Define Volatile Static Const ++ -- les petits mots doux des langage C et C++

Non non, il ne s'agit pas ici d'une nouvelle insulte inventée par Hergé dans la bouche du capitaine Hadock.
Nous ne referons pas ici le cours du C pour les nuls, mais il semble nécessaire de repréciser ces notions et leurs portés.

Sommaire :

I : L'identificateur const

Cet identificateur est juste une aide au développeur pour indiquer que tel ou tel nom représente en fait le contenu d'une case mémoire qui ne peut être modifié dans le programme.

En ce qui concerne les compilateurs, le fait de déclarer une variable const (si c'est une variable elle doit varier et bien non pas forcément !) permet de la stocker au fil de l'eau dans le programme, que celui-ci soit implanté en mémoire RAM, ROM ou EEPROM. Donc il faut bien comprendre que la notion de variable en informatique signifie case ou lot de cases mémoire étiquetée par un nom de baptême. En aucun cas, cette notion précise si ces zones mémoires sont ou non modifiables par le programme.
Exemple

```
byte b=2 ;
const byte c=3 ;

..
b=b+c; // pas de problèmes
```

c=b+c ; // ERREUR c étant défini comme *const*, il est normalement impossible de la modifier
b=c++ ; // aussi erreur car cela reviens plus ou moins à condenser les deux instruction suivantes b=c; suivi de c=c+1; // (C.F. §4.2) FATAL ERROR !! ;

Const VS #define le match !
Nous ne rentrerons pas ici dans toutes les possibilités offertes par les directives *#define* mais comparons simplement deux pratiques.

Un petit exemple qui semble trivial mais ne nous y fions pas.

Version N°1	Version N°2

const int x=0;
 	
y=x+1; // fonctionne
	#define x (0)
	
	y=x+1; // fonctionne

Les deux versions de programme bien qu'équivalentes dans le fonctionnement ne produisent pas le même code après compilation :

La directive #define est en fait une indication de traitement de texte qui signifie : dans une première passe d'analyse du programme, remplacer tous les x par le caractère 0.
La ligne y=x+1 se transforme en y=0+1 dans la première analyse

Opération après compilation cela pourrait donner :

Version N°1	Version N°2
Identifier 2 octets consécutifs en mémoire et les remplir de 0. Aller chercher à l'adresse précédement identifiée la valeur du contenu (2 octets), charger la valeur 1 dans un registre puis ajouter ce registre à la valeur chargée. Reposer la valeur calculée dans les deux octets pointés par l'adresse connue de y.	Charger 0 dans un registre Charger 1 dans un autre registre Additionner les registres et mettre le résultat dans l'octet de poids faible pointé par l'adresse de y Mettre 0 dans l'octet de poids fort pointé par l'adresse de y plus 1

Attention : des #define mal utilisés peuvent conduire à des erreurs de compilation difficile à interpréter car après la première passe, certains identificateurs peuvent paraitre communs.
Un exemple pour comprendre :

```
#define x y
.....

  int x=1;
  float y=x+z;
....
```

Le code ci-dessus va provoquer une erreur de compilation car après la passe de compilation de "traitement de texte". En effet , le code se transforme en

int y=1;
float y=y+z;
Nous avons donc une double définition de la variable y qui provoque un courroux bien normal de la deuxième passe du compilateur. Ces erreurs sont très difficile à détecter, car, dans la majorité des cas le code en italique n'est pas connu de l'utilisateur. Ce code réside dans un fichier intermédiaire et temporaire de la phase de compilation.

Une bonne pratique des #define est de remplacer des valeurs numériques complexes par un identificateur simple dont le nom est porteur de sens
ex :
#define _PI 3.1415926525 // Que j'aime à faire apprendre un nombre utile au sages ...
ainsi toute allusion à _PI sera remplacée par 3.141596525

Un autre écueil du #define est illustré dans le bout de code ci-dessous

#define x 0 int ma0variable = 3 ; … int maxvariable=2 ; Provoquera à la compilation une erreur de duplication d'identificateur ma0variable	Le processus « traitement de texte » génèrera le code suivant int ma0variable = 3 … int ma0variable=2; // le x aura été substitué par un 0 d'où la double definition de la variable.

Pour pallier à une partie de ce problème, une bonne pratique (mais qui peut poser problème lors de l'utilisation de macro mal écrites, est de définir les pseudo constantes numériques de la manière suivante :
#define _PI (3.141592625)
#define x (0)

Comme toute chose à son revers, dans l'exemple précédent nous allons générer la variable ma(0)variable qui n'est pas reconnue comme un mot par le compilateur…

Une solution plus globale est d'adopter les règles suivantes :
Tout *define* est défini avec un underscore en préfix
Tout *define* numérique est mis entre parenthèses
Les noms de variable ou fonctions ne possèdent pas de _ mais sont nommées à l'aide du camel back système :
ceciEstMaVariableCorrectementNommee pour une variable ou fonction
CeciEstMaClasseCorrectemetNommee pour une classe ou structure complexe

#define _PI (3.141592625)
#define _x (0)

Evidement ces règles limitent beaucoup les erreurs idiotes lors de la compilation mais par expérience, il en reste toujours difficile à débusquer surtout lors de l'utilisation de macros et de *define* en cascade.

II : L'identificateur volatile

Le Langage C et C++ sont autant des langage de haut niveaux que des langage adapté à la gestion de processus nous dirons "au raz des soudures". Plus souvent dans ce dernier cas, notre programme interagit (ou du moins une partie via les processus d'interruptions) avec le monde extérieur qui est imprévisible. D'un autre coté, le développement en C se pose souvent en concurrent du développement en assembleur. Les compilateurs de bonne facture intègrent des options de compilation pour accélérer l'exécution du code sur les machines afin de se placer sur le podium de l'efficacité des langages de développent. Parmi ces mécanismes de compilation (proche de l'intelligence artificielle), on peut relever une modification de l'ordre des opérations cité dans notre code source pour éviter un appel à une fonction ou une variable, la mise en mémoire temporaire d'une variable dans un registre du microprocesseur ou même un déplacement temporaire d'une partie de mémoire dans une zone plus rapide d'accès. Ces optimisations sont alors faites sans en référer à l'utilisateur (l'auteur du code source) et ne peuvent être signalés aux processus interruptionnels qui comme je me permet de le répéter peuvent se déclencher n'importe quand. Ces "tripatouillages" peuvent provoquer des "plantages" inexplicables ou au pire des résultats erronés.

Par exemple : En cas d'optimisation conduisant à une adresse mémoire "mouvante" de la variable, le contenu de la variable peut être utilisée (ou modifiée) de manière impromptue par une procédure d'interruption (ou par un autre processus dans le cas d'un traitement parallèle ou pseudo parallèle), Le deuxième processus qui accèdera à la variable par son adresse mémoire (initiale) , peut ne pas avoir la vraie valeur à se mettre sous la dent, et, peut être plus grave, voir son action sur la variable réduite à néant lorsque le processus principal se décidera à remettre la copie du registre en place dans la case mémoire correspondant à notre variable.

Vous avez compris, il faut dire au compilateur "bas les pattes", cette variable est protégée, n'essaie pas de l'optimiser. Cette opération se fait en rajoutant la directive volatile dans la déclaration de la variable

ex
```
volatile byte a=0;
volatile boolean etat=false;
```

III : La déclaration Static

Nota : Les lignes qui vont suivre vont aller crescendo en complication alors ne vous affolez pas si vous ne comprenez pas du premier coup c'est normal !

Les améliorations du C en C++ ont induit soit de nouveaux mots clefs (ex: class) ou de nouvelles interprétations des mots clefs existants dans le nouveau contexte du C++, sans pour autant enlever les anciennes. Le mot "static" fait parti de cette dernière catégorie. Il peut suivant son contexte invoquer plusieurs actions, et, il faut redoubler de vigilance pour l'employer et/ou interpréter un code source.

III.1 Cas du C

deux points sont ici à évoquer :

1. Une variable statique définie à l'intérieur d'une fonction conserve sa valeur entre les invocations de la fonction.
2. Une variable globale statique ou une fonction déclaré "static" n'est "vue" que dans le fichier dans lequel elle est déclarée (il s'agit ici d'un raffinement de visibilité (variable "globale", mais locale au fichier dans lequel elle est déclarée))

III.1.1 Le point 1

Le plus compliqué à comprendre dans le cas du C ancestral ; Voici un exemple:

Exemple de code	Le résultat

#include <stdio.h>
 	
void fonctionTest ()
{
int a = 10;
static int sa = 10; // variable qui semble
// locale mais ne pas s'y fier !
a + = 5;
sa + = 5;
 	
printf ("a =% d, sa =% d \ n", a, sa);
}
 	
int main()
{
for (int i = 0; i <10; ++ i)
fonctionTest();
}
	a = 15, sa = 15
	a = 15, sa = 20
	a = 15, sa = 25
	a = 15, sa = 30
	a = 15, sa = 35
	a = 15, sa = 40
	a = 15, sa = 45
	a = 15, sa = 50
	a = 15, sa = 55
	a = 15, sa = 60
	
	a : Se comporte comme une variable locale à la fonction réinitialisé à chaque invocation de la fonction.
	sa : Se comporte comme une variable globale qui n'a été initialisée qu'une fois en début de programme.

L'utilisation du modificateur "static" est utile dans les cas où une fonction doit conserver une trace numérique non détruite entre les invocations et que vous ne souhaitez pas utiliser de variables globales. L'utilisation d'une telle variable correspond à celle d'une variable globale qui ne peut être invoquée que dans la fonction ou elle a été déclarée. Cela peut éviter des conflits de nommage de variables mais peut rendre votre code difficilement compréhensible et deboggable.

III.I.2 le point 2

Il est largement utilisé comme une fonctionnalité de "contrôle d'accès". Lorsque vous avez un fichier .c implémentant certaines fonctionnalités, il n'expose généralement que quelques fonctions "publiques" aux utilisateurs. Le reste de ses fonctions devrait être

rendu statique, afin que l'utilisateur ne puisse pas y accéder. On appelle cela l'encapsulation.

dixit Wikipedia:

Dans le langage de programmation C, static est utilisé avec les variables globales et les fonctions pour définir leur portée dans le fichier conteneur. Dans les variables locales, static est utilisé pour stocker la variable dans la mémoire allouée statiquement au lieu de la mémoire allouée automatiquement. Bien que le langage ne dicte pas l'implémentation de l'un ou l'autre type de mémoire, la mémoire allouée statiquement est généralement réservée dans le segment de données du programme au moment de la compilation, tandis que la mémoire allouée automatiquement est normalement implémentée comme pile d'appels transitoires.

Dans le cas de microcontrôleur possédant pour la plupart une architecture Harvard la localisation mémoire des variables peut se faire de la sorte :
static ... (en mémoire RAM car modifiable)
const static (en EEPROM car non modifiable)
const (en EEPROM car non modifiable)

Le fait de déclarer une variable static ne permet pas d'économiser notre précieuse RAM.

III.2 Cas du C++

Ici aussi deux points seront évoqués Ils sont tout deux associés à la définition de class (objets).

1. Une variable statique définie à l'intérieur d'une classe conserve sa valeur quelque soit l'élément de cette classe.

2. Une procédure ou fonction d'une classe définie comme statique de comporte comme une procédure ou fonction globale "hors classe" mais peut être appelée comme un membre de la classe considéré par un objet issu de l'instanciation de cette classe (ouf cela va nécessiter quelques exemples pour comprendre !).

III.2.1 Le point 1

Une classe peut contenir des données membres statiques. Ces données sont soit des données membres propres à la classe, soit des données locales statiques des fonctions membres de la classe. Dans tous les cas, elles appartiennent à la classe, et non pas aux objets de cette classe. Elles sont donc communes à tous ces objets.

Il est impossible d'initialiser les données statiques d'une classe dans le constructeur de la classe, car le constructeur n'initialise que les données des nouveaux objets. Les données statiques ne sont pas spécifiques à un objet particulier et ne peuvent donc pas être initialisées dans le constructeur. En fait, leur initialisation doit se faire lors de leur

définition, en dehors de la déclaration de la classe. Pour préciser la classe à laquelle les données ainsi définies appartiennent, on devra utiliser l'opérateur de résolution de portée (.:).

Programme	***Résultat***

class test
{
  static int i; // Déclaration dans la classe.
  ...
};
 	
int test::i=3; // Initialisation en dehors de la classe.
	La variable *test::i* sera partagée par tous les objets de classe test, et sa valeur initiale est *3*.

Note : La définition des données membres statiques suit les mêmes règles que la définition des variables globales. Autrement dit, elles se comportent comme des variables déclarées externes. Elles sont donc accessibles dans tous les fichiers du programme (pourvu, bien entendu, qu'elles soient déclarées en zone publique dans la classe). De même, elles ne doivent être définies qu'une seule fois dans tout le programme. Il ne faut donc pas les définir dans un fichier d'en-tête qui peut être inclus plusieurs fois dans des fichiers sources, même si l'on protège ce fichier d'en-tête contre les inclusions multiples.

Les variables statiques des fonctions membres doivent être initialisées à l'intérieur des fonctions membres. Elles appartiennent également à la classe, et non pas aux objets. De plus, leur portée est réduite à celle du bloc dans lequel elles ont été déclarées. Ainsi, le code suivant :

Programme	***Résultat***

#include <stdio.h>
class test
{
public:
  int modifie_compte(void);
};
int test::modifie_compte(void)
{
  static int compte=0;
  return compte++;
}
int main(void)
{
  test objet1, objet2; // declaration variables
 	
  printf("%d ", objet1.modifie_compte()); //
Affiche 0
  printf("%d\n", objet2.modifie_compte()); //
Affiche 1
	0 1
	
	La variable statique *compte* est la même pour les deux objets
	
	La variable "presque globale" compte n'a été initialisée qu'une fois.
	
	La variable compte n'est accessible qu'a l'intérieur de la procédure modifie_compte

return 0; }	

III.2.2 Le point 2

Les classes peuvent également contenir des fonctions membres statiques. Cela peut surprendre à première vue, puisque les fonctions membres appartiennent déjà à la classe, c'est-à-dire à tous les objets. En fait, cela signifie que ces fonctions membres se comportent comme des fonctions « globale ». Attention, le caractère « universel » de ces fonctions dans le cadre d'une classe interdit l'emploi du pointeur sur l'objet *this (this->..)*, comme c'est le cas pour les autres fonctions membres. Par conséquent, elles ne pourront accéder directement qu'aux données statiques de l'objet qui sont en réalité des données génériques de la classe. Par contre, les fonctions non statiques d'une classe peuvent invoquer les données et fonction statiques de cette même classe.

Programme	**Résultat**

class Entier
{
int i;
static int j;
public:
static int get_value(void);
};
 	
int Entier::j=0; // affectation hors constructeur
 	
int Entier::get_value(void) // definition de la méthode
{
j=1; // Légal.
return i; // ERREUR ! get_value ne peut pas accéder à i.
}
	Ici pas de résultats possibles mais une erreur

La fonction get_value de l'exemple ci-dessus ne peut pas accéder à la donnée membre non statique i, parce qu'elle ne travaille sur aucun objet. Son champ d'action est uniquement la classe Entier. En revanche, elle peut modifier la variable statique j, puisque celle-ci appartient à la classe Entier et non aux objets de cette classe.

Lors de l'écriture de code de classe contenant des données et/ou fonctions statiques et « normales », le type d'erreur (essai d'accéder aux champs non statiques depuis une fonction *static*) cité plus haut est très fréquent. Les messages d'erreurs du compilateur semblent « venu de l'espace » et sont difficiles à interpréter.

L'appel des fonctions membre statiques se fait exactement comme celui des fonctions membres non statiques, en spécifiant l'identificateur d'un des objets de la classe et le nom de la fonction membre, séparés par un point. Cependant, comme les fonctions membres ne travaillent pas sur les objets des classes mais plutôt sur les classes elles-mêmes, la

présence de l'objet lors de l'appel est facultative. On peut donc se contenter d'appeler une fonction statique en qualifiant son nom du nom de la classe à laquelle elle appartient à l'aide de l'opérateur de résolution de portée (::).

Programme

```
class Entier
{
 boolean  in;
 static int i;
 public:
 Entier(); // Le constructeur
 static int get_value(void);
};

// Le constructeur par defaut
Entier ::Entier()
{
in=true ;
}
int Entier::i=3; // affectation par le biais de la classe
int Entier::get_value(void) // attention on ne redeclare pas le caractère static
{
   return i;
}

int main(void)
 {
   // Appelle la fonction statique get_value :
   int resultat=Entier::get_value(); // appel via la classe car get_value est déclaré static

   Entier E =Entier(); // Creation d'une entité de la classe Entier
   int res=E.get_value // appel via un membre de la classe
   return 0;
}
```

Les fonctions membres statiques sont souvent utilisées afin de regrouper un certain nombre de fonctionnalités en rapport avec leur classe. Ainsi, elles sont facilement localisables dans le code source. Les risques de conflits de nommage entre deux fonctions membres homonymes sont alors réduits. Il est possible par exemple de créer une classe regroupant toutes les fonctions utilitaires de votre application. Ces fonctions seront toutes déclarées de type *static* et public pour pouvoir être invoquées sans ambigüité tout au long de votre programme. Il n'est pas alors recommandé de mixer dans cette même classe des méthodes « standards » et vos fonctions utilitaires.

Programme

```
// Ce petit programme crée pour arduino affiche
// le temps de début et fin de boucle ainsi que le temps
// calculé d'une boucle
// notre classe de fonctions utilitaires
class Util
{
    public :
    static int maximum(int v1, int v2)// par exemple
    {
      if (v1 > v2) return v1;
      return v2;
    }
    static int minimum (int v1, int v2)
    {
      if (v1 > v2) return v2;
      return v1;
    }
    static unsigned long delaiEntre(unsigned long t1, unsigned long t2)
    {
      if (t1 > t2)return t1 - t2;
      return (4294967295ul - t2) + t1;
    }
};

// Les initialisations
void setup()
{
  Serial.begin(9600); // initialisation de la voie serie
}

// Les variables globales
unsigned long t2=micros(), t1;

// La procedure principale appelée cycliquement
void loop()
{
  t2=micros();
  Serial.print(t2); Serial.print(" ");
  Serial.print(t1); Serial.print("->");

  Serial.println(Util::delaiEntre(t2,t1));
// appel en utilisant l'identificateur de classe
// ce qui permet d'éviter des erreurs de nommage
  t1=t2;
}
```

Résultat

```
1964728 1939768->24960

1989688 1964728->24960

2014648 1989688->24960

2039608 2014648->24960
```

```
2064568 2039608->24960

2089528 2064568->24960

2114488 2089528->24960

2139448 2114488->24960

2164408 2139448->24960
```

IV Incrémentation Décrémentation

En C comme dans d'autre langages, proche de la machine, il existe pour des opérations simples d'incrémentation ou de décrémentation des méthodes permettant de limiter la taille du code produit et donc d'optimiser le temps calcul.

IV.1 ++ vs --

Les affectations les plus fréquentes sont du type:

```
I = I + 1      et      I = I - 1
```

En C et donc par la même en C++ , nous disposons de deux opérateurs inhabituels pour ces affectations:

`I++` ou `++I`	pour l'incrémentation	(augmentation d'une unité)
`I--` ou `--I`	pour la décrémentation	(diminution d'une unité)

Les opérateurs `++` et `--` sont employés dans les cas suivants:

incrémenter/décrémenter une variable (p.ex: dans une boucle). Dans ce cas il n'y a pas de différence entre la notation ***préfixe*** (`++I --I`) et la notation ***postfixe*** (`I++ I--`).

IV.2 affectation et incrémentation

incrémenter/décrémenter une variable et en même temps affecter sa valeur à une autre variable. Dans ce cas, nous devons choisir entre la notation préfixe et postfixe:

!	`X = I++`	passe d'abord la valeur de I à X et ***incrémente après***
	`X = I--`	passe d'abord la valeur de I à X et ***décrémente après***
	`X = ++I`	***incrémente d'abord*** et passe la valeur incrémentée à X
	`X = --I`	***décrémente d'abord*** et passe la valeur décrémentée à X

Exemple

Supposons que la valeur de N est égal à 5:

Incrém. postfixe: `X = N++;` Résultat: N=6 et X=5
Incrém. préfixe: `X = ++N;` Résultat: N=6 et X=6

IV.3 incrémentation quelconque

le ++ et le -- ne concernent que l'incrémentions et la décrémentation unitaire. Mais dans certains cas (les matheux penseront aux suites arithmétiques $U_{n+1}=U_n+r$) il est utile d'ajouter un terme numérique quelconque (réel compris) à un autre terme.

L'opération suivante
toto=toto+titi;
nécessite les opération machines suivantes
charger l'adresse de toto,
aller chercher la valeur à cette adresse,
charger l'adresse de titi,
aller chercher la valeur à cette adresse
ajouter les deux valeurs
charger l'adresse de toto
mettre le résultat à cette adresse

une autre manière d'écrire
toto+=titi;
permet d'optimiser les recherche d'adresse comme suit
charger l'adresse de titi
aller charger la valeur à cette adresse
charger l'adresse de toto
ajouter la valeur courante à la valeur de cette adresse

ceci permet d'optimiser le code

il en va de même pour le code **-=** qui soustrait la valeur mentionnée à droite à la valeur de la variable de gauche.

par exemple le code suivant est valide

```
for (int i=0; i<10; i++)
    {
     for (int j=0; j<10; j++)
         {T[i][j]+=10.23;}
     }
```

et incrémente un tableau de 10x10 d'une valeur constante 10.23

```
for (int i=0; i<10; i++)
    {
     for (int j=0; j<10; j++)
         {T[i][j]+=U[i][j];}
     }
```

Ce code permet d'effectuer l'opération matricielle T=T+U
(car T+=U ne peut fonctionner car T et U sont des variables complexes)

Bibliographie :

Cet article étant en jachère, modification depuis plus de … x ans, il constitue un condensé d'expériences, de morceaux cours et de discussions issus de forum. Je suis dans l'impossibilité de vous citer toutes les sources mais remercie la communauté informatique pour toutes les informations qu'elle permet d'obtenir.