateur ##

L'opérateur `##` concatène l'argument de gauche avec celui de droite tout en restant 'macro'. Les espaces contenus entre les différents arguments sont ignorés (`A ## B` est égal à `A##B`).
Comme la résultante reste un argument 'macro', il faut que celle-ci soit définie auparavant pour être utilisée à bon escient. Si elle n'est pas définie au moment de l'appel, le compilateur nous indiquera une erreur.
Exemple :

Code : C

```
#define AFFICHE_INT(x, y) printf( #x #y " = %d\n", x##y );
```

Dans ce cas-là si on appelle notre macro comme ça : `AFFICHE_INT(a, b)`, elle sera remplacée par : `printf( "a" "b" " = %d\n", ab );`.
Il faut donc que l'argument ab soit défini avant l'appel de la macro. Notez que j'ai mis 'avant l'appel' et pas 'avant la déclaration', ce qui signifie que AFFICHE_INT peut être défini sans que ab ne soit connu. ab peut être soit défini par le préprocesseur (`#define`) ou alors comme une simple variable.

Code : C

```
#include <stdio.h>
#include <stdlib.h>

#define AFFICHE_INT(x, y) printf( #x #y " = %d\n", x##y );

#define ab 5

int main(void) {
  int a = 5, b = 6;
  int ba = 10;
  AFFICHE_INT(a, b)    /* Affiche 'ab' défini par le préprocesseur */
  AFFICHE_INT(b, a)    /* Affiche la variable 'ba' */
  return EXIT_SUCCESS;
}
```

Ce code nous affichera :

Code : Console

```
ab = 5
ba = 10
```

Utilisation des opérateurs # et ## dans la même expression.

Si vous avez fait des tests en essayant d'utiliser à la fois `#` et `##` vous aurez remarqué que le compilateur apprécie modérément. De ce fait, pour pouvoir utiliser ces 2 opérateurs en même temps, il va falloir faire plusieurs macros afin de la créer.

Un exemple pour montrer comment concaténer 2 chaînes de caractères :

Code : C

```
#include <stdio.h>
```

```c
#include <stdlib.h>

#define CREER_CHAINE(chaine) #chaine
#define TMP(chaine) CREER_CHAINE(chaine)
#define CONCAT_CHAINE(chaine1, chaine2) TMP(chaine1 ## chaine2)

int main(void) {
  printf("%s", CONCAT_CHAINE(Hello\n, Ca va ?));
  return EXIT_SUCCESS;
}
```

Comme je vous l'ai dit pour pouvoir utiliser les 2 opérateurs, il faut faire plusieurs macros. C'est en fait pour 'isoler' chaque paramètre afin que la macro qui contient un `#` ou `##` ne 'voit' pas le `#` ou `##` de l'autre.

Petit exercice

Écrivez une macro PRINT qui prend 2 paramètres. Le premier est le type de l'élément à afficher et le second est l'expression à afficher.
Le 'prototype' est celui-là : `#define PRINT(type, expr)`.
Exemple d'utilisation :

Code : C

```c
PRINT(int, 1+3);
PRINT(double, 4.0 * atan(1.0));
PRINT(char, 'c');
```

Dois nous retourner :

Code : Console

```
1+3 = 4
4.0 * atan(1.0) = 3.141593
'c' = c
```

Indice :

Secret (cliquez pour afficher)

Il faut définir, à partir du type, le formateur approprié. On peut définir une macro du type : `#define PRINT_(type) PRINT_##type`.
Il faut ensuite définir les différents types qui seront utilisés (`PRINT_int`, `PRINT_double`, `PRINT_char`) et leur associer le formateur adéquat. 

Correction :

Secret (cliquez pour afficher)

Á partir de l'indice on peut arriver à ce résultat-là :

Code : C

```c
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <math.h>

#define PRINT_(type) PRINT_##type
#define PRINT_int "%d"
#define PRINT_double "%f"
#define PRINT_char "%c"

#define PRINT(type, expr) printf(#expr " = " PRINT_(type) "\n",
expr)

int main(void) {
  PRINT(int, 1+3);
  PRINT(double, 4.0 * atan(1.0));
  PRINT(char, 'c');
  return EXIT_SUCCESS;
}
```

#line, #error, #pragma

#line

Cette directive est assez intéressante. Elle permet de définir le numéro de ligne en cours, ainsi que le nom de fichier.
Vous pouvez définir le numéro de ligne en cours en spécifiant tout simplement le numéro de ligne souhaité.
Par exemple si votre fichier s'appelle main.c et vous voulez le redéfinir en test.c vous pouvez.
Son utilisation est la suivante :

Code : C

```c
#line ligne ["nom du fichier"]
```

Notez que renommer le fichier est facultatif.

Cette directive ne renomme pas le fichier, elle ne fait qu'interpréter le fichier en tant que.

Code : C

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
 printf("Ligne : %5d\tFichier : %s\n", __LINE__, __FILE__);
#line 300
 printf("Ligne : %5d\tFichier : %s\n", __LINE__, __FILE__);
#line 55 "test.c"
 printf("Ligne : %5d\tFichier : %s\n", __LINE__, __FILE__);
 return EXIT_SUCCESS;
}
```

La sortie donne :

Code : Console

```
Ligne :    45        Fichier : main.c
Ligne :   300        Fichier : main.c
```

```
Ligne :    55        Fichier : test.c
```

#error

Cette directive sert tout simplement à mettre une erreur.
Attention, il faut la mettre avec des `#if` ou `#ifdef` sinon ça ne compilera tout simplement pas (bah oui ça met une erreur).
Mon interprétation préférée est celle-là :

Code : C

```
#ifdef __cplusplus
#error Compilez en C ...
#endif
```

Si vous avez une erreur quand vous compilez ce code c'est que vous compilez en C++ et pas en C.

#pragma

La directive `#pragma` est spécifique à chaque compilateur. Elle indique au compilateur de prendre en compte certains paramètres.
Toute directive `#pragma` qui n'est pas reconnue est tout simplement ignorée.

Un exemple de directive : `#pragma pack(1)` .
Cette directive permet de forcer l'alignement mémoire tous les octets. C'est-à-dire que par défaut quand on crée une structure comme ça :

Code : C

```
struct non_alignee_s {
 char a;
 int b;
};
```

Si vous faites un **`sizeof`** de cette structure il apparaîtra qu'elle occupe 8 octets en mémoire ! (considérant qu'un int fait 4 octets)
Avec la directive décrite plus haut vous aurez 5.
Les valeurs de cette directive sont : 1, 2, 4.

Comme dit plus haut chaque directive est propre à son compilateur. De ce fait la directive `#pragma pack(1)` ne fonctionnera peut-être pas chez vous.

Un exemple de non-portabilité : sous GCC on peut utiliser la directive `#pragma unused( ma_variable )` pour ne pas avoir de warning sur la variable non utilisée spécifiée.
Sous Xcode (Mac et GCC) on peut utiliser la directive `#pragma mark XXX` qui sert à structurer l'arborescence de son code pour une recherche plus facile et plus rapide (très utile !).
Sous Visual C++ on peut linker les bibliothèques avec la directive `#pragma comment( lib, "emapi" )` .

Pour plus d'infos vous pouvez aller ici pragmas pour GCC, et là pragmas pour Visual C++.

Les macros qui en appellent d'autres et les macros à nombre variable d'arguments.

Une macro qui en appelle une autre.

Une macro peut en appeler une autre, ça vous le savez. Mais une macro peut prendre en paramètre le nom d'une autre pour l'appeler.
Par exemple, j'ai une macro MIN, une autre MAX et une dernière EVALUE. Cette dernière prend 3 paramètres : le nom de la macro à appeler et deux variables.
Voilà comment ce groupe se compose :

Code : C

```
#define MIN(x, y) (((x) < (y)) ? (x) : (y))
#define MAX(x, y) (((x) > (y)) ? (x) : (y))

/* Je préfère mettre l'underscore '_' plutôt que 'macro', mais vous
faîtes comme vous voulez.
* Par la suite je mettrai _ . */
#define EVALUE(macro, x, y) macro(x, y)
```

Si je veux calculer le minimum de 2 nombres, il suffit que j'appelle ma macro EVALUE comme ça : `EVALUE(MIN, 5, 3)` .

Les macros à nombre variable d'arguments.

Tout comme une fonction, une macro peut recevoir un nombre variable d'argument. Oui, mais la macro 'finale' doit avoir une nombre fini d'arguments.

La différence entre une fonction et une macro, c'est que la fonction est évaluée à l'exécution et la macro à la compilation. De ce fait, il faut que celle-ci soit entièrement connue à la compilation.
Pour qu'une macro prenne un nombre variable d'arguments, il faut tout simplement utiliser les trois petits points : ... (comme une fonction). Aucun intérêt, parce que la macro n'est pas entièrement connue ...
Vous vous souvenez qu'une macro peut en appeler une autre ?
Hé bien nous allons nous servir de ça ! Pour passer les arguments 'variables' il va falloir utiliser la macro `__VA_ARGS__` .

Prenons notre exemple de EVALUE plus haut. Pour l'instant elle prend 3 arguments. Nous allons la modifier pour qu'elle prenne un nombre d'arguments variable. Ainsi on va faire une macro générique pour appeler plusieurs macros.
Donc maintenant, sans toucher ni à MIN, ni à MAX, on obtient ça :

Code : C

```
#define MIN(x, y) (((x) < (y)) ? (x) : (y))
#define MAX(x, y) (((x) > (y)) ? (x) : (y))

/* Souvenez-vous, j'utilise '_' mais vous pouvez utiliser 'macro' ou
ce que vous voulez.
* Du moment que ça reste un nom valide. */
#define EVALUE(_, ...) _(__VA_ARGS__)
```

Note si vous utilisez Visual: la macro __VA_ARGS__ n'a été introduite qu'à partir de la version 2005, de ce fait, si vous avez une version antérieure, ça ne fonctionnera pas.

Les X-macros.

Nous arrivons à la partie la plus intéressante (mais aussi la plus compliquée) : les X-macros !

Que sont les X-macros ?

Rassurez-vous ça n'a rien à voir avec des trucs cochons.
Ça permet, dans le cas le plus simple, de faire simplement correspondre différentes valeurs entre elles (énumérations, structures, tableaux).
Je prends l'exemple d'un marchand de voitures. Chaque voiture a une marque, un modèle, un prix, une couleur (et tout ce qui

vous passe par la tête).
Première émission de votre programme vous rentrez tout à la main (bon pas de problème c'est la première émission).
Deuxième émission de votre programme vous devez rajouter plusieurs voitures, en modifier certaines et en supprimer. Oui mais voilà pour chercher les valeurs à modifier et à supprimer ce n'est pas forcement tâche facile.
C'est là qu'interviennent les X-macros ! Grâce à ces macros particulières on verra que la tâche va être grandement simplifiée.

Prenons notre exemple de voiture.
En temps normal nous aurions fait 4 tableaux de chaînes de caractères. Je vais faire aussi une énumération pour permettre une recherche facile de notre voiture.
(les valeurs ont été prises au hasard, ne venez pas me dire que le prix n'est pas bon etc. ce n'est qu'un exemple)

Code : C

```
enum index_e {
  Renault_Clio,
  Peugeot_207,
  Citroen_C4,
  NOMBRE_VOITURES
};

char const * const marque_a [] = {
  "Renault", "Peugeot", "Citroen"
};

char const * const modele_a [] = {
  "Clio", "207", "C4"
};

int const prix_a [] = {
  10000, 12000, 15000
};

char const * const couleur_a [] = {
  "blanc", "rouge", "bleu"
};
```

Ce code est pratique. Si on cherche une voiture en particulier il suffit de prendre la valeur de l'énumération comme indice de chaque tableau :

Code : C

```
printf("%s %s %d %s", marque_a[Citroen_C4], modele_a[Citroen_C4],
prix_a[Citroen_C4], couleur_a[Citroen_C4]);
```

Mais voilà, dès qu'on doit modifier, rajouter ou supprimer des modèles ça se révèle très peu pratique.
Nous allons donc faire une X-macro qui va se charger de créer tout ça toute seule. C'est une macro qui fait appel à toute une série de macros qui portent le même nom. J'ai choisi ici de la nommer X_VOITURE. Elle appellera 3 'sous-macros' (normal on a 3 voitures) nommées X et contenant les éléments de nos voitures.

Voilà comment elle est constituée :

Code : C

```
#define X_VOITURE \
X(Renault, Clio, 10000, blanc) \
X(Peugeot, 207, 12000, rouge) \
X(Citroen, C4, 15000, bleu)
```

C'est à partir de maintenant que toute la magie des X-macros se fait.
Le principe est tout bête : afin de définir l'action de X_VOITURE, il faut tout d'abord définir X. Une fois X_VOITURE appelée il suffit d'utiliser la directive `#undef X` et de continuer. On redéfinit X, on appelle X_VOITURE et on la supprime. On recommence le processus partout où vous voulez utiliser votre X-macro.
Grâce à ça, pour rajouter, modifier ou supprimer des informations sur nos voitures nous n'avons pas de problèmes ! Tout est répertorié au même endroit, il suffit de trouver la ligne concernée et de la modifier/supprimer ou d'en rajouter une.
Si vous avez compris le principe vous ne devriez pas avoir de mal à implémenter ça !

Secret (cliquez pour afficher)

Code : C

```c
#include <stdio.h>
#include <stdlib.h>

#define X_VOITURE \
X(Renault, Clio, 10000, blanc) \
X(Peugeot, 207, 12000, rouge) \
X(Citroen, C4, 15000, bleu)

/* Définit une macro qui concatène la marque, un underscore et le
modèle. */
#define X(marque, modele, prix, couleur) marque ## _ ## modele ,
enum index_e {
  /* On appelle X_VOITURE qui appellera notre macro définie juste
avant */
  X_VOITURE
  NOMBRE_VOITURES
};
/* On supprime notre macro X */
#undef X

/* On définit de nouveau une macro X qui créera cette fois une
* chaîne de caractère de la marque. */
#define X(marque, modele, prix, couleur) # marque ,
char const * const marque_a [] = {
  X_VOITURE
};
#undef X

/* On définit notre macro pour créer le modèle. */
#define X(marque, modele, prix, couleur) # modele ,
char const * const modele_a [] = {
  X_VOITURE
};
#undef X

/* On définit notre macro pour créer le prix. */
#define X(marque, modele, prix, couleur) prix ,
int const prix_a [] = {
  X_VOITURE
};
#undef X

/* On définit notre macro pour créer la couleur. */
#define X(marque, modele, prix, couleur) # couleur ,
char const * const couleur_a [] = {
  X_VOITURE
};
#undef X

int main(void) {
  int i;
  /* Liste toutes les voitures répertoriées.
* Le '-' juste après le % sert à justifier le texte à gauche. */
  for (i = 0; i < NOMBRE_VOITURES; i++)
    printf("Marque: %-8s - Modele: %-5s - Prix: %-6d - Couleur: %-
6s\n",
           marque_a[i], modele_a[i], prix_a[i], couleur_a[i]);
```

```
                marque_a[i], modele_a[i], prix_a[i], couleur_a[i]);
    return EXIT_SUCCESS;
}
```

C'est à partir de maintenant qu'on voit bien l'utilité d'utiliser un tel truc !
Maintenant si vous voulez modifier votre liste vous n'avez aucun mal, et tout le code sera modifié en conséquence.

Code : C

```
#define X_VOITURE \
X(Renault, Clio, 10000, blanc) \
X(Renault, Laguna, 15000, vert) \
X(Peugeot, 207, 12000, rouge) \
X(Peugeot, 1007, 13000, noir) \
X(Citroen, C4, 15000, bleu)
```

Une autre alternative est de créer un fichier qui contiendra les appels aux macros et ensuite on inclut tout simplement notre fichier. L'extension du fichier n'a pas d'importance, mais on choisira en général .def.
Fichier contenant les appels :

Code : C - Mes_Voitures.def

```
X(Renault, Clio, 10000, blanc)
X(Renault, Laguna, 15000, vert)
X(Peugeot, 207, 12000, rouge)
X(Peugeot, 1007, 13000, noir)
X(Citroen, C4, 15000, bleu)
```

Et le fichier contenant les inclusions :

Secret (cliquez pour afficher)

Code : C

```
#include <stdio.h>
#include <stdlib.h>

#define X(marque, modele, prix, couleur) marque ## _ ## modele ,
enum index e {
# include "Mes_Voitures.def"
  NOMBRE_VOITURES
};
#undef X

#define X(marque, modele, prix, couleur) # marque ,
char const * const marque a [] = {
# include "Mes_Voitures.def"
};
#undef X

#define X(marque, modele, prix, couleur) # modele ,
char const * const modele a [] = {
# include "Mes_Voitures.def"
};
#undef X

#define X(marque, modele, prix, couleur) prix ,
int const prix a [] = {
# include "Mes Voitures.def"
```

```
};
#undef X

#define X(marque, modele, prix, couleur) # couleur ,
char const * const couleur_a [] = {
# include "Mes_Voitures.def"
};
#undef X

int main(void) {
  int i;
  /* Liste toutes les voitures répertoriées.
* Le '-' juste après le % sert à justifier le texte à gauche. */
  for (i = 0; i < NOMBRE_VOITURES; i++)
    printf("Marque: %-8s - Modele: %-7s - Prix: %-6d - Couleur: %-
6s\n",
           marque_a[i], modele_a[i], prix_a[i], couleur_a[i]);
  return EXIT_SUCCESS;
}
```

Si vous avez compris le principe des X-macros, ainsi que celui des macros à nombre variable d'arguments, vous allez pouvoir combiner les macros et les X-macros entre elles.

Exercice :

Vous devez créer, à l'aide des macros et X-macros, une structure contenant 2 nombres de type int nommés x et y ainsi que 3 autres de type double nommés pythagore, sinx, siny.
Vous devez définir une macro qui initialisera tous les champs de la structure à 0, et une autre qui servira à afficher tous les champs de la structure.

Solution :

Secret (cliquez pour afficher)

Code : C

```
#include <stdio.h>
#include <stdlib.h>

#define X_DEF_STRUCT(type, variable, rien) type variable;

#define X_PRINT_FORMAT_(type) X_PRINT_FORMAT_##type
#define X_PRINT_FORMAT_double "%f"
#define X_PRINT_FORMAT_int "%d"

#define X_PRINT(type, variable, struct_nombres) \
printf("%-7s %-10s : " X_PRINT_FORMAT_(type) "\n", \
#type, #variable, struct_nombres->variable);

#define X_INIT(type, variable, struct_nombres) \
struct_nombres->variable = 0;

#define DEC_NBVAR(_, ...) \
_(int, x, __VA_ARGS__) \
_(int, y, __VA_ARGS__) \
_(double, pythagore, __VA_ARGS__) \
_(double, sinx, __VA_ARGS__) \
_(double, siny, __VA_ARGS__)

struct nombres {
  DEC_NBVAR(X_DEF_STRUCT, )
};

void initStruct(struct nombres *s) {
  DEC_NBVAR(X_INIT, s)
```

```
}

void printStruct(struct nombres *s) {
  DEC_NBVAR(X_PRINT, s)
}

int main(void) {
  struct nombres s;
  initStruct(&s);
  printStruct(&s);
  puts("");
  return EXIT_SUCCESS;
}
```

Q.C.M.

Le premier QCM de ce cours vous est offert en libre accès.
Pour accéder aux suivants

Connectez-vous Inscrivez-vous
Est-ce que ce code est correct ?

Code : C

```
#include <stdio.h>
#include <stdlib.h>

#define PLUS(x, y) ((x) + (y))

int main(void) {
  PLUS(3+1, 4-3);
#undef PLUS
  return EXIT_SUCCESS;
}
```

- ◯ Oui, pas de problèmes.
- ◯ Non, il y a un #undef.
- ◯ Peut-être...
- ◯ Je ne sais pas.

Ce code affiche-t'il bonjour ou bonsoir ?

Code : C

```
#include <stdio.h>
#include <stdlib.h>

#define AFFICHE_BONJOUR(jour_ou_nuit) \
#if jour_ou_nuit == 1 \
# /* Si jour_ou_nuit vaut 1 on affiche bonjour */ \
printf("%s", "Bonjour !"); \
#else \
# /* Sinon on affiche bonsoir */ \
printf("%s", "Bonsoir !"); \
#endif

int main(void) {
  AFFICHE_BONJOUR(1)
  return EXIT_SUCCESS;
}
```

- ◯ Il affiche bonjour.
- ◯ Il affiche bonsoir.
- ◯ Il est midi, il ne peut qu'afficher bonjour !
- ◯ Il n'affiche rien.

Quelle expression est correcte ?

- ◯ #define VAL_ABS(x) (((x) > 0) ? (x) : (-(y)))
- ◯ #define VAL_ABS(x) (((x) > 0) ? (x) : -(x))
- ◯ #define VAL_ABS(x) (((x) (>) (0)) ? ((x)) : (-(x)))
- ◯ #define VAL_ABS(x) (x > 0) ? x : -x)

Correction !

Statistiques de réponses au QCM

Voilà vous en savez un peu plus sur le préprocesseur.
Cependant en ce qui concerne les X-macros il faut éviter de les utiliser (voire pas du tout), ou si vous les utilisez il faut le faire avec précaution ! C'est source d'erreurs, et des erreurs difficiles à débugger.
Faites attention à ne pas abuser des macros non plus, si elles sont mal utilisées, ça peut faire du code très difficile à lire. Exemple (merci SpaceFox) :

Code : C

```
#define x =
#define double(a,b) int
#define char k['a']
#define union static struct
```

J'espère que ça ne vous a pas paru trop compliqué.

A bientôt !