INF155 – SÉANCE 8(PART2)
CLASSES DE STOCKAGE
VISIBILITÉ

Anis Boubaker, Ph.D.
Maître d'enseignement
École de Technologie Supérieure

ÉTS

PLAN DE LA SÉANCE

- Les classes de stockage et la visibilité des variables
  - auto
  - register
  - static
  - extern
- Variables globales et fonctions statiques (static)
- Les variables constantes
  - Variables locales constantes
  - Paramètres constants
- La définition de types (***typedef***)

3 LES CLASSES DE STOCKAGE

LES VARIABLES...

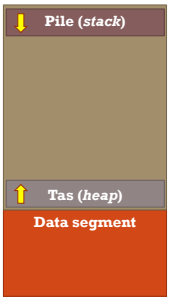
- Jusqu'à maintenant, les variables que nous avons déclarées sont:
  - Des variables locales à une fonction: elles sont désallouées dès que la fonction se termine;
  - Des variables globales: visibles dans toute l'unité de compilation (tout le programme, si le programme est constitué d'un seul fichier)

LES CLASSES DE STOCKAGE

- Il est possible de déclarer des variables de façon plus précise, afin d'agir sur:
  - La visibilité de la variable (d'où est-ce qu'elle est accessible?)
  - Sa durée de vie (Pendant combien de temps elle est accessible?)
  - Son espace de stockage (Où elle sera stockée?)

CLASSE "AUTO"

- C'est la classe par défaut d'une variable locale
- Une variable de cette classe est:
  - **Visible** dans la fonction où elle été déclarée;
  - **Stockée** dans la pile de fonctions (dans l'espace réservé à la fonction);
  - **Sa durée de vie** est la même que la fonction où elle a été déclarée – Elle est désallouée dès que la fonction a terminé son exécution.



CLASSE "AUTO"

- La classe auto est la classe de stockage par défaut.
- Il est inutile de préciser la classe, mais ce n'est pas une erreur de le faire:

Déclaration:

```
auto int un_entier;
```

- Si vous utilisez le compilateur en mode C++ (extension .cpp) vous aurez une erreur car le mot clé auto a une autre signification.

CLASSE "REGISTER"

- Une variable **locale** de cette classe est:
  - **Visible** dans la fonction où elle été déclarée (comme auto);
  - **Stockée** dans un registre du processeur **ou** comme une variable de la classe auto (c'est au compilateur de prendre la décision finale);
  - **Sa durée de vie** est la même que la fonction où elle a été déclarée (variable locale) **ou** du programme (variable globale)
- Utile dans le cas d'une variable accédée très fréquemment dont on veut optimiser le temps d'accès.

Déclaration:

```
register int un_entier;
```


CLASSE EXTERNE OU GLOBALE

- La classe ***extern*** est la classe des variables globales;
  - **Visible** dans l'unité de compilation (fichier .c) où elle a été déclarée (on peut l'étendre avec le clé ***extern***);
  - **Stockée** dans le segment de données – son espace est alloué tout au long de l'exécution du programme;
  - **Sa durée de vie** est la même que la durée de vie du programme.

MOT CLÉ "EXTERN"

- Il est possible de rendre une variable globale accessible depuis une autre unité de compilation;
- Pour y accéder, il suffit de la déclarer comme variable globale externe.

foo.c:

```
int main(void)
{
   //Réfère à la variable x
   //dans une autre unité de
   //compililation (bar.c)
   extern int x;

   //Affiche 10
   printf("x=%d", x);
}
```

bar.c:

```
int x = 10;
```

10 ÉTS

CLASSE STATIQUE (STATIC)

- Une variable **locale** de cette classe est:
  - **Visible** dans la fonction où elle été déclarée (comme auto);
  - **Stockée** dans le segment de données – Elle est allouée une seule fois à la compilation

    ➔ **Par conséquent, la variable préserve sa valeur entre les appels de fonction.**
  - **Sa durée de vie** est la même que la durée du vie du programme.

11 ÉTS

CLASSE STATIQUE - EXEMPLE

```
void visiter(void)
{
   static int nb_visites=0;
   nb_visites++;
   printf("Vous m'avez visitée %d fois.\n%d",nb);
}

int main(void)
{
   int i;

   for(i=0; i< 10; i++)
   {
      visiter();
   }

   return 0;
}
```

Affiche:
Vous m'avez visitée 1 fois
Vous m'avez visitée 2 fois
Vous m'avez visitée 3 fois
Vous m'avez visitée 4 fois
Vous m'avez visitée 5 fois
Vous m'avez visitée 6 fois
Vous m'avez visitée 7 fois
Vous m'avez visitée 8 fois
Vous m'avez visitée 9 fois
Vous m'avez visitée 10 fois

12 ÉTS

CLASSE STATIQUE - UTILISATION

- Les cas d'utilisation de variables de la classe statique sont rares.
- Généralement utile lorsqu'on veut traiter des données par lot (on doit se rappeller où on est rendus entre les appels)
- Il ne faut pas en abuser car:
  - C'est un gaspillage de mémoire;
  - Ça rend le code moins maintenable.

LES VARIABLES GLOBALES ET FONCTIONS STATIQUES

LE MOT CLÉ STATIC

- Nous venons de voir que le mot clé **static** permet de définir une variable locale de façon persistante dans une fonction.
- Ce mot clé a aussi un autre usage: il permet de restreindre la visibilité de variables globales et de fonctions.
- Il faut distinguer le cas le mot clé static est utilisé:
  - Avec une variable locale;
  - Avec une variable globale ou une fonction.

FONCTIONS ET V. GLOBALES STATIQUES

- Il est possible de limiter la visibilité d'une variable globale ou d'une fonction à l'unité de compilation (fichier .c) où elle a été définie.
- Pour ce faire, il suffit de précéder sa déclaration (variable) ou son prototype (fonction) par le mot clé **static**.

VARIABLES GLOBALES STATIC

foo.c:

```
static int x = 10; //Accessible uniquement dans foo.c

int main(void)
{
    x++;
}
```

- Une variable globale déclarée static n'est plus accessible depuis un autre fichier source – même en utilisant extern!

FONCTIONS STATIQUES

- Une fonction statique est une fonction limitée à l'unité de compilation (fichier .c) à laquelle appartient.
- Elle peut être appelée par toute fonction du même module.
- Elle ne peut pas être appelée par des fonctions d'un autre module.

foo.c:

```
static int carre(int x)
{
     return x*x;
}

void une_fonction(void)
{
    printf("Le carre de 10:%d", carre(10));
}
```

bar.c:

```
#include "foo.h"

void une_autre_fonction(void)
{
    //ERREUR!
    printf("10^2=%d", carre(10));
}
```

FONCTION STATIQUES - USAGE

- Les fonctions statiques sont utiles pour limiter l'accès à une fonction aux fonctions d'un module;
- Cela favorise l'encapsulation du module: on choisit quelles fonctions exposer au monde extérieur;
- L'encapsulation favorise une bonne maintenabilité du module;
- **Le prototype d'une fonction statique ne doit pas être déclaré dans le fichier d'en-tête!**

VARIABLES CONSTANTES

LES CONSTANTES AVEC #DEFINE

- Nous avons vu, dans les cours précédents, qu'il est possible de déclarer des constantes à l'aide de la directive de précompilation:

  #define

- Les constantes (macros) déclarées avec #define ne sont pas des variables: le précompilateur remplacera toute occurrence de la macro par sa valeur.

LES VARIABLES CONSTANTES

- Il est également possible de définir des variables constantes;
- Une variable constante est une variable, dont la valeur ne peut changer après son initialisation;
- On utilise le mot clé **const**:

Déclaration:

```
const int un_entier_constant=10;
un_entier_constant = 20; //ERREUR
```

22 ÉTS

LES VARIABLES CONSTANTES - USAGE

- Quand utiliser #define:
  - Lorsque la mémoire est limitée;
  - Lorsque la constante a une portée globale (tout le module ou tout le programme)
- Quand utiliser une variable constante:
  - Lorsque l'utilisation de la constante est limité à une fonction (toujours)
  - Lorsqu'on a besoin de transmettre un pointeur vers la constante
- Notez qu'une variable constante est une variable: elle a un type, et le compilateur peut détecter des conversion illicites (→ meilleure maintenabilité)

23 ÉTS

LES PARAMÈTRES CONSTANTS

- Il est possible d'utiliser le mot clé const avec des paramètres de fonctions
- Le paramètre devient alors en lecture seule dans la fonction.

```
int somme_tableau(const int tab[], int nb_elts)
{
      int i;
      int somme;
      tab[0]=10; //ERREUR
      for(i=0; i<nb_elts; i++)
      {
            somme+=tab[i];
      }
      return somme;
}
```

24 ÉTS

PARAMÈTRES CONSTANTS - USAGE

- Ils permettent de s'assurer qu'une variable ne sera pas modifiées dans la fonction: Si une instruction tente de la modifier, le système nous en avise par une erreur.
- Très utile lorsque ledit paramètre est un tableau ou un pointeur.
  - Nous n'avons pas le choix de transmettre un tableau par référence;
  - En modifiant le tableau dans la fonction, on modifie le tableau chez l'appelant de de la fonction.
  - Dans bien des cas, ceci n'est pas désirable. Alors, on se protège de nous-mêmes en déclarant le paramètre comme une constante.

26 DÉFINITIONS DE TYPES

DÉFINITION DE TYPE?

- Jusqu'à maintenant, nous avons utilisé les types de données prédéfinis du C : int, double, char, etc.
- La **définition de type** consiste à définir notre propre type de données et de lui donner un nom distinctif
- Une fois avoir défini un type, nous seront en mesure de déclarer des variables de type de données que nous avons défini
- Pour définir un type, nous utilisons le mot clé **typedef**

DÉFINITION DE TYPE

Syntaxe:

```
typedef type id_type1 [,id_type2, id_type3, ...];
```

Exemple:

```
typedef int t_entier; //On peut utiliser le type t_entier à la
                      // place de int

int main(void)
{
   t_entier ma_var=10;

   printf("ma_var = %d", ma_var);
}
```

28 ÉTS

DÉFINITION DE TYPE TABLEAU

- Il est aussi possible de définir un type tableau, ce qui permet d'uniformiser et d'alléger les déclarations de tableaux du même type.

Exemple:

```
#define TAILLE_MAX 10;
//tableau d'entiers 10x10
typedef int t_tab_entier[TAILLE_MAX][TAILLE_MAX];

int main(void)
{
   t_tab_entier mon_tableau = { {10, 15} };

   printf("Case [0][1]: %d", mon_tableau[0][1]);
}
```

29 ÉTS

DÉFINITION DE TYPES - UTILITÉ

- La définition de type est très peu utile (voir inutile) quand il s'agit de juste renommer des types de base (ex.: int)
- Elle est utile lorsque la déclaration d'une variable est complexe et que le même type de variable se répète (ex.: Dans le TP2: on pourrait créer un type grille)
- Très utile lorsque nous utiliserons les types complexes (enregistrements et énumérations).

30 ÉTS

EXAMEN FINAL DANS 4 SEMAINES!