Chapitre 2

September 22, 2022

1 Chapitre 1 - Concepts de Programmation

2 Chapitre 2 - Aspects Mémoire: les spécificités du C

2.1 La mémoire vive (RAM)

On se représente la mémoire vive comme une succession de "cases mémoire" qui chacune contient un octet. Un octet est une chaine de 8 bits (un bit vaut 0 ou 1). Les cases sont numérotées de 0 à la taille de la RAM, le numéro d'une case s'appelle son *adresse* dans la mémoire. Les adresses sont donc des entiers, qui il y a peu était stockés sur 32 bits. Par conséquent la taille de la RAM était limité à $2^{32} - 1 = 4294967295$, soit un peu plus de 4 Giga octets. Entre 2000 et 2010 s'est donc opéré un changement du système d'adressage: les adresses sont maintenant encodées sur 64 bits.

2.2 Les pointeurs

Si "var" est une variable, alors

`&var`

désigne l'adresse mémoire de cette variable dans la RAM. Cette adresse n'est rien d'autre qu'un entier 64 bits, pour autant le type de l'expression "&var" n'est pas "long" mais "t*" où t désigne le type de de var.

La raison est la suivante: si "var" est de type "int", alors elle n'a pas **une** adresse mémoire mais **quatre** adresses mémoire, en effet un "int" occupe 4 octets en mémoire.

[ ]:
```
int var = 12 ;
int*  p = &var ;
printf("var = %d;\np = %p;\n(long)p = %lu", var, p, (long)(p)) ;
```

La variable "p" désigne donc l'adresse de la première case mémoire occupée par "var" et le type "int*" permet de savoir que la variable var est stockée à cette adresse **plus les trois suivantes**.

Remarque: le système d'adressage virtuel de ce serveur est très déterministe. Pour tous les exemples qui affichent des adresses, il faut privilégier l'exécution en environnement réel. Pour l'exemple ci-dessus, observer l'adresse retournée sur plusieurs exécution (cours2-0.c).

Une variable qui contient une adresse mémoire (telle que "p" ci dessus) s'appelle un **pointeur**: il pointe vers une autre adresse en mémoire. Etant donné un pointeur, on peut récupérer la valeur contenue dans la case mémoire qu'il pointe comme ci dessous, on parle de **déréférencement de pointeur**.

[ ]:
```c
int var = 12 ;
int* p = &var ;
int varbis = *p ;
printf("La variable var vaut %d\n", var) ;
printf("La variable varbis vaut \"le contenu de la case pointee par p\", soit␣
 ↪%d aussi", varbis) ;
```

La présence de pointeur dans un programme rend son exécution un peu plus difficile à suivre.

Exercice: prédire les valeurs des différentes variables du programme suivant en fin d'exécution.

[ ]:
```c
int entier1 = 3;
int entier2 = 5;
int entier3 = 8;
int* pointeur1 = &entier1;
int* pointeur2 = &entier2;
int* pointeur3 = &entier3;
pointeur1 = pointeur3;
*pointeur3 = 10;
entier2 = *pointeur1 + 7;
```

[ ]:
```c
printf("entier1: %d \nentier2: %d \nentier3: %d\n", entier1, entier2, entier3) ;
printf("Adresse de Entier1: %p \n", &entier1) ;
printf("Adresse de Entier2: %p \n", &entier2) ;
printf("Adresse de Entier3: %p \n", &entier3) ;
printf("Adresse pointée par pointeur1: %p \n", pointeur1) ;
printf("Adresse pointée par pointeur2: %p \n", pointeur2) ;
printf("Adresse pointée par pointeur3: %p \n", pointeur3) ;
```

Illustration PythonTutor: cours2-32.c.

Résumé: - Si var est de type t alors son adresse &var est de type t. - *Si p est un pointeur de type t*, alors *p est une valeur de type t.

Attention: un pointeur p contient une adresse, mais il a lui même une adresse (comme toute variable !) et les deux ne coincident pas:

[ ]:
```c
int a = 12 ;
int* p = &a ;
printf("Adresse de p: %p \nValeur de p = adresse de a: %p", &p, p) ;
```

On dit du C que c'est un langage **bas niveau** (*proche de la machine*): les instructions se traduisent *relativement simplement* en langage machine. Cela contraste avec les langage **haut niveau** (comme Python) où certaines instructions sont non élémentaires: une seule instruction sera traduite en un grand nombre d'instructions machine. De plus, avec les pointeurs, on a un accès direct à la mémoire, ce qui permet d'être très précis dans les manipulations effectuées.

L'avantage du C est que l'on a alors un meilleur contrôle sur ce qui sera réellement exécuté par la machine (cependant le *compilateur* effectue d'énormes optimisations à la compilation).

Enfin, un langage **haut niveau** se distingue du C par la présence de davantage de *structures avancés* natives: en C il n'y a ni listes, ni séquences, ni dictionnaires (comme vous avez peut être vu en Python), encore moins les arbres ou les types récursifs comme on verra en OCaml. Il n'y a que trois structures *non élémentaires* en C que nous allons aborder: les tableaux, les chaines de caractères et les STRUCT déjà rencontrées. Et nous allons voir que leur manipulation nécessite des précautions qui sont gérées **automatiquement** dans des langaages plus haut niveau.

Les pointeurs sont à la fois un avantage et un danger: ils permettent une grande liberté, mais leur manipulation engendrent facilement des erreurs.

[ ]:
```c
// Le programme de MP2I mentionne que l'arithmetique des pointeurs est hors programme.
// La compréhension du code ci dessous n'est pas tout à fait exigible.
char c1 = 'a' ;
char c2 = 'a' ;
char c3 = 'a' ;
char c4 = 'a' ;
int a = 12 ;
int b = 24 ;

printf("Contenu de la case a droite de a: %d\n", *(&a+1)) ; // Ligne hors programme
*(&a+1) = 13 ; // Ligne hors programme
printf("Valeur de b après L11: %d\n", b) ;
printf("Contenu la case a gauche de a: %d\n", *(&a-1)) ; // Ligne hors programme
printf("Codage du caractere \'a\': %d\n", (int)(c1)) ;
printf("97 en binaire: 01100001\n") ;
printf("Le nombre binaire 01100001 01100001 01100001 01100001 s'ecrit en decimal: 1633771873") ;
```

Lorsque le programme s'exécute, l'OS ne lui alloue qu'une partie (un segment) de la RAM de l'ordinateur. Si le programme tente d'accéder à une adresse "qui ne lui appartient pas", il échoue avec l'erreur "Segmentation Fault". C'est typiquement le cas si vous déclarez un pointeur sans le définir; alors sa valeur est 0 par défaut et l'adresse nulle est invalide par convention. On utilise généralement la constante NULL (en majuscule en C) pour désigner le pointeur nul, qui pointe vers l'adresse 0 (cours2-1.c).

[ ]:
```c
int* p ;
printf("%d",*p) ;
```

L'interpréteur C de ce serveur est beaucoup trop poli. Comparer avec cours2-1.c en environnement réel.

[ ]:
```c
int* p;
int* q = 0 ;
int* r = NULL ;
printf("%p %p %p\n",p,q,r) ;
```

```c
if (p == q && q == r)
    printf("0 == NULL == initialisation par defaut\n") ;
```

Si l'on tire un entier au hasard, avec grande probabilité il désigne une adresse hors du segment de RAM alloué au programme (cours2-2.c).

[ ]:
```c
bool choice = true ;
long n ;
if (choice)
    n = 140634921422984 ;
else
    n = 250634921422984 ;
int* p = (int*)(n) ;
*p = 12 ;
printf("%d", *p) ;
```

2.3 Gestion des variables en mémoire: la pile.

2.3.1 Adresses des variables

Lorsque l'on crée des variables, elles sont stockées sur la pile. Le nom est explicite: imaginez une pile de livre, et à chaque acquisition d'un nouveau livre vous vous contentez de le poser sur la pile existante.

[ ]:
```c
void printBool (bool b)
{
    b? printf("true "):printf("false ") ; // Syntaxe hors programme
}
```

[ ]:
```c
void printAdd (char* name, void* p)
{
    printf("Adresse de %s: %p %ld\n", name, p, (long)(p)) ;
}
```

[ ]:
```c
// On déclare 5 variables de type CHAR
char a ;
char b ;
char c ;
char d ;
char e ;

// On affiche les adresses: &a = l'adresse de la variable a.
// ignorer la conversion (char*): c'est à cause du C++ !
// voir la version C dans cours2-6.c
printAdd( (char*)"a", &a) ;
printAdd( (char*)"b", &b) ;
printAdd( (char*)"c", &c) ;
printAdd( (char*)"d", &d) ;
```

```
printAdd( (char*)"e", &e) ;


// On constate que les adresses sont côte à côte:
printf("\n&b == &a + 1: ") ;
printBool(&b == &a + 1);
printf("\n&c == &b + 1: ") ;
printBool(&c == &b + 1) ;
printf("\n&d == &c + 1: ") ;
printBool(&d == &c + 1) ;
printf("\n&e == &d + 1: ") ;
printBool(&e == &d + 1) ;
```

A chaque exécution d'un programme, l'OS (operating system) alloue un espace mémoire pour la pile. Cet espace change à chaque exécution: notez que lorsque vous réexécutez la cellule précédente, les adresses allouées changent.

L'allocation mémoire est la raison principale pour laquelle on doit donner le type d'une variable lorsqu'on la déclare: le système doit savoir quelle place il réserve pour cette variable:

[ ]:
```
// Observons la place que prennent en mémoire les variables suivantes, de type␣
 ↪différents:
int z ;
char a1 ;
char a2 ;
char a3 ;
char a4 ;
int b ;
float c ;
long d ;
double e ;
char f ;


// On affiche les adresses:
printAdd((char*)"a4", &a4) ;
printAdd((char*)"b", &b) ;
printAdd((char*)"c", &c) ;
printAdd((char*)"d", &d) ;
printAdd((char*)"e", &e) ;
printAdd((char*)"f", &f) ;


// On en déduit les tailles des types associés:
printf("\nUn CHAR est stocke \t sur 1 octet  ( 8 bits): &b - &a = %ld",␣
 ↪(long)&b - (long)&a4) ;
printf("\nUn INT \t \t \t sur 4 octets (32 bits): &c - &b = %ld", (long)&c -␣
 ↪(long)&b) ;
```

```
printf("\nUn FLOAT \t \t sur 4 octets (32 bits): &d - &c = %ld", (long)&d -␣
 ↪(long)&c) ;
printf("\nUn LONG \t \t sur 8 octets (64 bits): &e - &d = %ld", (long)&e -␣
 ↪(long)&d) ;
printf("\nUn DOUBLE  \t \t sur 8 octets (64 bits): &f - &e = %ld",(long)&f -␣
 ↪(long)&e) ;
```

Inutile de connaitre ces valeurs par coeur:

[ ]:
```
printf("Taille d'un CHAR: %lu\n", sizeof(char)) ;
printf("Taille d'un INT: %lu\n", sizeof(int)) ;
printf("Taille d'un FLOAT: %lu\n", sizeof(float)) ;
printf("Taille d'un LONG: %lu\n", sizeof(long)) ;
printf("Taille d'un DOUBLE: %lu\n", sizeof(double)) ;
```

2.3.2 Portée des variables (l'explication)

[ ]:
```
void f ()
{
    int a = 11;
    int b = 12;
    int c = 13;
    printf("Adresse de a (dans fct f): %p %lu\n", &a, (long)(&a)) ;
    printf("Adresse de b (dans fct f): %p %lu\n", &b, (long)(&b)) ;
    printf("Adresse de c (dans fct f): %p %lu\n", &c, (long)(&c)) ;
}
```

[ ]:
```
int main ()
{
    int a = 2;
    int b = 3;
    int c = 4;
    printf("Adresse de a (dans main) : %p %lu\n", &a, (long)(&a)) ;
    printf("Adresse de b (dans main) : %p %lu\n", &b, (long)(&b)) ;
    printf("Adresse de c (dans main) : %p %lu\n", &c, (long)(&c)) ;
    printf("\n") ;
    f () ;
    printf("\n") ;
    printf("Adresse de a (dans main) : %p %lu\n", &a, (long)(&a)) ;
    printf("Adresse de b (dans main) : %p %lu\n", &b, (long)(&b)) ;
    printf("Adresse de c (dans main) : %p %lu\n", &c, (long)(&c)) ;
}
```

[ ]:
```
main () ;
```

Le phénomène est (presque) le même lorsque l'on déclare une variable au sein d'une boucle FOR:

[ ]:
```
int myVar = -1 ;
printf("%lu\n\n", (long)(&myVar)) ;
for (int i = 0; i < 12; i++)
{
    int myVar = i ;
    printf("Adresse a l'interieur de la boucle: %p %lu\n", &myVar,␣
 ↪(long)(&myVar)) ;
}
printf("\nAdresse a l'exterieur de la boucle: %p %lu\n", &myVar,␣
 ↪(long)(&myVar));
```

Retenez cependant: il est rarement justifié de définir une nouvelle variable dans une boucle FOR, on peut la déclarer en amont et la mettre à jour dans la boucle.

Les variables declarées dans la boucle sont depilées a chaque exécution de la boucle:

[ ]:
```
int main ()
{
  int var = -1 ;
  for (int i = 0; i < 12; i++)
    {
      printf("%d ", var) ;
      int var = i ;
    }
}
main () ;
```

Exercice: le code ci-dessous est-il correct ? Pensez en terme de graphe de flot de contrôle.

[ ]:
```
int main ()
{
  for (int i = 0; i < 12; i++)
    {
      if (i != 0)
        printf("%d\n", var) ;
      int var = i ;
    }
}
```

2.4 Tableaux (1ere entrevue)

2.4.1 Manipulation naïve

Déclaration d'un tableau:

[ ]:
```
int tab[10] ;
```

Initialisation d'un tableau:

[ ]:
```
// 1. A la main
int tab[10] ;
for (int i = 0; i < 10; i++)
    tab[i] = i ;

// 2. En listant les éléments:
int tab2[] = { 1,2,3,4,5,6,7,8,9,10 } ;

// 3. En listant les éléments avec taille explicite:
int tab3[10] = { 1,2,3,4,5,6,7,8,9,10 } ;
```

On ne peut pas lister plus d'éléments qu'annoncé:

[ ]:
```
int tab4[5] = { 1,2,3,4,5,6,7,8,9,10 } ;
```

Mais on peut en lister moins:

[ ]:
```
int tab4[15] = { 0,1,2,3,4,5,6,7,8,9 } ;
```

Accès aux éléments d'un tableau:

`tab[i]`

[ ]:
```
printf("tab = ") ;
for (int i = 0; i < 10; i++)
    printf("%d ", tab[i]) ;
printf("\ntab2 = ");
for (int i = 0; i < 10; i++)
    printf("%d ", tab2[i]) ;
```

Gros détail: il n'existe pas de fonction qui permettent de connaitre la taille d'un tableau. Mis à part cela, vous avez ci-dessus suffisament de matière pour manipuler les tableaux comme vous le faisiez en Python (or do you ?):

[ ]:
```
int sumTab (int longueur, int tab[])
{
    int sum = 0 ;
    for (int i = 0; i < longueur; i++)
        sum = sum + tab[i] ;
    return(sum) ;
}
printf("Somme des elts de tab: %d\n",sumTab(10,tab)) ;
printf("Somme des elts de tab2: %d\n",sumTab(10,tab2)) ;
```

Et si on se trompe dans la longueur ?

[ ]:
```
printf("Somme des elts de tab: %d\n",sumTab(5,tab)) ;
```

Et si on se trompe de l'autre côté ? (à exécuter en env réel: cours2-3.c)

[ ]:
```
printf("Somme des elts de tab: %d\n",sumTab(15,tab)) ;
printf("Somme des elts de tab4: %d\n",sumTab(15,tab4)) ;
```

Conclusion: un tableau doit toujours se promener avec sa longueur, et lorsqu'on écrit "tab[i]", on prend le temps de se convaincre que i est un indice valide !

2.4.2 Implémentation des tableaux en mémoire

L'avantage d'avoir une longueur de tableau connue à la compilation (on dit *statique*), c'est que le compilateur détecte les débordements:

[ ]:
```
int main ()
{
  int tab[10] ;
  printf("%d\n", tab[10]) ;
}
```

Mais ce n'est qu'un warning:

[ ]:
```
main () ;
```

Et il ne le détecte pas forcément:

[ ]:
```
int main ()
{
    int tab[10] ;
    int n = 10 ;
    printf("%d\n", tab[n]) ;
}
```

[ ]:
```
main () ;
```

Bonus: exécuter ces deux exemples en environnement réel pour observer les valeurs affichées (cours2-5.c et cours2-7.c)

Question: à quoi correspondent ces valeurs ?

[ ]:
```
int a = 4 ;
  int b = -1 ;
  int aa = 26 ;
  int bb = 27 ;
  int ttab[4] = { 121,122,123,124 } ;
  int ttab2[4] = { 212,213,214,215 } ;
  int c = 12 ;
  printf("ttab[4] = %d \n", ttab[a]) ;
  printf("ttab[-1] = %d \n", ttab[b]) ;
  printf("ttab2[4] = %d \n", ttab2[a]) ;
  printf("ttab2[-1] = %d \n", ttab2[b]) ;
```

[ ]:
```
printf("Explication: \n") ;
printAdd((char*)"a",&a) ;
printAdd((char*)"b",&b) ;
printAdd((char*)"aa",&aa) ;
printAdd((char*)"bb",&bb) ;
printAdd((char*)"ttab[0]",&ttab[0]) ;
printAdd((char*)"ttab[1]",&ttab[1]) ;
printAdd((char*)"ttab[2]",&ttab[2]) ;
printAdd((char*)"ttab[3]",&ttab[3]) ;
printAdd((char*)"ttab2[0]",&ttab2[0]) ;
printAdd((char*)"ttab2[1]",&ttab2[1]) ;
printAdd((char*)"ttab2[2]",&ttab2[2]) ;
printAdd((char*)"ttab2[3]",&ttab2[3]) ;
printAdd((char*)"c",&c) ;
```

Attention: la gestion des variables locales (notamment l'ordre dans lequel on les empile) dépend du compilateur. Comparer avec cours2-4.c.

Finalement, déclarer un tableau d'entiers de taille 10 (int tab[10]), cela revient à déclarer d'un coup 10 variables de type int, ces variables s'appellent tab[0], ..., tab[9]. Elles ont une adresse: &tab[i] ; On peut les modifier: tab[i] = 3 ;

La variable tab n'est qu'un pointeur vers le premier élément du tableau:

[ ]:
```
printf("Adresse de tab[0]: %lu \n", (long)&tab[0]) ;
printf("Adresse pointée par tab: %lu\n", (long)tab) ;
if (tab == &tab[0])
    printf("Le prof a raison !") ;
else
    printf("Le prof a tort, et le tortue.") ;
```

Bonus (hors programme): tab[i] c'est simplement un calcul *d'arithmétique des pointeurs*: on part de l'adresse de tab et on ajoute le nombre de cases à parcourir (i) fois la taille d'une case:

`tab[i] = *(tab + i*sizeof(type))`

[ ]:
```
int hauteurs[3] = {1, 2, 3};
int* pValeur = &hauteurs[0];
pValeur = pValeur + 1;
*pValeur = 4;
pValeur = pValeur + 1;
*pValeur = 9;
printf("%d %d %d\n", hauteurs[0], hauteurs[1], hauteurs[2]) ;
```

D'après ce qui précède, la seconde ligne

`int* pValeur = &hauteurs[0] ;`

peut être remplacée par

`int* pValeur = hauteurs ;`

Remarque: une variable a un type explicite, afin que le compilateur sache quel espace mémoire réserver (char = 8 bits, int = 32 bits, ...). Tous les types pointeurs (char, *int*, ...) ne sont que des adresses, et toutes les adresses sont sur 64 bits. Pourquoi n'y a-t-il pas qu'un seul type pointeur ? Afin de connaitre totalement l'espace pointé: c'est l'espace qui commence à l'adresse du pointeur et dont la taille est donnée par le type du pointeur (4 cases pour un int, *1 case pour un char*, ...). Cela permet de simplifier l'arithmétique des pointeurs: l'opérateur + dépend du type:

[ ]:
```
int a = 0;
int* p = &a ;
char* q = (char*)p ;
printAdd((char*)"p", p) ;
printAdd((char*)"q", q) ;
printAdd((char*)"p+1", p+1) ;
printAdd((char*)"q+1", q+1) ;
printf("Si on convertit avant l'addition: (long)p + 1 = %lu \n", (long)p+1) ;
```

Ceci explique pourquoi dans le programme précédent, on effectue

`pValeur = pValeur + 1`

au lieu de

`pValeur = pValeur + 4`

bien que dans un tableau d'entier, la case suivante se situe 4 cases mémoires plus loin.

Ce qu'il faut retenir: - Les tableaux sont stockés de manière contigue en mémoire. - Chaque case à la même taille, et on peut donc par un calcul très simple connaitre l'adresse d'une case, sans parcourir tout le tableau. - Lorsque le système réserve un emplacement pour un tableau, vous ne savez pas ce qui se situe autour du tableau en mémoire. Il est donc impossible d'augmenter la taille d'un tableau après coup. La seule solution est d'initialiser un nouveau tableau plus grand, et de copier l'ancien tableau case par case. - Une variable de "type tableau" n'est qu'un pointeur vers l'adresse du premier élément du tableau. Ce point sera détaillé dans la suite.

2.5 Gestion des appels de fonctions: la pile d'appels

2.5.1 Fonctionnement

[1]:
```
// Fonction à 0 argument:
void f0 ()
{
    int a_f = 1;
    int b_f = 2;
    int* p = &b_f ;
    for (int i = 0; i < 16; i++)
    {
        printf("Contenu %d*4 cases avant b_f: %d\n", i, *(p+i)) ;
    }
}
```

[2]:
```
int main ()
{
    int a = 123 ;
    int b = 243 ;
    f0 () ;
}
```

[3]:
```
main () ;
```

```
Contenu 0*4 cases avant b_f: 2
Contenu 1*4 cases avant b_f: 1
Contenu 2*4 cases avant b_f: -685923968
Contenu 3*4 cases avant b_f: 32764
Contenu 4*4 cases avant b_f: -955293534
Contenu 5*4 cases avant b_f: 32511
Contenu 6*4 cases avant b_f: -685923808
Contenu 7*4 cases avant b_f: 32764
Contenu 8*4 cases avant b_f: 243
Contenu 9*4 cases avant b_f: 123
Contenu 10*4 cases avant b_f: -685923904
Contenu 11*4 cases avant b_f: 32764
Contenu 12*4 cases avant b_f: -955293647
Contenu 13*4 cases avant b_f: 32511
Contenu 14*4 cases avant b_f: -685923808
Contenu 15*4 cases avant b_f: 32764
```

[4]:
```
// Fonction à 1 arguments:
void f1 (int x)
{
    int a_f = 1;
    int b_f = 2;
    int* p = &b_f ;
    for (int i = 0; i < x; i++)
    {
        printf("Contenu %d*4 cases avant b_f: %d\n", i, *(p+i)) ;
    }
}
```

[5]:
```
int main ()
{
    int a = 123 ;
    int b = 243 ;
    f1(16) ;
}
```

[6]:
```
main () ;
```

```
Contenu 0*4 cases avant b_f: 2
Contenu 1*4 cases avant b_f: 1
Contenu 2*4 cases avant b_f: 16
Contenu 3*4 cases avant b_f: -685923968
Contenu 4*4 cases avant b_f: 32764
Contenu 5*4 cases avant b_f: -955563857
Contenu 6*4 cases avant b_f: 32511
Contenu 7*4 cases avant b_f: -685923808
Contenu 8*4 cases avant b_f: 16
Contenu 9*4 cases avant b_f: 243
Contenu 10*4 cases avant b_f: 123
Contenu 11*4 cases avant b_f: -685923904
Contenu 12*4 cases avant b_f: 32764
Contenu 13*4 cases avant b_f: -955563983
Contenu 14*4 cases avant b_f: 32511
Contenu 15*4 cases avant b_f: -685923808
```

[7]:
```
// Fonction à 2 arguments:
void f2 (int x, int y)
{
    int a_f = 1;
    int b_f = 2;
    int* p = &b_f ;
    for (int i = 0; i < x; i++)
    {
        printf("Contenu %d*4 cases avant b_f: %d\n", i, *(p+i)) ;
    }
}
```

[8]:
```
int main ()
{
    int a = 123 ;
    int b = 243 ;
    f2 (16, 332) ;
}
```

[9]:
```
main () ;
```

```
Contenu 0*4 cases avant b_f: 2
Contenu 1*4 cases avant b_f: 1
Contenu 2*4 cases avant b_f: 332
Contenu 3*4 cases avant b_f: 16
Contenu 4*4 cases avant b_f: -685923968
Contenu 5*4 cases avant b_f: 32764
Contenu 6*4 cases avant b_f: -957230916
Contenu 7*4 cases avant b_f: 32511
Contenu 8*4 cases avant b_f: 332
Contenu 9*4 cases avant b_f: 16
```

```
Contenu 10*4 cases avant b_f: 243
Contenu 11*4 cases avant b_f: 123
Contenu 12*4 cases avant b_f: -685923904
Contenu 13*4 cases avant b_f: 32764
Contenu 14*4 cases avant b_f: -957231055
Contenu 15*4 cases avant b_f: 32511
```

[10]:
```
// Fonction à 3 arguments:
void f3 (int x, int y, int z)
{
    int a_f = 1;
    int b_f = 2;
    int* p = &b_f ;
    for (int i = 0; i < x; i++)
    {
        printf("Contenu %d*4 cases avant b_f: %d\n", i, *(p+i)) ;
    }
}
```

[11]:
```
int main ()
{
    int a = 123 ;
    int b = 243 ;
    f3 (20, 332, 156) ;
}
```

[12]:
```
main () ;
```

```
Contenu 0*4 cases avant b_f: 2
Contenu 1*4 cases avant b_f: 1
Contenu 2*4 cases avant b_f: 156
Contenu 3*4 cases avant b_f: 332
Contenu 4*4 cases avant b_f: 20
Contenu 5*4 cases avant b_f: -685923968
Contenu 6*4 cases avant b_f: 32764
Contenu 7*4 cases avant b_f: -957239078
Contenu 8*4 cases avant b_f: 32511
Contenu 9*4 cases avant b_f: 160451280
Contenu 10*4 cases avant b_f: 332
Contenu 11*4 cases avant b_f: 20
Contenu 12*4 cases avant b_f: 156
Contenu 13*4 cases avant b_f: -957239056
Contenu 14*4 cases avant b_f: 32511
Contenu 15*4 cases avant b_f: 243
Contenu 16*4 cases avant b_f: 123
Contenu 17*4 cases avant b_f: -685923904
Contenu 18*4 cases avant b_f: 32764
Contenu 19*4 cases avant b_f: -957239247
```

[14]:
```
void f (int x)
{
    int a = x+1 ;
    int b = x+2 ;
    printf("Adresse de a: %lu\n", (long)(&a)) ;
    printf("Adresse de b: %lu\n", (long)(&b)) ;
}
```

[15]:
```
int main ()
{
    int x = 12 ;
    printf("Adresse x: %lu\n", (long)(&x)) ;
    printf("1er appel: \n") ;
    f(x + 10) ;
    printf("\nAdresse x: %lu\n", (long)(&x)) ;
    printf("2e appel: \n") ;
    f(x+20) ;
    printf("\nAdresse x: %lu", (long)(&x)) ;
}
```

[16]:
```
main () ;
```

```
Adresse x: 140723917529468
1er appel:
Adresse de a: 140723917529400
Adresse de b: 140723917529396

Adresse x: 140723917529468
2e appel:
Adresse de a: 140723917529400
Adresse de b: 140723917529396

Adresse x: 140723917529468
```

Tout ce qui a été ajouté sur la pile pour le premier appel de f a été *dépilé* à la fin de ce premier appel.

[17]:
```
void ff (int y, int z)
{
    int a = y+12 ;
    int b = y-7 ;
    int c = 12 ;
    printf("Adresse de a: %lu\n", (long)(&a)) ;
    printf("Adresse de b: %lu\n", (long)(&b)) ;
}
```

[18]:
```
void fff (int y)
{
    int c = 12 ;
    int a = y+12 ;
    int b = y-7 ;
    printf("Adresse de a: %lu\n", (long)(&a)) ;
    printf("Adresse de b: %lu\n", (long)(&b)) ;
}
```

[19]:
```
int main ()
{
    int x = 12 ;
    printf("Adresse x: %lu\n", (long)(&x)) ;
    printf("Appel f: \n") ;
    f(x + 10) ;
    printf("\nAdresse x: %lu\n", (long)(&x)) ;
    printf("Appel ff: \n") ;
    ff(x+20,13) ;
    printf("\nAdresse x: %lu\n", (long)(&x)) ;
    printf("Appel fff: \n") ;
    fff(x+30) ;
    printf("\nAdresse x: %lu\n", (long)(&x)) ;
}
```

[20]:
```
main () ;
```

```
Adresse x: 140723917529468
Appel f:
Adresse de a: 140723917529384
Adresse de b: 140723917529380

Adresse x: 140723917529468
Appel ff:
Adresse de a: 140723917529380
Adresse de b: 140723917529376

Adresse x: 140723917529468
Appel fff:
Adresse de a: 140723917529380
Adresse de b: 140723917529376

Adresse x: 140723917529468
```

Ce qu'il faut retenir: - Les appels de fonction ont un coût en mémoire. - Lors d'un appel de fonction, les arguments sont copiés plusieurs fois.

Pararaphe du programme officiel:

"On met l'accent sur la gestion au niveau de la machine, en termes d'occupation mémoire,

de la pile d'exécution, et de temps de calcul, en évoquant les questions de sauvegarde et de re

Petit bonus: prédire la valeur qui va être affichée par la fonction main (credit: université de Rennes).

[21]:
```c
int foo ()
{
    char buffer[8];
    char * ret;
    ret = buffer + 24 ;
    (*ret) += 8 ;
    return 0 ;
}
```

[ ]:
```c
int main ()
{
    int i = 26;
    foo() ;
    i = 17 ;
    printf("%d\n", i) ;
    return 0 ;
}
```

[ ]:
```c
// Cet exemple ne fonctionne pas ici, ni sur onlinegdb.
// Voir cours2-11.c
main () ;
```

2.5.2 Digression: l'évaluation paresseuse

Les expressions booléennes sont évaluées de manière *paresseuse*:

[23]:
```c
bool f (bool b)
{
  if (b)
    printf("Mon argument est vrai.\n") ;
  else
    printf("Mon argument est faux.\n");
  return b;
}
```

[24]:
```c
int main ()
{
  printf("Test 1:\n") ;
  if (f(true) || f(true))
    printf("La disjonction est vraie.\n\n") ;
  else
    printf("La disjonction est fausse.\n\n");

  printf("Test 2:\n") ;
```

```
  if (f(true) || f(false))
    printf("La disjonction est vraie.\n\n") ;
  else
    printf("La disjonction est fausse.\n\n");

  printf("Test 3:\n") ;
  if (f(false) || f(true))
    printf("La disjonction est vraie.\n\n") ;
  else
    printf("La disjonction est fausse.\n\n");

  printf("Test 4:\n") ;
  if (f(false) || f(false))
    printf("La disjonction est vraie.\n\n") ;
  else
    printf("La disjonction est fausse.\n\n");

  printf("Test 5:\n") ;
  if (f(false) && f(false))
    printf("La conjonction est vraie.\n\n") ;
  else
    printf("La conjonction est fausse.\n\n");

  printf("Test 6:\n") ;
  if (f(true) && f(false))
    printf("La conjonction est vraie.\n\n") ;
  else
    printf("La conjonction est fausse.\n\n");

    printf("Test 6:\n") ;
  if (f(false) && f(true))
    printf("La conjonction est vraie.\n\n") ;
  else
    printf("La conjonction est fausse.\n\n");

    printf("Test 8:\n") ;
  if (f(true) && f(true))
    printf("La conjonction est vraie.\n\n") ;
  else
    printf("La conjonction est fausse.\n\n");
}
```

[25]:
```
main () ;
```

```
Test 1:
Mon argument est vrai.
La disjonction est vraie.
```

```
Test 2:
Mon argument est vrai.
La disjonction est vraie.

Test 3:
Mon argument est faux.
Mon argument est vrai.
La disjonction est vraie.

Test 4:
Mon argument est faux.
Mon argument est faux.
La disjonction est fausse.

Test 5:
Mon argument est faux.
La conjonction est fausse.

Test 6:
Mon argument est vrai.
Mon argument est faux.
La conjonction est fausse.

Test 6:
Mon argument est faux.
La conjonction est fausse.

Test 8:
Mon argument est vrai.
Mon argument est vrai.
La conjonction est vraie.
```

En C, comme dans beaucoup de langages, l'évaluation des fonctions n'est PAS paresseuse:

[26]:
```c
void f ( int n)
{
  printf("f: Utiliser ses arguments c'est une norme sociale de l'ancien temps.
  ↪\n ") ;
}
```

[27]:
```c
int g ()
{
  printf("g: On m'appelle ?\n") ;
  return 0 ;
}
```

[28]:
```
int main ()
{
  f(g()) ;
}
main () ;
```

```
g: On m'appelle ?
f: Utiliser ses arguments c'est une norme sociale de l'ancien temps.
```

Dans un langage avec évaluation paresseuse (comme Haskell), la ligne

```
"g: On m'appelle ?"
```

n'apparaitrait pas, puisque la fonction "f" n'utilise jamais son argument, il est donc inutile de l'évaluer.

En OCaml, il existe une commande (lazy) qui force une évaluation à être paresseuse.

2.5.3 Passage (d'arguments) par valeurs

On a vu plus haut que lorsque l'on appelle une fonction, ses arguments sont copiés sur la pile. On a même aperçu qu'ils sont copiés deux fois: une première fois au sommet de la pile pour appeler l'instruction "call" en assembleur (hors programme), puis copié une seconde fois dans un paramètre local.

[29]:
```
int f (int x)
{
    printf("Adresse de x dans f: %lu\n",(long)&x) ;
    return (x+1) ;
}
```

[30]:
```
int main ()
{
    int n = 42 ;
    printf("Adresse de n dans main: %lu\n",(long)&n) ;
    f(n) ;
}
main () ;
```

```
Adresse de n dans main: 140723917529468
Adresse de x dans f: 140723917529420
```

Par conséquent, si la fonction "f" modifie son argument "x", elle le modifie localement et cela n'a aucune incidence sur la variable passée en argument à f (ici "n"): ce n'est pas la variable qui est argument, c'est sa **valeur**: on parle de **passage par valeurs**. Dans le code précédent, comme "n" vaut 42 au moment de l'appel, tout se passe exactement comme si l'on avait directement écrit "f(42)".

[31]:
```
// From France IOI
void afficheEtModifie(int parametre)
{
```

```
   printf("Début fonction : %d\n", parametre);
   parametre = 68;
   printf("Fin fonction : %d\n", parametre);
}
```

[32]:
```
// From France IOI
int main()
{
   int valeur = 42;
   printf("Début programme : %d\n", valeur);
   afficheEtModifie(valeur);
   printf("Fin programme : %d\n", valeur);
}
main () ;
```

```
Début programme : 42
Début fonction : 42
Fin fonction : 68
Fin programme : 42
```

En C, tous les appels se font par valeur. L'argument est constamment copié, ce qui a des conséquences:

[33]:
```
struct hordeDeNombres {
    int entier01 ;
    int entier02 ;
    int entier98 ;
    int entier99 ;
} ;

typedef struct hordeDeNombres hDN
```

[34]:
```
int sum (hDN horde)
{
    printf("Adresses de la horde: %lu %lu %lu %lu \n",
           (long)&horde.entier01, (long)&horde.entier02, (long)&horde.entier98,␣
 ↪(long)&horde.entier99) ;
    return (horde.entier01 + horde.entier02 + horde.entier98 + horde.entier99) ;
}
```

[35]:
```
int main ()
{
    hDN horde { 12, 15, 16, 3 } ;
    printf("Adresses de la horde: %lu %lu %lu %lu \n",
           (long)&horde.entier01, (long)&horde.entier02, (long)&horde.entier98,␣
 ↪(long)&horde.entier99) ;
    printf("%d\n", sum(horde)) ;
}
```

```
main () ;
```

```
Adresses de la horde: 140723917529456 140723917529460 140723917529464
140723917529468
Adresses de la horde: 140723917529368 140723917529372 140723917529376
140723917529380
46
```

Heureusement (?), en C il n'existe pas d'objets de grosse taille. A part les STRUCT que nous venons de voir (et en pratique on ne crée pas un type avec 1000 champs), tous les objets sont stockés sur 64 bits. En effet, les seuls types que l'on aurait pu imaginer être volumineux sont les tableaux et les chaines de caractères. Mais ces derniers sont en réalité des pointeurs vers un bloc de mémoire, et un pointeur est stocké sur 64 bits. C'est une contrainte dictée par le langage sous-jacent à C: en assembleur, la valeur de retour d'une fonction est stocké dans un registre dédiée du micro-processeur, lequel peut contenir 64 bits.

Dans les langages plus haut niveau, certains objets sont passés par valeurs (notamment les entiers, flottants, booléens, caractères, ...) et d'autres par **référence** (on parle de *passage par référence*), notamment les plus gros objets tels que les listes, les tableaux, les dictionnaires, les arbres, ... Dans le passage par référence, la fonction ne reçoit pas une copie des arguments, mais un *lien* vers les arguments d'origine. Souvent, ces langages haut niveau sont compilés (traduits) vers le C, et ce "passage par référence" est en réalité un passage par valeur, mais d'un pointeur, et non d'une valeur. Dans la plupart des langages de haut niveau, la notion de pointeur n'existe pas, et donc on parle de *passage par référence*. Ce mode de passage des arguments présente des inconvénients:

[36]:
```
void afficheEtModifie(int* parametre)
{
   printf("Début fonction : %d\n", *parametre);
   *parametre = 68;
   printf("Fin fonction : %d\n", *parametre);
}
```

[37]:
```
int main()
{
   int valeur = 42;
   printf("Début programme : %d\n", valeur);
   afficheEtModifie(&valeur);
   printf("Fin programme : %d\n", valeur);
}
main () ;
```

```
Début programme : 42
Début fonction : 42
Fin fonction : 68
Fin programme : 68
```

[38]:
```
int* increment (int longueur, int tab[])
{
```

```
    for (int i = 0; i < longueur; i++)
        tab[i] = tab[i]+1 ;
    return tab ;
}
```

[39]:
```
int main ()
{
    int myTab[10] = { 0,1,2,3,4,5,6,7,8,9 } ;
    int* newTab = increment(10, myTab) ;
    printf("myTab = ") ;
    for (int i = 0 ; i < 10; i++)
        printf("%d ", myTab[i]) ;
    printf("\n") ;
    if (myTab == newTab)
        printf("myTab == newTab") ;
    else
        printf("myTab != newTab") ;
}
main () ;
```

```
myTab = 1 2 3 4 5 6 7 8 9 10
myTab == newTab
```

Illustration avec PythonTutor.

2.5.4 Simulation de valeurs de retour multiples (par passage par référence)

Le passage par référence permet également de "simuler des valeurs de retour multiples" (citation du programme officiel). On a déjà vu comment renvoyer plusieurs variables en les groupant dans une STRUCT. On peut également placer la valeur retournée à une adresse donnée en argument.

[ ]:
```
void f (int a, int* p, int* q, int* r)
{
    int x = a+1 ;
    int y = a+2 ;
    int z = a+3 ;
    // Simulation de return:
    *p = x ;
    *q = y ;
    *r = z ;
}
```

[ ]:
```
// cours2-12.c
int main ()
{
    int a = 12 ;
    int p ;
    int q ;
```

```
    int r ;
    f(a,&p,&q,&r) ;
    printf("%d %d %d\n",p,q,r) ;
}

main () ;
```

2.5.5 Saturation de la pile (StackOverflow)

Exercice: écrire un programme qui ne termine pas, le plus simple et court possible.

[ ]:
```
int toInfinity ()
{
    while (true)
    {
    }
}
toInfinity () ;
```

[ ]:
```
// cours2-8.c
void andBeyond ()
{
    andBeyond () ;
}
andBeyond () ;
```

Pourtant, l'un de ces programme termine. Lequel ?

[40]:
```
// Explication: cours2-9.c
void andBeyond ()
{
  int a ;
  printf("%p %lu\n", &a, (long)&a) ;
  andBeyond () ;
}
```

[ ]:
```
// Explication plus précise: cours2-10.c
void andBeyond (int n)
{
  printf("%d %p %lu\n", n,&n,(long)&n) ;
  andBeyond (n+1) ;
}
```

Par défaut, l'exécution d'un programme est lancée avec une pile d'exécution de 8 Mb. Lorsque cette mémoire est intégralement utilisée, le sommet de la pile se situe dans une zone interdite de la mémoire: Segmentation Fault.

Et comme on l'a vu plus haut, appeler une fonction occupe environ 50 cases mémoire. Sur mon

ordinateur, “andBeyond” plante au bout d’environ 175 000 appels, 175 000 * 50 = 8 750 000 soit environ 8Mb.

C’est un inconvénient des programmes récursifs: ils consomment de l’espace mémoire sur la pile. On verra en fin de premier semestre comment contourner ce problème avec la *récursion terminale*.

Un exemple célèbre est la suite de Fibonacci, censée modéliser l’évolution dans le temps d’une population de lapins: - $u_0 = u_1 = 1$, - $u_{n+2} = u_{n+1} + u_n$.

[ ]:
```c
// cours2-13.c
int fibonacci(int n)
{
    if (n==0 || n==1)
        return 1 ;
    else
        return (fibonacci(n-1) + fibonacci(n-2)) ;
}
```

[ ]:
```c
printf("%d\n",fibonacci(24)) ;
```

Sur mon ordinateur, ça Segfault à 140 000. Les ressources du serveur sont différentes semble-t-il. Voir cours2-13.c.

2.5.6 Ce qu’il faut retenir:

Lors de l’exécution d’un programme, le système d’exploitation alloue un espace mémoire relativement restreint (8 Mb) réservé au programme, que l’on appelle la pile (stack en anglais). Le programme y stocke: - les variables locales, - tout un tas d’information nécessaires lors d’un appel de fonction.

Le premier point explique la portée des variables: les variables locales d’une fonction qui se termine sont dépilées, elles disparaissent. Il faut retenir le second point pour se souvenir que les programmes récursifs consomment de la mémoire: les appels imbriqués les uns dans les autres accumulent des informations sur la pile.

Il faut également retenir les concepts de passage par valeurs et référence. Notamment qu’en C, les arguments sont copiés. Mais lorsque l’on se passe un pointeur en argument (une “référence”), c’est bel et bien un pointeur vers la copie originale, les modifications ont donc lieu sur l’originale.

2.5.7 Stackoverflow

https://stackoverflow.com/ est un forum d’entraide pour développeurs dont les informations sont fiables. Ce site fait partie du réseau https://stackexchange.com/, la version math a également très bonne réputation: https://math.stackexchange.com/

2.6 Les tableaux: C pas si facile ...

2.6.1 Stackoverflow (again !)

Au paragraphe précédent, on a provoqué une “stack overflow” en saturant la pile avec des appels de fonctions. Avec quoi d’autre pouvons nous saturer la pile d’exécution ?

[ ]:
```
// cours2-14.c
// Ne fonctionne pas ici. Je ne connais pas les ressources de ce serveur,
// manifestement pas celles par défaut
int main ()
{
    int tab[2100000] ;
}
```

2.6.2 Mon tableau a disparu !

[41]:
```
int* initTableau ()
{
    int tab[6] ;
    printf("Entrez 6 entiers: ") ;
    for (int i = 0; i < 6; i++)
        // scanf("%d", &tab[i]) ;
        tab[i] = i ;
    return tab ;
}
```

```
input_line_47:8:12: warning: address of stack memory associated with local
variable 'tab' returned [-Wreturn-stack-address]
    return tab ;
           ^~~
```

[42]:
```
void traitementTableau (int longueur, int tab[longueur])
{
    int a = 27 ;
    int b = 37 ;
    int c = 47 ;
    int d = 57 ;
    for (int i = 0; i < longueur; i++)
        tab[i] = 2*tab[i] ;
}
```

[43]:
```
int main ()
{
    int* tab = initTableau() ;
    traitementTableau(6, tab) ;
    printf("tab = %d %d %d %d %d %d\n",␣
 ↪tab[0],tab[1],tab[2],tab[3],tab[4],tab[5]) ;
}
```

[44]:
```
main () ;
```

```
Entrez 6 entiers: tab = 0 6 57 94 74 54
```

Conclusion: la structure “chronologique” de la pile est incompatible avec les objets dont on souhaite

qu'ils restent longtemps en mémoire (au delà de la portée de la variable). C'est notamment un problème avec les objets que l'on ne peut pas copier.

La solution: **le tas !**

2.7 Le tas (heap), Malloc, Free

Comme son nom l'indique, le tas est moins "ordonné" que la pile. La pile a une structure: les objets les plus récents se trouvent au sommet de la pile, et lorsqu'on "descend" dans la pile, on accède à des variables déclarées plus tôt. Dans le tas en revanche, aucune structure. Il est impossible d'y trouver quelque chose sans connaitre exactement son adresse. Pour demander au système un espace mémoire sur le tas, on utilise la fonction "malloc" (memory allocation), dont le seul argument est le nombre d'octets demandés. Attention, il faut charger la librairie stdlib pour pouvoir utiliser malloc.

[ ]:
```c
#include <stdlib.h>
int* p ;
for (int i = 0; i < 10; i++)
{
    p = (int*)(malloc(sizeof(int))) ;
    printf("Adresse de p: %p   %lu ;   Valeur de p: %d \n", p, (long)(p), *p) ;
    free(p) ;
}
```

Le système se souvient de quels sont les espaces mémoire *alloués* et ceux qui sont *libres*. Lorsque la fonction malloc est appelée, elle renvoie l'adresse d'un espace libre et le système note alors cet espace comme alloué. Si l'on itère suffisament la boucle du programme ci dessus, le système va nous allouer toute la mémoire; la fonction malloc va alors échouer car il n'y a plus d'espace libre.

Si un programme a réellement besoin d'autant de mémoire (ça ne sera pas le cas en CPGE), on l'exécute sur une machine spécifique avec davantage de RAM. Mais la plupart du temps, une consommation excessive de mémoire provient d'une **fuite mémoire**: votre programme alloue de l'espace sans jamais le **libérer**. Pour libérer un espace alloué, on utilise la fonction "free()". Contrairement aux variables locales qui ont une durée de vie (dictée par le fonctionnement de la pile), les objets sur le tas y persiste jusqu'à la fin d'exécution du programme. Il faut donc manuellement "mettre fin à leur vie" lorsqu'ils ne seront plus utilisés.

A tout appel de "malloc" doit donc correspondre un appel de "free".

[ ]:
```c
// PythonTutor !!!
// cours2-33.c
int* initTableau (int* longueur)
{
    printf("Combien de nombres allez-vous entrer ? ") ;
    int n ;
    // scanf("%d",&n) ;
    n = 28 ;
    printf("28\n") ;
    printf("Entrez %d nombres, un par ligne.\n",n) ;
    // Conversion obligatoire ici ?
```

```
    int* tab = (int*)(malloc(sizeof(int)*n)) ;
    for (int i = 0; i < n; i++)
        // scanf("%d", &tab[i]) ;
        tab[i] = i ;
    *longueur = n ; // Retour multiple
    return tab ;
}
```

[ ]:
```
void traitementTableau (int longueur, int tab[longueur])
{
    for (int i = 0; i < longueur; i++)
        tab[i] = 2*tab[i] ;
}
```

[ ]:
```
int main ()
{
    int longueur ;
    int* tab = initTableau(&longueur) ;
    traitementTableau(longueur, tab) ;
    printf("tab = ") ;
    for (int i = 0; i < longueur; i++)
        printf("%d ", tab[i]) ;
    // LIGNE IMPORTANTE:
    free(tab) ;
}
```

[ ]:
```
main () ;
```

Techniquement, le type de retour de la fonction “malloc” est “void”, *soit un pointeur vers une donnée de type void. Or ici, mon tableau est de type ”int*”, d’où la conversion explicite. D’après votre programme officiel, c’est le seul cas où vous êtes censés utiliser des conversions:

`"Transtypage de données depuis et vers le type void* dans l’optique stricte de l’utilisation d`

Ceci étant dit, sur ce serveur (avec noyau C++) j’ai un warning si j’omets la conversion explicite, je n’en ai pas avec gcc.

Remarque: j’aurais voulu illustrer la fuite mémoire, mais j’ai eu pitié pour ma machine. Plusieurs erreurs sont possibles: - “Malloc cannot allocate region” - “Malloc cannot allocate memory” - malloc retourne le pointeur NULL, auquel cas l’erreur est un Segmentation Fault

Le tas est bien plus large que la pile: par défaut le système doit allouer environ 1Gb de RAM. Il est donc peu probable que vous ayiez l’une de ces erreurs en CPGE, et ce même si vous ne libérez pas votre mémoire. Il est néanmoins indispensable d’écrire les choses correctement tout de même, pour convaincre vos correcteurs que vous avez compris le fonctionnement. Et pour être tout à fait correct, il faut envisager que “malloc” renvoie le pointeur NULL en cas de saturation du tas. Je vous recommande donc de toujours vérifier que ce n’est pas le cas. On peut le faire avec un “if” mais l’écriture la plus concise et claire reste:

```
int* p = malloc(12) ;
```

```
assert(p != NULL) ;
```

Ce qu'il faut retenir: lors de l'exécution d'un programme, l'OS alloue deux espaces dans la mémoire: la pile (stack) et le tas (heap). La pile est utilisée comme une pile "on y empile" les données que l'on veut stocker, dans l'ordre où elles interviennent dans le programme. Le tas ressemble davantage à votre chambre d'étudiant et sert à stocker des données non-copiables de manière persistente.

Dès que les tableaux sont grands, voire de taille non pré-déterminée (non déterminée à la compilation), il convient de les placer sur le tas, en utilisant la fonction "malloc". Il faut absolument libérer (avec "free") la mémoire allouée avec "malloc" dès qu'elle ne sera plus utilisée. Il faut toujours vérifier que "malloc" ne renvoie pas un pointeur nul.

Pour les tableaux statiques (définis sur la pile) il faut également faire attention à leur "disparition": leur durée de vie est la même que celle d'une variable classique. Comme "main" est la première et dernière fonction appelée (outtermost dans l'arbre d'appels), les variables locales de "main" ne seront jamais dépilées avant la fin du programme. C'est donc une solution pour la persistance du tableau, mais cette solution n'est pas viable pour un projet de taille conséquente.

[ ]:
```
void initTableau (int longueur, int* tab)
{
    printf("Entrez %d entiers: ",longueur) ;
    for (int i = 0; i < longueur; i++)
        // scanf("%d", &tab[i]) ;
        tab[i] = i ;
}
```

[ ]:
```
int main ()
{
    int tab[6] ;
    initTableau(6,tab) ;
    traitementTableau(6, tab) ;
    printf("tab = ") ;
    for (int i = 0; i < 6; i++)
        printf("%d ", tab[i]) ;
}
```

Souvenez-vous bien que les tableaux sont "passés par référence" (car ils SONT en réalité des références), ainsi le tableau d'origine et modifié par la fonction "traitementTableau".

2.8 Quelques précisions sur les tableaux

2.8.1 Tableaux en argument d'une fonction

Les 3 lignes suivantes sont équivalentes pour passer un tableau en argument:

```
type f(int longueur, int* tab)
type f(int longueur, int tab[])
type f(int longueur, int tab[longueur])
```

La 3e ligne donne une indication au développeur, mais aucune vérification n'est effectuée: si tab n'est pas de longueur "longueur", cela ne pose aucun problème.

[ ]:
```
//int f(int longueur, int* tab)
// int f(int longueur, int tab[])
//int f(int longueur, int tab[longueur])
// int f(int tab[], int longueur)  // On peut inverser l'ordre des arguments,
 ↪sauf pour la 3e ligne.
{
    return(tab[longueur-1]) ;
}
```

[ ]:
```
int main()
{
    int tab[10] = { 1,2,3,34,4,5,6,7,7,54 } ;
    printf("%d\n",f(10,tab)) ;
}
main () ;
```

2.8.2 Conflit avec le programme officiel

J'insiste encore une fois: l'instruction

```
int tab[12] ;
```

crée le tableau sur la pile. Il faut soigneusement vérifier que la durée de vie de cette variable est celle attendue, et **n'utiliser cette syntaxe qu'avec des valeurs numériques**. Interdiction formelle d'écrire:

```
int tab[n] ;
```

et ce **même si vous déclarer à la ligne précédente**

```
int n = 12 ;
```

La syntaxe "int tab[n]" existe en C et est acceptée. C'est moi qui vous l'interdit, pas le langage. Je détaille plus bas. Le programme officiel autorise (et encourage) le code suivant:

[ ]:
```
const int n = 12 ;
int tab[n] ;
```

Je vous interdit cette syntaxe malgré tout. Le mot clé "const" force la variable à rester constante, il n'est plus possible d'affecter une valeur à n:

[ ]:
```
const int n = 12 ;
n = 13 ;
```

Je m'attends à ce que les correcteurs aient comme consigne de sanctionner la syntaxe "int tab[n]", sans même vérifier si la variable "n" a été définie comme constante avec le mot clé "const". Si votre tableau n'a pas une taille qui est un entier numérique, alors vous **devez** passer par la fonction malloc et l'allocation dynamique sur le tas.

[ ]:
```
int n = 12 ;
int* tab = (int*)(malloc(sizeof(int)*12)) ;
```

2.8.3 Copier un tableau, subtilité de typage

En C, toutes les instructions du langages sont élémentaires. Il n'existe pas de commande qui se traduisent en réalité en un grand nombre d'opérations machine. Par conséquent, il est impossible de copier un tableau en une seule instruction, il faut pour cela faire une boucle.

[ ]:
```
int tab1[5] = { 1,2,3,4,5 } ;
int tab2[5] ;
tab2 = tab1 ;
```

Pourtant, lorsque l'on défini un tableau dynamiquement, on le définit comme un pointeur, et à ce moment là ce n'est plus interdit:

[ ]:
```
int* tab1 = (int*)(malloc(sizeof(int)*5)) ;
tab1[0] = 1; tab1[1] = 2; tab1[2] = 3; tab1[3] = 4; tab1[4] = 5 ;
int* tab2 ;
tab2 = tab1 ;
printf("tab1 = %d %d %d %d %d\n", tab1[0], tab1[1], tab1[2], tab1[3], tab1[4]) ;
printf("tab2 = %d %d %d %d %d\n", tab2[0], tab2[1], tab2[2], tab2[3], tab2[4]) ;
```

Mais l'effet produit par cette affectation n'est peut être pas celui que vous imaginiez (notamment ce n'est PAS une copie):

[ ]:
```
// PythonTutor
tab2[0] = 12;
tab2[1] = 17;
tab2[2] = 24 ;
printf("tab1 = %d %d %d %d %d\n", tab1[0], tab1[1], tab1[2], tab1[3], tab1[4]) ;
printf("tab2 = %d %d %d %d %d\n", tab2[0], tab2[1], tab2[2], tab2[3], tab2[4]) ;
```

Le type "array" n'existe pas vraiment, le compilateur distingue "int" *et "array" pour vous protéger d'erreur telle que celle-ci. Mais cette sécurité disparait dès que vous passez à travers une fonction. En effet, les syntaxes du 2.8.1 sont équivalentes, la fonction appelée voit votre tableau comme un int*, ni plus ni moins. Elle n'a aucun moyen de savoir que c'était un tableau à l'origine:

[ ]:
```
void f(int longueur, int tab1[], int tab2[])
{
    tab2 = tab1 ;
    printf("Dans f: tab1 = %d %d %d %d %d\n", tab1[0], tab1[1], tab1[2],␣
 ↪tab1[3], tab1[4]) ;
    printf("Dans f: tab1 = %d %d %d %d %d\n", tab2[0], tab2[1], tab2[2],␣
 ↪tab2[3], tab2[4]) ;
}
```

[ ]:
```
int main()
{
    int tab1[5] = { 1,2,3,4,5 } ;
    int tab2[5] ;
    f(5,tab1, tab2) ;
    // tab2 = tab1 ;
    printf("Dans main: tab1 = %d %d %d %d %d\n", tab1[0], tab1[1], tab1[2],␣
 ↪tab1[3], tab1[4]) ;
    printf("Dans main: tab1 = %d %d %d %d %d\n", tab2[0], tab2[1], tab2[2],␣
 ↪tab2[3], tab2[4]) ;
    return 0 ;
}
main () ;
```

2.9 Quelques précisions sur les structures

2.9.1 Passage par valeurs et copie

Comme pour les objets élémentaires du langage, les éléments de type “STRUCT trucmuche” sont passés par valeurs: ils sont notamment copiés lorsqu'ils sont passés en argument, ou renvoyé par “return”:

[ ]:
```
struct montype { int champ1; int champ2 ;} ;
typedef struct montype montype ;
```

[ ]:
```
void stackScribbler ()
{
    int a = 12 ;
    int b = 12 ;
    int c = 12 ;
    int d = 12 ;
    int e = 12 ;
    printf("Scribbled over: %lu %lu %lu %lu %lu\n"
           , (long)&a, (long)&b, (long)&c, (long)&d, (long)&e) ;
}
```

[ ]:
```
montype init (int n, int m)
{
  montype mavar;
  mavar.champ1 = n;
  mavar.champ2 = m;
  printf("Adresse de mavar (dans init): %lu %lu %lu\n"
         ,(long)&mavar, (long)&mavar.champ1, (long)&mavar.champ2) ;
  return mavar;
}
```

[ ]:
```
int main()
{
  montype monobjet = init (42,12) ;
  stackScribbler () ;
  printf("Adresse et valeur de monobjet (dans main): %lu %d %d\n"
         , (long)&monobjet, monobjet.champ1, monobjet.champ2) ;
}
main () ;
```

A priori, un objet de type “STRUCT montype” peut être assez volumineux, comme un tableau. Pourquoi alors est-ce que le langage sait les copier, mais pas les tableaux ? Parce que la taille est statiquement connue: elle est fixée lors de la définition du type:

[ ]:
```
printf("%lu", sizeof(montype)) ;
```

alors que tous les tableaux ont le même type, et donc le type ne suffit pas à indiquer la taille. D’ailleurs, le langage sait faire de la copie de structure:

[ ]:
```
montype obj1 = { 1, 2 } ;
montype obj2 = obj1 ;
montype obj3 ;
obj3 = obj2 ;
printf("Obj1 = %d %d \n", obj1.champ1, obj1.champ2) ;
printf("Obj2 = %d %d \n", obj2.champ1, obj2.champ2) ;
printf("Obj3 = %d %d \n", obj3.champ1, obj3.champ2) ;
printf("On modifie l'objet 1 et l'objet 3.\n") ;
obj3.champ1 = 36 ;
obj1.champ2 = 42 ;
printf("Obj1 = %d %d \n", obj1.champ1, obj1.champ2) ;
printf("Obj2 = %d %d \n", obj2.champ1, obj2.champ2) ;
printf("Obj3 = %d %d \n", obj3.champ1, obj3.champ2) ;
```

2.9.2 Passage par référence

Si on veut malgré tout minimiser la copie de données, on peut utiliser des pointeurs et ne de passer que la référence des STRUCT en argument et return. La mise en oeuvre est la même que pour les tableaux:

[ ]:
```
montype* init (int n, int m)
{
  montype* p = (montype*)(malloc(sizeof(montype))) ;
  (*p).champ1 = n; // Les parenthèses sont obligatoires.
  p->champ2 = m; // Syntaxe alternative.
  printf("Adresse et valeur de p: %lu %lu\n",(long)&p,(long)p);
  return p;
}
```

[ ]:
```
int main()
{
  montype* p = init (42,12) ;
  stackScribbler () ;
  printf("Adresse et valeur de p (dans main): %lu %lu\n", (long)&p, (long)p) ;
  printf("Valeur des champs (dans main): %d %d\n",p->champ1, (*p).champ2) ;
  free(p) ;
}
main () ;
```

Erreurs classiques sur ce code: - garder le même format que dans le 2.9.1 et simplement retourner "&mavar" dans la fonction "init": cela retourne effectivement l'adresse où est stocké l'objet "mavar", mais ce dernier va être écrasé par "stackScribbler". - Ligne 3 de la fonction "init" du paragraphe 2.9.2: déclarer le pointeur sans l'initialiser:

```
montype* p ;
```

Il est alors initialiser à NULL ou à une valeur aléatoire, et les affectations

```
(*p).champ1
(*p).champ2
```

échouent (tentative d'accéder à une adresse invalide: segmentation fault).

2.10 Bonus Fonte de Cerveaux: interaction STRUCT et Tableaux

Dans l'exemple de "profilEleve" présenté au cours précédent pour illustrer les STRUCT, on avait des chaines de caractères dans la structure (pour le nom et commentaires). En C, les chaines de caractères sont ni plus ni moins que des tableaux de char, nous avions donc un tableau dans une STRUCT. Il y a deux implémentations possibles:

[ ]:
```
struct type1 { int tab[5] ; };
typedef struct type1 type1 ;
struct type2 { int* tab ; } ;
typedef struct type2 type2 ;
```

2.10.1 Taille des objets:

Prévoir le résultat de ces commandes:

[ ]:
```
printf("Taille du type1: %lu\n", sizeof(type1)) ;
printf("Taille du type2: %lu\n", sizeof(type2)) ;
```

2.10.2 Initialisation, accès et modification des objets:

[ ]:
```
type1 obj1 = { {1,2,3,4,5} } ; // fonctionne
```

[ ]:
```
type2 obj2 = { { 1,2,3,4,5} } ; // ne fonctionne pas
```

[ ]:
```
int montab[5] = {1,2,3,4,5} ;
type2 obj2 = { montab } ;
```

Pour l'accès et les mises à jour, cela ne change absolument rien:

[ ]:
```
printf("obj1.tab[2] = %d\n",obj1.tab[2]);
printf("obj2.tab[2] = %d\n",obj2.tab[2]);
obj1.tab[2] = 12 ;
obj2.tab[2] = 13 ;
printf("obj1.tab[2] = %d\n",obj1.tab[2]);
printf("obj2.tab[2] = %d\n",obj2.tab[2]);
```

Ou presque:

[ ]:
```
int autretab[5] = { 12,13,15,17,19 } ;
obj2.tab = autretab ;
```

[ ]:
```
obj1.tab = autretab ;
```

2.10.3 Passage par valeur et persistance mémoire

[ ]:
```
type1 init1 (int n)
{
    type1 obj1 = { { n,n,n,n,n } } ;
    return obj1 ;
}
```

[ ]:
```
type2 init2 (int n)
{
    int tab[5] = { n,n,n,n,n } ;
    type2 obj2 = { tab } ;
    return obj2 ;
}
```

[ ]:
```
int main1 ()
{
    type1 obj1 = init1(42) ;
    stackScribbler() ;
    printf("obj1.tab = %d %d %d %d %d\n",
           obj1.tab[0], obj1.tab[1], obj1.tab[2],obj1.tab[3],obj1.tab[4]) ;
    return 0 ;
}
main1 () ;
```

[ ]:
```
int main2 ()
{
    type2 obj2 = init2(42) ;
```

```
    stackScribbler() ;
    printf("obj2.tab = %d %d %d %d %d\n",
           obj2.tab[0], obj2.tab[1], obj2.tab[2],obj2.tab[3],obj2.tab[4]) ;
    return 0;
}
main2 () ;
```

2.10.4 Copie et Hack ultime

[ ]:
```
type1 obj11 = { {1,2,3,4,5} } ;
type1 obj12 = { {3,4,5,6,7} } ;
obj12 = obj11 ;
printf("obj11.tab = %d %d %d %d %d\n",
           obj11.tab[0], obj11.tab[1], obj11.tab[2],obj11.tab[3],obj11.tab[4]) ;
printf("obj12.tab = %d %d %d %d %d\n",
           obj12.tab[0], obj12.tab[1], obj12.tab[2],obj12.tab[3],obj12.tab[4]) ;
printf("Modification obj12\n") ;
obj12.tab[3] = 26 ;
printf("obj11.tab = %d %d %d %d %d\n",
           obj11.tab[0], obj11.tab[1], obj11.tab[2],obj11.tab[3],obj11.tab[4]) ;
printf("obj12.tab = %d %d %d %d %d\n",
           obj12.tab[0], obj12.tab[1], obj12.tab[2],obj12.tab[3],obj12.tab[4]) ;
```

On vient de copier un tableau !!!!

Alors qu'avec le type2:

[ ]:
```
int tab1[5] = { 1,2,3,4,5 };
int tab2[5] = { 3,4,5,6,7 };
type2 obj21 = { tab1 };
type2 obj22 = { tab2 };
obj22 = obj21 ;
printf("obj21.tab = %d %d %d %d %d\n",
           obj21.tab[0], obj21.tab[1], obj21.tab[2],obj21.tab[3],obj21.tab[4]) ;
printf("obj12.tab = %d %d %d %d %d\n",
           obj22.tab[0], obj22.tab[1], obj22.tab[2],obj22.tab[3],obj22.tab[4]) ;
printf("Modification obj22\n") ;
obj22.tab[3] = 26 ;
printf("obj21.tab = %d %d %d %d %d\n",
           obj21.tab[0], obj21.tab[1], obj21.tab[2],obj21.tab[3],obj21.tab[4]) ;
printf("obj12.tab = %d %d %d %d %d\n",
           obj22.tab[0], obj22.tab[1], obj22.tab[2],obj22.tab[3],obj22.tab[4]) ;
```

2.10.5 Conclusion

Dans le cas du type1, le tableau est complètement contenu dans la STRUCT. Les objets du type1 sont donc volumineux et il faut adopter le style du paragraphe 2.9.2: toujours les faire passer par référence. Cela impose donc de créer ces objets directement sur le tas:

```
type1* obj = (type1*)(malloc(sizeof(type1))) ;
```

On peut tout de même les garder sur la pile, et l'appel par valeur fonctionnera. C'est d'ailleurs un hack pour pouvoir copier des tableaux dont la taille est statiquement connue.

Dans le cas du type2, le champs du STRUCT pointe vers un tableau, qui se situe ailleurs. Si ce ailleurs est la pile, on aura les problèmes de disparition de tableaux sus-mentionnés. Il faut donc impérativement créer un tableau sur le tas, et y faire référence dans le champ du STRUCT.

```
int* tab = (int*)(malloc(sizeof(int)*longueur)) ;
type2 obj = { tab } ;
```

Par conséquent, le STRUCT sera peu volumineux, on peut donc laisser faire l'appel par valeur et ne pas s'embêter à déclarer la STRUCT sur le tas (puisque le tableau y est déjà). Il faut par contre garder en tête que toutes les copies de ce STRUCT pointent vers un même tableau, déclaré sur le tas. Toute modification du tableau s'appliquera à toutes les copies du STRUCT.

Un avantage du type2 est qu'il ne restreint pas la taille du tableau, là où dans le type1, la taille du tableau est définitivement fixée. Voyez d'ailleurs le message d'erreur associé au code suivant:

[ ]:
```
int n = 12 ;

struct myTab {
  int tab[n] ;
} ;
```

2.11 Les Chaines de Caractères (string)

2.11.1 Ce qu'il faut savoir:

En C, il n'y a pas de stype "string" comme dans la plupart des autres langages. Les chaines de caractères sont **littéralement** des tableaux de *char*.

[3]:
```
char maChaine[50] ;
maChaine[0] = 'j' ;
maChaine[1] = 'e' ;
maChaine[2] = ' ' ;
maChaine[3] = 'm' ;
maChaine[4] = '\'' ;
maChaine[5] = 'a' ;
maChaine[6] = 'p' ;
maChaine[7] = 'p' ;
maChaine[8] = 'e' ;
maChaine[9] = 'l' ;
maChaine[10] = 'l' ;
maChaine[11] = 'e' ;
maChaine[12] = ' ' ;
maChaine[13] = 'S' ;
maChaine[14] = 'i' ;
maChaine[15] = 'm' ;
maChaine[16] = 'o' ;
```

```
maChaine[17] = 'n' ;
for (int i = 0; i < 18; i++)
  printf("%c",maChaine[i]) ;
printf("\n%s\n", "Ma vie est passionnante n'est-ce pas ?") ;
```

```
je m'appelle Simon
Ma vie est passionnante n'est-ce pas ?
```

A la **déclaration seulement**, on peut les initialiser plus rapidemment (ouf !):

[4]:
```
char monNom[] = { 'S','i','m','o','n' } ;
```

Une autre syntaxe reservée aux chaines de caractères:

[1]:
```
char maPhrase[] = "Bonjour, je m'appelle Simon !" ;
```

Comme pour les tableaux, la *détection automatique* de la taille à l'initialisation choisit la taille la plus petite possible:

[6]:
```
printf("Longueur de maPhrase: %lu\n",strlen(maPhrase)) ;
```

```
Longueur de maPhrase: 29
```

- Titouan: Euh … On a une fonction qui donne la longueur pour les tableaux ???
- Prof: Non toujours pas, mais pour les chaines de caractère oui.
- Titouan ?: pourtant c'est la même chose ???
- C'est grace à la magie du **caractère nul**, aussi appelé **caractère de fin de chaine**.

Table ASCII: http://www.mon-club-elec.fr/mes_images/reference_francais/asciifull.gif

Extrait du document:

Le code ASCII (American Standard Code for Information Interchange) date de 1960. C'est la façon standard dont le texte est codé numériquement. Noter que les 32 premiers caractères (0-31) sont des caractères non-imprimables, souvent appelés caractères de contrôle.

[1]:
```
for (int j = 65-26; j < 100; j+=26)
{
    for (int i = j; i < j+26 && i < 256; i++)
        printf("%c ", (char)i) ;
    printf("\n") ;
}
```

```
' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
[ \ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t
```

Toutes les chaines créés ci-dessus, peu importe la méthode d'initialisation, se terminent par un caractère nul:

[9]:
```
printf("maChaine[18-25] = %d %d %d %d %d %d %d %d\n"
       ,(int)maChaine[18], (int)maChaine[19],(int)maChaine[20],(int)maChaine[21]
      ␣
 ↪,(int)maChaine[22],(int)maChaine[23],(int)maChaine[24],(int)maChaine[25]) ;
printf("Longueur de monNom: %lu\n", strlen(monNom)) ;
printf("monNom[5]: %d\n", (int)monNom[5]) ;
printf("maPhrase[29]: %d\n", (int)(maPhrase[29])) ;
```

```
maChaine[18-25] = 0 0 0 0 0 0 0 0
Longueur de monNom: 5
monNom[5]: 0
maPhrase[29]: 0
```

[33]:
```
int i = 7 ;
char a = 'a' ;
char monPrenom[] = "Halfon" ;
char b = 'b' ;
printf("etat stack: ... &a |  &prenom | &b ...\n") ;
printf("etat stack: ... %d %d ... %d %d %d ...\n",
       (int)monPrenom[-1], (int)monPrenom[0],
       (int)monPrenom[5], (int)monPrenom[6], (int)monPrenom[i]) ;
```

```
etat stack: … &a |  &prenom | &b …
etat stack: … 97 72 … 110 0 98 …
```

Cela permet d'écrire des itérateurs sur les chaines dont on ne dispose pas sur les tableaux:

[34]:
```
// printf("%s", maPhrase) ;
int i = 0 ;
while (maPhrase[i] != '\0')
  {
    printf("%c", maPhrase[i]) ;
    i++ ;
  }

printf("\n%s\n", maPhrase) ;
```

```
Bonjour, je m'appelle Simon !
Bonjour, je m'appelle Simon !
```

[ ]:
```
// strlen(maPhrase) ;
int i = 0 ;
while (maPhrase[i] != '\0')
  {
    i++ ;
  }
printf("strlen(maPhrase) = %d\n", i) ;
printf("strlen(maPhrase) = %lu\n", strlen(maPhrase)) ;
```

[2]:
```
// strcpy(newPhrase, maPhrase) ;
char newPhrase[35] ;
int i = 0 ;
while (maPhrase[i] != '\0')
  {
    newPhrase[i] = maPhrase[i] ;
    i++ ;
  }
printf("newPhrase: %s\n", newPhrase) ;
char newnewPhrase[32] ;
strcpy(newnewPhrase, maPhrase) ;
printf("newnewPhrase = %s\n", newnewPhrase) ;
```

```
newPhrase: Bonjour, je m'appelle Simon !
newnewPhrase = Bonjour, je m'appelle Simon !
```

Et plein d'autres dans la librairie <string.h>. Une autre librairie intéressante pour travailler avec les chaines de caractères: <ctype.h>. Elle définit des fonctions permettant de tester le type d'un caractère (majuscule, minuscule, alpha-numérique, numérique, lettre de l'alphabet, …). **Ces deux librairies n'apparaissent PAS dans votre programme.** Cependant, les fonctions "strlen", "strcpy" et "strcat" apparaissent dans le programme.

Extrait du programme:

```
À l'écrit, on travaille toujours sous l'hypothèse que les entêtes suivants ont tous été inclus
```

Notamment, si vous voulez utiliser les fonctions de <string.h> et <ctype.h> à l'écrit, il faut inclure la ligne

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

Attention, les fonctions "strlen", "strcpy" etc. sont bien des fonctions écrites en C et absolument pas des éléments du langages. Les string sont des tableaux, et les opérations de copie et d'affectation globales restent impossible:

[3]:
```
char newnewnewPhrase[36] = maPhrase ;
```

```
input_line_9:2:7: error: array initializer must be an initializer list or string
literal
 char newnewnewPhrase[36] = maPhrase ;
      ^
```

```
Interpreter Error:
```

[4]:
```
char newnewnewPhrase[36] ;
newnewnewPhrase = maPhrase ;
```

```
input_line_10:3:17: error: array type 'char [36]' is not assignable
newnewnewPhrase = maPhrase ;
~~~~~~~~~~~~~~~ ^
```

```
Interpreter Error:
```

2.11.2 L'exercice "date" du thème "fonctions"

Dans cet exercice, il fallait écrire une fonction qui renvoie le jour de la semaine associé à une date, sous la forme d'une chaine de caractère.

[12]:
```
// cours2-36.c
char* jourSemaine (int n)
{
    char jour[] = "lundi" ;
    return jour ;
}
```

```
input_line_18:4:12: warning: address of stack memory associated with local
variable 'jour' returned [-Wreturn-stack-address]
    return jour ;
           ^~~~
```

[14]:
```
int main ()
{
    printf("Nous sommes un %s.\n", jourSemaine(12)) ;
}
main () ;
```

```
Nous sommes un .
```

Comportement propre au string:

[15]:
```
// cours2-38.c
char* jourSemaine2 (int n)
{
    return "lundi" ;
}
// Warning propre à C++
```

```
input_line_21:3:12: warning: ISO C++11 does not allow conversion from string
literal to 'char *' [-Wwritable-strings]
    return "lundi" ;
           ^
```

[16]:
```
int main ()
{
    printf("Nous sommes un %s.\n", jourSemaine2(12)) ;
}
main () ;
```

```
Nous sommes un lundi.
```

[17]:
```
// cours2-39.c
int* tab ()
{
    return ({1,2,3}) ;
}
```

```
input_line_23:3:19: error: expected ';' after expression
    return ({1,2,3}) ;
                  ^
                  ;
input_line_23:3:12: error: cannot initialize return object of type 'int *' with
an rvalue of type 'int'
    return ({1,2,3}) ;
           ^~~~~~~~
```

```
Interpreter Error:
```

2.11.3 Bonus: Buffer Overflow

Comme pour les tableaux, il faut expliciter la taille d'une string lorsqu'on la déclare. C'est particulièrement problématique lorsque l'on veut récupérer une entrée de l'utilisateur; dont on ne connait pas encore la taille:

[6]:
```
// Attention: scanf ==> passer en environnement réel.
char nom[50] ;
scanf("%s", nom) ; // Pas de & ici: pourquoi ?
```

[6]:
```
-1
type: int
```

Que se passe-t-il si l'utilisateur entre un nom trop long ? Illustration: cours2-25.c.

Réponse: ça plante lamentablement (abort). Aucune chance d'utiliser "assert" ici puisqu'il faut d'abord stocker une première fois la chaine pour en faire quoique ce soit.

Idée: travaillons sur le tas plutôt que sur la pile. Illustration: cours2-26.c.

Super ça marche ! Or does it ? ...

Non c'est une très mauvaise idée ! Sur la pile, le système était capable de détecter que l'on tentait d'écrire hors de la string et interrompait l'opération. Sur le tas il laisse faire, mais c'est terriblement dangereux: les derniers caractères de notre chaine dépassent de l'espace alloué par malloc: on est potentiellement en train d'écrire sur les cases d'une autre variable ! Et comme c'est mal rangé dans le tas, on a aucune idée de quel variable nous serions en train d'écraser (corruption de données).

Solution simple: utiliser

```
scanf("%50s",nom) ;
```

qui tronque l'entrée utilisateur au delà de 50 caractères lus (cours 2-27.c, cours2-28.c, cours2-29.c). Attention, "scanf(%s)" s'arrête au premier espace (il ne lit qu'un seul mot).

Solution expert: séparer la lecture de l'entrée et le parsing. La pair fgets + sscanf est alors recommandée. On verra (peut-être) cela en TP dédié, mais ces questions dépassent le cadre du programme de MP2I.

2.12 Tableaux 2D

2.12.1 Tableaux 2D sur la pile: déclaration, lecture, écriture

[12]:
```
int tab[12][5] ; // Tableau, 12 lignes, 5 colonnes.
for (int i = 0; i < 12; i++)
    for (int j = 0; j < 5; j++)
        tab[i][j] = i + j ;
```

[23]:
```
for (int i = 0; i < 12; i++)
{
    for (int j = 0; j < 5; j++)
    {
        int n = tab[i][j] ;
        if (n > 9)
            printf("%d ", n) ;
        else
            printf(" %d ", n) ;
    }
    printf("\n") ;
}
```

```
 0  1  2  3  4
 1  2  3  4  5
 2  3  4  5  6
 3  4  5  6  7
 4  5  6  7  8
 5  6  7  8  9
 6  7  8  9 10
 7  8  9 10 11
 8  9 10 11 12
 9 10 11 12 13
10 11 12 13 14
11 12 13 14 15
```

On peut initialiser à la déclaration:

[25]:
```
int tab2[2][3] = { { 0,1,2 }, {1,2,3 } } ;

for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 3; j++)
```

```
        printf("%d ", tab2[i][j]) ;
    printf("\n") ;
}
```

```
0 1 2
1 2 3
```

Et tout ceci marche en dimension quelconque:

[26]:
```
int tab[12][3][17][25] ;
  for (int i = 0; i < 12; i++)
    for (int j = 0; j < 3; j++)
      for (int k = 0; k < 17; k++)
        for (int l = 0; l < 25; l++)
          tab[i][j][k][l] = 0 ;
```

[27]:
```
int tab[2][2][2] = { { { 1,2 }, { 3,4 } }, { { 5,6 }, { 7,8 } } } ;
// Dessin en 3D au tableau ?
```

2.12.2 La magie du pré-compilateur

[31]:
```
int tab2[2][3] = { { 0,1,2 }, {1,2,3 } } ;
for (int i = 0; i < 6; i++)
    printf("%d ", *((int*)tab2+i)) ;
// Conversion obligatoire à cause du type tableau
```

```
0 1 2 1 2 3
```

[34]:
```
int tab3[6] = { 0,1,2,1,2,3 } ;
for (int i = 0; i < 6; i++)
    printf("%d ", *(tab3+i)) ;
```

```
0 1 2 1 2 3
```

`tab[i][j] = *(tab + i*nbCol + j)`

[36]:
```
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 3; j++)
        printf("%d ", *((int*)tab2 + i*3 + j)) ;
    printf("\n") ;
}
```

```
0 1 2
1 2 3
```

2.12.3 Passage en argument

Nous avions vu 3 syntaxes **équivalentes** pour les tableaux 1D:

```
void f (int longueur, int* tab)
void f (int longueur, int tab[])
void f (int longueur, int tab[longueur])
```

Cela donne 5 syntaxes **imaginables** pour les tableaux 2D:

```
void f (int nbRow, int nbCol, int* tab)
void f (int nbRow, int nbCol, int tab[][])
void f (int nbRow, int nbCol, int tab[][nbCol])
void f (int nbRow, int nbCol, int tab[nbRow][])
void f (int nbRow, int nbCol, int tab[nbRow][NbCol])
```

Fonctionnent: 3 et 5 **SEULEMENT**.

[37]:
```
// cours2-21.c
// même erreur ici et avec gcc
void f (int nbRow, int nbCol, int tab[][])
{
  printf("%d\n", tab[nbRow-1][nbCol-1]) ;
}
```

```
input_line_46:2:38: error: array has incomplete element type 'int []'
void f (int nbRow, int nbCol, int tab[][])
                                     ^
```

```
Interpreter Error:
```

[38]:
```
// cours2-22.c
// Erreur différente en gcc, mais erreur quand même
void f (int nbRow, int nbCol, int* tab)
{
  printf("%d\n", tab[nbRow-1][nbCol-1]) ;
}
```

```
input_line_47:3:30: error: subscripted value is not an array, pointer, or vector
  printf("%d\n", tab[nbRow-1][nbCol-1]) ;
                 ~~~~~~~~~~~~~^~~~~~~~
```

```
Interpreter Error:
```

[39]:
```
void f (int nbRow, int nbCol, int tab[nbRow][])
{
  printf("%d\n", tab[nbRow-1][nbCol-1]) ;
}
```

```
input_line_48:1:38: error: array has incomplete element type 'int []'
void f (int nbRow, int nbCol, int tab[nbRow][])
                                     ^
```

```
Interpreter Error:
```

[43]:
```c
// cours2-24.c
void f (int nbRow, int nbCol, int tab[][nbCol])
{
  printf("%d\n", tab[nbRow-1][nbCol-1]) ;
}
// f(2,3,tab2) ;
// Ne fonctionne pas ici à cause du noyau C++
```

[46]:
```c
void f (int nbRow, int nbCol, int tab[nbRow][nbCol])
{
  printf("%d\n", tab[nbRow-1][nbCol-1]) ;
}
// f(2,3,tab2) ;
// Ne fonctionne pas ici à cause du noyau C++
```

[5]:
```c
void printTab2D (int nbRows, int nbCol, int tab[nbRows][nbCol])
{
    for (int i = 0; i < nbRows; i++)
    {
        for (int j = 0; j < nbCol; j++)
            printf("%d ", tab[i][j]) ;
        printf("\n") ;
    }
}
// f(2,3,tab2) ;
// Ne fonctionn pas ici à cause du noyau C++
```

2.12.4 Exemple d'utilisation:

[5]:
```c
void quelJour(int n)
{
    char week[7][10] = { "lundi", "mardi", "mercredi", "jeudi", "vendredi",␣
 ↪"samedi", "dimanche" } ;
    printf("Ce sera un %s.\n", week[n % 7]) ;
    // week[0] = "Monday" ;
    strcpy(week[0],"Monday") ;
    strcpy(week[1],"Tuesday") ;
    strcpy(week[2],"Wednesday") ;
    strcpy(week[3],"Thursday") ;
    strcpy(week[4],"FridayFridayFriday") ;
    printf("It will be a %s.\n", week[n % 7]) ;
}
```

[11]:
```
quelJour(0) ;
```

```
Ce sera un lundi.
It will be a Monday.
```

2.12.5 Tableaux 2D sur le tas (allocation dynamique)

Linéarisation “à la main”

[49]:
```
int* createArray2D (int nbRow, int nbCol)
{
    int* tab = (int*)(malloc(nbRow * nbCol * sizeof(int))) ;
    for (int i = 0; i < nbRow; i++)
        for (int j = 0; j < nbCol; j++)
            tab[i*nbCol + j] = 12 ;
            // Attention la ligne ci-dessous ne fonctionne pas !!!
            // tab[i][j] = 12 ;
    return tab ;
}
```

[50]:
```
void printTabLin2D (int nbRows, int nbCol, int* tab)
{
    for (int i = 0; i < nbRows; i++)
    {
        for (int j = 0; j < nbCol; j++)
            printf("%d ", tab[i * nbCol + j]) ;
        printf("\n") ;
    }
}
```

[51]:
```
int* tab = createArray2D(5,12) ;
printTabLin2D(5,12,tab) ;
```

```
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
```

Inconvénient (léger): - nécessite nbRow * nbCol cases mémoires contigues. - on perd la syntaxe tab[i][j]

Tableaux de tableaux

[2]:
```
int** createArray2D (int nbRow, int nbCol)
{
    int** tab = (int**)(malloc(nbRow * sizeof(int*))) ;
    for (int i = 0; i < nbRow; i++)
    {
        tab[i] = (int*)(malloc(nbCol * sizeof(int))) ;
```

```
        for (int j = 0; j < nbCol; j++)
            tab[i][j] = 12 ;
    }
    return tab ;
}
```

[12]:
```
void printTab2D (int nbRows, int nbCol, int* tab[])
{
    for (int i = 0; i < nbRows; i++)
    {
        for (int j = 0; j < nbCol; j++)
            printf("%d ", tab[i][j]) ;
        printf("\n") ;
    }
}
// f(2,3,tab2) ;
// Ne fonctionn pas ici à cause du noyau C++
```

[11]:
```
int** tab = createArray2D (5,12) ;
printTab2D(5,12,tab) ;
```

```
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
```

Remarque: dans cette implémentation des tableaux 2D, tab est réellement un tableau 1D de pointeurs. Pour déclarer la fonction printTab2D, on a donc à nouveau le choix entre les 3 syntaxes rigoureusement équivalentes:

```
void printTab2D (int nbRows, int nbCol, int* tab[])
void printTab2D (int nbRows, int nbCol, int** tab)
void printTab2D (int nbRows, int nbCol, int* tab[nbRows])
```

Avantages: - On retrouve la syntaxe tab[i][j] - On a besoin nbRow fois d'une allocation mémoire de taille nbCol.

Inconvénient: - L'accès à un élément est en temps linéaire en la dimension (contre logarithmique dans l'autre implémentation). Cette implémentation est un mix entre tableau et liste chainées.

2.13 Listes Chainées

Illustration PythonTutor: cours2-31.c.

[ ]: