Introduction
Langage C

Partie 2: Langage C

Responsable : Florian Brandner (4D59)
Auteurs : Bertrand Dupouy et Etienne Borde

Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs(1)
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie
- Deuxième partie
  - Pointeurs et tableaux (2)
  - Allocation en mémoire
    - Pile et arguments
    - Passage d'adresses en paramètres
    - Variables globales et locales statiques
    - Allocation dynamique
- Annexe : Java/C

Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs(1)
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie
- Deuxième partie
  - ☞ **Pointeurs et tableaux (2)**
  - Allocation en mémoire
    - Pile et arguments
    - Passage d'adresses en paramètres
    - Variables globales et locales statiques
    - Allocation dynamique
- Annexe : Java/C

Pointeurs : rappels de la partie 1

- Le type d'un pointeur est déclaré celui de l'objet pointé suivi d'un astérisque (*) ,
- Le programme ci-contre indique comment sont déclarés et initialisés deux pointeurs, l'un sur des réel, l'autre sur des entiers.
- Attention :
  - Tous les pointeurs sont implantés sur le même nombre d'octets, quel que soit le type de l'élément pointé,
  - Le type d'un pointeur va jouer sur son arithmétique.

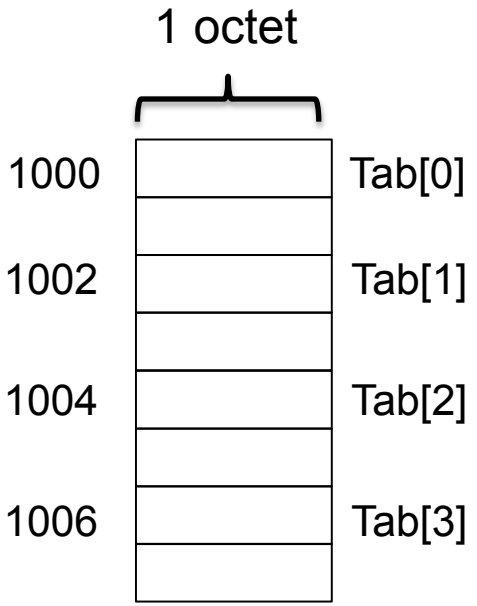
```
int i ;
float r ;
/* declaration */
int * ptr1 ;
float * ptr2;

/* initialisation */
ptr1 = &i;
ptr2 = &r;
```

Introduction à l'arithmétique des pointeurs (1/3)

- Il n'y a pas de type tableau en C : le nom d'un tableau est un pointeur constant sur l'adresse de début du tableau. Il ne peut donc pas être modifié.
  - Exemple : soit la déclaration :

    ```
    short Tab[4];
    ```

  - On suppose que le tableau est implanté à l'adresse 1000,
  - La case dont le nom est `Tab` contient donc la valeur 1000



- Si on écrit, dans le même programme :

  ```
  short  * Ptr1;
  Ptr1 = Tab;
  ```

  Alors ces deux instructions sont équivalentes : elles déposent toutes deux la valeur 0 à l'adresse 1002, sur deux octets :

  `Tab[1]= 0;` et `Ptr1[1]= 0;`

Introduction à l'arithmétique des pointeurs (2/3)

- Si on a déclaré :

```
int Tab[100];
int * Ptr;
Ptr = Tab;
```

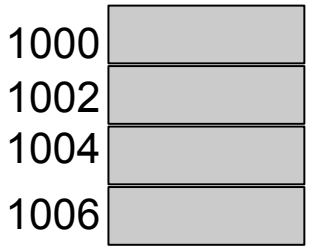
- Alors les notations suivantes sont équivalentes :

```
Tab[i]= 0;
*(Tab + i)= 0;
Ptr[i]= 0;
*(Ptr + i) = 0;
```

- On appelle cela l'***arithmétique des pointeurs***:

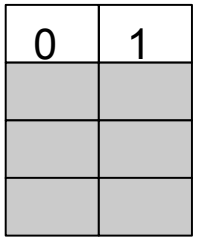
Si on a : `TYPE * ptr ;`
alors `ptr + k` désigne la case mémoire d'adresse : (valeur de ptr) + k*taille de TYPE ;

Introduction à l'arithmétique des pointeurs (3/3)

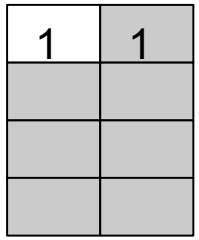


- Soit le programme :

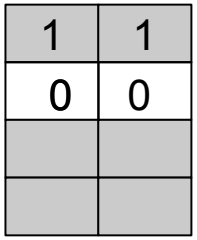
```
short Tab[4];
short  * Ptr1;
char   * Ptr2;
Ptr1  = (short *)Tab;
Ptr2  = (char  *)Tab;

*Ptr1 = 1;
*Ptr2 = 1;
*(Tab + 1) = 0;

/* valeur de Ptr1 après l'instruction
suivante : 1000 */
*(Ptr1 + 1) = 2;

/* valeur de Ptr1 après l'instruction
suivante : 1002 */
*(Ptr1 ++) = 3;

/* Instruction suivante incorrecte : on ne
peut modifier un pointeur constant */
*(Tab++) = 0;
```



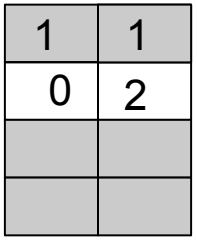
*Ptr1=1



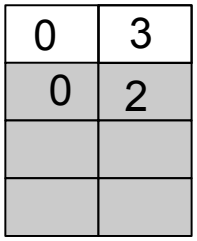
*Ptr2=1



*(Tab+1)=0



*(Ptr1+1)=2



*(Ptr1++)=3

Pointeurs : les chaînes de caractères

- Il n'y a pas, comme en java, de type `String` :
  - Une chaîne de caractère est implantée sous forme d'un tableau de `char`,
  - La fin d'un tableau de caractères est indiquée par le caractère `NULL` : `'\0'` (la valeur 0),
  - Une bibliothèque de fonctions fournit les opérations classiques sur les chaînes de caractères (`strcpy, strlen, strcat, …`) mais la vérification du bon usage de la mémoire reste à la charge du programme

- ***Exercice***: proposez une implémentation de strcpy:
  - `char * strcpy(char *dest, const char *src)`
  - *Parameters*

    `dest` -- *This is the pointer to the destination array where the content is copied.*

    `src` -- *This is the address of the first element in the char array to copy.*
  - *Return Value*

    This return a pointer to the destination string `dest`.

Pointeurs & Structures

- Il existe une notation spécifique pour accéder aux champs d'une structure via un pointeur.

Considérons le code ci-contre suivante :

Les notations

`(*c).reel` et `c->reel`

sont équivalentes.

```
struct complexe {
    float reel;
    float imagine;
};

int reinit_complexe(struct complexe * c){

  (*c).reel = 0.0 ;
  (*c).imagine = 0.0 ;

  c->reel = 0.0;
  c->imagine = 3.0;

  return 0;
}
```

Note: attention, les parenthèses sont nécessaires dans la notation `(*c).reel` pour lever une ambiguïté avec `*c.reel`

Pointeurs : résumé

1. **Opérateurs spécifiques :**
   - & : pour obtenir l'adresse d'une variable
   - * : déréférencement : aller chercher une valeur à l'adresse contenue dans la pointeur

   Attention : * sert aussi à déclarer une variable pointeur, par exemple : int * ptr ;

2. **Arithmétique :**
   - Les opérations + et - se font suivant le type de l'objet pointé :

     Si on a : TYPE * ptr ;

     alors ptr + k désigne la case mémoire d'adresse : (valeur de ptr) + k*taille de TYPE ;

3. **Allocation mémoire : en fonction du type de l'objet pointé**
4. C traite les pointeurs comme des tableaux et vice-versa, on peut donc utiliser la notation « pointeur » ou « indexée » sur un tableau ou un pointeur variable.

Plan

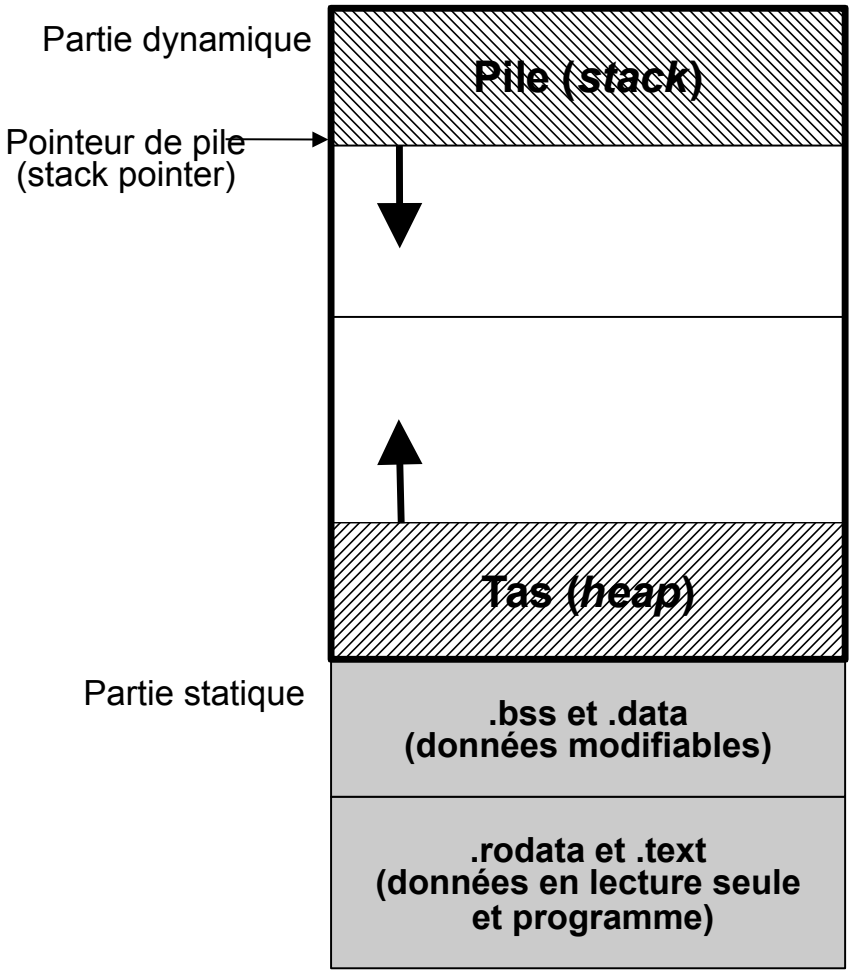
- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs(1)
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie

- Deuxième partie
  - Pointeurs et tableaux (2)
  - ☞ **Allocation en mémoire**
    - Pile et arguments
    - Passage d'adresses en paramètres
    - Variables globales et locales statiques
    - Allocation dynamique

- Annexe : Java/C

Les zones mémoire associées à un programme (Unix)

- La partie dynamique :
  1. la pile
  2. le tas où se trouvent les variables allouées par `malloc` (C), `new`(java),
- La partie statique :
  1. Une zone accessible à la fois en lecture et en écriture (on y trouve les variables globales et celles de type static):
     a) `.bss`, zone initialisée à zéro,
     b) `.data`,
  2. Une zone accessible en lecture seule :
     `.rodata`



Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs(1)
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie

- Deuxième partie
  - Pointeurs et tableaux (2)
  - Allocation en mémoire
    - ☞ **Pile et arguments**
    - Passage d'adresses en paramètres
    - Variables globales ou locales statiques
    - Allocation dynamique

- Annexe : Java/C

Utilisation de la pile: exemple illustratif

- Exemple illustratif pour justifier l'utilisation de la pile

```
int factorielle (int n){
  int i;
  if (n == 0)
    i= 1;
  else
    i = n * factorielle (n-1);
  return (i) ;
}
```

➔ Combien de cases mémoire sont nécessaires pour exécuter ce code?

Solution: utilisation dynamique sur la pile

- Les étapes de calcul intermédiaires sont stockées dynamiquement sur la pile
- Cela concerne (entre autres): les variables locales, paramètre de retour, et paramètre(s) d'entrée
- Les paramètres des fonctions sont copiés sur la pile pour ne pas écraser les valeurs calculées précédemment
- A la fin de l'exécution, l'espace utilisé par la fonction est rendu pour l'exécution de nouvelles fonctions
- Le registre stack pointer identifie le dernier élément ajouté sur la pile

Les fonctions : pile et appels récursifs

- Exemple : calcul de factorielle 3.
- **1-** On commence par une **première phase d'empilement des appels :**
  - Appel à factorielle (3)
  - Appel à 3 * factorielle (2)
  - Appel à 2 * factorielle (1)
  - Appel à 1 * factorielle (0)
  - On arrive à la condition de fin, la fonction renvoie la valeur , on passe en phase 2

```
int factorielle (int n){
  int i;
  if (n == 0)
    i= 1;
  else
    i = n * factorielle (n-1);
  return (i) ;
}
```

- **2- Phase de dépilement des appels** :
  - on sort de factorielle (0) par return(1), on dépile,
  - on retourne dans factorielle (1) qui renvoie 1*1,
  - on sort factorielle (2) qui renvoie 2*1,
  - on sort de factorielle (3) qui renvoie 3*2,
  - et on revient dans main…

Fonctions : passage des arguments par valeur

- Le passage des arguments se fait par valeur :
  - On donne le nom des variables et, à l'exécution du programme, lors de l'appel à la fonction, ce sont leurs valeurs qui sont **copiées** sur la pile,
  - Les fonctions appelée et appelante ne **partagent pas de mémoire,**
  - Si on veut que la fonction appelée modifie des variables dans la fonction appelante, il faut en passer les **références** (ou utiliser des variables globales)**,** *on passe la valeur de la référence*

- Certains langages passent les adresses de façon transparente (e.g. Java), en C il faut le faire explicitement en utilisant les pointeurs ou les opérateurs associés (l'opérateur & qui renvoie une adresse)

Fonctions : utilisation de la pile

- C'est une zone mémoire dédiée, de taille limitée dont le début est repéré par un registre spécial (le pointeur de pile ou stack pointer),
- Elle abrite les variables dites « dynamiques » :
  - arguments des fonctions et procédures,
  - variables locales,

- A propos de la fonction `main` : `main` admet des arguments qui peuvent être initialisés sur la ligne de commande (exemple : `int main(int argc, char *argv[])` :
  - `argc`, un entier : le nombre de mots donnés sur la ligne de commande,
  - `argv`[ ], un tableau de chaînes de caractères contenant la liste de ces mots,
  - *`arge`[ ] , un tableau de chaînes de caractères,* ***rarement utilisé,*** *contenant les variables d'environnement,*

- Remarque : en Java `main` prend un seul argument : `args[]`

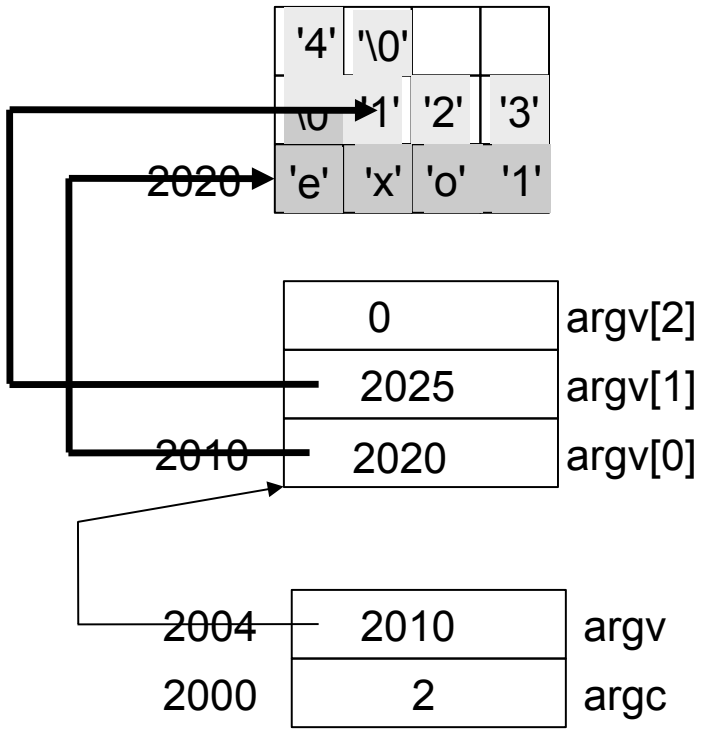
La pile : appel à la fonction main

- La commande :

  `exo1 1234`

a été entrée au clavier, donc les mots `exo1` et `1234` ont été rangés respectivement dans `argv[0]` et `argv[1]`),

- Voici un schéma illustrant l'utilisation de la pile par le système lors de l'appel à la fonction `main` :
- L'initialisation des paramètres `argc` et `argv` sur la pile est réalisée par un code binaire (`crt0.o`) qui est ajouté à l'exécutable par gcc.



La pile : effet d'une mauvaise utilisation de `strcpy`

- On a involontairement copié un texte sur le premier mot de ceux passés à main (`texte` copié en `argv[0]` au lieu de l'inverse).

```
./a.out  mot1 mot2 234

message : bonjour tout le monde!
texte : bonjour tout le monde!



ptr : bonjour tout le monde!
argv[0] : bonjour tout le monde!
argv[1] : tout le monde!
argv[2] : le monde!
argv[3] : nde!
```

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
int i =0;

char message[] ="bonjour tout le
monde!";
char texte [80];
char * ptr;

printf("message : %s\n", message);
ptr = message;
strcpy(texte, message);

printf("texte : %s\n", texte);
printf("ptr : %s\n", ptr);

/* au lieu de strcpy(texte, argv[0]): */
strcpy(argv[0], texte);

while (argv[i] != 0) {
  printf("argv[%d] : %s\n",i, argv[i]);
  i++;
  }
     return 0;
}
```

La pile : appel à la fonction main

- Dans le programme suivant, on affiche :
  - la **valeur** (format `%p`) des pointeurs (`argv[i]` et `arge[i]`) passés sur la pile à main,
  - le **contenu** des chaînes de caractères repérées par ces pointeurs (format `%s`)

```
int main(int argc, char* argv[], char* arge[]){
  short i;
  for(i=0; argv[i] != 0; i++)
    printf("argv[%d](0x%p) : %s\n", i, argv[i], argv[i]);

  for(i=0; i < 3; i++)
    printf("arge[%d](0x%p) : %s\n", i, arge[i], arge[i]);
 ...
```

```
Résultat :

argv[0](0x5fbffbb0) : ./a.out
arge[0](0x5fbffbb8) : TERM_PROGRAM=Apple_Terminal
arge[1](0x5fbffbd4) : TERM=xterm-color
arge[2](0x5fbffbe5) : SHELL=/bin/bash
```

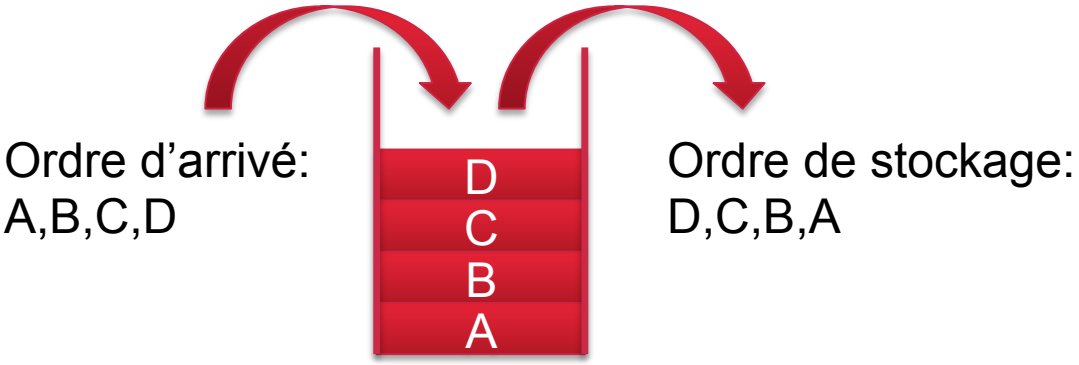
Les fonctions : pile et appels récursifs

- Exemple : calcul de factorielle 3.
- **1-** On commence par une **première phase d'empilement des appels :**
  - Appel à factorielle (3)
  - Appel à 3 * factorielle (2)
  - Appel à 2 * factorielle (1)
  - Appel à 1 * factorielle (0)
  - On arrive à la condition de fin, la fonction renvoie la valeur , on passe en phase 2
- **2- Phase de dépilement des appels** :
  - on sort de factorielle (0) par return(1), on dépile,
  - on retourne dans factorielle (1) qui renvoie 1*1,
  - on sort factorielle (2) qui renvoie 2*1,
  - on sort de factorielle (3) qui renvoie 3*2,
  - et on revient dans main…

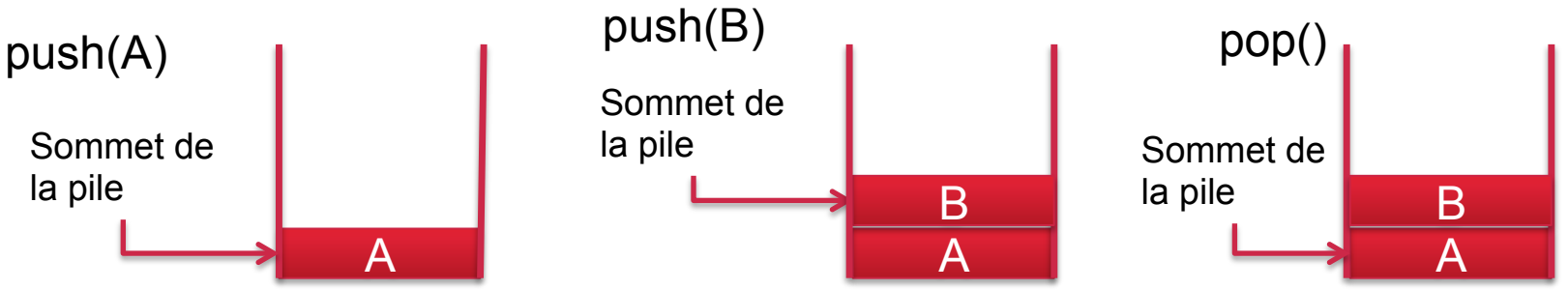
```
int factorielle (int n){
  int i;
  if (n == 0) i= 1;
    else
    i = n * factorielle (n-1);
  return (i) ;
}
```

La pile: définition générale

- En informatique, une pile est une **structure de donnée** dans laquelle les éléments sont stockés pour être ensuite récupérés dans un ordre **LIFO** (Last In First Out)



- Une pile est en générale caractérisée par **un pointeur sur le sommet de la pile**, et deux opérations:
  - **push**: ajoute un élément au sommet la pile, et décale le pointeur sur le nouveau sommet
  - **pop**: enlève l'élément au sommet de la pile et décale le pointeur sur le nouveau sommet



Exemple avec le code du factoriel, traduit en code assembleur (simplifié)

```
int factorielle (int n){
  if (n == 0) n = 1;
  else
    n = n * factorielle (n-1);
  return n ;
}

int main() {
  int ret = factorielle (3);
}
```

```
factorielle:
0000  cmp  0, %SP
0002  jne  8
0004  move 1, %SP
0006  jmp  16
0008  move %SP, %AX
0009  add  -1, %AX
000B  push %AX
000D  call 0
000E  pop  %AX
0010  move %SP, %BX
0012  mul  %AX, %BX
0014  move %AX, %SP
0016  Ret

main:
  move 3, %AX
  push %AX
  call 0
  pop %AX
```

Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs(1)
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie
- Deuxième partie
  - Pointeurs et tableaux (2)
  - Allocation en mémoire
    - Pile et arguments
    - ☞ **Passage d'adresses en paramètres**
    - Variables globales et locales statiques
    - Allocation dynamique
- Annexe : Java/C

Allocation mémoire : passage par valeur

- Les variables locales à une fonction, et les paramètres sont alloués sur la pile.
- La valeur des paramètres est copiée sur la pile.
- L'espace mémoire alloué est rendu à la sortie de la fonction.
- Dans l'exemple ci-contre, les variables i et j de main **ne sont pas** modifiées.

```
Résultat :

main :    i = 10, j = 20

echange : i = 20, j = 10

main :    i = 10, j = 20
```

```
#include <stdio.h>
/*******  main   *******/
int main(int argc, char*argv[]){
  int i, j;
  void echange(int, int);

  i = 10;
  j = 20;

  printf ("main : i = %d, j = %d\n",i , j);
  echange(i, j);
  printf ("main : i = %d, j = %d\n",i , j);
  return 0;
}

/*******   echange   *******/
void echange(int i, int j){
  int k;
  k = i;
  i = j;
  j = k;
  printf("echange : i = %d, j = %d\n",i , j);
  return;
}
```

Passage d'adresses ou de valeurs (1/2)

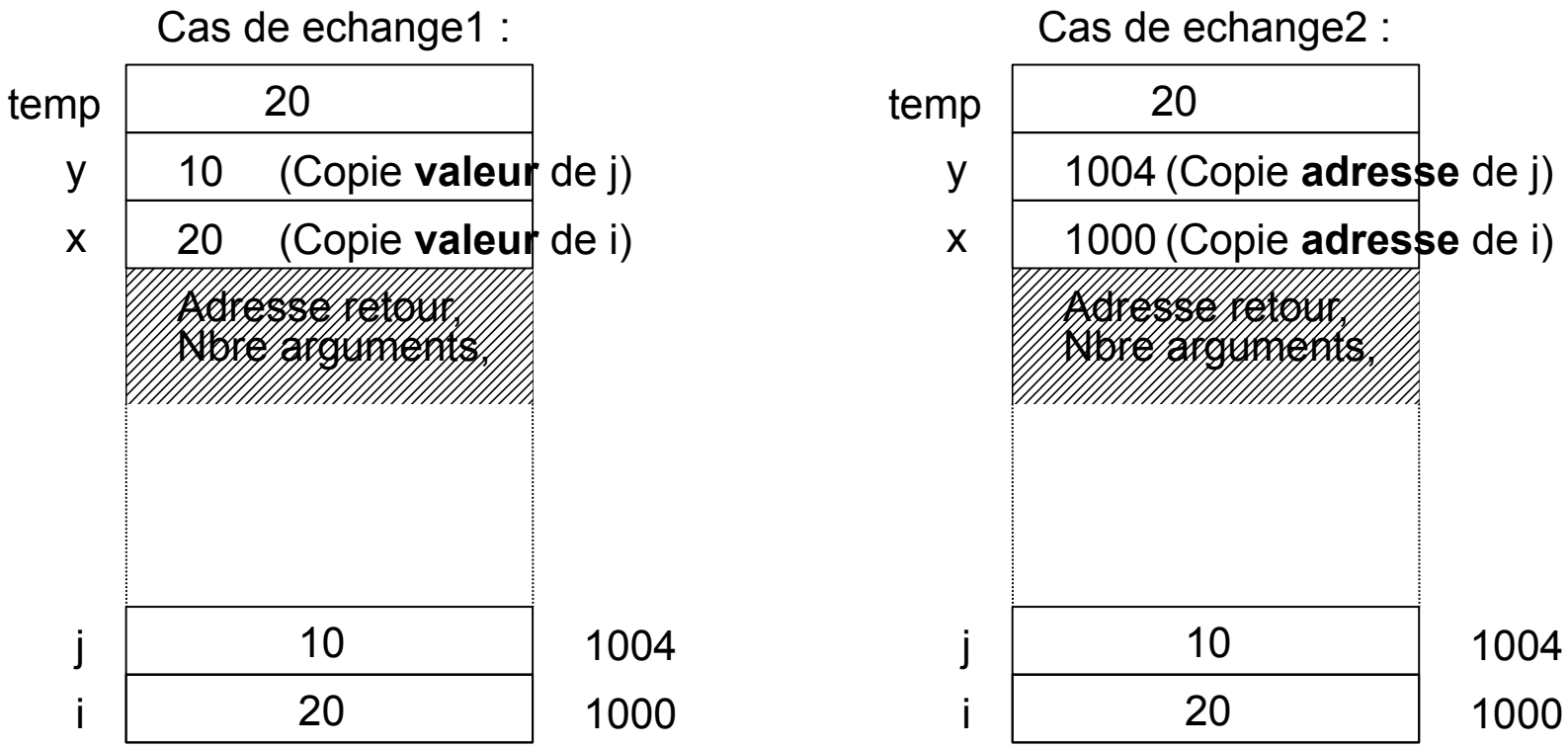
- Pour affecter la valeur de variables depuis une fonction, on passe leur adresse sur la pile,
- Dans l'exemple ci-contre :
  - la fonction echange1 **modifie les valeurs déposées sur la pile,** donc ne modifie pas i et j dans main,
  - La fonction echange2 utilise les **adresses** de ces variables déposées sur la pile et modifie le contenu des variables en utilisant des pointeurs (cf. schéma page suivante),

```
Resultat :
i : 20, j : 10
i : 10, j : 20
```

```
int main(int argc, char*argv[]) {
     void echange1 (int , int );
     void echange2 (int *, int *);
     int i = 20, j = 10;

     echange1 (i, j);
     printf(" i : %d, j : %d\n", i, j);
     echange2 (&i, &j);
     printf(" i : %d, j : %d\n", i, j);
     return;
}
void echange1 (int x, int y){
     int temp;
     temp = x;
     x = y;
     y = temp;
     return ;
}
void echange2 (int *x, int *y){
     int temp;
     temp = *x;
     *x = *y;
     *y = temp;
     return ;
}
```

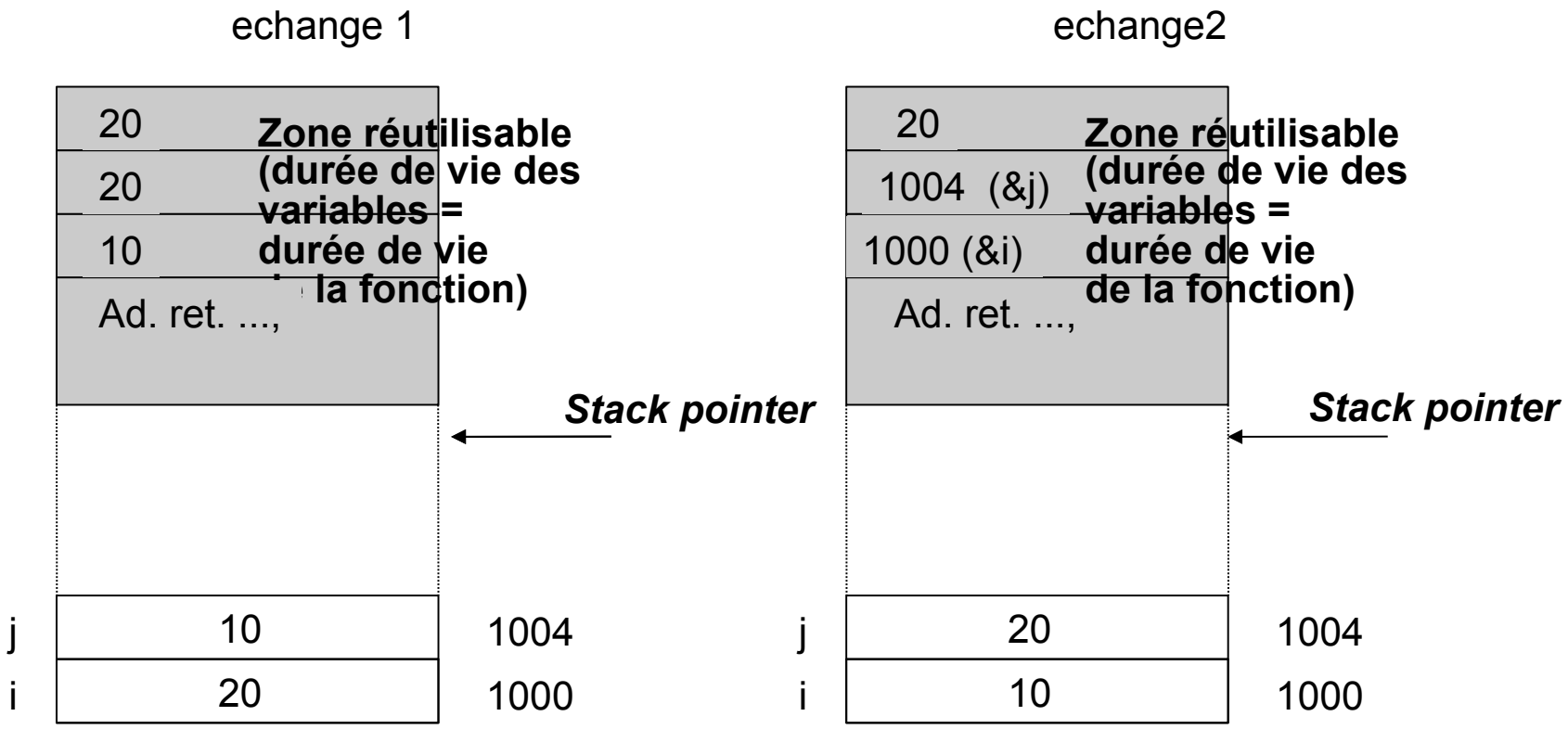
Passage d'adresses ou de valeurs (2/2)

- Etat de la pile après exécution de `temp=x` dans echange1 et `temp=*x` dans echange2



Gestion de la pile

- Etat de la pile après retour dans main :



Nous avions déjà vu des cas intéressants…

- Les fonctions scanf et fscanf

`str`, le paramètre est bien une adresse

`&i`, le paramètre est bien une adresse

```
#include <stdio.h>

int main ()
{
 char str [80];
 int i;

 printf ("Enter your family name: ");
 scanf ("%s", str);
 printf ("Enter your age: ");
 scanf ("%d", &i);
 printf ("Ms. or Mr. %s , %d years old.\n",str,i);
}
```

Fonctions ou procédure ? (passer une valeur ou une adresse) : exemple

- Exemple d'initialisation de struct, avec deux modes d'initialisation des variables :
  - par une fonction : une fonction peut renvoyer la valeur d'une struct,
  - par une procédure, dans ce cas il faut passer les variables par adresse :

```
typedef struct{
              float reel;
              float imagine;
} complexe_t ;

int main(int argc, char*argv[]){
  void init_1(complexe_t *ptr);
  complexe_t init_2(void);
  complexe_t c1;

/* initialisation par passage d'adresse */
  init_1(&c1);

/* initialisation par retour de valeur */
   c1=init_2();
   return 0;
}
```

```
/* fonction */
complexe_t init_2(void){
  complexe_t x;
  x.reel = 1.0;
  x.imagine = 1.0;
  return x;
}
```

```
/* procedure */
void init_1(complexe_t *ptr){
  (*ptr).imagine = 1.0;
  ptr->reel = 1.0;
  return;
}
```

Passer en paramètre une adresse ou une valeur

- Si la variable peut être modifiée par la fonction appelée : passage de l'adresse.
- Sinon, le critère de choix peut être la performance : pour une structure complexe (une image), passer la valeur est contre-productif.
- Vocabulaire :
  - On appelle paramètre formel un paramètre donné à la définition d'une fonction :

```
int fonc (int a, float b){
        …
}
```

  - On appelle paramètre "actual" le paramètre donné lors de l'appel d'une fonction :

```
...
fonc(5, 2.0) ;
...
```

Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs(1)
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie
- Deuxième partie
  - Pointeurs et tableaux (2)
  - Allocation en mémoire
    - Pile et arguments
    - Passage d'adresses en paramètres
    - ☞ **Variables globales et locales statiques**
    - Allocation dynamique
- Annexe : Java/C

Allocation mémoire : variables globales

- Ici une variable globale est utilisée par la fonction main et par une fonction se trouvant dans un autre fichier :
  - Elle est définie dans le fichier contenant main,
  - Cette variable est déclarée et référencée dans l'autre fichier :
- **extern** indique au compilateur que la variable est implantée dans un autre fichier

```
#include <stdio.h>
#define DIM 10

int table_glob[DIM];

void echange(int, int);

int main (int argc, char*argv[]){
   int i;

   for(i=0; i < DIM; i++)
      table_glob[i] = i ;
   echange(2, 5);
   return 0;
}
```

```
extern int table_glob[];

void echange (int i, int j){
   int k;
   k=table_glob[i];
   table_glob[i]=table_glob[j];
   table_glob[j]=k;
}
```

Allocation mémoire : variable locale déclarée « static »

- Une variable qualifiée de `static` :
  - Est implantée dans un espace mémoire qui lui est alloué lors de la compilation dans la partie statique de la région de données : `.data ou .bss` (respectivement : variables initialisées ou non),
  - Donc :
    - elle n'est pas implantée sur la pile pendant l'exécution de la fonction où elle est déclarée,
    - elle est référencée par une adresse fixe,
    - elle n'est pas réinitialisée à chaque appel à la fonction où elle a été déclarée,
- Un exemple d'utilisation :
  - compter dans une fonction le nombre d'appels à la fonction elle-même

(voir exemple suivant).

Allocation mémoire : variables locales dynamiques et statiques

- Rappel :

une variable locale est implantée sur la pile : à la sortie du bloc d'instructions dans lequel elle est utilisée l'espace alloué sur la pile est rendu.

- Conséquence :

Ici, dans la fonction `somme` la variable `compteur` est réallouée sur la pile à chaque appel, ce qui n'est pas la cas dans `carre`.

```
RESULTAT :

somme : compteur = 1
somme : compteur = 1
carre : compteur = 1
carre : compteur = 2
```

```
#include <stdio.h>
int main(void) {
  float somme(float x, float y);
  int carre(int );
  int i;
  float r;
  r = somme(1.0, 2.0);
  r = somme(10, 0.75);
  i = carre (3);
  i = carre (5);
  return 0;
}
float somme(float x, float y){
  int compteur = 0;
  compteur ++;
  printf("somme : compteur = %d\n", compteur);
  float z;
  z = x + y;
  return z;
}
int carre(int j){
  static int compteur = 0;
  compteur ++;
  printf("carre : compteur = %d\n", compteur);
 return (j*j);
}
```

Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs(1)
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie
- Deuxième partie
  - Pointeurs et tableaux (2)
  - Allocation en mémoire
    - Pile et arguments
    - Passage d'adresses en paramètres
    - Variables globales et locales statiques
    - ☞ **Allocation dynamique**
- Annexe : Java/C

Comment savoir la taille de la zone mémoire à allouer? opérateur `sizeof ()`

- Comme nous l'avons vu précédemment, l'espace mémoire associé à une variable dépend de son type ET de la machine…
- L'opérateur `sizeof()` renvoie, en octets, la taille d'une variable ou d'un type.
- Il prend en paramètre une variable, ou un identifiant de type.

```
#include <stdio.h>

int main (void){
 long k;
 int i;
 short l;
 char car;

 printf("sizeof(long) : %ld\n", sizeof(long));
 printf("sizeof(int) : %ld\n", sizeof(int));
 printf("sizeof(short) : %ld\n", sizeof(short));
 printf("sizeof(char) : %ld\n", sizeof(char));

 printf("sizeof(k) : %ld\n", sizeof(k));
 printf("sizeof(i) : %ld\n", sizeof(i));
 printf("sizeof(l) : %ld\n", sizeof(l));
 printf("sizeof(car) : %ld\n", sizeof(car));
 return 0;
}
```

Résultats affiché sur le terminal **(sur ma machine)** :

```
sizeof(long) : 8
sizeof(int) : 4
sizeof(short) : 2
sizeof(char) : 1

sizeof(k) : 8
sizeof(i) : 4
sizeof(l) : 2
sizeof(car) : 1
```

`%ld` : format "entier long"

Types composés : taille en mémoire

- Attention : la taille d'un type composé est supérieure à la somme de la taille de chacun de ses éléments…
- L'explication sera donnée dans le partie de cours sur les systèmes d'exploitation

```
#include <stdio.h>
typedef struct{
    int x;
    long y;
    char couleur;
} point_t ;

int main (void){
 printf("sizeof(point_t) : %d\n", (int)sizeof(point_t));

 printf("sizeof(int)  + sizeof(long) + sizeof(char) : %d\n",
        sizeof(int)  + sizeof(long) + sizeof(char) );

 return 0;
};
```

```
Résultat :

sizeof(point_t) : 24
sizeof(int)  + sizeof(long) + sizeof(char): 13
```

Pointeurs : utilisation de l'opérateur sizeof

- Dans ce programme, on a :

```
sizeof(tab1) = 128
sizeof(tab2) = 11

sizeof(tab3) = 512
sizeof(ptr) = 8
sizeof(ptr2) = 8
```

- Attention, sizeof n'est pas une fonction, et le résultat n'est pas « calculé »:

```
sizeof(param) = 8
```

```
#include <stdio.h>
#include <string.h>

int compute_erroneous_sizeof(int*);

int main(int argc, char *argv[]){

  char tab1[128] ;
  char tab2[] = "0123456789";
  sizeof(tab1);
  sizeof(tab2);

  int tab3[128] ;
  sizeof(tab3);

  char *ptr = tab1;
  int * ptr2 = tab3;
  sizeof(ptr);
  sizeof(ptr2);

  return 0;
}

int compute_erroneous_sizeof(int * param){
  return sizeof(param);
}
```

Pointeurs : allocation dynamique de mémoire (1/2)

- Les trois fonctions les plus utilisées pour gérer la mémoire sont les suivantes :
  - `void * malloc(size_t size)` : allocation, renvoie un `void *`,
  - `void free(void * ptr)` : libération, renvoie un `void`,
  - `void * realloc(void * ptr, size_t size)` : réallocation, renvoie un `void *`
- Pour demander l'allocation d'une zone en mémoire, on donne sa taille en octets, par exemple :

```
int * ptr, * tab ;
int dim = 100 ;

/* demande d'allocation d'un  entier */
ptr = (int *) malloc (sizeof(int));

/* demande d'allocation d'un tableau de 100 entiers */
tab = (int *) malloc (dim *sizeof(int)); /* tableau de 100 int */
```

Note: size_t est défini comme un *unsigned long* sur ma machine

Pointeurs : allocation dynamique de mémoire (2/2)

- Exemple d'utilisation des trois fonctions précédentes :

```
int * ptr;

/* ptr va pointer vers une zone de 24 entiers
 * alloues sur le tas
 */
ptr = (int *) malloc (24*sizeof(int));

...

/* realloc change la taille de cette zone allouee sur le tas
 * en la copiant à une nouvelle  adresse, les eventuels
 * octets   supplémentaires ne sont pas initialises
 */
 ptr = (int *) realloc (ptr, 48*sizeof(int));

…

/* on libère cette zone qui pourra etre réutilisée par le
système*/
free (ptr);
```

Allocation d'un tableau à deux dimensions (1/2)

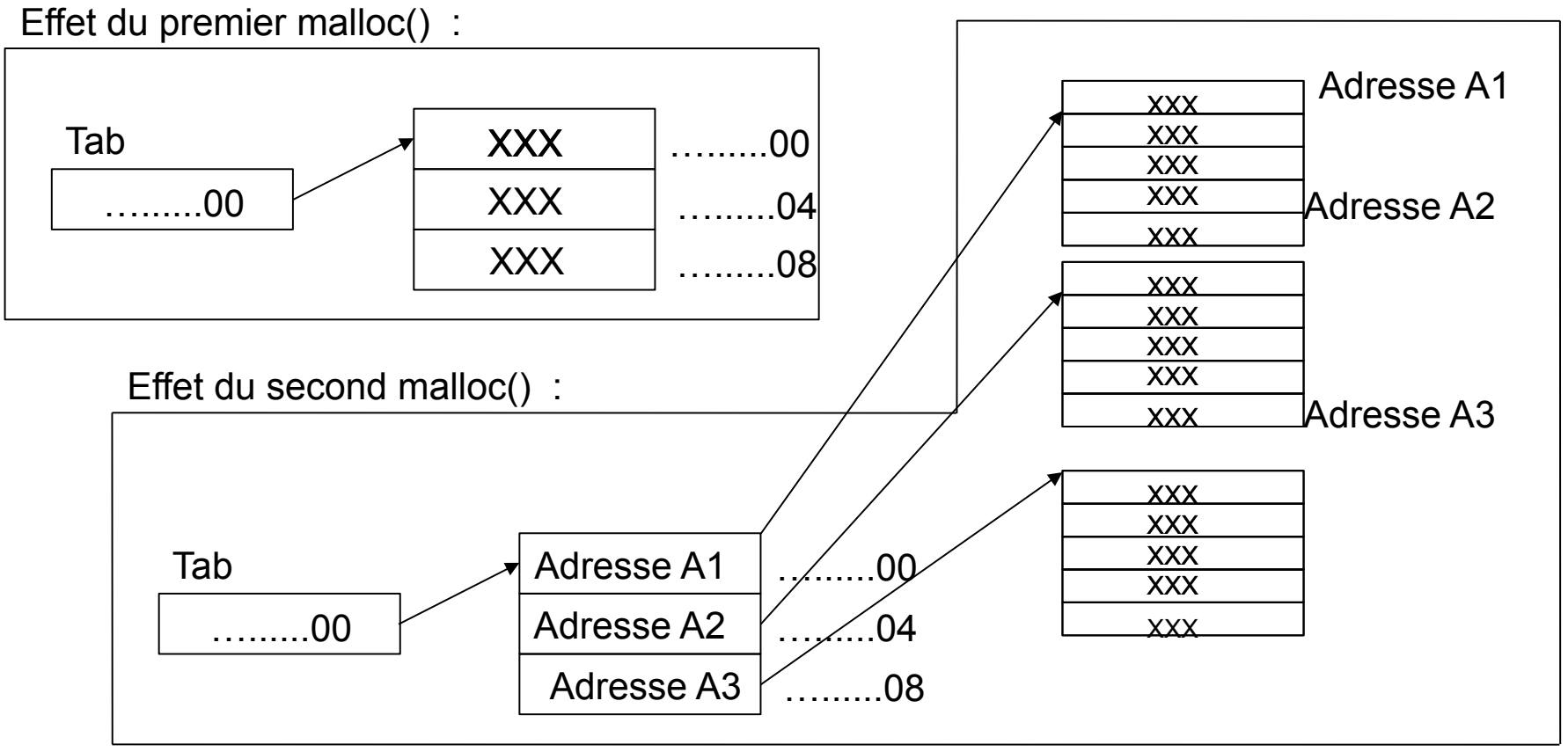
- Exemple pour un tableau à deux dimensions :
  - Un tableau à deux dimensions est un tableau à une dimension dont chaque élément est lui-même un tableau…
  - Le premier tableau est donc un tableau de **pointeurs** !

```
int main(int argc, char*argv[]){
int ** Tab;
int i, j;
int nlig = 3;
int ncol = 5;

/* Allocation des pointeurs vers chaque ligne */
Tab = (int **)malloc(nlig * sizeof(int *));

/* Allocation memoire pour chaque lignes */
for (i = 0; i < nlig; i++)
  Tab[i] = (int *)malloc(ncol * sizeof (int));

/*  Initialisation du  tableau */
for (i = 0 ; i < nlig ; i++)
  for (j = 0; j < ncol; j++){
    Tab[i][j] = i * ncol + j;
    printf("Tab[%d][%d] = %d adresse %08p\n",
            i,j,Tab[i][j],(int)&Tab[i][j]);
  }
  return 0;
}
```

```
Résultat :
Tab[0][0] = 0 adresse 001000a0
Tab[0][1] = 1 adresse 001000a4
Tab[0][2] = 2 adresse 001000a8
Tab[0][3] = 3 adresse 001000ac
Tab[0][4] = 4 adresse 001000b0
Tab[1][0] = 5 adresse 001000c0
Tab[1][1] = 6 adresse 001000c4
Tab[1][2] = 7 adresse 001000c8
Tab[1][3] = 8 adresse 001000cc
Tab[1][4] = 9 adresse 001000d0
Tab[2][0] = 10 adresse 001000e0
Tab[2][1] = 11 adresse 001000e4
Tab[2][2] = 12 adresse 001000e8
Tab[2][3] = 13 adresse 001000ec
Tab[2][4] = 14 adresse 001000f0
```

Allocation d'un tableau à deux dimensions (2/2)

- Au début, voici le contenu de Tab : Tab | XXX



Attention aux fuites mémoire…

- On appelle fuite mémoire (memory leak) le fait que la mémoire allouée n'est pas entièrement libérée (on ne fait pas autant de free que de malloc)

- Ce n'est pas important avec un programme qui finit rapidement, mais avec un programme fait pour s'exécuter plusieurs jours, ou mois....

- Sur un programme de 100 KLOC, corriger ce problème peut être difficile

- Exemple de source de memory leak, à méditer… :

```
void leak_example(void * addr)
{
    addr = malloc(8);
}
```

→ Pourquoi? Solutions?

Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs(1)
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie
- Deuxième partie
  - Pointeurs et tableaux (2)
  - Allocation en mémoire
    - Pile et arguments
    - Passage d'adresses en paramètres
    - Variables globales et locales statiques
    - Allocation dynamique
- ☞ **Annexe : Java/C**

Annexe : retour sur le langage Java

Quelques différences C/Java

- Niveau développement : objet vs fonctionnel. En simplifiant :

  Java : les objets encapsulent les données et leur traitement,

  ⇨ l'application **doit** respecter une structure hiérarchique (packages, classes) ;

  C : les données sont séparées de leur traitement, la fonction est le moyen d'organisation de l'application,

  ⇨ pas d'obligation de structuration, elle doit être volontaire,

- Niveau chaîne de production, pour passer du fichier source à une exécution :

  En java : fichier source → bytecode → exécution par la JVM

  En C : fichier source → fichier objet → fichier exécutable par la machine,

- Niveau support d'exécution :

  Gestion mémoire : le GC de la JVM gère automatiquement la mémoire en Java, en C ce travail est à la charge du programmeur

Environnement de développement (1/2)

- Java et Eclipse:
  - Pendant les TP Java, l'utilisation d'Eclipse a masqué les étapes intermédiaires qui permettent de passer du fichier source à l'exécution de l'application par la JVM.
  - Pour passer, par exemple, du fichier source `exo.java` à une exécution sur la JVM, vous auriez pu utiliser les deux commandes suivantes:

    1- `javac exo.java` (appel au compilateur java et production d'un fichier *bytecode* appelé `exo.class`)

    2- `java exo` (exécution du fichier `exo.class` par la JVM)

- Pendant les TP langage C, pour comprendre les différentes étapes qui font passer du fichier source au fichier exécutable on utilisera le langage de commandes, (cf. page suivante)

Environnement de développement (2/2)

- Comment passer d'un fichier source C à une exécution :

1. Produire un fichier exécutable (compilation et édition de liens) à partir du fichier source `exo.c` :

```
$gcc -Wall exo.c -o exo
```

   - `-Wall` (Warnings all), pour que le compilateur (par défaut très permissif) indique toutes les fautes, même si elles n'empêchent pas la production d'un fichier exécutable.

   - `-o` (output), le fichier exécutable s'appellera `exo` au lieu de `a.out`, appellation historique…

2. Lancer l'exécution :

```
$./exo
```

Fonction main : C vs Java

- Ci-contre, l'affichage du message « coucou ! » écrit en C (puis en Java). Ce que l'on observe :

1. Des commentaires inclus entre /* et */ ou précédés de // (comme en Java).
2. Une ligne qui commence par le caractère #, cette ligne est traitée par le **préprocesseur**, **particularité du C,**
3. La **fonction** `main` point d'entrée du système dans le programme :
   - main accepte **deux** arguments (un seul en Java) :
     - `argc`: nombre de mots
     - `argv.` : tableau contenant ces mots
4. Affichage écran : `printf` vs `System.out.println`

```
/*  Commentaire  */
//  Commentaire
#include <stdio.h>

int main(int argc, char* argv[]){
     printf("coucou !\n");
     return 0;
}
```

```
// Commentaire

import java.io.*;

public class coucou{
 public static void main (String args [])
{
        System.out.println("coucou !");
        }
}
```

Affichage sur l'écran : C vs Java

- Entrées-sorties : différences C/Java

Le fichier **`stdio.h contient les signatures`** des fonctions d'entrées-sorties

```c
#include <stdio.h>

int main(int argc, char*argv[]) {
   int  k = 5;
   float r = 7.0;
   printf("k = %d, r = %f\n", k, r):

}
```

```java
import java.io.*;
import java.util.Formatter;

public static void main(String args[]){
   Formatter fmt = new Formatter();
   int  k = 5;
   float r = 7.0F;
   fmt.format("k = %d,  r = %f", k, r);
   System.out.println(fmt);
// equivalent à :
   System.out.println("k = " + k + "r = "+ r);

}
```

Structure d'un programme : exemple

- Le point d'entrée : la fonction main

```
/*******************************
      main
 *******************************/
#include <stdio.h>

int main(int argc, char*argv[]) {
  int k;
  float r, s;
  float somme(float, float );
  int carre(int);

  s = 3.0;
  r = somme(s, 2.0);
  k = carre(3);

  printf("k = %d, r = %f\n", k, r);
  return 0;
}
```

Le rôle de cette ligne sera expliqué dans la section « préprocesseur »

Exemple de jeu de fonctions :

```
/*******************************
    Somme de deux réels
 *******************************/
float somme(float x, float y){
  float z;
  z = x + y;
  return z;
}
```

```
/*******************************
     Carre d un entier
 *******************************/
int carre(int j){
  return (j * j);
}
```

Saisies clavier : exemple

- Comment saisir des valeurs au clavier

- Attention : pour saisir les valeurs des variables on donne leur adresse.

```
#include <stdio.h>
int main(int argc, char* argv[]){
     int dim = 0;
     float x = 0.0;

     printf("Donner dim (entier) ");
     scanf("%d", &dim);

     printf("Donner x (reel) ");
     scanf("%f", &x);

     printf("dim = %d et x =  %f\n", dim, x);
     return 0;
}
```

```
Exemple d'exécution :

Donner dim (entier) 3
Donner x (reel) 2
dim = 3 et x =  2.000000
```

Passer des valeurs sur la ligne de commande

- Comment saisir des valeurs sur la ligne de commande:

```
/***********  io2.c ********** /
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
 int dim = 0;
 float x = 0.0;
 if (argc != 3 ){
   printf("Utilisation : %s dim (entier x(reel)\n", argv[0]);
   return 1;
   }

  dim = atoi(argv[1]);
  x = atof(argv[2]);
  printf("dim = %d et x =  %f\n", dim, x);
 return 0;
}
}
```

```
Exemple d'exécution du programme :
$ gcc -Wall io2.c -o monprog
$ monprog
Utilisation : ./monprog dim (entier x(reel)
$ monprog  34 2
Dim = 34 et x =  2.000000
```

Java et les chaînes de caractères

- En Java, les caractères sont codés sur un ou deux octets (byte : un octet) :

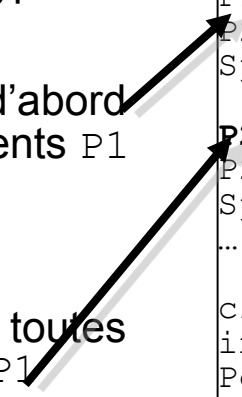
```
#public static void main(String[] args) {
  Formatter fmt = new Formatter();
  String chaîne1 = new String("año : année");
  System.out.println(chaîne1);
  byte[] octets = chaîne1.getBytes();

  // taille du tableaux d octets :
  System.out.println("taille du tableau d octets  : "
                     + octets.length);
  // Contenu du tableau d octets :
  for (int k = 0; k < octets.length; k++)
           fmt.format("0x%x, ", octets[k]);
  System.out.println(fmt);

  // Re-conversion des octets  en string  :
  String chaîne2 = new String(octets);
  System.out.println("chaîne2 = " + chaîne2);
```

Les caractères ñ et é sont codés sur 2 octets, les autres sur un seul :

```
Résultat :
año : année
taille du tableau d octets  : 13
0x61, 0xc3, 0xb1, 0x6f, 0x20, 0x3a,
0x20, 0x61, 0x6e, 0x6e, 0xc3, 0xa9,
0x65,
chaîne2 = año : année
```

Retour sur les références en Java (1/2)

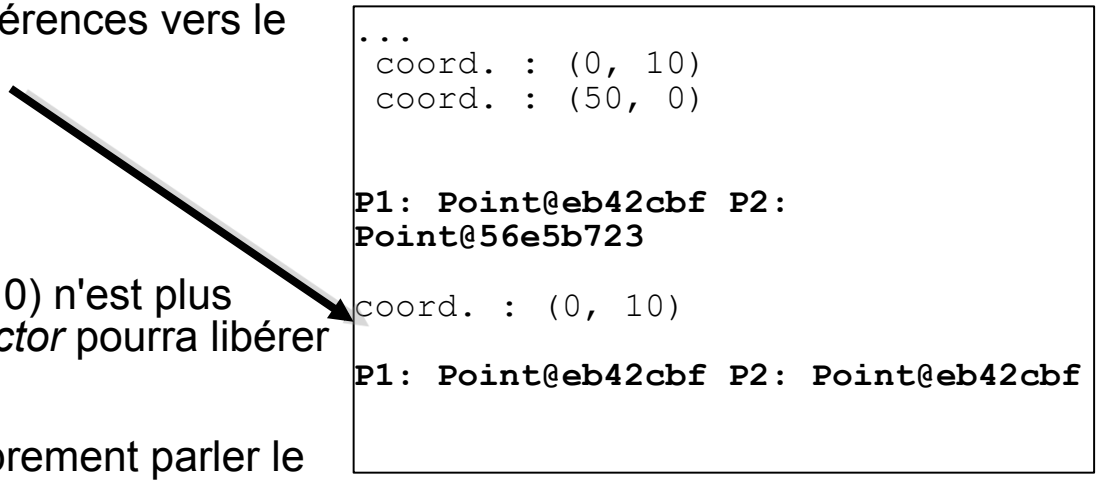
- Dans le programme ci-contre, `P1` et `P2` sont des références vers deux objets du type `Point` :
  - Elles référencent d'abord deux objets différents `P1` et `P2`
  - Puis elles pointent toutes deux sur le point `P1`



```
...
Point P1 = new Point(0, 10);
Point P2 = new Point(50, 0);

P1.affiche();
P2.affiche();
System.out.println("P1: "+P1+" P2: "+P2);

P2 = P1;
P2.affiche();
System.out.println("P1: "+ P1+" P2: "+P2);
…

class Point{
int x, y;
Point(int x, int y){
 this.x = x;
 this.y = y;
 }
void affiche(){
System.out.println(”coord.: (" +x+", "+ y+")");
  }
}
```

Retour sur les références en Java (2/2)

- Exécution du programme précédent : P1 et P2 sont maintenant des références vers le même objet de type Point.



- L'objet de coordonnées (50,0) n'est plus référencé, le *garbage collector* pourra libérer la mémoire qu'il occupe
- P1 et P2 ne sont pas à proprement parler le nom des objets. On pourrait ajouter un champ `nom` du type `String` et les appeler A et B, par exemple.

Java et le *bytecode*

Le compilateur java (`javac`) produit un fichier contenant du *bytecode* :

- ce code n'est exécutable sur aucun processeur,
- il sera interprété par la JVM(*Java Virtual Machine*) : chaque instruction *bytecode* sera traduite en langage machine.
- la JVM commence l'exécution en chargeant la classe contenant la méthode main.

Exemple :

`$javac HelloWorld.java` (production de `HelloWorld.class`)

`$java HelloWorld` (appel à la JVM qui charge `HelloWorld.class`)

Java : du source au bytecode

Les fichiers contenant du *bytecode* peuvent être désassemblés grâce à l'utilitaire `javap`.
Par exemple, pour le fichier bytecode créé à partir du fichier source ci-contre, on obtient (commentaires ajoutés en français…) :

```
public class HelloWorld{
public static void main(String args[]){
   int i;
   for(i=0 ; i<10 ; i++);
   System.out.println(" Hello world,  i = " + i);
   }
}
```

```
$javac  HelloWorld.java
$javap -c HelloWorld.class

 0: iconst_0 0          # mettre 0 sur la pile
 1: istore_1            # vider la pile dans la première variable
 2: iload_1             # empiler la premiere variable
 3: bipush        10    # mettre 10 sur la pile
 5: if_icmpge     14    # comparer les deux valeurs en sommet de pile
                        # si >= aller à l'adresse 14
 8: iinc          1,1   # ajouter 1 à la première variable
11: goto          2     # boucler : aller à l'adresse 2
14: ...
    ...
39: return
}
```