Objectif 03
Les pointeurs et la gestion de la mémoire

Aziz Salah
salah.aziz@uqam.ca

Département d'informatique
UQÀM

Automne 2013

Plan

Pointeur

Définition

Un *pointeur* est une variable dont la valeur est une adresse en mémoire

Syntaxe

```
<type> *<identificateur>;
```

Exemple

```
int *ptri;
float *ptrf;
```

- `ptri` est un pointeur de type `int *`
- `ptri` sert à retenir l'adresse (la référence) en mémoire d'une valeur de type `int`

Attention

"`int *ptri;`" permet de réserver en mémoire où stocker une adresse mais ne réserve pas l'endroit pointé (référencé)

Opérateur de référencement & ("adresse de")

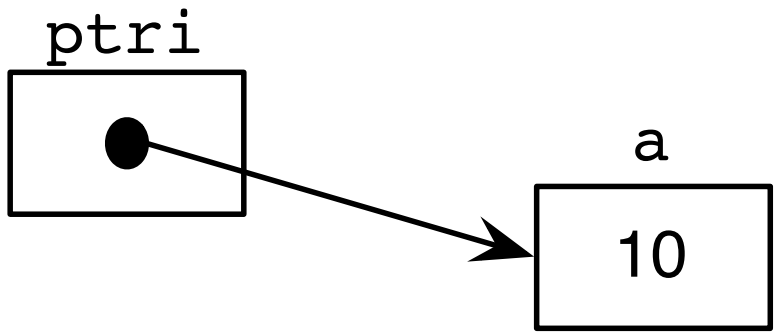
- L'opérateur `&` appliqué à une variable permet de retourner l'adresse mémoire de cette variable (attention : ne pas confondre avec opérateur conjonction bit à bit `&`)
- L'adresse d'une variable est du même type qu'un pointeur du type de la variable

Exemple

```
1 int *ptri;
2 int a = 10;
3 ptri = &a; //ptri et &a ont le même type qui est int *
4
5 scanf("%d",&a);   //La fonction scanf() exige des pointeurs pour ses
      paramètres
6
7 scanf("%d",ptri);
```

Représentation de la mémoire

```
1 int *ptri;
2 int a = 10;
3 ptri = &a;
```



Comment accéder à 10 à travers `ptri` ?

Opérateur d'indirection *

L'opérateur * permet l'accès au contenu de l'adresse pointée par un pointeur

```
1  int *ptri; //pointeur de int
2  int b, a = 10;
3  ptri = &a;  // *ptri devient la même chose que a
4
5  printf("%d",*ptri); //imprimer la valeur référencée par ptri
6
7  *ptri = 20; //la valeur à l'adresse ptri est 20
8              //la valeur référencée par ptri est 20
9
10 b = *ptri; //la valeur de b prend la valeur référencée par ptri (b
       devient 20)
11 printf("%p",ptri); //%p permet d'afficher la valeur de ptri
```

Pointeurs constants

Prérequis

```
const int c = 3;
const int t[3] = {10, 20, 30};
```

- La variable `c` est initialisée à 3 et ne peut être modifiée durant l'exécution par la suite
- Toute instruction essayant de modifier `c` donne lieu à une erreur de compilation

```
1 const int c = 3;
2 int v = 3;
3 int *b = &c; // invalid
4 int * const p = &v; // p est constant
5 const int *pc = &v; // *pc est constant mais pas v
6 const int  * const pcc = &c; //pointeur et pointé constants
```

Importance : protection des paramètres dans les fonctions

Plan

Passage de paramètres par référence

Toute toute en un petit exemple

```
3  int reset(int *ptr){
4      int old = *ptr ;
5      *ptr = 0 ;
6      return old ;
7  }
8  int
9  main()
10 {
11     int val = 33, valold ;
12
13     valold = reset(&val) ; // &val représente l'adresse de val
14
15     printf("%d %d\n",val,valold) ;    //affiche 0 33
```

Passage de paramètres par référence

Encore un exemple pour m'assurer

```
1  void
2  permuter(int *a, int *b){
3
4    int t ; //variable tonpon
5
6    t = *a ;
7    *a = *b ;
8    *b = t;
9
10 }
```

Peut-on faire l'appel "`permuter(p,&q);`" ? Dites oui !

Plan

Relation entre pointeurs et tableaux

- Le nom d'un tableau représente en fait l'adresse de son premier élément
  - Soit la déclaration : `type tableau[exp_entière];`
  - `&tableau[0]` et `tableau` sont syntaxiquement équivalents
  - `tableau` est un pointeur constant de type (`type const *`) référençant le premier élément du tableau (`tableau++;` invalide)
- Une expression utilisant les opérations avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs grâce à l'arithmétique des pointeurs
  - `tableau[i]` s'écrit aussi `*(tableau+i);`

Arithmétique des pointeurs

```
type tableau[exp_entière];
type *ptr, *ptrm, *ptrv;
int i,j;
...
```

- Après `ptr = tableau;`
  - `*ptr` ou `*tableau` désigne `tableau[0]`
  - `*(ptr+1)` désigne `tableau[1]`
  - `*(ptr+i)` désigne `tableau[i]`
- Après `ptrm = &tableau[i];`
  - `*ptrm` désigne `tableau[i]`
  - `*(ptrm+j)` désigne `tableau[i+j]` avec `j` un entier
- Après `ptrv = &tableau[i];`
  - `*ptrv++` ou `*(ptrv++)` retourne `tableau[i]` puis `ptrv` est incrémenté
  - `*++ptrv` ou `*(++ptrv)` retourne `tableau[i+1]` car `ptrv` est incrémenté d'abord

L'opérateur `sizeof`

- `sizeof` est un opérateur souvent évalué durant la compilation (exception de `int v[n];` avec `n` variable)
- `sizeof` retourne la taille occupée en mémoire par une variable, une expression ou un type
  - `sizeof (<expr>)` ou `sizeof <expr>`
  - `sizeof (<type>)`
- `sizeof` utilise `sizeof(char)` comme unité de mesure
- Le retour de `sizeof` est de type `size_t`, un entier sans signe défini dans `<stddef.h>`
- `sizeof` est important pour comprendre l'arithmétique des pointeurs
- `sizeof` sert lors de l'allocation dynamique de la mémoire

Exemple de tests avec `sizeof` I

Cas des types de base

```
6  printf("sizeof(char) : %zu\n", sizeof(char) );
7  printf("sizeof(int) : %zu\n", sizeof(int) );
8  printf("sizeof(long) : %zu\n", sizeof(long) );
9  printf("sizeof(long long) : %zu\n", sizeof(long long) );
10 printf("sizeof(float) : %zu\n", sizeof(float) );
11 printf("sizeof(double) : %zu\n", sizeof(double) );
12 printf("sizeof(long double) : %zu\n", sizeof(long double) );
13 printf("sizeof('a') : %zu  ;)\n", sizeof('a') );
14 printf("sizeof('a'+'a') : %zu\n", sizeof('a'+'a') );
15 printf("sizeof(2.0) : %zu\n", sizeof(2.0) );
16 printf("sizeof(4) : %zu\n", sizeof(4) );
17 int i; printf("int i; sizeof(i) : %zu\n", sizeof(i) );
18 i=10; int tabInt[i];
```

Exemple de tests avec `sizeof` I

Affichage (sur une machine 64bits)

```
sizeof(char) : 1
sizeof(int) : 4
sizeof(long) : 8
sizeof(long long) : 8
sizeof(float) : 4
sizeof(double) : 8
sizeof(long double) : 16
sizeof('a') : 4  ;)
sizeof('a'+'a') : 4
sizeof(2.0) : 8
sizeof(4) : 4
int i; sizeof(i) : 4
```

Exemple de tests avec `sizeof` II

Cas des pointeurs et tableaux

```
17 int i; printf("int␣i;␣sizeof(i)␣:␣%zu\n", sizeof(i) );
18 i=10; int tabInt[i];
19 printf("int␣tabInt[10]␣;␣sizeof(tabInt)␣:␣%zu\n", sizeof(tabInt));
20 int *pInt;
21 printf("int␣*pInt␣;␣sizeof(pInt)␣:␣%zu\n", sizeof(pInt));
22 char *pChar;
23 printf("char␣*pChar;␣sizeof(pChar)␣:␣%zu\n", sizeof(pChar));
24 double T[4][3];
25 printf("double␣T[4][3];␣sizeof(T):%zu;␣sizeof(*T):%zu;"
26     "␣sizeof(T[1]):%zu\n", sizeof(T), sizeof(*T),sizeof(T[1]));
27 double (* ptr)[3];
28 printf("double␣(*ptr)[3];␣sizeof(ptr)␣:␣%zu␣;␣sizeof(*ptr)␣:␣%zu\n",
29     sizeof(ptr),sizeof(*ptr)); //pointeur d'un tableau de 3 double
```

Exemple de tests avec `sizeof` II

Affichage (sur une machine 64bits)

```
int tabInt[10] ; sizeof(tabInt) : 40
int *pInt ; sizeof(pInt) : 8
char *pChar; sizeof(pChar) : 8
double T[4][3]; sizeof(T):96; sizeof(*T):24; sizeof(T[1]):24
double (*ptr)[3]; sizeof(ptr) : 8 ; sizeof(*ptr) : 24
```

Importance de l'arithmétique des pointeurs

- Un pointeur représente une adresse
- L'arithmétique des pointeurs tient compte de la taille des données lors des calculs d'adresses effectués principalement avec les opérateurs `++` `--` `+` et `-`
- Étant donné `p` un pointeur,
  - `p+1` pointe la donnée à l'adresse :
    `valeur(p) + sizeof(type_pointé_par_p)`
  - `p+n` pointe la donnée l'adresse :
    `valeur(p) + sizeof(type_pointé_par_p) * n`

Faites vos jeux !

Code

```
6     int tableau[] = { 7,6 , -10, 19, 0, 33 };
7     int *ptr = tableau;
8
9     while(*++ptr) //++ s'applique avant le test
10        printf("[%d] ",*ptr) ;
11    printf("%d\n",*ptr) ;    // ici ptr pointe 0
12
13    ptr = tableau;
14    while(*ptr++) //++ s'applique après le test
15        printf("[%d] ",*ptr) ;
16    printf("%d\n",*ptr) ;    // ici ptr pointe 33
```

Trace d'exécution

```
[6] [-10] [19] 0
[6] [-10] [19] [0] 33
```

Représentation d'un tableau 2D en mémoire

```
un_type T[4][3];
```

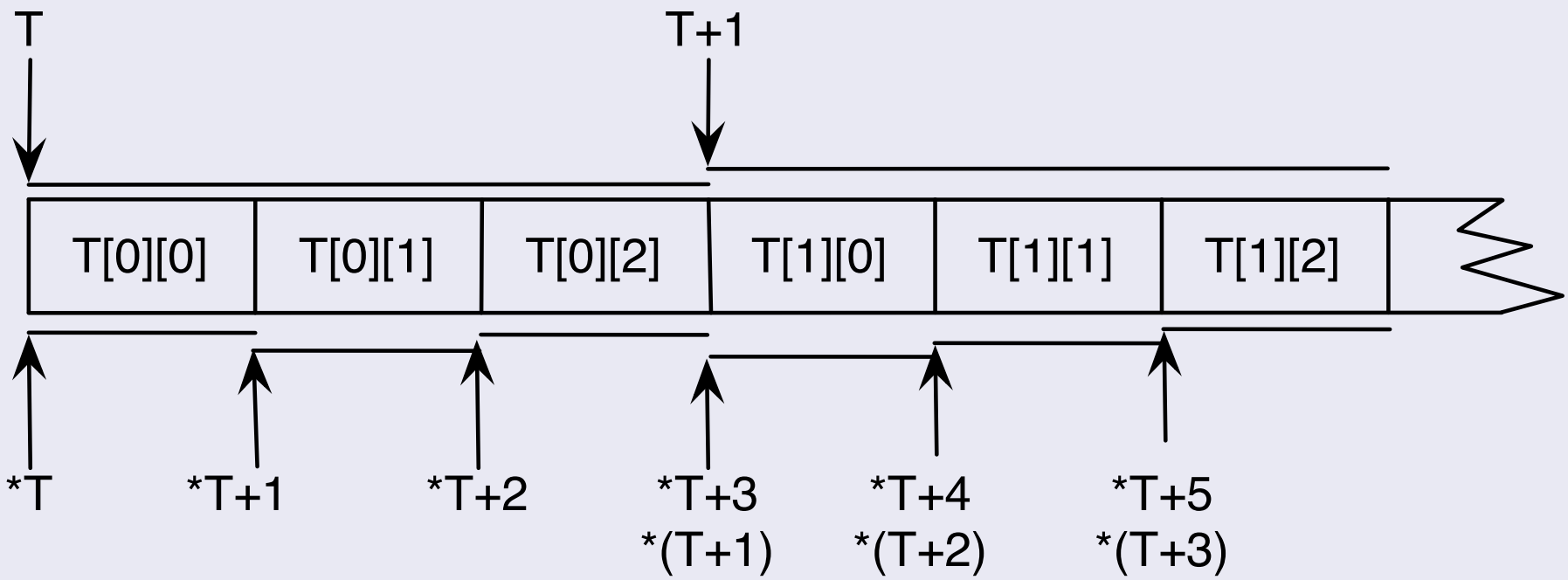
T[0][0]	T[0][1]	T[0][2]
T[1][0]	T[1][1]	T[1][2]
T[2][0]	T[2][1]	T[2][2]
T[3][0]	T[3][1]	T[3][2]

Représentation conceptuelle du tableau T

Représentation interne en mémoire du tableau T (ligne 1 suit ligne 2)

Pointeur sur le début d'une ligne

`int` T[4][3] ;



T[1] ou *(T+1) est constitué de 3 `int`

Arithmétique de pointeurs et tableaux 2D I

Alignement mémoire

```
7  int T[4][3] = {{0, 1, 2},{10,11,12},{20,21,22},{30,31,32}};
8
9  printf("|   T : %p |     *T : %p |   T[0] : %p |\n", T, *T, T[0]);
10 printf("| T+1 : %p | *T+1 : %p | T[0]+1 : %p |\n", T+1, *T+1, T[0]+1);
```

Affichage

```
   T:0x7fff54572a28   *T:|0x7fff54572a28|   T[0]:|0x7fff54572a28|
 T+1:0x7fff54572a34 *T+1:|0x7fff54572a2c| T[0]+1:|0x7fff54572a2c|
-------------------------------------------------------------------
 +12 : 3*sizeof(int)    +4 : sizeof(int)      +4 : sizeof(int)
```

Arithmétique de pointeurs et tableaux 2D II

Écritures équivalentes pour éléments d'un tableau 2D

```
15 int T[4][3] = {{0, 1, 2},{10,11,12},{20,21,22},{30,31,32}} ;
16 int *p;
17
18 int i=1;
19 p = T[i]; //car p[j] represente T[i][j]
20
21 for(int j=0; j< 3; j++) //Affiche |10|10|10| |11|11|11| |12|12|12|
22    printf( "|%d|%d|%d|␣", *(p+j), p[j], T[i][j]) ;
```

Arithmétique de pointeurs et tableaux 2D III

Écritures équivalentes pour éléments d'un tableau 2D

```
26 int T[4][3] = {{0, 1, 2},{10,11,12},{20,21,22},{30,31,32}} ;
27
28 for(int i = 1, j  = 0; j< 3; j++) //Affiche |10|10|10| |11|11|11|
       |12|12|12|
29     printf( "|%d|%d|%d|␣", T[i][j] , *(T[i]+j), *( *(T+i) + j ) );
```

Arithmétique de pointeurs et tableaux 2D IV

Pointeur de lignes d'un tableau 2D

```
34 int T[4][3] = {{0, 1, 2},{10,11,12},{20,21,22},{30,31,32}} ;
35 int (*ligneptr)[3]; //ligneptr est un pointeur d'un tableau de 3 entiers
       (ligneptr de type int (*)[3])
36
37 // int *ptrT[3];  donne que ptrT soit un tableau de 3 pointeurs d'entier
38
39 ligneptr = &T[1];//car ligneptr[0] représente T[1] et donc ligneptr[0][j
       ] représente T[1][j]
40
41 for(int j=0; j< 3; j++) //Affiche |10|10|10| |11|11|11| |12|12|12|
42    printf( "|%d|%d|%d|| ", ligneptr[0][j] , *(*ligneptr + j ) , (*
          ligneptr)[j] ) ;
43
44 ligneptr++; //passer à la ligne suivante
45 for(int j=0; j< 3; j++) //Affiche |20| |21| |22|
46    printf( "|%d| ", *( *ligneptr + j )) ;
```

Arithmétique de pointeurs et tableaux 2D V

Parcours linéaire d'un tableau 2D

```
50 int T[4][3] = {{0, 1, 2},{10,11,12},{20,21,22},{30,31,32}} ;
51
52 for(int i=0; i< 5; i++) //Affiche |0|1|2|10|11
53    printf( "|%d", *(*T+i)) ;
```

Bonnes Pratiques

- Privilégiez les tableau par rapport aux pointeurs pour avoir l'aide du compilateur
- Adoptez une nomenclature des variables qui permet de distinguer facilement les pointeurs des tableaux (nom de pointeur avec ptr)

Plan

Chaine de caractères

Définition

Une chaine de caractères (string) est un tableau de caractères terminé avec le caractère `'\0'`

Exemples

```
char mot[10]="Bonjour";
```

'B'	'o'	'n'	'j'	'o'	'u'	'r'	'\0'		

`printf("%s",mot);` // affiche `"Bonjour"`

`mot[3]='\0';`

'B'	'o'	'n'	'\0'	'o'	'u'	'r'	'\0'		

`printf("%s",mot);` // affiche `"Bon"`

`printf("%s",&mot[5]);` // affiche `"ur"`

```
char * str = "Bonne journée";
```

`printf("%s",str);` // affiche `"Bonne journée"`

`str[3]='\0';` // arrête l'exécution du programme car la zone pointée par `str` ne peut être modifiée (`Bus error`) ! !

Bonnes pratiques

```
1 #define LONG_MAX 128
2 ...
3 char mot[LONG_MAX+1] = "bonjour";//Explicitement +1 pour '\0'
4
5 gets(mot); //c'est  risqué
6
7 fgets(0,mot,LONG_MAX); // c'est sécuritaire
```

Plan

Fonction avec un tableau comme paramètre

```c
#include <stdio.h>
int min_tab(int tab[] , int taille)
{
   int i, min;
   for (min = tab[0], i = 1 ; i < taille ; i++)
      if (min > tab[i])
           min = tab[i] ;
   return(min);
}
int
main ()
{
    int t[] = {45 -2, 6, 19};
    int n = sizeof(t)/sizeof(int); //la taille du tableau t
    int resultat = min_tab(t, n); //Appel de la fonction min_tab()
    printf("taille : %d, resultat : %d\n", n, resultat);
    return 0;
}// Affichage taille : 3, resultat : 6
```

Fonction avec un pointeur (tableau) comme paramètre I

```
1  #include <stdio.h>
2  int min_tab(int *tab, int taille)
3  {
4      int min;
5      int *limite = tab + taille;
6      for (min = *tab; tab < limite ; tab++)
7          if (min > *tab)
8              min = *tab ;
9      return(min);
10 }
```

Fonction avec un pointeur (tableau) comme paramètre II

Suite

```
11 int
12 main ()
13 {
14     int *p, t[] = {45, -2, 6, 19};
15     int n = sizeof(t)/sizeof(int); //n est la taille du tableau t
16     p = t;
17     int n2 = sizeof(p)/sizeof(int);//n2 n'est pas la taille du tableau!
18
19     int resultat = min_tab(t, n); //Appel de la fonction min_tab()
20
21     printf("n2 : %d, n : %d, resultat : %d\n", n2, n, resultat);
22
23     return 0;
24 } // Affichage taille : 4, resultat : -2
```

Usage de `const` pour les paramètres

```
2  int min_tab(const int tab[] , int taille)
3  {
4     int i, min;
5     //-----------------
6     tab[0]=2;          // cette instruction donne erreur de compilation
7     //-----------------
8     for (min = tab[0], i = 1 ; i < taille ; i++)
9        if (min > tab[i])
10            min = tab[i] ;
11    return(min);
12 }
```

Bonne pratique de programmation

Cette fonction n'est pas censée modifier les éléments de `tab` alors on les protège avec `const`

Fonction avec tab 2D comme paramètre (définition)

```
3  int somme(int M[][3],int lig, int col){
4      int res = 0;
5
6      for(int i=0; i<lig; i++)
7          for(int j=0; j<col; j++)
8              res += M[i][j];
9
10     return res;
11 }
12
13 int sommep(int (*ptrl)[3],int lig, int col){
14     int res = 0;
15
16     for(int i=0; i<lig; i++)
17         for(int j=0; j<col; j++)
18             res += ptrl[i][j];
19
20     return res;
21 }
```

Fonction avec tab 2D comme paramètre (appel)

```
23 int
24 main()
25 {
26     int T[4][3] = {{0, 1, 2},{10,11,12}}; //Le compilateur complète avec
           des 0 pour [4][3]
27     printf( "somme() : |%d| - sommep() : |%d|\n", somme(T,4,3), sommep(T
           ,4,3) ) ;
28
29     int U[3][3] = {{0, 1, 2},{10,11,12}};//Le compilateur complète avec
           des 0 pour [3][3]
30     printf( "somme() : |%d| - sommep() : |%d|\n", somme(U,4,3), sommep(U
           ,4,3) ) ;
31
32     return 0;
33 }
34 //Affichge :
35 //somme() : |36| - sommep() : |36|
36 //somme() : |39| - sommep() : |39|  //cette ligne est inattendue!!
```

Plan

Fonction des strings par l'exemple

```c
#include <stdio.h>
#include <string.h> //déclare strcpy(), strcat(), etc

int main (int argc, char const *argv[])
{
    char * str = "Bonne journée";
    char ch1[20], ch2[50];

    strcpy(ch1,str);
    puts(ch1); //Affiche : Bonne journée

    strcpy(ch2,"Je vous aime");
    puts(ch2);//Affiche : Je vous aime
    strcat(ch2," et je vous respecte.");
    puts(ch2); //Affiche : Je vous aime et je vous respecte.
    return 0;
}
```

Fonctions des strings de la librairie C standard

<string.h>

- `size_t strlen(const char *s);` // retourne le nombre de caractères dans `s` sans le caractère nul.
- `char *strcpy(char *restrict s1, const char *restrict s2);` // copie `s2` dans `s1` et retourne `s1`
- `char *strncpy(char *restrict s1, const char *restrict s2, int n);` // copie au plus les `n` premiers caractères de `s2` dans `s1`, y place le caractère nul à sa fin et retourne `s1`
- `char *strcat(char *restrict s1, const char *restrict s2);` // ajoute `s2` à la fin de `s1`, place le caractère nul et retourne `s1`
- `char *strncat(char *restrict s1, const char *restrict s2, int n);` // ajoute au plus les `n` premiers caractères de `s2` dans `s1` et y place le caractère nul à sa fin.

Fonctions des strings de la librairie C standard

<string.h> (suite)

- `int strcmp(const char *s1, const char *s2);//` retourne 0 si `s1` et `s2` sont identiques ou bien un nombre >0 si `s1>s2` et un nombre <0 sinon
- `int strncmp(const char *s1, const char *s2, int n);//` compare au plus les n premiers caractères

- Toutes ces fonctions supposent disponibles les zones mémoire où copier ou concaténer
- C'es la responsabilité du programmeur de réserver l'espace nécessaire avant d'appeler ces fonctions
- Pour plus de détail ou d'autres fonctions, consultez les pages man de `string.h`

Plan

Allocation dynamique ?

- L'allocation dynamique désigne la réservation d'une zone mémoire dans le *tas* (*heap*) pour une donnée au moment de l'exécution du programme.
- Il n'est pas toujours facile de décider statique de la bonne taille d'une donnée
- Les zones mémoire des variables locales à une fonction se situent dans la pile (pas dans le tas) et leurs durées de vie sont restreintes à la durée d'exécution de leur fonction
- La durée de vie des zones mémoire allouées dans le tas est la durée de vie du programme !

Allocation dynamique dans le language C

L'allocation dynamique fait appel aux fonctions déclarée dans `<stdlib.h>` :

- `void *malloc(size_t size);`
  retourne un pointeur sur une zone de taille `size` et NULL en cas d'échec
- `void *calloc(size_t count, size_t size);`
  retourne un pointeur sur une zone de taille `size*count` initialisée à zéro
- `void free(void *ptr);`
  libère la zone pointée par `ptr` (précédemment allouée avec malloc())
- `void *realloc(void *ptr, size_t size);`
  Change la taille de la zone pointée par `ptr` à la nouvelle taille `size` ou la déplace et la recopie ; et retourne un pointeur sur la nouvelle zone

(Voir les pages man pour les détails)

`void *` type de pointeur générique compatible avec tous les pointeurs

Exemple simple avec `malloc()`

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

int main (int argc, char const *argv[])
{
    int *pi;
    int taille = 10;
    pi = malloc(taille*sizeof(int));
    for (int i = 0; i < taille ; i++)
        pi[i]= i*i;

    for (int *p=pi ; p < taille+pi ; p++)
         printf("%d ",*p);

    free(pi);
    return 0;
}
// Affiche : 0 1 4 9 16 25 36 49 64 81
```

Quelle est l'intruse ?

L'intruse

```
1 char * cloner(char c,int n){
2     char clone[n+1];
3     for(int i = 0 ; i<n ; i++){
4         clone[i]=c;
5     }
6     clone[n] = '\0';
7     return clone;
8 }
```

La bonne

```
1 char * cloner(char c,int n){
2     char *ptrclone = malloc(n+1)
          ;
3     for(int i = 0 ; i<n ; i++){
4         ptrclone[i]=c;
5     }
6     ptrclone[n] = '\0';
7     return ptrclone;
8 }
```

Message du compilateur pour repérer l'intrus

```
warning: function returns address of local variable
```

Bonnes pratiques

Cloner une chaine de caractères

```
char * strcloner(const char *s){

   return strcpy( malloc(strlen(s) + 1), s);

}
```

Soyez explicite

- `const char *s`
- `malloc(strlen(s) + 1)`

Plan

Arguments de programme I

Exigence

```
% ./a.out "ab cd" 'ef 4' gg  e
Le nombre de paramètres est 5
argv[0]=a.out
argv[1]=ab cd
argv[2]=ef 4
argv[3]=gg
argv[4]=e
a.out
ab cd
ef 4
gg
e
```

Arguments de programme II

Une réalisation

```
1  #include <stdio.h>
2
3  int main (int argc, char *argv[])
4  // ou bien int main(int argc, char **argv)
5  {
6      printf("Le nombre de paramètres est %d\n",argc);
7      for(int i = 0 ; i < argc ; i++) {
8          printf("argv[%d]=%s\n",i, argv[i]);
9      }
10
11     argv[1][0] = 'Y';
12     puts(argv[1]);
13     do
14     {
15         printf("%s\n", *argv);
16     }
17     while (*++argv);
18
```

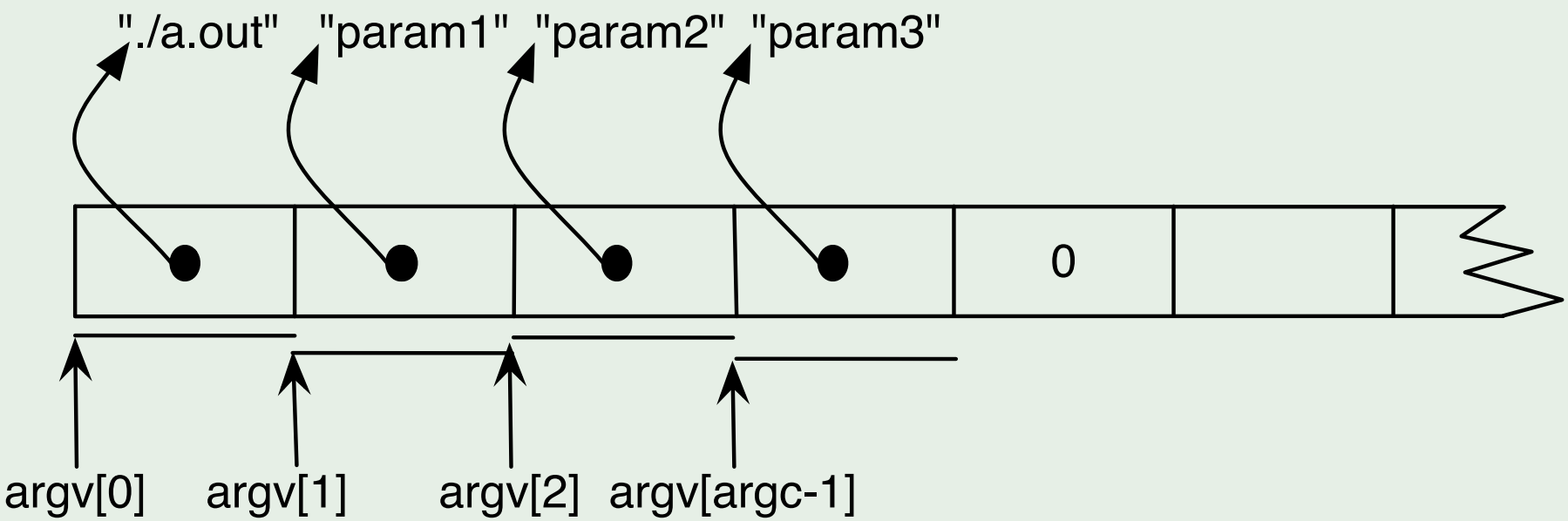
A retenir

- Prototype de la fonction main()

```
int main (int argc, char const *argv[])
```

- La commande à exécuter

```
% ./a.out param1 param2 param3
```



Une petite détente

Exercice

Comparez ces trois déclarations

- `char motConst[] = "bonjour papa!";`
- `char mot20[20] = "bonjour papa!";`
- `char *motEtoile = "bonjour papa!";`

La grande détente

Solution de l'exercice

```
5     char motConst[]="bonjour␣papa!";
6     char mot20[20]="bonjour␣papa!";
7     char *motEtoile="bonjour␣papa!"; // Soyez explicite char const *
8
9     printf("sizeof(motConst)␣:␣%zu\n", sizeof(motConst));//14
10    printf("sizeof(mot20)␣:␣%zu\n", sizeof(mot20));//20
11    printf("sizeof(motEtoile)␣:␣%zu\n", sizeof(motEtoile));//8
12
13    motConst[0] = 'A'; //c'est permis
14    //motConst++; non permis
15    mot20[0] = 'A'; // c'est permis
16    //mot20++; non permis
17    //motEtoile[0]='A';non permis
18    motEtoile++; //c'est permis
```

Allez on reprend !

Exercice

- Comparez ces deux déclarations

```
int m[5][7];

int * v[5];
```

- Et celles-là

```
char *name[]={"toi","moi","toi","lui","elle"};

char name[][20]= {"toi","moi","toi","lui","elle"};
```

Solution

```
char *name[]= {"toi","moi","toi","lui","elle"};
```

```
24      char *name[]={"toi","moi","toi","lui","elle"};
25      printf("sizeof(name) : %zu\n", sizeof(name));//32
26      //name est tab de 5 pointeurs de char
27
28      // name ++; est invalide
29      // name[0][0]='R'; est invalide
30      printf("%c\n",name[0][0]);//affiche t
31      for(int i = 0 ; i <  5 ; i++)
32          printf("adresse : %p valeur %s\n", name[i], name[i]);
33      puts("----");
34
35      for(char **ptr = name ; ptr < name + 5 ; ptr++)
36          printf("adresse : %p valeur %s\n", *ptr, *ptr);
```

```
char *name[]= {"toi","moi","toi","lui","elle"};
```

```
adresse : 0x10896eeb8 valeur : toi
adresse : 0x10896eebc valeur : moi
adresse : 0x10896eeb8 valeur : toi
adresse : 0x10896eec0 valeur : lui
adresse : 0x10896eec4 valeur : elle
----
adresse : 0x10896eeb8 valeur : toi
adresse : 0x10896eebc valeur : moi
adresse : 0x10896eeb8 valeur : toi
adresse : 0x10896eec0 valeur : lui
adresse : 0x10896eec4 valeur : elle
```

Solution

```
char name[][20]= {"toi","moi","toi","lui","elle"};
```

```
41      char name[][20]= {"toi","moi","toi","lui","elle"};
42      printf("sizeof(name) : %zu\n", sizeof(name));//80
43      //name est tab 2D de char
44
45      // name ++; est invalide
46      for(int i = 0 ; i <  5 ; i++)
47          printf("adresse : %p valeur %s\n", name[i], name[i]);
48      puts("----");
49
50      for(char (*ptr)[20] = name ; ptr < name + 5 ; ptr++)
51          printf("adresse : %p valeur %s\n", ptr, ptr);
```

Solution

```
char name[][20]= {"toi","moi","toi","lui","elle"};
```

```
adresse : 0x7fff57291964 valeur : toi
adresse : 0x7fff57291978 valeur : moi
adresse : 0x7fff5729198c valeur : toi
adresse : 0x7fff572919a0 valeur : lui
adresse : 0x7fff572919b4 valeur : elle
----
adresse : 0x7fff57291964 valeur : toi
adresse : 0x7fff57291978 valeur : moi
adresse : 0x7fff5729198c valeur : toi
adresse : 0x7fff572919a0 valeur : lui
adresse : 0x7fff572919b4 valeur : elle
```

Conclusion : mode d'emploi

Les pointeurs deviennent faciles

1. Si vous faites un bon schéma de la mémoire
2. Si vous maitrisez leur arithmétique
3. Si vous prévoyez l'allocation dynamique requise
4. Si vous vous pratiquez