SCIENCES ET MONTAGNE
UNIVERSITÉ SAVOIE MONT BLANC

UNIVERSITÉ SAVOIE MONT BLANC

INFO 505 - Programmation C II
L3 – 2024-25

CM2 : Adressage, Portées, Pointeurs, Tableaux

J.Y. RAMEL

Bureau 2D- 204 - Polytech - Bourget du Lac

On se réveille...

- Portée des variables ?

  Faire un prg qui déclare une variable globale, une variable dans main() et une fonction d'affichage de ces variables *void affiche(void)*

- Fonctions et types de paramètres ?

  Ajouter une fonction *diviser_par_2(…)* avec des variantes : sans / avec paramètre, affichage du resultat dans la fonction / dans main(), avec / sans *return*

Portée des variables

Portée :

- Définie par l'emplacement de la déclaration de la variable au sein du code.
- Définie la portion du code source où une variable et son contenu sont reconnus.

Durée de vie d'une variable :

- Temps pendant lequel une variable a une existence en mémoire.
  - Création => allocation d'une zone mémoire
  - Destruction => libération de la zone mémoire

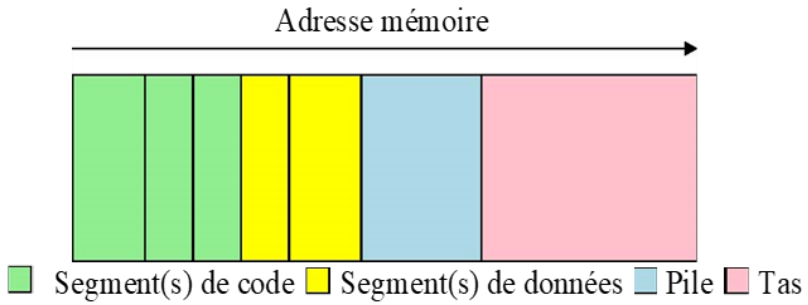
- Variable automatique : variable locale
  - Allocation en début de bloc et libération en sortie du bloc

- Variables statique : variable globale
  - Allocation en début de programme et libération en fin de programme

Portée des variables

Accès à une variable :

- Variable stockée en mémoire à une adresse @variable
- Accéder au contenu d'une variable -> accéder à la zone mémoire @variable
- Modifier le contenu de la zone mémoire @variable
- Différentes zones mémoire
  - Segment de données : stockage des variables globales
  - Pile (Stack) :
    - Stockage des variables locales
    - Stockage des paramètres de fonctions



Portée des variables

```c
#include <stdio.h>
#include <stdlib.h>

/**
 * @brief Illustrer la notion de portée d'une variable
 *   -> variable locale et globale
 */

int Ma_variable_globale = 100; // Variable globale

void affiche_var (void);

void affiche_var (void) {
    printf ("2. Ma_variable = %d\n", Ma_variable);
    printf ("3. Ma_variable_globale = %d\n", Ma_variable_globale);
}

int main(void) {
    int Ma_variable = 10; // Variable locale
                          // Portée limitée à la fonction main

    printf ("1. Ma_variable = %d\n", Ma_variable);
    affiche_var ();
    printf ("3. Ma_variable_globale = %d\n", Ma_variable_globale);
    return EXIT_SUCCESS;
}
```

Output View

Build | Search | Replace | References | Output | Terminal

```
C:/msys64/mingw64/bin/mingw32-make.exe -j12 -e -f  Makefile
----------Building project:[ PorteeVariable - Debug ]----------
mingw32-make[1]: Entering directory 'D:/Documents/ProgrammationLangageC/PorteeVariable'
C:/msys64/mingw64/bin/gcc.exe -c  "d:/Documents/ProgrammationLangageC/PorteeVariable/main.c" -gdwarf-2 -O0 -Wall  -o ../build-Debug/PorteeVariable/main.c.o -I. -I.
d:/Documents/ProgrammationLangageC/PorteeVariable/main.c: In function 'affiche_var':
d:/Documents/ProgrammationLangageC/PorteeVariable/main.c:14:38: error: 'Ma_variable' undeclared (first use in this function)
   14 |     printf ("2. Ma_variable = %d\n", Ma_variable);
      |                                      ^~~~~~~~~~~
d:/Documents/ProgrammationLangageC/PorteeVariable/main.c:14:38: note: each undeclared identifier is reported only once for each function it appears in
mingw32-make[1]: *** [PorteeVariable.mk:100: ../build-Debug/PorteeVariable/main.c.o] Error 1
mingw32-make: *** [Makefile:5: All] Error 2
mingw32-make[1]: Leaving directory 'D:/Documents/ProgrammationLangageC/PorteeVariable'
==== build ended with errors (1 errors, 1 warnings) ===
```

Fonctions et paramètres

Déclaration de fonctions

- Déclaration du prototype
  - Avant utilisation de la fonction, éventuellement dans le fichier en-tête .h
  - Spécifiant les paramètres → *type* Ma_fonction(liste des paramètres)
  - Spécifiant le *type* de la valeur renvoyée
  - Type *void* si rien n'est renvoyé et/ou pas de paramètre

- Codage de la fonction
  - *type* Ma_fonction(liste des paramètres)

```
{
        // Le code de la fct
}
```

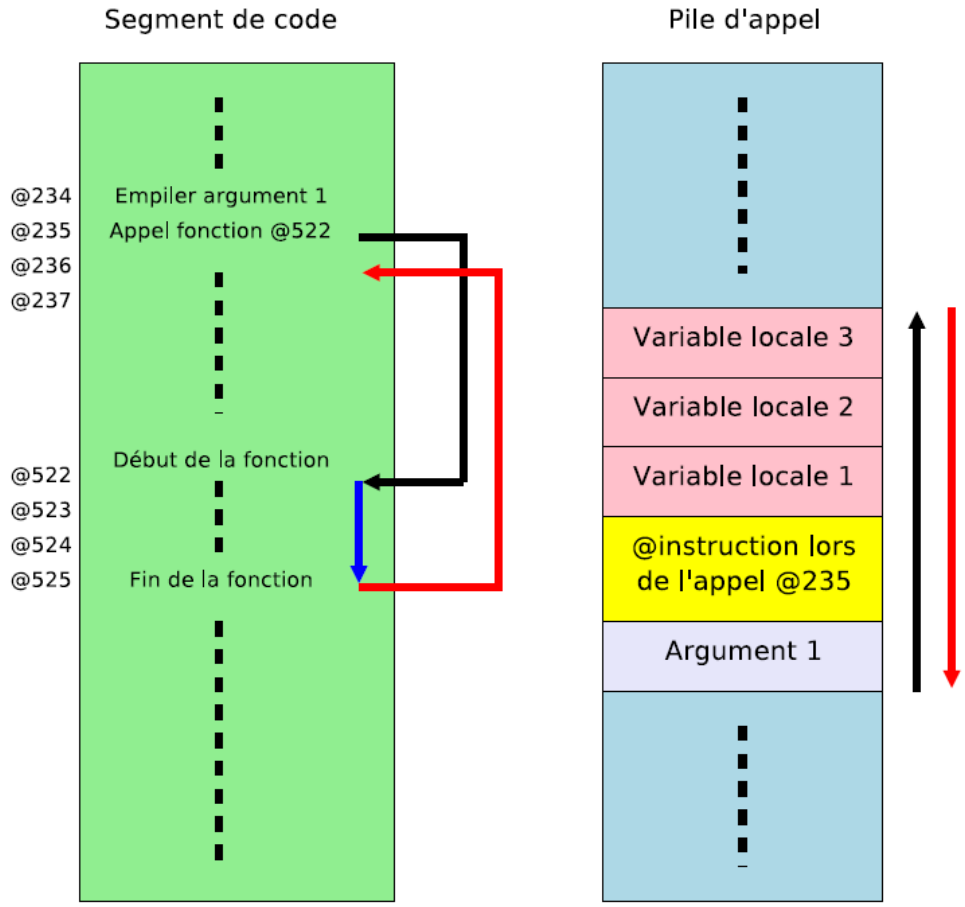
  - Si valeur renvoyée le code doit contenir
    → return(valeur renvoyée) /* de type *type* */

Fonctions et paramètres

Appel de fonction

- Appel : exécuter le code défini dans le corps de la fonction, avec la liste des paramètres

- Passage des paramètres : 2 modalités
  - Par valeur
  - Par adresse



Fonctions et paramètres

Passage par valeur

- La fonction manipule uniquement la valeur de la variable.
- Les paramètres transmis à l'appel de la fonction sont évalués ;
- Le résultat des expressions sont **copiés** dans la pile ;
- La fonction est appelée :

⇒ Recopie de la valeur des paramètres dans la pile.

⇒ La fonction manipule la copie des paramètres situés dans la pile

⇒ A la sortie de la fonction, l'espace mémoire dans la pile est restitué.

Fonction_PassageParValeur\main.c

```c
#include <stdio.h>
#include <stdlib.h>

/**
 * @brief Illustration appel de fonction
 * Passage de paramètres par valeur
 */

// Prototype de fonction
void divise_par_2 (int n);

/**
 * @brief divise par 2 une valeur entière
 * @fn divise_par_2
 * @param n entier
 */
void divise_par_2 (int n) {
    n /= 2;
    printf("valeur de n : %d\n", n);
}

int main(void)
{
    int y = 10;
    divise_par_2 (y);
    printf("valeur de y : %d\n", y);
    return EXIT_SUCCESS;
}
```

```
d:\Documents\ProgrammationLangageC\build-Debug\bin\Fonction_PassageParValeur.exe
valeur de n : 5
valeur de y : 10

==== Program exited with exit code: 0 ====
Time elapsed: 000:00.047 (MM:SS.MS)
Press any key to continue...
```

Fonctions et paramètres

Fonction_PassageParValeur\main.c

```c
#include <stdio.h>
#include <stdlib.h>

/**
 * @brief Illustration appel de fonction
 * Passage de paramètres par valeur
 */

// Prototype de fonction
void divise_par_2 (int n);

/**
 * @brief divise par 2 une valeur entière
 * @fn divise_par_2
 * @param n entier
 */
void divise_par_2 (int n) {
    n /= 2;
    printf("valeur de n : %d\n", n);
}

int main(void)
{
    int y = 10;
    divise_par_2 (y);
    printf("valeur de y : %d\n", y);
    return EXIT_SUCCESS;
}
```

n : paramètre de fonction → equivalent à 1 variable locale de la fct

y : variable locale à la fonction main

Output View

Build | Search | Replace | References | Output | Terminal

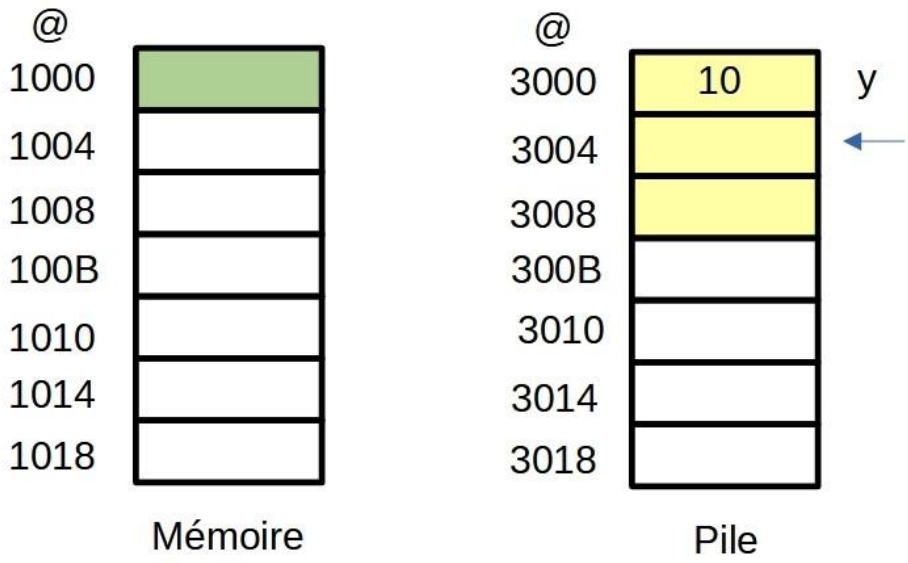
```
C:/msys64/mingw64/bin/mingw32-make.exe -j12 -e -f  Makefile
----------Building project:[ Fonction_PassageParValeur - Debug ]----------
mingw32-make[1]: Entering directory 'D:/Documents/ProgrammationLangageC/Fonction_PassageParValeur'
C:/msys64/mingw64/bin/gcc.exe -c  "d:/Documents/ProgrammationLangageC/Fonction_PassageParValeur/main.c" -gdwarf-2 -O0 -Wall  -o ../build-Debug/Fonction_PassageParValeur/m
C:/msys64/mingw64/bin/g++.exe -o ..\build-Debug\bin\Fonction_PassageParValeur.exe @../build-Debug/Fonction_PassageParValeur/ObjectsList.txt -L.
mingw32-make[1]: Leaving directory 'D:/Documents/ProgrammationLangageC/Fonction_PassageParValeur'
=== build completed successfully (0 errors, 0 warnings) ===
```

Fonction et passage de paramètres par valeur

Fonction_PassageParValeur\main.c

```c
#include <stdio.h>
#include <stdlib.h>

/**
 * @brief Illustration appel de fonction
 * Passage de paramètres par valeur
*/

// Prototype de fonction
void divise_par_2 (int n);

/**
 * @brief divise par 2 une valeur entière
 * @fn divise_par_2
 * @param n entier
*/
void divise_par_2 (int n) {
    n /= 2;
    printf("valeur de n : %d\n", n);
}

int main(void)
{
    int y = 10;
    divise_par_2 (y);
    printf("valeur de y : %d\n", y);
    return EXIT_SUCCESS;
}
```

1. Appel fonction main

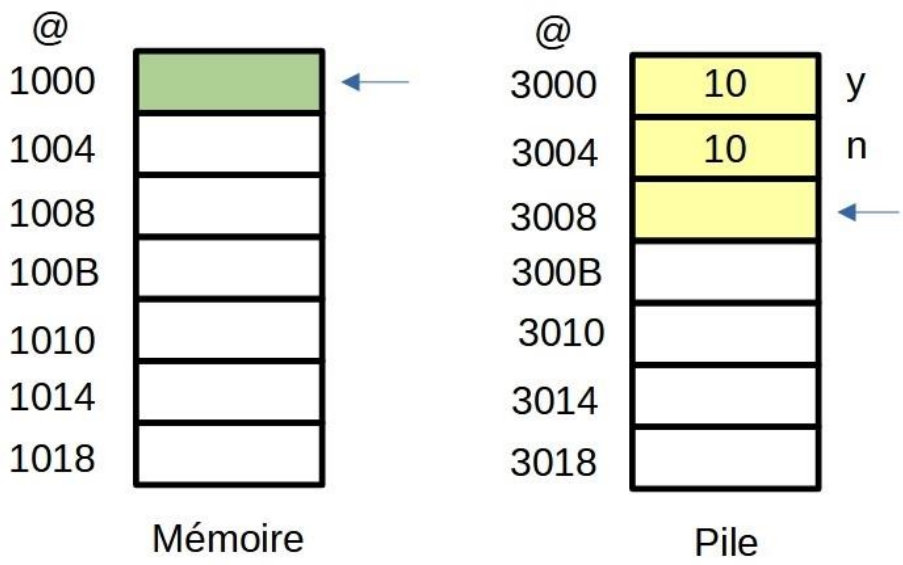


Fonction et passage de paramètres par valeur

Fonction_PassageParValeur\main.c

```c
#include <stdio.h>
#include <stdlib.h>

/**
 * @brief Illustration appel de fonction
 * Passage de paramètres par valeur
 */

// Prototype de fonction
void divise_par_2 (int n);

/**
 * @brief divise par 2 une valeur entière
 * @fn divise_par_2
 * @param n entier
 */
void divise_par_2 (int n) {
    n /= 2;
    printf("valeur de n : %d\n", n);
}

int main(void)
{
    int y = 10;
    divise_par_2 (y);
    printf("valeur de y : %d\n", y);
    return EXIT_SUCCESS;
}
```

2. Appel fonction divise_par_2



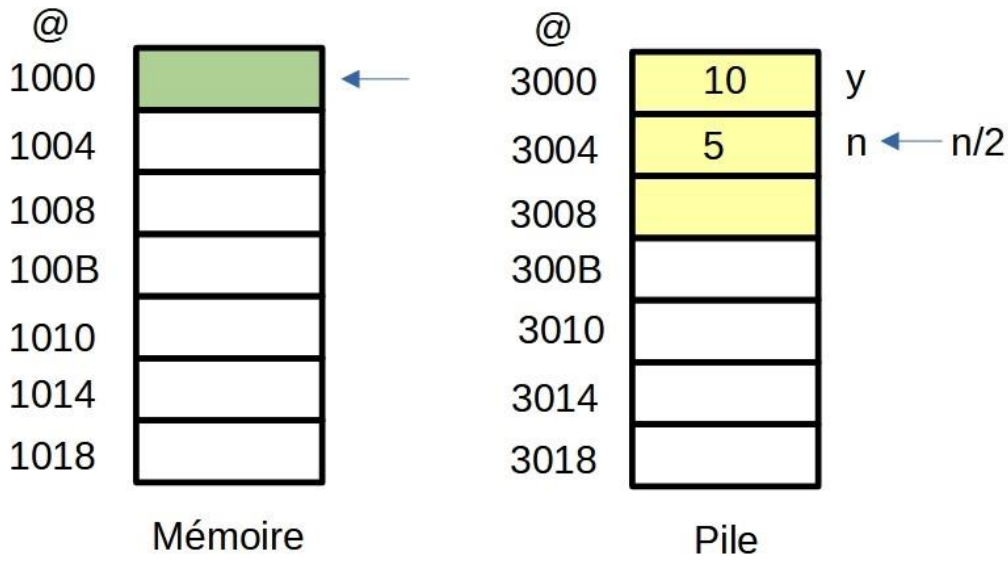
Copie du contenu de la variable y

Fonction et passage de paramètres par valeur

Fonction_PassageParValeur\main.c

```c
#include <stdio.h>
#include <stdlib.h>

/**
 * @brief Illustration appel de fonction
 * Passage de paramètres par valeur
*/

// Prototype de fonction
void divise_par_2 (int n);

/**
 * @brief divise par 2 une valeur entière
 * @fn divise_par_2
 * @param n entier
*/
void divise_par_2 (int n) {
    n /= 2;
    printf("valeur de n : %d\n", n);
}

int main(void)
{
    int y = 10;
    divise_par_2 (y);
    printf("valeur de y : %d\n", y);
    return EXIT_SUCCESS;
}
```

3. Exécution de n/=2;

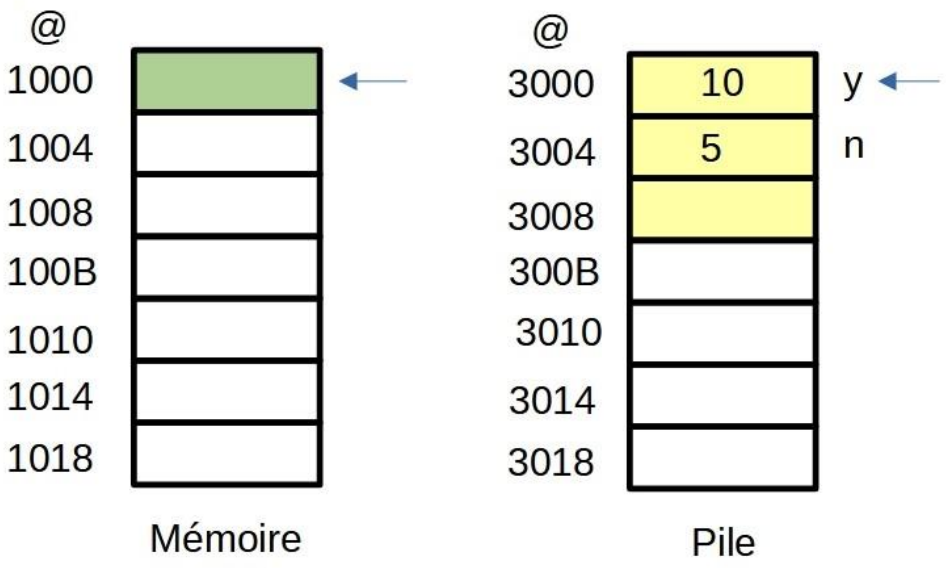


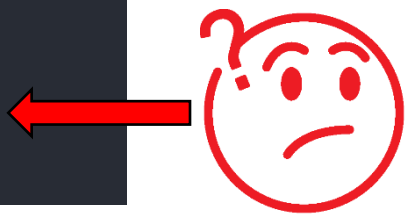
Fonction et passage de paramètres par valeur

Fonction_PassageParValeur\main.c

```c
#include <stdio.h>
#include <stdlib.h>

/**
 * @brief Illustration appel de fonction
 * Passage de paramètres par valeur
*/

// Prototype de fonction
void divise_par_2 (int n);

/**
 * @brief divise par 2 une valeur entière
 * @fn divise_par_2
 * @param n entier
*/
void divise_par_2 (int n) {
    n /= 2;
    printf("valeur de n : %d\n", n);
}

int main(void)
{
    int y = 10;
    divise_par_2 (y);
    printf("valeur de y : %d\n", y);
    return EXIT_SUCCESS;
}
```

4. Sortie de divise_par_2





Fonction et passage de paramètres par valeur

Fonction_PassageParValeur\main.c

```c
#include <stdio.h>
#include <stdlib.h>

/**
 * @brief Illustration appel de fonction
 * Passage de paramètres par valeur
 */

// Prototype de fonction
void divise_par_2 (int n);

/**
 * @brief divise par 2 une valeur entière
 * @fn divise_par_2
 * @param n entier
 */
void divise_par_2 (int n) {
    n /= 2;
    printf("valeur de n : %d\n", n);
}

int main(void)
{
    int y = 10;
    divise_par_2 (y);
    printf("valeur de y : %d\n", y);
    return EXIT_SUCCESS;
}
```

Passage par valeur :

- Recopie de la valeur du paramètre dans la pile.
- Pas de modification du contenu d'une variable passée par valeur en paramètre d'une fonction.

Passage par valeur :

- Si modification du contenu d'une variable passée en paramètre d'une fonction
  ⇒ passage de paramètre par adresse pour accéder à la zone mémoire de la variable.
  ⇒ Adresse : notion de pointeur

On se réveille...

- Les pointeurs…
- Que fait l'instruction → `int *p_int = 6;`
- Que fait l'instruction → `*p_int = 6;`

```
int i = 0;
int *p_int = &i;
*p_int =  6;
```

Pointeurs

Définitions

- **Pointeur** : variable qui contient l'adresse d'une autre variable stockée en mémoire.
- **Variable pointeur** : variable qui **contient** une **adresse**. L'adresse contenue permet de **faire référence** à une variable, une fonction ou tout autre type de donnée valide.

- Déclaration d'une variable pointeur : syntaxe

type * Id_Variable;

La variable déclarée contient une adresse ;
Le type défini lors de la déclaration permet de spécifier quel est le type de données auquel on accède

```
int * p_i;
float * p_f;
```

Pointeurs

Interprétation déclaration d'une variable pointeur

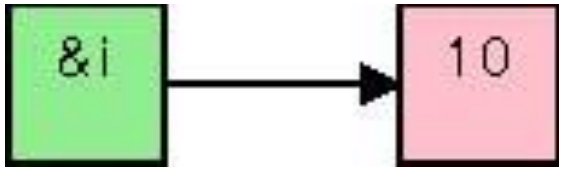
Exemple : int * p_int ;

L'Interprétation s'effectue de droite à gauche :

1. p_int est une variable ; *int* * p_int ;
2. p_int est une variable pointeur ; *int* * p_int ;
3. p_int est une variable pointeur sur un entier de type int ; *int* * p_int ;

Pointeurs

Opérateurs



Exemple : int * p, i, j ;

& : adresse de l'opérande ;
* : déréférencement / indirection / valeur de la zone pointée

```
int *p, i, j ;
i = 10 ;
p = &i ;
j = *p ;
```

p contient l'adresse à laquelle la variable i est stockée en mémoire
&i : adresse à laquelle la variable i est stockée
*p : valeur de l'entier stocké à l'adresse contenue dans la variable p ;

Pointeurs

Utilisation usuelle :

int i = 0;

int * p_i = & i ;

Erreur classique :

int i = 0 ;

int * p_i = num ;

Opérateur &

Operateur_&\main.c

```c
#include <stdlib.h>

int main(void)
{
    int i, *p_i = NULL;
    i = 10;
    p_i = i;
    *p_i = 20;

    return EXIT_SUCCESS;
}
```

Output View

Build | Search | Replace | References | Output | Terminal

```
C:/msys64/mingw64/bin/mingw32-make.exe -j12 -e -f  Makefile
=== build completed successfully (0 errors, 0 warnings) ===
----------Building project:[ Operateur_& - Debug ]----------
mingw32-make[1]: Entering directory 'D:/Documents/ProgrammationLangageC/Operateur_&'
C:/msys64/mingw64/bin/gcc.exe -c  "d:/Documents/ProgrammationLangageC/Operateur_&/main.c" -gdwarf-2 -O0 -Wall  -o ../build-Debug/Operateur_&/main.c.o -I. -I.
d:/Documents/ProgrammationLangageC/Operateur_&/main.c: In function 'main':
d:/Documents/ProgrammationLangageC/Operateur_&/main.c:7:9: warning: assignment to 'int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
    7 |     p_i = i;
      |         ^
'/main.c.o' n'est pas reconnu en tant que commande interne
ou externe, un programme excutable ou un fichier de commandes.
mingw32-make[1]: *** [Operateur_&.mk:100: ../build-Debug/Operateur_&/main.c.o] Error 1
mingw32-make: *** [Makefile:5: All] Error 2
mingw32-make[1]: Leaving directory 'D:/Documents/ProgrammationLangageC/Operateur_&'
=== build ended with warnings (0 errors, 1 warnings) ===
```

Compilation -> un avertissement (warning) voire une erreur, indiquant que l'on effectue une conversion du type *int* vers le type *int* *

Pointeurs

Opérateur *

L'opérateur unaire * permet de retourner la valeur contenue dans la zone mémoire dont l'adresse est stockée dans une variable pointeur.

int num = 10;
int * p_int = & num ;
printf ("%d \n ",* p_in t) ; permet d'afficher la valeur 10

Le même opérateur * peut être utilisé lors d'affectation : * p_int = 25 ;

Arithmétique des Pointeurs

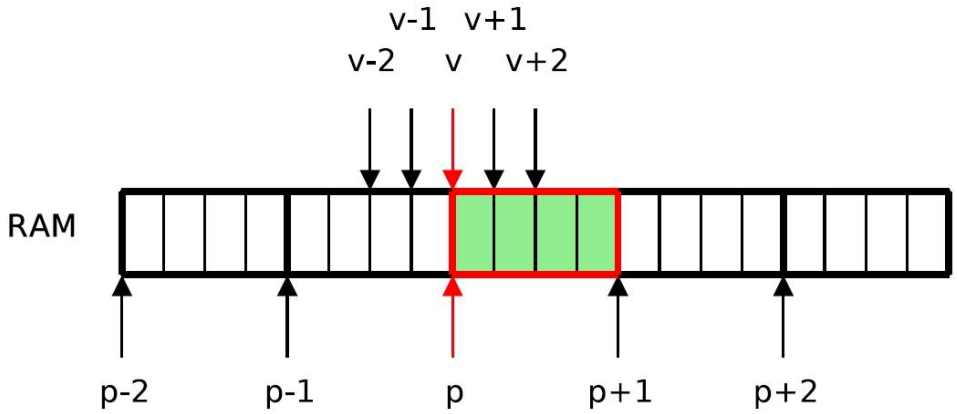
Opérateur arithmétiques sur pointeurs

Opérateurs valides : +, -, ++ et - -
L'effet d'un opérateur arithmétique sur un pointeur est liée au type de donnée pointée

Déclaration :
type * ptr ;
ptr ++ ; /* ptr += 1; ⇔ ptr = ptr + 1;*/

Résultat :
L'adresse contenue dans p est incrémenté de sizeof (type) octets ;
ptr pointe sur le prochain enregistrement de type *type* en mémoire.

Exemple :
int i ;
int * ptr_i = &i ;
char *v = (char *) ptr_i



Pointeurs

Opérateur arithmétiques

Arithmetique_pointeur\main.c

```c
 * @brief  Illustration arithmétique des pointeurs
 * pour accéder aux éléments d'un tableau 1D
 *
 */

#include <stdio.h>
#include <stdlib.h>

#define N_TAB 4

int main(void)
{
    int tab_int[N_TAB] = {2, 4, 8, 16};
    int *p_tab_int = &tab_int[0]; // équivalent à int *p_tab_int = tab_int

    for (int i = 0; i < N_TAB; i++) {
        printf("@ de tab_int[%d] : %p\n", i, (p_tab_int + i));
        printf("tab_int[%d] : %d\n", i, *(p_tab_int + i));
    }

    return EXIT_SUCCESS;
}
```

d:\Documents\ProgrammationLangageC\build-Debug\bin\Arithmetique_pointeur.exe

```
@ de tab_int[0] : 0000009530dffc10
tab_int[0] : 2
@ de tab_int[1] : 0000009530dffc14
tab_int[1] : 4
@ de tab_int[2] : 0000009530dffc18
tab_int[2] : 8
@ de tab_int[3] : 0000009530dffc1c
tab_int[3] : 16

==== Program exited with exit code: 0 ====
Time elapsed: 000:00.015 (MM:SS.MS)
Press any key to continue...
```

Output View

Build | Search | Replace | References | Output | Terminal

```
C:/msys64/mingw64/bin/mingw32-make.exe -j12 -e -f  Makefile
----------Building project:[ Arithmetique_pointeur - Debug ]----------
mingw32-make[1]: Entering directory 'D:/Documents/ProgrammationLangageC/Arithmetique_pointeur'
mingw32-make[1]: Leaving directory 'D:/Documents/ProgrammationLangageC/Arithmetique_pointeur'
=== build completed successfully (0 errors, 0 warnings) ===
```

Pointeurs

Pointeur de type void

Le **type** *void ** permet de définir des variables pointeur pour lesquelles le type de données adressée n'est pas spécifié.

void * ptr ;

Il est possible de définir par la suite le type des données adressées par un **cast (conversion explicite) :**

(float *) ptr → permet de spécifier que ptr adresse une donnée de type float → CF *malloc()*

Propriétés :

1. Un pointeur de type *void* possède la même représentation et le même alignement mémoire qu'un pointeur sur le type *char*.
2. Une variable pointeur de type *void* * ne peut pas être utilisé comme pointeur de fonction.
3. L'opérateur *sizeof* ne peut être utilisé que sur les variables de type *void* * (et donc des pointeurs), jamais sur le type *void*.

Pointeurs

Pointeur *NULL*

NULL : constante (valeur par défaut : 0)
NULL représente une valeur

*float *c = NULL; // Pour préciser que le pointeur n'adresse aucun objet de type float*
Le pointeur c ne pointe sur aucun objet
Le test
if (c == NULL) permet de savoir si la variable c contient une adresse valide

cf. Allocation dynamique ou gestion de liste ou gestion d'arbre

Passage de paramètres par adresse

Le paramètre passé est l'adresse de l'argument → le contenu de la variable peut être modifié au sein de la fonction, puisque l'on peut accéder à la zone mémoire

Un paramètre passé par adresse est déclaré à l'aide d'un pointeur dans la liste des arguments de la fonction.

Passage_Par_Adresse\main.c

```
 * Passage de paramètres par adresse
*/

// Prototype de fonction
void divise_par_2 (int *n);

/**
 * @brief divise par 2 une valeur entière
 * @fn divise_par_2
 * @param n entier
*/
void divise_par_2 (int *n) {
    *n /= 2;
    printf("valeur de n : %d\n", *n);
}

int main(void)
{
    int y = 10;
    divise_par_2 (&y);
    printf("valeur de y : %d\n", y);
    return EXIT_SUCCESS;
}
```

Pointeur

Adresse

```
d:\Documents\ProgrammationLangageC\build-Debug\bin\Passage_Par_Adresse.exe
valeur de n : 5
valeur de y : 5

==== Program exited with exit code: 0 ====
Time elapsed: 000:00.031 (MM:SS.MS)
Press any key to continue...
```

On se réveille...

- Les tableaux…
  Faire un prg qui déclare un tableau de 10 float et affiche l'espace mémoire occupé par ce tableau

- Initialisation d'un tableau ?
  Ajouter l'initialisation du contenu avec des 0.0
  Faire une fonction d'initialisation qui spécifie un pas et une valeur de début

Les tableaux de dimension 1

Définitions

- Tableau : ensemble d'emplacements mémoire contigus, portant le même nom, et contenant le même type de donnée.
- Elément d'un tableau : un élément particulier du tableau.

Tableau 1D : déclaration

type Id_tableau[nb_elements];

type : le type de données commun à tous les éléments du tableau
Id_tableau : nom du tableau (identificateur au sens du langage C)
nb_elements : nombre d'éléments du tableau (constante littérale ou symbolique de type entier)

Les tableaux de dimension 1

- Exemple : tableau de 10 entiers
  int tab_entier[10] ;

- Occupation d'un tableau en mémoire : nombre d'octets

type Id_tableau[nb_elements];
sizeof (type) * nb_elements
sizeof (Id_tableau)

Remarque : pour définir le nombre d'éléments du tableau, privilégier les constantes symboliques (directive *#define*)

Les tableaux de dimension 1

Occupation d'un tableau en mémoire : illustration

Occupation_Memoire_Tableau1D\main.c

```c
/**
 * @brief Illustration calcul de
 * l'occupation mémoire d'un tableau 1D
*/

#include <stdio.h>
#include <stdlib.h>

#define NB_ELMTS 10

int main(void)
{
    float tab_float[NB_ELMTS];

    printf ("Nombre d'octets pour stocker un reel de type float : %llu\n", sizeof (float));
    printf ("Nombre d'octets pour stocker un tableau de %d float : %llu\n", NB_ELMTS,sizeof (float) * NB_ELMTS);
    printf ("Nombre d'octets pour stocker un tableau de %d float : %llu\n", NB_ELMTS,sizeof (tab_float));
    return EXIT_SUCCESS;
}
```

```
d:\Documents\ProgrammationLangageC\build-Debug\bin\Occupation_Memoire_Tableau1D.exe
Nombre d'octets pour stocker un reel de type float : 4
Nombre d'octets pour stocker un tableau de 10 float : 40
Nombre d'octets pour stocker un tableau de 10 float : 40

==== Program exited with exit code: 0 ====
Time elapsed: 000:00.047 (MM:SS.MS)
Press any key to continue...
```

Les tableaux de dimension 1

Accès à un élément d'un tableau :

type Id_tableau[nb_elements];

Id_tableau[indice_elements] pour accéder à l'élément d'indice " indice_elements":
indice_elements :

- expression renvoyant une valeur entière
- ou bien constante entière
- valeur dans l'ensemble {0,…, nb_elements – 1}

Indice du premier élément : toujours 0 ;
Indice du dernier élément : nb_elements – 1
Pas de mécanisme de vérification sur la valeur de l'indice de l'élément d'un tableau
=> possibilité de sortir de la zone mémoire réservée au stockage du tableau !!!

Les tableaux de dimension 1

Initialisation des éléments d'un tableau :

```
int tab [10] ;
tab [2] = 64 ; /* le 3ème élément du tableau tab reçoit la valeur 64 */
```

Lors de la déclaration :

```
int tab [4] = {2, 4, 6, 8} ;              /* Contenu
        int tab [4] = {0} ;               /* Contenu
        int tab [] = {2, 4, 6, 8} ;       /* Contenu
        int tab [4] = {
           [0] = 2,
           [1] = 4,
           [2] = 6,
           [3] = 8} ;
```

Les tableaux de dimension 1

Initialisation des éléments d'un tableau :

Init_Tab_1d\main.c

```c
/**
 * @brief Illustration Initialisation des
 * éléments d'un tableau 1D
*/

#include <stdio.h>
#include <stdlib.h>

#define NB_ELMTS 4

int main(void)
{
    int tab_int[NB_ELMTS];
    float tab1_float[NB_ELMTS] = {1.1, 2.2, 3.3, 4.4};
    float tab2_float[] = {2.2, 4.4, 6.6, 8.8};
    double tab1_double[NB_ELMTS] = {0.0};
    double tab2_double[NB_ELMTS] = {5.5, 10.10};
    for (int i = 0; i < NB_ELMTS; i++){
        tab_int[i] = 3 * i;
    }
```

d:\Documents\ProgrammationLangageC\build-Debug\bin\Init_Tab_1D.exe

```
tab_int[0] = 0; tab_int[1] = 3; tab_int[2] = 6; tab_int[3] = 9;
tab1_float[0] = 1.100000; tab1_float[1] = 2.200000; tab1_float[2] = 3.300000; tab1_float[3] = 4.400000;
tab2_float[0] = 2.200000; tab2_float[1] = 4.400000; tab2_float[2] = 6.600000; tab2_float[3] = 8.800000;
tab1_double[0] = 0.000000; tab1_double[1] = 0.000000; tab1_double[2] = 0.000000; tab1_double[3] = 0.000000;
tab2_double[0] = 5.500000; tab2_double[1] = 10.100000; tab2_double[2] = 0.000000; tab2_double[3] = 0.000000;

==== Program exited with exit code: 0 ====
Time elapsed: 000:00.063 (MM:SS.MS)
Press any key to continue...
```

Pointeurs et tableau de dimension 1

Soit la déclaration d'un tableau : type *Id_tableau*[nb_elements];

Un tableau n'est pas un pointeur!
L'identificateur d'un tableau *Id_tableau* correspond au pointeur sur le premier élément du tableau de type *type*

char tab[4] = {2, 4, 6, 8} ;

Conséquence1 : tab ⇔ &tab[0]

tab[0] : valeur du premier élément du tableau tab
& tab[0] : adresse du premier élément du tableau tab
tab : de type *type* *

On se réveille...

- Pointeurs et tableaux ?
  tab , tab[i] , tab + i , *(tab + i)

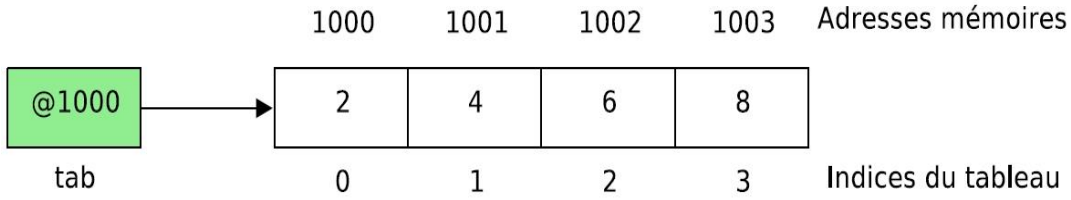
```
int t1[3];
int t2[3];
t1 = t2;
```

- Incorrect => un tableau ne peut pas être recopié avec = □ Un tableau n'est pas un pointeur!

- Tableau et paramètre de fonctions ?
  Quels paramètres pour une fonction d'initialisation d'un tableau

Pointeurs et tableau de dimension 1

char tab[4] = {2, 4, 6, 8} ;



Conséquence2 : *tab ⇔ tab[0]

tab : adresse du premier élément du tableau tab

* tab : valeur du premier élément du tableau, soit tab[0]

Accès à un élément d'un tableau 1D : arithmétique des pointeurs

tab[i] ⇔ *(tab + i)

tab : adresse du premier élément du tableau tab

tab + i : adresse de l'élément d'indice i du tableau tab

*(tab + i) : valeur de l'élément d'indice i du tableau tab (attention à la priorité des opérateurs + et * => **parenthèses obligatoires**!)

Fonction et passage de tableau 1D en paramètre

Comment manipuler un tableau de dimension 1 en paramètre d'une fonction ?

Préambule :

```
int t1[3];
int t2[3];
```

t1 = t2; /* Incorrect */ => un tableau ne peut pas être recopié
Un tableau n'est pas un pointeur!

Un tableau ne peut pas être passé par valeur en paramètre d'une fonction
=> passage par adresse, soit l'adresse du premier élément du tableau

Il est indispensable de conserver l'information sur la taille du tableau

```
#define N 10
int tab[N] ;
```

Prototype fonction :

```
void affiche_t (int * t, int n);
void affiche_t (int t[], int n);
```

Appel fonction :

```
affiche_t (tab, N);
```

Fonction et passage de tableau 1D en paramètre

Tab1D_et_fonction\main.c

```c
#include <stdio.h>
/**
 * @brief Illustration manipulation tableau 1D et paramètre
 *   de fonction
 */

#include <stdio.h>
#include <stdlib.h>

#define NB_ELMTS_TAB 4

void init_tab_int (int * tab1D, int n);
void affiche_tab_int (int tab[], int n);

void init_tab_int (int * tab1D, int n){
    for (int i = 0; i < NB_ELMTS_TAB; i++){
        *(tab1D + i) = 2 * i;
    }
}

void affiche_tab_int (int tab[], int n){
   for (int i = 0; i < NB_ELMTS_TAB; i++){
       printf ("tab[%d] : %d\n", i, *(tab + i)); // *(tab + i) <=> tab[i]
   }
}

int main(void) {
    int t_int[NB_ELMTS_TAB] = {0};

    init_tab_int (t_int, NB_ELMTS_TAB);
    affiche_tab_int (t_int, NB_ELMTS_TAB);
    return EXIT_SUCCESS;
}
```

```
Sélection d:\Documents\ProgrammationLangageC\build-Debug\bin\Tab1D_et_fonct...
tab[0] : 0
tab[1] : 2
tab[2] : 4
tab[3] : 6

==== Program exited with exit code: 0 ====
Time elapsed: 000:00.016 (MM:SS.MS)
Press any key to continue...
```

On se réveille...

- Chaine de caractères ?
- Caractère spécial → ‘\0’
- char * t_s = "Hello" ; ar t_s[] = "Hello" ;

Les chaines de caractères

Pas de type de données explicite pour les chaîne de caractères
Chaine de caractères : manipulée sous la forme de tableau 1D:

- Dont les éléments sont de type char ;
- Dont le dernier caractère est le caractère de fin de chaîne '\0'

Conséquence :
Toute chaine de caractères peut aussi être manipulée en utilisant les pointeurs.

Définition :
char t_s[12] = " Hello world";

Les chaines de caractères

Distinction entre char * t_s et char t_s[]

Soit la chaîne de caractères ‘’Une chaine” :

‘U’	‘n’	‘e’	‘ ’	‘c’	‘h‘	‘a’	‘i’	‘n’	‘e’	‘\0’	?

char t_s[12] = ” Une chaine”; ‘

Gestion implicite du caractère de fin de chaîne ‘\0’

char t_s[12] ={‘U’, ‘n’, ‘e’, ‘ ’, ‘c’, ‘h’, ‘a’ ‘i’ ,’n’, ‘e’, ‘\0’};

Gestion explicite du caractère de fin de chaîne ‘\0’

char * t_s = ” Une chaine”;

‘U’	‘n’	‘e’	‘ ’	‘c’	‘h‘	‘a’	‘i’	‘n’	‘e’	‘\0’

Gestion implicite du caractère de fin de chaîne ‘\0’

Les chaines de caractères

Distinction entre char * t_s et char t_s[]

char * t_s = ”Une chaine”;
char t_s[] = ”Une chaine”;

Mais subtilité :

1. char * t_s = ”Une chaine”; ⇔ Pointeur vers une chaine constante
   ⇔ const char * t_s = ”Une chaine”;
   // On ne peut donc plus modifier le contenu de la chaine de caractères.
   t_s[0] = ‘u’; // non valide, provoque une erreur lors de l’exécution.

2. char t_s[] = ”Une chaine”; ⇔ Creation d’un tableau de char dans lequel est copié le contenu
   t_s[0] = ‘u’ ; // est valide

Compléments : manipulations de chaine de caractères <string.h>

Fonctions dédiées de manipulation de chaines de caractères

Nécessite l'inclusion de string.h par #include<string.h>
Ressource : https://koor.fr/C/cstring/cstring.wp

Quelques fonctions :

- strlen : longueur d'une chaine de caractères
- strcpy : copier une chaine de caractères
- strcat : concaténer 2 chaines de caractères en une seule
- strcmp : comparer le contenu de 2 chaines de caractères

size_t strlen(const char *chaine);

Paramètre d'entrée : chaine de caractères
Valeur retournée : le nombre de caractères de la chaine (le caractére '\0' n'est pas comptabilisé)

On se réveille...

- Les paramètres de main() ???
  ARGV et ARGC ???
  Faire un prgm qui affiche la liste des arguments passés en parametre

Retour sur main()

```c
/*
 * Première manière :
 *      - le type de retour void, implique que votre programme n'indique pas au système
 *        d'exploitation comment il se termine (en succès ou en échec).
 */
void main(void) {

    // TODO

}

/*
 * Seconde manière :
 *      - le type de retour int, permet de renvoyer un code de retour au système d'exploitation.
 *          0 (ou constante EXIT_SUCCESS) : indique que le programme se termine correctement.
 *          Toute autre valeur (ou constante EXIT_FAILURE) : le programme se termine suite à une erreur.
 *      - Les constantes (EXIT_SUCCESS et EXIT_FAILURE) sont définies dans l'entête <stdlib.h>.
 */
int main(void) {

    // TODO

    return EXIT_SUCCESS;
}
```

https://koor.fr/C/Tutorial/Fonctions.wp

Retour sur main()

```c
/*
 * Troisème manière :
 *      - ce main permet de manipuler les arguments passés sur la ligne de commande.
 *        argc (argument counter) : indique combien de paramètres sont passés au main.
 *        argv (argument values) : un tableau de chaînes de caractères contenant les
 *                                 différents arguments.
 *      - le type de retour void, implique que votre programme n'indique pas au système
 *        d'exploitation comment il se termine (en succès ou en échec).
 */
void main( int argc, char * argv [] ) {

    // TODO

}

/*
 * Quatrième manière :
 *      - ce main permet de manipuler les arguments passés sur la ligne de commande.
 *        argc (argument counter) : indique combien de paramètres sont passés au main.
 *        argv (argument values) : un tableau de chaînes de caractères contenant les
 *                                 différents arguments.
 *      - le type de retour int, permet de renvoyer un code de retour au système d'exploitation.
 *           0 (ou constante EXIT_SUCCESS) : indique que le programme se termine correctement.
 *           Toute autre valeur (ou constante EXIT_FAILURE) : le programme se termine suite à une erreur.
 *      - Les constantes (EXIT_SUCCESS et EXIT_FAILURE) sont définies dans l'entête <stdlib.h>.
 */
int main( int argc, char * argv [] ) {

    // TODO

    return EXIT_SUCCESS;
}
```

Retour sur main()

argc : signifie "argument count" (nombre d'arguments). Il contient le nombre total d'arguments passés à votre programme, y compris le nom du programme lui-même.

argv : signifie "argument vector" (vecteur d'arguments). Il s'agit d'un tableau de chaînes de caractères (tableau de char*), où chaque élément est un argument passé au programme.

Le1er élément, argv[0], est toujours le nom du programme ou le chemin d'accès au programme.

```c
int main(int argc, char *argv[]) {
    // Affichage du nombre d'arguments
    printf("Nombre d'arguments : %d\n", argc);

    // Affichage de chaque argument passé
    for (int i = 0; i < argc; i++) {
        printf("Argument %d : %s\n", i, argv[i]);
    }

    return 0;
```