Filière IGE – 2017/2018
Algorithmique & Programmation
Leçon 6 : Pointeurs

Prof. *A. DARGHAM*

Pointeurs

- **Notion de pointeur**
  - Supposons que **var** est une variable d'un certain type **T**.
  - Le compilateur va automatiquement lui allouer de l'espace mémoire de taille **sizeof**(**T**) = **sizeof**(**var**).
  - L'expression **&var** représente **l'adresse de la variable var**.
  - Maintenant, **affectons l'adresse** de **var** à une **autre variable**, disons à **pvar** :

    pvar = &var;

Pointeurs

- **Notion de pointeur**
  - Cette nouvelle variable constitue un **pointeur** sur **var**, car elle **indique l'adresse de l'emplacement mémoire** dans laquelle la variable **var** a été rangée en mémoire.
  - Le schéma suivant illustre la relation qui lie les deux variables **var** et **pvar** :

Pointeurs

- **Définition :**
  - Un **pointeur** est une variable dont la valeur est une **adresse mémoire**.
- **Déclaration :**
  - Un **pointeur se déclare comme une variable ordinaire**, mais avec **une petite modification** :

    On doit ajouter le symbole '*' **entre** le **type** et **l'identificateur** de la variable.

Pointeurs

- **Déclaration :**
  - Par exemple, si **var** est une variable de type **int** et **pvar** est un **pointeur sur var**, alors on écrit :

```
int var;
int * pvar;
/* pvar est un pointeur sur un int */
pvar = &var;
/* pvar reçoit l'adresse de var : on dit alors
   que pvar pointe sur var */
```

Pointeurs

- **Déclaration :**
  - Lorsque « **pvar** » **pointe sur** « **var** », alors :
    - La **valeur** de « **pvar** » est **l'adresse** de « **var** », c'est-à-dire que :

      pvar == &var
    - L'expression « ***pvar** » **représente la valeur** de « **var** », c'est-à-dire que :

      ***pvar == var**

pvar == &var ⇔ *pvar == var

Relation fondamentale entre variable pointée et variable pointant

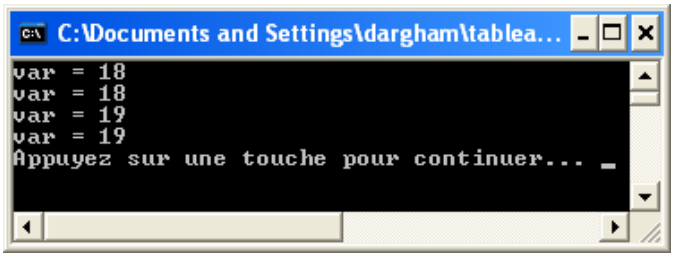
Pointeurs

- La **relation fondamentale** signifie que si une variable « **var**» **est pointée** par un **pointeur** « **pvar** », alors on peut **manipuler la variable pointée** « **var** » **indirectement** en utilisant l'expression « ***pvar** ».

Pointeurs

```
#include <stdio.h>
main()
{
   int var = 18;
   int * pvar;
   pvar = &var;
   printf("var = %d\n", var);
   printf("var = %d\n", *pvar);
   *pvar = 19;
   printf("var = %d\n", var);
   printf("var = %d\n", *pvar);
   system("pause");
}
```

Pointeurs



Pointeurs

- **Déclaration :**
  - On peut **pointer** vers une **variable de n'importe quel type**.
  - Ainsi, on peut avoir :
    - Un **pointeur** sur un « **int** »
    - Un **pointeur** sur un « **char** »
    - Un **pointeur** sur un « **float** »
    - Un **pointeur** sur un « **double** ».

Pointeurs

- **Exemples :**

```
int * p1;
/* p1 est un pointeur sur un int */
char * p2;
/* p2 est un pointeur sur un char */
float * p3;
/* p3 est un pointeur sur un float */
double * p4;
/* p4 est un pointeur sur un double */
```

Pointeurs

- **Déclaration :**
  - On peut également :
    - **Pointer sur un pointeur**
    - **Pointer sur un pointeur pointant sur un pointeur**
    - …etc.

Pointeurs

- **Exemples :**

```
int ** p5;
/* p5 est un pointeur sur un pointeur
   sur un int */
float *** p6;
/* p6 est un pointeur sur pointeur
   pointant sur un pointeur pointant sur
   un float */
```

Pointeurs

- **Taille d'un pointeur :**
  - **Tous les pointeurs ont la même taille**, que ce soit un **pointeur** sur un « **char** », un **pointeur** sur un « **int** », …etc.
  - Nous avons alors la relation suivante :

    sizeof(**char***) = **sizeof**(**int***) = **sizeof**(**float***) = **sizeof**(**double***)
  - En effet, un **pointeur stocke** toujours **une adresse**.

Pointeurs

```
#include <stdio.h>
main()
{
   printf("Taille d'un pointeur sur char : %d\n",
          sizeof(char*));
   printf("Taille d'un pointeur sur int : %d\n",
          sizeof(int*));
   printf("Taille d'un pointeur sur float : %d\n",
          sizeof(float*));
   printf("Taille d'un pointeur sur double : %d\n",
          sizeof(double*));
   system("pause");
}
```

Pointeurs

Pointeurs

- **Valeur NULL :**
  - Lorsqu'on veut "**dépointer**" un **pointeur**, c'est-à-dire **lui rendre « libre »,** on doit lui affecter une **valeur spéciale** : la valeur **NULL**.
  - Exemple :

```
int var = 7;
int * pvar = &var; /* pvar pointe sur var */
pvar = NULL;
/* pvar ne pointe vers aucune variable */
*pvar = 11;
/* Erreur à l'exécution !!! */
```

Pointeurs

- **Arithmétique des pointeurs :**
  - Avec les pointeurs, il est possible de réaliser les opérations suivantes :
    - **Ajouter un entier** à un **pointeur**.
    - **Retrancher un entier** à un **pointeur**.
    - **Incrémenter** un **pointeur**.
    - **Décrémenter** un **pointeur**.
    - **Soustraire** deux **pointeurs**.
    - **Comparer** deux **pointeurs**.

Pointeurs

- **Ajouter un entier à un pointeur :**
  - Soient **p** est un **pointeur sur un type T**, et **k** un **nombre entier**. L'expression :

    p + k

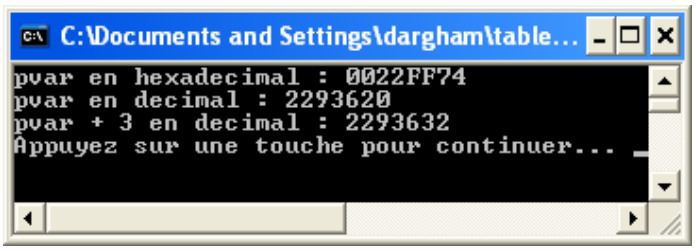
    désigne **l'adresse du k^{ème} objet de type T situé juste après l'objet de type T pointé** par **p**.
  - Cela signifie que si **x** est la valeur de **p** (c-à-d si l'adresse de l'objet pointé par **p** est égale à **x**), alors la valeur de l'expression **p + k** est égale à :

    x + k × sizeof(T)

Pointeurs

```
#include <stdio.h>
main()
{
   int var = 7;
   int * pvar;

   pvar = &var;
   printf("pvar en hexadecimal : %p\n", pvar);
   printf("pvar en decimal : %d\n", pvar);
   printf("pvar + 3 en decimal : %d\n", pvar + 3);
   system("pause");
}
```

Pointeurs



Pointeurs

- **Retrancher un entier à un pointeur :**
  - Soient **p** est un **pointeur sur un type T**, et **k** un **nombre entier**. L'expression :

    p - k

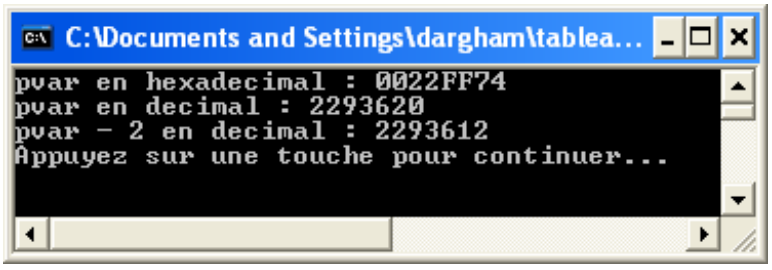
    désigne **l'adresse du k^{ème} objet de type T situé juste avant l'objet de type T pointé** par **p**.
  - Cela signifie que si **x** est la valeur de **p**, alors la valeur de l'expression **p - k** est égale à :

    x - k × sizeof(T)

Pointeurs

```c
#include <stdio.h>
main()
{
   int var = 7;
   int * pvar;
   pvar = &var;

   printf("pvar en hexadecimal : %p\n", pvar);
   printf("pvar en decimal : %d\n", pvar);
   printf("pvar - 2 en decimal : %d\n", pvar - 2);
   system("pause");
}
```

Pointeurs



Pointeurs

- **Incrémenter un pointeur :**
  - Soient **p** est un **pointeur sur un type T**.
  - Les expressions :

    p++ et ++p

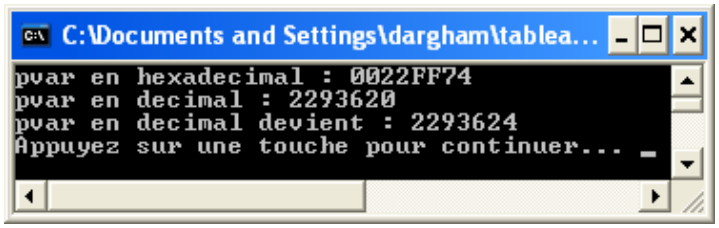
    désignent toutes les deux **l'adresse de l'objet de type T situé juste après l'objet de type T pointé** par **p**. (**p++ et ++p ⇔ p = p + 1**)
  - Cela signifie que si **x** est la valeur de **p**, alors la valeur de l'expression **p++** e **++p** st égale à :

    x + sizeof(T)

Pointeurs

```
#include <stdio.h>
main()
{
   int var = 7;
   int * pvar;
   pvar = &var;

   printf("pvar en hexadecimal : %p\n", pvar);
   printf("pvar en decimal : %d\n", pvar);
   pvar++;
   printf("pvar en decimal devient : %d\n", pvar);
   system("pause");
}
```

Pointeurs



Pointeurs

- **Décrémenter un pointeur :**
  - Soient **p** est un **pointeur sur un type T**.
  - Les expressions :

    p-- et --p

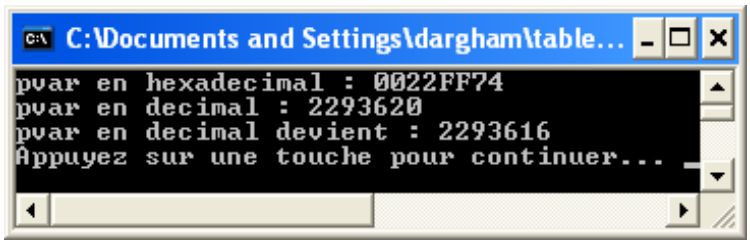
    désignent toutes les deux **l'adresse de l'objet de type T situé juste avant l'objet de type T pointé** par **p**. (**p-- et --p ⇔ p = p - 1**)
  - Cela signifie que si **x** est la valeur de **p**, alors la valeur de l'expression **p--** e **--p** st égale à :

    x - sizeof(T)

Pointeurs

```c
#include <stdio.h>
main()
{
   int var = 7;
   int * pvar;
   pvar = &var;

   printf("pvar en hexadecimal : %p\n", pvar);
   printf("pvar en decimal : %d\n", pvar);
   --pvar;
   printf("pvar en decimal devient : %d\n", pvar);
   system("pause");
}
```

Pointeurs



Pointeurs

- **Soustraire deux pointeurs :**
  - Soient **p** et **q** deux **pointeurs sur un type T**.
  - L'expression :

    p - q

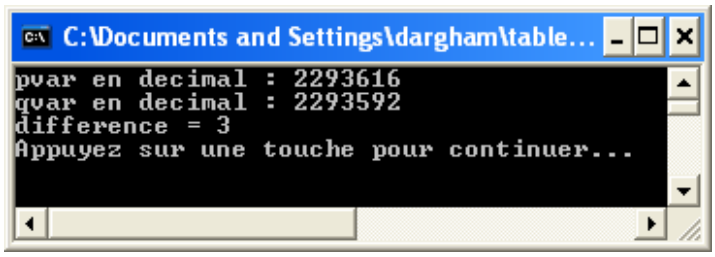
    représente le **nombre d'objets de type T situés entre l'objet pointé** par **p** et **celui pointé** par **q**.
  - Cela signifie que si **x** est la valeur de **p** et si **y** est la valeur de **q**, alors la valeur de l'expression **p - q** est égale à :

    (x - y) / sizeof(T)

Pointeurs

```
#include <stdio.h>
main()
{
   double var = 2.75;
   double * pvar;
   double * qvar;
   pvar = &var;
   qvar = &var - 3;
   printf("pvar en decimal : %d\n", pvar);
   printf("qvar en decimal : %d\n", pvar);
   printf("difference = %d\n", pvar - qvar);

   system("pause");
}
```

Pointeurs



Pointeurs

- **Comparer deux pointeurs :**
  - Soient **p** et **q** deux **pointeurs sur un type T**.
  - On peut comparer les valeurs de **p** et **q** en écrivant les expressions :

p > q
p >= q
p < q
p <= q
p == q
p != q

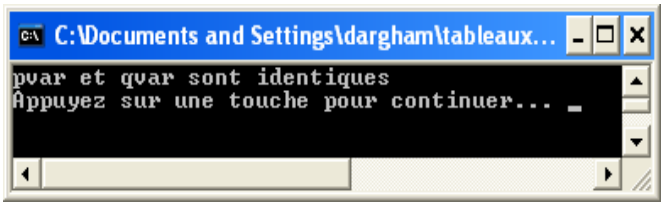
Pointeurs

- **Comparer deux pointeurs :**
  - Ces expressions sont interprétées comme des comparaisons entre des valeurs entières (**les adresses**).
  - Par exemple, **p == q** sera vraie, si **p** et **q**, ont des valeurs identiques, c'est-à-dire s'ils **pointent sur le même objet de type T**.

Pointeurs

```
#include <stdio.h>
main()
{
   double var = 2.75;
   double * pvar;
   double * qvar;

   pvar = &var;
   qvar = &var;
   if(pvar == qvar)
     printf("pvar et qvar sont identiques\n");
   else
     printf("pvar et qvar sont differents\n");
   system("pause");
}
```

Pointeurs



Pointeurs

- **Pointeurs et tableaux :**
  - Les **pointeurs** et les **tableaux** en C sont très liés entre eux.
  - En fait, **le nom d'un tableau** est **l'adresse de son premier élément** : c'est un **pointeur constant**.
  - De même, un **pointeur** peut être vu **comme un tableau** (**non constant !!**).

Pointeurs

- **Pointeurs et tableaux :**
  - Soient **A** un **tableau de type T** et **P** un **pointeur sur le type T**.

```
#define  N  10

T A[N];

T * P;
```

Pointeurs

- **Pointeurs et tableaux :**
  - Les formules de la **relation universelle tableau / Pointeur** sont :

A[i] ≡ *(A + i) **&A[i] ≡ (A + i)**

***(P + i) ≡ P[i]** **(P + i) ≡ &P[i]**

« i » étant un entier.

Pointeurs

- **Gestion dynamique de la mémoire :**
  - On veut calculer la **somme** de **N variables entières**, où la **valeur de N n'est pas connue à l'avance, mais à l'exécution**.
  - La résolution de ce problème nécessite une **technique fondamentale de la programmation** : la **gestion dynamique de la mémoire**.

Pointeurs

- **Gestion dynamique de la mémoire :**
  - Une première solution (**inefficace**) à ce problème, consiste à **préciser une valeur maximale** de **N**, N_{max} et **d'utiliser un tableau de longueur N** pour **stocker les N variables**.
  - Il y a deux inconvénients à cette solution :
    - **Il ne faut pas dépasser** la valeur N_{max}.
    - Si la **valeur effective** de **N** est **trop petite à** N_{max}, il y **a vraiment un grand gaspillage de la mémoire**.

Pointeurs

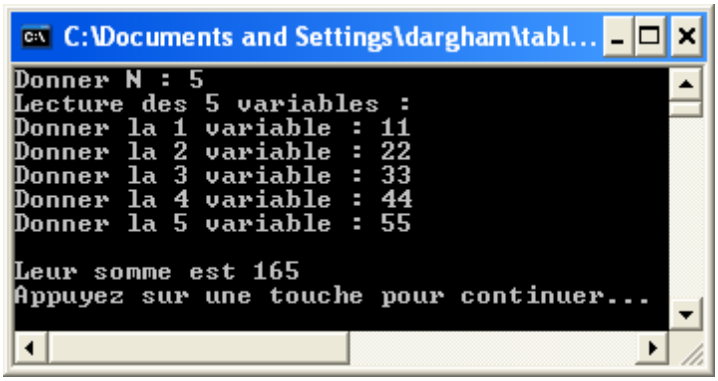
```
#include <stdio.h>
#define N_MAX 20
main()
{
   int N, s = 0, i;
   int T[N_MAX];

   printf("Donner N : ");
   scanf("%d", &N);
   if(N > NMAX || N <= 0)
     printf("Desole, probleme impossible\n");
```

Pointeurs

```
   else
   {
      printf("Lecture des %d variables :\n", N);
      for(i = 0; i < N; i++)
      {
         printf("Donner une variable : ");
         scanf("%d", &T[i]);
         s += T[i];
      }
      printf("\nLeur somme est %d\n", s);
   }
   system("pause");
}
```

Pointeurs

Pointeurs



Pointeurs

- **Gestion dynamique de la mémoire :**
  - Une seconde solution (**efficace**) à ce problème, consiste à **utiliser la technique de la gestion dynamique de la mémoire**.
  - On a besoin de **N** variables, mais la valeur de **N n'est pas disponible au moment de la compilation** (on va la lire au cours de l'exécution du programme).
  - Il **faut donc utiliser** de la **mémoire dynamique**.
  - Celle-ci se trouve dans une **zone appelée le tas** (**Heap** en anglais).

Pointeurs

- **Gestion dynamique de la mémoire :**
  - La fonction **malloc** de la bibliothèque <**stdlib.h**> **permet d'allouer** de la **mémoire dynamique**.
  - Cette fonction, lorsqu'elle est appelée, **alloue une zone mémoire** (**contiguë**) **de taille spécifiée comme argument de cette fonction**.
  - Si elle **réussie**, elle **retournera un pointeur sur le premier octet de la zone allouée**.
  - Si elle **échoue**, elle **retournera le pointeur NULL**.

Pointeurs

- **Gestion dynamique de la mémoire :**
  - La **zone allouée** par la fonction **malloc est sans format**.
  - Elle faut donc la **formater** en **spécifiant le type de pointeur désiré** (par une **opération de conversion de type**).
  - Comme on vient de le savoir, le **résultat retourné** par **malloc** est **un pointeur**.
  - On peut donc **l'affecter** à **un pointeur de même type**.

Pointeurs

- **Gestion dynamique de la mémoire :**
  - Par exemple, voilà ce qu'il faut faire pour allouer de la **mémoire dynamique** pour 4 entiers (**int**) :
  - On déclare un pointeur sur un **int** :

```
int * ptr;
```

  - On appelle la fonction **malloc** pour allouer 4 **int** (donc 4 x **sizeof**(**int**)) et l'on **affecte son résultat**, **après formatage** (vers **int***) à `ptr` :

```
ptr = (int*) malloc(4 * sizeof(int));
```

Pointeurs

- **Gestion dynamique de la mémoire :**
  - En fait, nous avons crée un **tableau dynamique**, nommé `ptr` : `ptr[0], ptr[1], ptr[2], ptr[3]`.
  - Nous pouvons alors résoudre le problème posé précédemment en utilisant la **gestion dynamique de la mémoire**, et plus exactement, en appelant la fonction **malloc**.

Pointeurs

```
#include <stdio.h>
#include <stdlib.h>
main()
{
   int N, s = 0, i;
   int * T;  /* tableau dynamique */

   printf("Donner N : ");
   scanf("%d", &N);
   if(N <= 0)
     printf("Desole, probleme impossible\n");
```

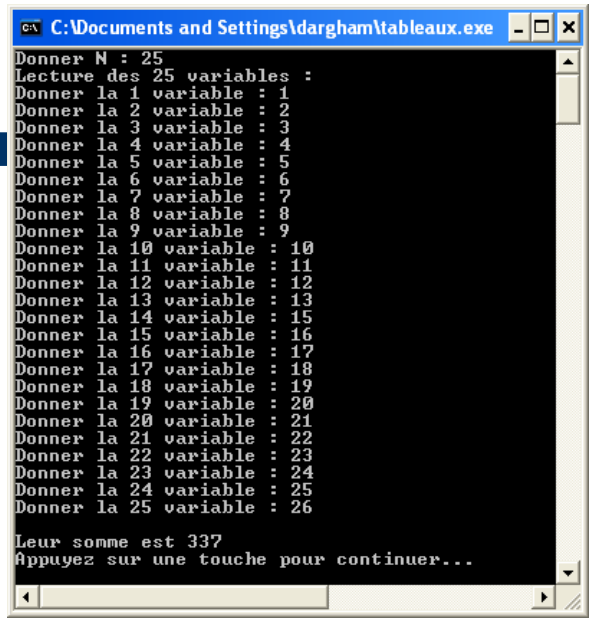
Pointeurs

```
    else
    {
       T = (int*) malloc(N * sizeof(int));
       if(T == NULL)
         printf("Erreur d'allocation de la memoire
                 dynamique\n");
       else
       {
        printf("Lecture des %d variables :\n", N);
        for(i = 0; i < N; i++) {
          printf("Donner une variable : ");
          scanf("%d", &T[i]);
          s += T[i];
        }
```

Pointeurs

```
          printf("\nLeur somme est %d\n", s);
       }
    }
    system("pause");
}
```

Pointeurs



Pointeurs

- **Gestion dynamique de la mémoire :**
  - La fonction **free** de la bibliothèque <**stdlib.h**> **permet de désallouer** (**libérer**) la **mémoire dynamique** déjà allouée par **malloc**.
  - La syntaxe est simple :

```
free(ptr);
```

  - Ici, `ptr` est le **nom du tableau dynamique** crée par **malloc**.