SI4 - Bases de la programmation

Les pointeurs

Allocations dynamiques

Références

Chaînage

Introduction

* Le Langage C/C++ fait beaucoup appel aux pointeurs
* Dans certains cas, il n'est pas possible de travailler autrement (allocation dynamique, fonction, …)
* Les pointeurs sont présents en : C, C++, PASCAL, COBOL, ASM (naturel).
* Les pointeurs permettent d'accéder à des composants (matériels) mappés en mémoire centrale.
* Pas de pointeurs directement manipulables en Java, Swift, C#.

1. Pseudo-code

```
VAR
        i : entier
        p : pointeur sur entier

DEBUT
        i <-10;
        p<-(adresse)i // &i
        AFFICHER (contenu)p // *p
FIN
```

L'exemple impose de travailler directement en C, le pseudo-code n'étant pas suffisamment normalisé.

2. Langage C

```
void main()
{
        int intVariable; // une variable entier

        intVariable = 10;

        int * pInt; // un pointeur sur entier pas initialisé

        pInt = &intVariable; //le pointeur pointe !
        printf("%d",*pInt); // affichage de la valeur pointée
        getch();
}
```

3. Opérations de base

* & : récupère une adresse
* * : contenu du pointeur
* malloc : allocation dynamique (malloc.h)
* free : désallocation mémoire
* new : allocation dynamique en C++
* delete : désallocation mémoire
* Un pointeur est initialisé à NULL : pointe sur rien

* Les pointeurs sont utilisables sur tous les types et les tableaux.

4. Allocation dynamique en C

```
char * charVariable =  ”\0"; // un pointeur
int size = 256;
charVariable = (char *) malloc(size * sizeof(char)); // allocation dynamique
charVariable = "Ceci est une chaîne de caractères relativement longue !";
printf("%s ", charVariable);
```

5. Tableaux et pointeurs

* Un pointeur sur tableau :

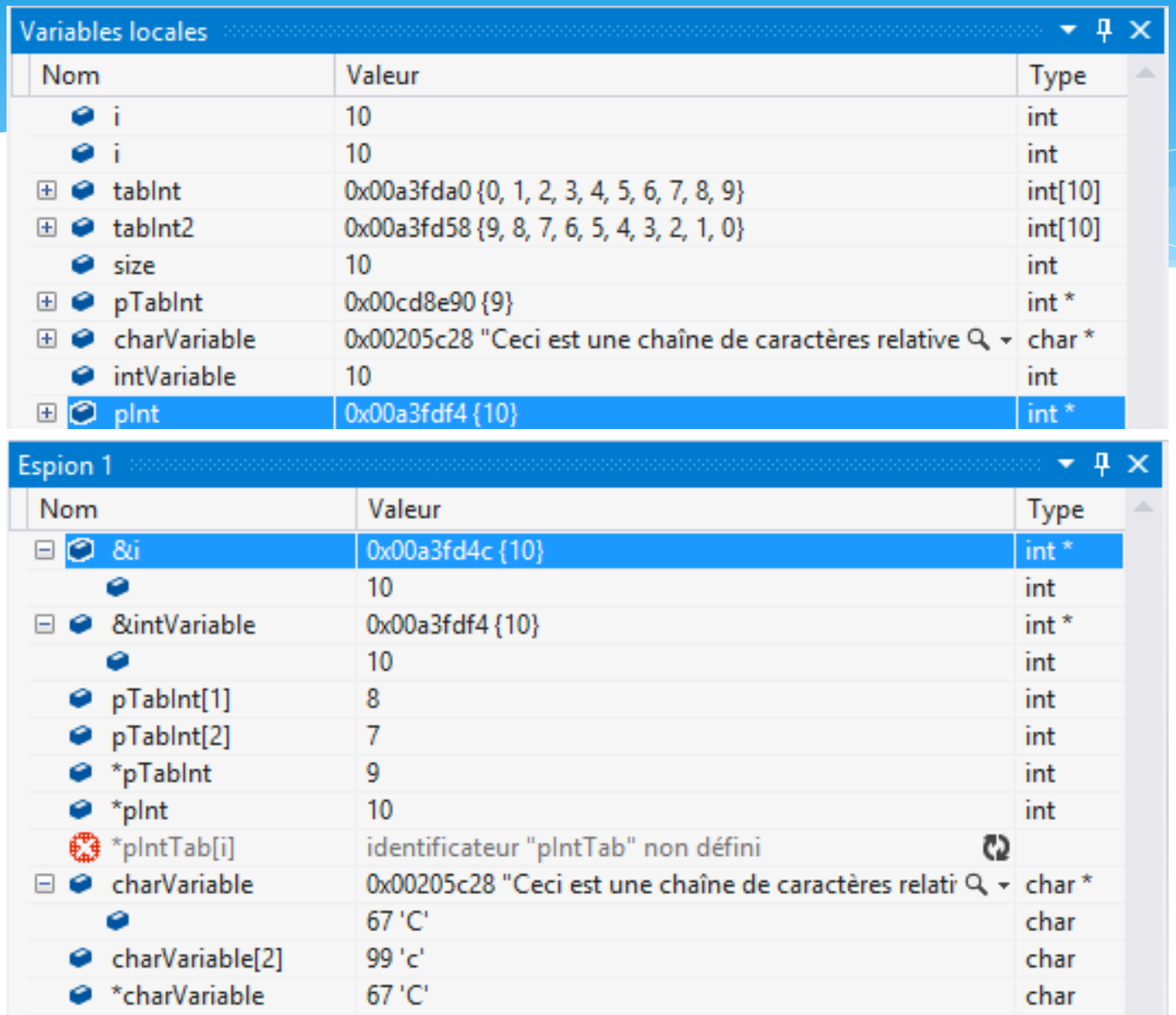
```
int tabInt[10] = {0,1,2,3,4,5,6,7,8,9}; // initialisation d'un tableau
int * pTabInt; // un pointeur simple
pTabInt = tabInt; // pointe sur le tableau
for(int i = 0; i < 10; i++)
{
        printf("%d",pTabInt[i]);
}
```

5. Tableaux et pointeurs

* Un tableau de pointeurs dynamiques, avec affectation de valeurs :

```
size = 10;
int tabInt2[10] = {9,8,7,6,5,4,3,2,1,0};
pTabInt = (int *) malloc (size * sizeof(int));
for (int i = 0; i < 10; i++)
{
        pTabInt[i] = tabInt2[i]; //affectation
        printf("%d",pTabInt[i]);
}
```

6. Etat mémoire



Variables locales

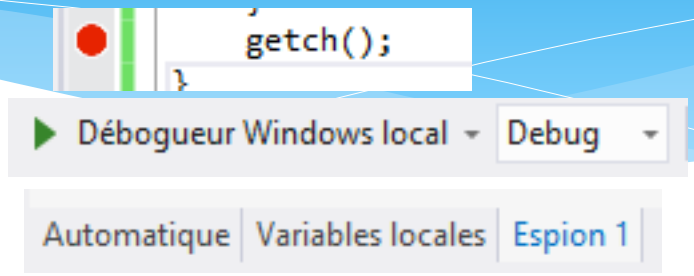
Nom	Valeur	Type
i	10	int
i	10	int
tabInt	0x00a3fda0 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}	int[10]
tabInt2	0x00a3fd58 {9, 8, 7, 6, 5, 4, 3, 2, 1, 0}	int[10]
size	10	int
pTabInt	0x00cd8e90 {9}	int *
charVariable	0x00205c28 "Ceci est une chaîne de caractères relative	char *
intVariable	10	int
pInt	0x00a3fdf4 {10}	int *

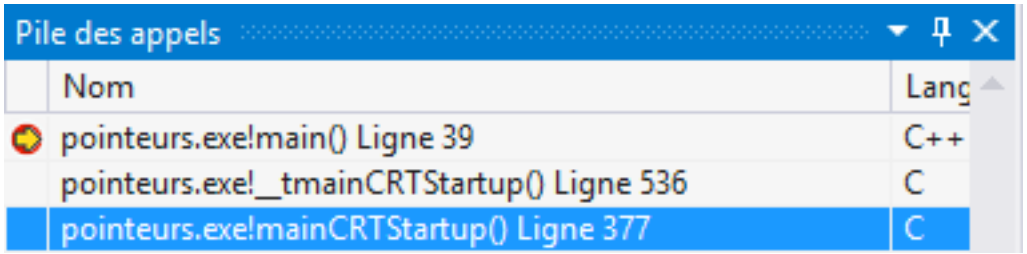
Espion 1

Nom	Valeur	Type
&i	0x00a3fd4c {10}	int *
	10	int
&intVariable	0x00a3fdf4 {10}	int *
	10	int
pTabInt[1]	8	int
pTabInt[2]	7	int
*pTabInt	9	int
*pInt	10	int
*pIntTab[i]	identificateur "pIntTab" non défini	
charVariable	0x00205c28 "Ceci est une chaîne de caractères relati	char *
	67 'C'	char
charVariable[2]	99 'c'	char
*charVariable	67 'C'	char

7. Utilité du debugger

* Poser des points d'arrêts :
* Lancer le programme :
* A chaque point d'arrêt :
* La pile des appels (sous-programmes) :





* F5 : point d'arrêt suivant
* F10 : pas à pas détaillé
* SHIFT F5 : arrêt

8. Transtypage (CAST)

* Un transtypage :

```
float reel;
reel = (float)intVariable; // transtypage
printf(" %f ",reel);
```

* Pas de transtypage :

```
float * pReel;
pReel = &reel; // pointage
printf(" %f ",pReel);
```

9. Procédures et références

* Les références : permutation

```
void permute(int &x, int &y) // travail sur l'adresse
{
    int z;
    z = x;
    X = y;
    Y = z;
}
```

* Sans la référence le travail se fait sur une copie uniquement valable dans la portée de la procédure.

9. Procédures et pointeurs

* Les pointeurs : permutation

```
void permute(int *x, int *y)
{
        int z=*x;
        *x=*y;
        *y=z;
}
```

9. Procédures qui retournent un tableau

* Une procédure peut retourner un tableau :

```
int *tab;
tab = (int *) malloc(10 * sizeof(int)); // tableau vide !
renvoiTab(tab); // tableau plein
```

* Procédure :

```
void renvoiTab(int * tab)
{
        for (int i =0 ; i<10; i++)
        {
                tab[i] = i; // remplissage
        }
}
```

10. Fonctions et pointeurs

* Fonction qui renvoie un tableau de pointeur :

```
int *tabR;
tabR = (int *) malloc(10 * sizeof(int)); // tableau vide
tabR = renvoiTab();
```

* Fonction :

```
int * renvoiTab()
{
        int *tab;
        tab = (int *) malloc(10 * sizeof(int)); // tableau vide
        for (int i =0 ; i<10; i++)
        {
                tab[i] = i; // remplissage
        }
        return tab;
}
```

* La signature sur les paramètres permet de faire la différence entre la procédure et la fonction. Le type de retour ne constitue pas une signature.

10. Fonctions et références

* Fonction qui renvoie une référence :

```
char & chaine = renvoiChaine();
printf("%s",&chaine);
getch();
```

* Fonction :

```
char & renvoiChaine()
{
        char * chaine = "La réponse est une chaîne";
        char & ref =*chaine;
        return ref;
}
```

11. Allocation en C

* Remarque : la fonction calloc est identique à malloc, mais elle fait une initialisation des emplacements.

```
int *tabR;
tabR = (int *) calloc(10,10 * sizeof(int)); // tableau vide
```

* La fonction realloc, permet d'ajouter des éléments à un tableau.

```
tabR = (int *) realloc(tabR,11 * sizeof(int)); // tableau avec une case en plus soit 11.
tabR[10] = 88; // fonctionne
for (int i = 0; i< 11; i++)
{
        printf("%d",tabR[i]);
}
```

11. Allocation en C

* Voyons le problème de la réallocation diminuée :

* Realloc ne permet pas de diminuer le tableau.
* Il faut donc le sauvegarder.
* Le libérer avec free ou delete
* Le réallouer avec la nouvelle longueur

11. Allocation en C++

* new et delete sont utilisables à travers Visual Studio, puisque le mélange C/C++ est assumé.

```
char * tabChar;
tabChar = new char[256];
tabChar[1] = 'n';
printf("%c",tabChar[1]);
delete tabChar;
```

* Cela peut choquer les puristes !!

12. Pointeur sur fonction

* Un pointeur peut contenir l'adresse d'une fonction :

```
void (* pFonction)(int *, int *); // permute
void (* pFonction2)(int *tab); // renvoiTab
pFonction = &permute;
//printf("%d",(*pFonction)(a,b)); si la fonction avait renvoyé un résultat
pFonction(a,b);
pFonction2 = &renvoiTab;
pFonction2(tab);
```

13. Tableaux à 2 dimensions dynamiques

```
int ** tabDeuxDim;
  tabDeuxDim = (int **) calloc(10, 10 * sizeof(int));
  //tabDeuxDim = new  int*[10]; // équivalent
  for (int i = 0; i < 10; i++)
  {
              tabDeuxDim[i] = (int *) calloc(10, 10 * sizeof(int));
               //tabDeuxDim[i] = new int[10]; // equivalent
  }
  for (int i = 0; i< 10; i++)
  {
               for (int j = 0; j <10; j++)
               {
                           tabDeuxDim[i][j] = i*j;
                           printf("%d %d = %d \n",i,j,tabDeuxDim[i][j]);
               }
  }
```

14. Pointeur et liste

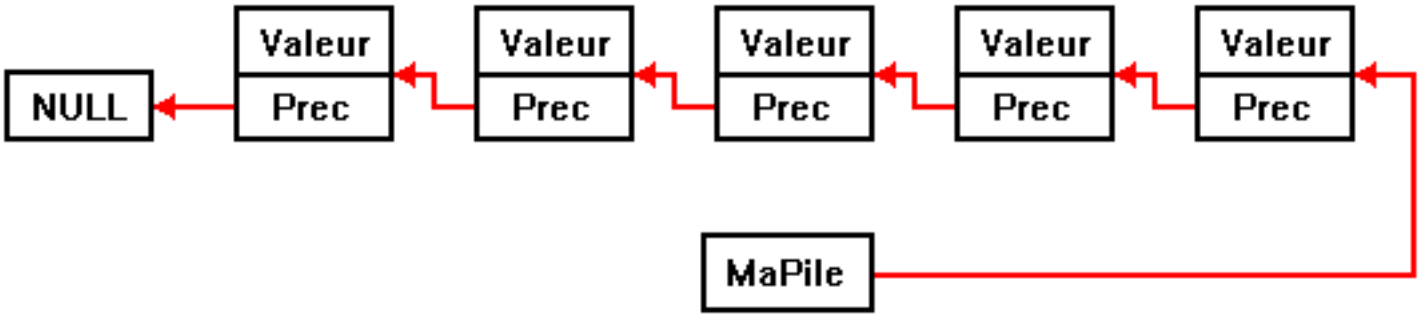
* Les listes chaînées mettent à profit les pointeurs :

  * Liste chaînée dans un sens
  * Liste chaînée dans les deux sens

  * Opération d'insertion
  * Opération de suppression
  * Opération de parcours

14. Pointeur et liste

* Liste chaînée simple :



14. Pointeur et liste

* Liste chaînée double