École de technologie supérieure

THÈME 8
Accès aux bagages: destination locale ou globale?

INF125 Introduction à la programmation
Sylvie Ratté et Hugues Saulnier

1. Portée, accès, mémoire et modification

A. Le village local et le village global

Une variable est de portée LOCALE lorsqu'elle est déclarée à l'INTÉRIEUR d'un BLOC. Elle existe uniquement lorsque le bloc s'exécute et disparaît lorsque l'exécution du bloc se termine. Elle est donc connue et est accessible uniquement de l'intérieur du bloc.

Une variable est de portée GLOBALE lorsqu'elle est déclarée à l'EXTÉRIEUR de tout BLOC. Elle existe pendant toute la durée de l'exécution. Elle est donc connue et est accessible de l'intérieur de TOUS les blocs à moins qu'une variable locale ne porte le même nom.

Considérez le fichier suivant:

<table>
<tr><th>CODE</th><th>BLOCS MÉMOIRES</th></tr>
<tr><td><pre>int P;
int x;
int z;

void exemple(int a, int x, int y){
  int P, Q;
  int b;
  ...
}

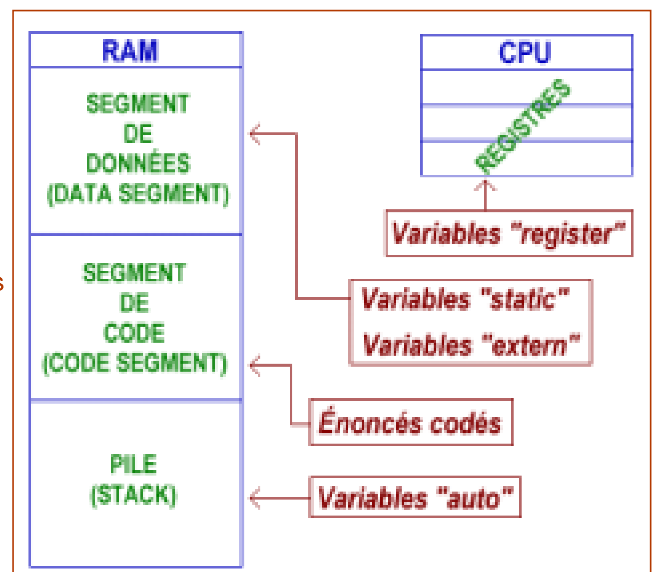
int main(void){
   int a,b,c;
        a = b = c = 5;
        exemple(a,b,c);
        return 0;
}</pre></td><td>GLOBALES: P, x, z

Fonction "exemple"
LOCALES: a, x, y, P, Q, b
La fonction a de plus accès à la variable globale "z"

Fonction "main"
LOCALES: a, b, c
La fonction "main" a de plus accès aux variables globales P, x et z.</td></tr>
</table>

B. Classes d'allocation

B.1. INTRODUCTION



Un programme qui s'exécute sur un ordinateur (on dit aussi "un processus") vit dans l'environnement RAM illustré ci-contre.

Cet environnement baigne évidemment dans la mémoire RAM de la machine; en ce qui concerne les registres, ils font partis du processeur (le CPU).

Qu'est-ce qui détermine où vont les objets (les variables)? Évidemment, le code se retrouve dans le segment CODE. En ce qui concerne les variables, elles peuvent se retrouver dans le segment des données, dans la pile ou dans les registres.

Une variable automatique se retrouve dans la pile, une variable registre se retrouve évidemment dans un registre du CPU, une variable statique est placée dans le segment de données et il en est de même pour une variable "externe".

Les mots clefs "auto", "static", "register" et "extern" permettent de mentionner explicitement la classe d'allocation désirée. De toute façon, selon le type d'identificateur visé, il y a toujours une classe d'allocation par défaut. Le tableau suivant résume les classes par défaut utilisées par

le compilateur et mentionne également les classes d'allocation que vous pouvez utiliser.

identificateur	classe d'allocation par défaut	classe d'allocation autorisée
variable globale	extern	static
variable locale	auto	static, register
fonction	extern	static

Examinons maintenant ces quatre classes d'allocation.

B.2. AUTO

À partir du moment où l'on déclare une variable dans un bloc, celle-ci appartient à la classe d'allocation AUTO. Si l'on tient absolument à être explicite, on écrira:

```
auto int a;
```

plutôt que:

```
int a;
```

La variable est ainsi placée dans la PILE (on notera que la pile est ainsi "remplie" en allant VERS le segment contenant le CODE... c'est donc dire que si la pile déborde, c'est le code qui risque d'être endommagé). Le programmeur ne peut espérer que sa variable soit toujours placée au même endroit dans la pile.

B.3. REGISTER

Il est possible de mentionner au compilateur que l'on aimerait, pour des questions de rapidité, que notre variable appartienne à la classe d'allocation REGISTER. Les variables seraient ainsi placées directement dans les registres du CPU. On utilise alors le mot "register" pour effectuer cette demande.

Notez cependant qu'il s'agit d'un simple SOUHAIT. S'il y a de la place dans les registres, on obtiendra l'effet souhaité sinon, le compilateur placera vos variables en mode auto. Remarquez que les compilateurs récents font, de toute façon, beaucoup d'optimisation et que lors de ce travail, il peut très bien mettre en registre des variables qui sont indiquées auto.

Question:
Si la classe auto est explicitement mentionnée, est-ce que l'optimisation effectuée par le compilateur peut remplacer cette allocation par la classe register ou si ce remplacement ne peut se faire que si la classe auto n'est pas explicitement mentionnée.
Réponse: Lors d'une optimisation, le compilateur peut très bien attribuer la classe "register", par contre, si la classe "volatile" est utilisée, le compilateur ne pourra pas effectuer une optimisation (nous n'avons pas vu cette classe mais je la mentionne à tout hasard).

Depuis le début, nous parlons des variables déclarées dans un bloc. Quand est-il des paramètres des fonctions?

Question:
Quelle est la classe d'allocation par défaut des paramètres d'une fonction?
Réponse: auto puisque ce sont en fait des variables locales. Mais si on y tient, on peut explicitement mentionner auto:

```
int f_toto(auto int ca1, auto int ca2)...
```

Question:
La déclaration suivante serait-elle acceptable?

```
int f_toto(register int ca1)...
```

Réponse: Oui, mais il s'agit d'un souhait. Sans doute verrions-nous une différence en utilisant un vieux compilateur.

Question:
Que pensez-vous maintenant de la déclaration suivante?

```
int f_toto(static int a)...
```

Réponse: Ouch! Comment devons-nous interpréter "a"? En utilisant "static", on indique que "a" doit avoir une vie permanente, ce qui n'a aucun sens. Cette demande n'est pas du tout cohérente avec la notion même de paramètres. Un paramètre, par définition, a une vie de courte durée. Impossible de rendre ça permanent. De plus, on dit qu'un paramètre est une variable locale qui a la particularité d'être initialisée lors de l'appel. Cette dernière particularité n'est pas le propre des variables "static" qui ne sont initialisées qu'une fois.

Question:
Que pensez-vous maintenant de la déclaration/définition suivante?

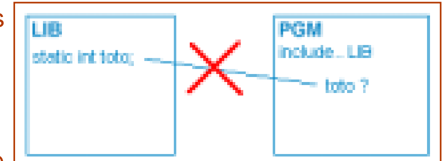
```
int f_toto(int ca1, int ca2){
    register int i;
    ...
}
```

Mais tout est parfait ici.

B.4 STATIC

Une variable préfixée de "static" vit aussi longtemps que le programme. Elle est placée dans le segment de DONNÉES. Il est impossible d'avoir accès à cette variable ailleurs que dans le bloc (ou le fichier) dans lequel elle a été déclarée. Ces variables sont très utiles dans les librairies. Le mot "static" nous permet de déclarer une variable globale mais qui n'est globale que pour la librairie. Elle n'est pas accessible ailleurs.

Le mot "static" devant un identificateur de fonction rend, pour ainsi dire, le bâtisseur de liens (linker) "aveugle". La fonction n'est pas disponible ailleurs que dans la librairie. Un programme qui tente d'utiliser une fonction préfixée de "static" et appartenant à une librairie provoque une erreur de "linker" lors de sa construction.



Si vous préfixez la déclaration d'une variable locale à un bloc du mot "static", la définition de cette variable (i.e. son initialisation) sera effectuée la première fois que vous entrez dans ce bloc. De plus, cette variable conservera sa valeur d'un appel à l'autre de ce même bloc.

```
void F_toto() {
    static int toto = 5; // déclaration et définition
    printf("\n%d", toto);
    toto += 10;
}
```

La première fois, F_toto affiche 5, ensuite elle affichera: 15, 25, 35, 45, 55, etc

B.5 EXTERN

La classe d'allocation "extern" est simple et complexe en même temps. Disons simplement que l'on place "extern" devant la déclaration d'une variable (ou de tout identificateur) qui, de fait, appartient à autre fichier (une librairie). C'est utile, par exemple, lorsqu'on désire utiliser une variable qui appartient à une librairie. Nous avons préalablement vu comment bloquer cette utilisation (avec static), avec extern c'est totalement l'inverse. De plus, le mot "extern" apparaît, non pas dans la librairie, mais bien dans le programme qui utilise la librairie.

Prenons une librairie quelconque que nous appellerons "LIB". Voici le contenu de son .H et de son .C. Vous remarquez que la librairie contient la déclaration d'une variable globale "a".

Contenu de LIB.H

```
void ditBonjour(int n);
```

Contenu de LIB.C

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

int a;

static void allo(void){
        printf("allo %i\n", a);
}

void ditBonjour(int n){
        int i;
        for (i = 0; i < n; i++)
                allo();
}
```

La fonction "allo" n'est pas dans le .H. En conséquence, cette fonction n'est pas accessible par l'utilisateur-programmeur. Cette fonction est tout à fait "privée"; elle est d'ailleurs précédée du mot "static". Elle ne peut donc être utilisée que par les fonctions de la librairie et pas ailleurs.

Voici un petit "main" utilisant la librairie.

Contenu de PGM.C

```
#include "lib.h"

int main(void){
      extern int a;
      a = 5;
      ditBonjour(2);
      a = 101;
      ditBonjour(2);
      return 0;
}
```

Si l'on désire avoir accès à la variable "a" (qui appartient à la librairie), on doit alors la déclarer et faire précéder sa déclaration du mot "extern" indiquant par là que nous désirons avoir accès à cette variable externe... Ainsi:

Le "main" suivant ne serait pas valide car pour le compilateur, la variable "a" n'a pas été déclarée.

```
#include "lib.h"

int main(void){
      a = 5;
      ditBonjour(2);
      a = 101;
      ditBonjour(2);
      return 0;
}
```

Inversement, le "main" suivant ne produit pas l'accès désiré:

```
#include "lib.h"

int main(void){
      int a;
      a = 5;
      ditBonjour(2);
      a = 101;
      ditBonjour(2);
      return 0;
}
```

car le "a" utilisé ne sera pas celui de la librairie.

C. Qualificatifs de types

Les classes d'allocation (auto, register, static et extern) fournissent des informations sur la durée de vie des variables et leur visibilité (c'est-à-dire leur portée). Les qualifieurs de type fournissent des informations sur la façon dont elles pourront être utilisées. Il en existe deux: "const" et "volatile". Ces deux qualificatifs se placent directement devant le type de la variable que vous déclarez.

C.1. CONST

Le qualificatif "const" spécifie que l'objet (la variable) ne peut être modifié. En fait, c'est la façon la plus appropriée de déclarer des constantes.

Exemple: la déclaration "const int NBLIG = 24;" spécifie que l'identificateur NBLIG est une constante de type "int" dont la valeur sera 24.

L'utilisation de "const" pour définir des constantes devrait remplacer la directive #define car elle permet de limiter plus facilement la portée de l'identificateur. On ne peut malheureusement pas l'utiliser pour définir la taille de nos tableaux. Ainsi, la suite de déclarations suivante n'est pas possible et l'on doit alors, dans ce cas précis, continuer d'utiliser #define.

```
const int TAILLE = 24;
double Tablo[TAILLE];
```

Le qualificatif "const" peut également être utilisé pour rendre des tableaux constants. Le tableau "jours" déclaré comme suit:

```
const int jours[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

NE PEUT PAS être modifié.

La directive "const" peut également être utilisée devant les déclarations de paramètres afin d'indiquer que la fonction ne modifie pas ses arguments.

B.2. VOLATILE

Le qualificatif "volatile" signifie exactement l'inverse de "const". Il signifie en fait que la valeur de la variable peut être modifiée à tout moment. On utilise essentiellement ce qualificatif lorsque notre programme utilise des variables qui seront mises à jour par des sources externes: fonctions du système d'exploitation ou procédures d'interruption. Nous n'utiliserons pas ce qualificatif pendant ce cours.

2. Créez et gardez vos inventions à l'abris des accès indiscrets

A. Définition de types

A.1. INTRODUCTION À " typedef"

Avec l'ajout des qualificatifs de type et les classes d'allocation, sans mentionner les *, les [ ] et les types préfixés de "unsigned", les déclarations de variables peuvent devenir très longues:

```
const static unsigned long * T[50];   /* ouf ! */
```

Pour simplifier nos déclarations mais également pour les rendre plus lisibles, on peut utiliser le mot réservé "typedef". Le "typedef" permet ainsi d'associer un nom au type que nous utiliserions. Ainsi, après la déclaration suivante, "Compteur" devient un synonyme de "unsigned long".

```
typedef unsigned long Compteur;
```

On peut alors déclarer nos variables de type "Compteur" au lieu de "unsigned long":

```
int main() {
   Compteur cc;
   ...
```

De la même manière, on peut donner un synonyme à n'importe quel type:

```
typedef int Index;
```

Le programme devient ainsi beaucoup plus lisible.

A.2. UTILISATIONS DIVERSES

Pour donner un nom à un type tableau, on procède comme suit:

```
typedef type-de-base nom-du-type[taille];
```

Ainsi, suite à la déclaration suivante:

```
typedef int serie[45];
```

On peut effectuer la déclaration de variable (a) qui remplacement avantageusement la déclaration (b).

```
(a) serie T;
(b) int[45] T;
```

On procède de la même manière avec les tableaux à deux dimensions et plus.

Pour donner un nom à un type "pointeur", on procède de la manière habituelle. Ainsi, suite au "typedef" présenté en (a), on pourra effectuer la déclaration (b) qui remplace ainsi la déclaration (c).

```
(a) typedef int * tpt;
(b) tpt P;
(c) int * P;
```

A.3. LIBRAIRIES ET TYPEDEF

Si le nouveau type ainsi défini par le "typedef" doit être accessible de l'utilisateur-programmeur qui utilise votre librairie, le "typedef" se retrouvera dans le .H de la librairie, sinon, il se retrouvera dans le .C de cette librairie.

A.4. EXEMPLE

Le programme "vie.c" que nous avons vu au thème 7 utilise les déclarations suivantes:

```
#define NBLIG 10
#define NBCOL 20

long modifierMonde(char mondeCourant[][NBCOL], char mondeSuivant[][NBCOL]);
void echangerMondes(char mondeCourant[][NBCOL], char mondeSuivant[][NBCOL]);
void afficherMonde(char monde[][NBCOL]);
void initMonde(char monde[][NBCOL]);
```

En utilisant des "typedef", on rend la lecture du programme beaucoup plus agréable.

```
#define NBLIG 10
#define NBCOL 20

typedef char tmonde[NBLIG][NBCOL];
long modifierMonde(tmonde mondeCourant, tmonde mondeSuivant);
void echangerMondes(tmonde mondeCourant, tmonde mondeSuivant);
void afficherMonde(tmonde monde);
void initMonde(tmonde monde);
```

ÉTUDE DE CAS

Voici à nouveau une version du Bingo mais cette fois, nous avons placé le système dans une librairie et nous créons un programme qui joue maintenant avec 10 cartes!

La librairie BINGO comprend BINGO.H et BiNGO.C. Le programme utilisant cette librairie se nomme MBINGO.C.

A. BINGO.H

Dans le .H nous avons placé les "typedef" et les prototypes des fonctions qui sont accessibles par tout utilisateur de la librairie.

```
typedef char tnumero;
typedef char tboulier[75];
typedef char tcolonne[5];
typedef tcolonne tcarte[5];
void remplirCarte(tcarte carte);
void afficherCarte(tcarte carte);
void marquerCarte(tcarte carte, tnumero numero);
int Bingo(tcarte carte);
tnumero tirerNumero(tboulier tirage);
void initTirage(tboulier boulier);
void afficherMessageGagnant(int R);
```

La fonction "initTirage" est nouvelle et permet d'initialiser notre "boulier" (tableau de 75 emplacements) à zéro, signifiant par là qu'aucun des chiffres n'est encore tiré.

B. BINGO.C

Le .C comprend évidemment la définition de toutes les fonctions précédentes mais elle contient aussi celles des fonctions non visibles. Ces dernières sont évidemment toutes précédées du mot réservé "static". Voici le début de la librairie.

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

static void remplirColonne(tcolonne colonne, tnumero min, tnumero max);
static int present(tboulier T, tnumero elem, int A, int B);
static int rechercher(tcarte carte, int A, int P, int D, int B, int Q, int E);
static double alea(void);
static tnumero alea75(void);
static tnumero aleaMinMax(tnumero min, tnumero max);
les définitions de toutes les fonctions suivent...
```

Vous pouvez consulter le code complet ici: BINGO.C.

C. MBINGO.C

Le programme principal permet de jouer avec 10 cartes. Nous avons donc besoin d'un tableau de 10 éléments de type "tcarte". Rien de plus simple. Nous utilisons un petit "typedef" pour alléger le programme:

```
#define NBCARTES 10

typedef tcarte tmadame[NBCARTES];
```

C.1. LES DÉCLARATIONS DES FONCTIONS

Puisque nous devons manipuler 10 cartes, nous utiliserons quelques fonctions de manière à rendre le code du "main" = plus concis. Voici leur prototype.

```
void RemplirLesCartes(tmadame cartes);
void AfficherLesCartes(tmadame cartes);
void MarquerLesCartes(tmadame cartes, tnumero No);
int VerifierBingoSurLesCartes(tmadame cartes, int * noCarteGagnante);
```

C.2. LE "MAIN"

Le "main" peut désormais être écrit. Notez l'utilisation des fonctions de la librairie (indiquée en bleu) et celle des fonctions appartenant au programme (indiquée en vert).

```
int main(void){
   int Resultat;
   int noGagnant = 0;
   tmadame cartes; /* nous jouons avec 10 cartes ! */
   tboulier tirage;
   srand(time(NULL));
   initTirage(tirage);
   RemplirLesCartes(cartes);
   AfficherLesCartes(cartes);
   do{
      MarquerLesCartes(cartes, tirerNumero(tirage));
      AfficherLesCartes(cartes);
      getch();
   }while (!(Resultat = VerifierBingoSurLesCartes(cartes,  &noGagnant)));
   printf("La carte no %i est gagnante.\n");
   afficherCarte(cartes[noGagnant]);
   afficherMessageGagnant(Resultat);
   return EXIT_SUCCESS;
}
```

C.3 LES DÉFINITIONS DES FONCTIONS

Les définitions des fonctions sont évidentes.

```
void RemplirLesCartes(tmadame cartes){
  int i;
  for (i = 0; i < NBCARTES; i++)
    remplirCarte(cartes[i]);
}
```

```
void AfficherLesCartes(tmadame cartes){
  int i;
  for (i = 0; i < NBCARTES; i++)
    afficherCarte(cartes[i]);
}
```

```
void MarquerLesCartes(tmadame cartes, tnumero No){
  int i;
  for (i = 0; i < NBCARTES; i++)
    marquerCarte(cartes[i], No);
}
```

```
int VerifierBingoSurLesCartes(tmadame cartes, int * noCarteGagnante){
  int i, Resultat;
  for (i = 0; i < NBCARTES; i++){
    Resultat = Bingo(cartes[i]);
    if (Resultat){ /* carte gagnante ! */
       *noCarteGagnante = i;
       return Resultat;
    }
  }
  return 0;
}
```

Les trois fichiers peuvent être récupérés ici: BiNGO.H, BINGO.C, MBINGO.C.