Licence Informatique et Vidéoludisme

Programmation avancée
de C à C++

Farès Belhadj
Revekka Kyriakoglou

Cours et exercices
10 octobre 2025

Plan

Présentation générale

(les points cités ci-après ne sont pas nécessairement abordés dans l'ordre)

- Rapides révisions de notions abordées en *Programmation Impérative*.
- Maîtriser les concepts avancés de la programmation C : renforcement pointeurs, allocation mémoire et organisation multi-fichiers ; structures de données efficaces ; mesure de performances.
- Apprendre à utiliser une sélection d'outils : `make`, utilisation/création de bibliothèques, `gnuplot`, `GraphViz`, `gprof`, `valgrind`, ...
- La généricité en C (usage des `void *`).
- Utilisation poussée des macros (*C preprocessor*).
- Introduction au C++ sans paradigme objet.
- Utilisation de la STL (C++).

Préambule 1/3

Saisir ce code, sans faire un copier/coller, dans un éditeur convenable ayant au minimum une coloration syntaxique et une capacité à l'indenter correctement.

```c
/* Pour utiliser printf */
#include <stdio.h>
/* Fonction principale du programme */
int main(int argc, char ** argv) {
  int i;
  printf("Ce␣programme␣se␣nomme␣<<␣%s␣>>\n", argv[0]);
  for(i = 1; i < argc; ++i)
    printf("Son␣paramètre␣%d␣est␣<<␣%s␣>>\n", i, argv[i]);
  return 0; /* Pourquoi 0 ? Peut-être remplacé par EXIT_SUCCESS */
}
```

Code source 2.1 – Premier code du semestre `print_argv.c`

Puis le compiler

```
$ gcc -Wall -Wextra print_argv.c -o print_argv
```

Puis, si tout va bien, l'exécuter

```
$ ./print_argv "Hello World"
Ce programme se nomme << ./print_argv >>
Son paramètre 1 est << Hello World >>
```

Des questions ?

Préambule 2/3

Pour le précédent code, avez-vous créé un répertoire spécifique pour le tester ?

Disons que votre répertoire de travail soit `work`, un bonne option de travail aurait été d'avoir un répertoire :
`~/work/cours/ProgIAvancee/Seance01/print_argv/`


Au fait, le `~/` est la référence vers votre *home directory* (racine de votre compte utilisateur

Préambule 3/3

Qu'est censé faire le code ci-après ?
Le fait-il vraiment ?

```c
#include <stdio.h>
void swap(int a, int b) {
  int c;
  c = a; /* c sauvegarde la valeur stockée dans a */
  a = b; /* a prend la valeur stockée dans b */
  b = c; /* b prend la valeur stockée dans c, soit le a initial */
}
int main(void) {
  int a = 7, b = 42;
  /* affichage de a et b avant l'échange */
  printf("a = %d, b = %d\n", a, b);
  /* échange (?) à l'aide de swap */
  swap(a, b);
  /* affichage de a et b après l'échange */
  printf("a = %d, b = %d\n", a, b);
  return 0;
}
```

Code source 2.2 – Problème simple ? swap.c

Quelqu'un·e souhaite apporter des modifications ?

Notions de base – les variables 1/6

- Qu'est-ce qu'une variable ?
- Comment déclarer une variable ?
- Quels types de variable et quel impact sur la mémoire ?
- Comment affecter une **valeur** à une **variable** ?

Notions de base – les variables 2/6

- Qu'est-ce qu'une variable (pour nous) ?
  - « **symbole** qui associe un **nom** à une **valeur** » 
    (ça n'est pas plutôt le rôle d'une constante ?)
  - Une variable est un symbole, identifié par son nom, qui associe ce nom à zone mémoire. La taille et l'emplacment de cette dernière diffère en fonction du type – ou *type specifier* – et de l'emplacement où a été déclarée la variable. Certains « *specifiers* » supplémentaires, appelées *qualifiers*, peuvent modifier l'emplacement et la nature de la variable.
- Comment déclarer une variable ?
- Quels types de variable et quel impact sur la mémoire ?
- Comment affecter une **valeur** à une **variable** ?

Variable https://fr.wikipedia.org/wiki/Variable_(informatique)
Qualifier https://en.wikipedia.org/wiki/Type_qualifier

Notions de base – les variables 3/6

- Qu'est-ce qu'une variable ?
- Comment déclarer une variable ?

```
[qualifier1 [qualifier2 [...]]] <type> [* [* [...]]] nom_de_la_variable;
```

Exemples :

```
float moyenne;
static int compteur;
char * ptr;
```

Notez qu'un nom de variable bien choisi vaut mieux que plusieurs lignes de commentaires. Notez aussi qu'il n'y a aucune garantie (en C) qu'une variable dynamique (*i.e.* non statique) soit initialisée à zéro. À l'inverse, une variable ayant un *qualifier* `static` est mise à zéro (ou `NULL` si pointeur) et mise dans le *BSS segment* (voir aussi *Data segment*) du binaire ou éxécutable généré.

- Quels types de variable et quel impact sur la mémoire ?
- Comment affecter une **valeur** à une **variable** ?

BSS Segment https://en.wikipedia.org/wiki/.bss
Data Segment https://en.wikipedia.org/wiki/Data_segment

→ Plus tard, nous ferons un usage et une analyse poussés de ces notions.

Notions de base – les variables 4/6

- Qu'est-ce qu'une variable ?
- Comment déclarer une variable ?
- Quels types de variable et quel impact sur la mémoire ?
  1. **Bien retenir** que le type d'une variable a pour principal effet de déterminer le nombre de bits que cette dernière occupe en mémoire.
  2. L'autre utilité du type est liée à **l'interprétation** qui est faite de ce type ; principalement par le compilateur mais parfois par des fonctions comment `printf`. Nous donnerons une attention particulière au fait qu'une valeur affectée à une variable corresponde à son type.

     Arrêtez de penser qu'on stocke littéralement des caractères en RAM !

     ```
     char byte1 = 'A', byte2 = 65;
     /* byte1 et byte2 contiennent strictement la même valeur c-à-d (65)10 ou (01000001)2 */
     ```
- Comment affecter une **valeur** à une **variable** ?

La table ASCII `https://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange`
est une table d'indirection (LookUpTable) utilisée par le compilateur ou autres fonctions traitant des entrées/sorties

Notions de base – les variables 5/6

- Qu'est-ce qu'une variable ?
- Comment déclarer une variable ?
- Quels types de variable et quel impact sur la mémoire ?
- Comment affecter une **valeur** à une **variable** ?
  - À l'aide de l'opérateur affectation « = », lors de la déclaration de la variable ou plus tard, **ou bien**, par passage de paramètres (*i.e.* valable pour les arguments d'une fonction). Il y a d'autres moyens tels que `memset`/`memcpy` ou autres manipulations passant par des pointeurs : à voir plus tard.
  - Dans tous les cas il faut bien distinguer ce qu'est une **l-value** et une **r-value** ; **l** et **r** faisant respectivement référence à *left* et *right*, et ceci est lié à l'emplacement du symbole par rapport à l'opérateur d'affectation « = ». Ainsi, une **l-value** est un contenant, comme par exemple une boîte, et une **r-value** fait référence à une donnée, ou une information, qui correspond *in fine* à une valeur.

Cette notion est fondamentale, assurons-nous qu'elle soit maîtrisée par tou·te·s !

↓

Notions de base – les variables 6/6

- **(SUITE 1)** Comment affecter une **valeur** à une **variable** ?

  Insistons sur la **l-value** et la **r-value** :

  l-value : est un symbole identifiant une zone mémoire permettant le stockage d'une donnée. Ce symbole peut donc fournir au compilateur une information sur :
  - l'emplacement de la zone, soit l'adresse du premier octet de cette zone ;
  - la taille et le type de la donnée que nous pouvons y mettre, cela sert à vérifier la cohérence de l'affectation, car *in fine* ce que nous y mettons est toujours un packet de 0 et de 1 → du binaire.

  La **l-value** ne peut donc être une constante (comme `7`, `42`, `0x7ffee9239a68`, `...`), une fonction ou une expression (comme `(a + 1)`, `(a * b)`, `...`). Nous excluons de cette restriction les situations où nous ferons usage de **l'opérateur de déréférencement « * »**.

  r-value : est une donnée (au sens une ou plusieurs valeurs) qui peut être extraite à partir de symboles : une variable (ex. `nb_elements`), une constante (ex. `42` ou `mon_approximation_de_pi`), un retour de fonction (ex. `pgcd(p, q)`) ou le résultat d'une quelconque expression (ex. `((a + b) / c)`).

```c
#include <stdio.h>
int carre(int x) {
  return x * x; /* r-value, le contenu de x est récupéré pour le calcul */
}
int main(void) {
  int a, b, c, * p; /* a, b et c sont des conteneurs d'int, et p est
                       un conteneur d'adresse de int */
  a = 7; /* la valeur 7 (111 en binaire) est affectée à la zone
            mémoire "labélisée a" (l-value) */
  b = carre(a); /* a est une r-value, donc la r-value renvoyée par
                   carre(7) -> 49 est affectée à la zone mémoire
                   "labélisée b" (l-value) */
  c = (b - a); /* la différence entre le contenu extrait de b et celui
                  de a est affectée à la l-value c */
  p = &a; /* p (l-value) stocke l'adresse du conteneur "labellisé a" */
  /* &p = NULL; impossible, l'adresse de p est constante, indice en
     mémoire du conteneur, il est fixe et ne peut etre une l-value */
  /* a + 2 = 11; impossiblr, (a + 2) est une expression qui vaut, pas
     de sens d'affecter une valeur à un nombre */
  *p = 9; /* p stocke l'adresse de a, *p est physiquement l'entité qui
             se trouve à l'adresse stockée dans p, donc *p est a,
             ainsi 9 est affectée à la case mémoire dont l'adresse est
             stockée dans p, c-à-d la case labellisée a */
  *(p + 1) = carre(a); /* syntaxiquement correct mais dangeureux, p
                          contient l'adresse de a => (p + 1) est l'@
                          du int voisin (à sa droite en RAM), *(p + 1)
                          est donc physiquement ce voisin de droite,
                          on écrit dedans le carré de a */
  printf("a = %d, b = %d, c = %d, p = %p (&a = %p)\n", a, b, c, p, &a);
  return 0;
}
```

Code source 2.3 – Concepts de **l&r-values** `lvaluervalue.c`, à tester

Notions de base – les fonctions (en C) 1/4

- Qu'est-ce qu'une fonction ?
- Déclarer ET définir une fonction ?
- Quelques *qualifiers* et leurs impacts ?

Notions de base – les fonctions (en C) 2/4

- Qu'est-ce qu'une fonction ?

  Dans un programme, une fonction peut être résumée à une séquence d'instructions regroupées pour réaliser une tâche (généralement actions ou calculs) particulière, dans un cadre délimité [1] ou restreint, lié à ses paramètres, s'ils existent. Une fonction peut donc être appelée (ou utilisée) plusieurs fois au sein du programme, ou, dans le cas spécifique de l'écriture d'une bibliothèque de fonctions, être réservée à un usage ultérieur par d'autres programmes utilisant la dite bibliothèque.
- Déclarer ET définir une fonction ?
- Quelques *qualifiers* et leurs impacts ?

1. Voir néanmoins le cas particulier de la fonction à effet de bord : https://fr.wikipedia.org/wiki/Effet_de_bord_(informatique)

Notions de base – les fonctions (en C) 3/4

- Qu'est-ce qu'une fonction ?
- Déclarer ET définir une fonction ?

Déclarer une fonction consiste à donner un prototype, le plus complet possible, permettant de la qualifier[2], de donner son type de retour et d'informer sur ses paramètres. En donnant, à la suite, le type de chaque paramètre, nous déduisons leur nombre. Si le premier paramètre est `void` cela indique que la fonction n'en a pas. Il est important de déclarer une fonction avant de la définir ; la déclaration se fait en haut du fichier C quand les fonctions sont statiques et dans un fichier d'en-têtes (fichier `.h`) quand les fonctions sont externes. Ci-après, une forme générale d'un prototype :

```
[qualifier1 [...]] <type> [* [...]] nom_de_la_fonction(<void> | <type1>[, <type2>[...]]);
```

Définir une fonction consiste à donner « ***dans les grandes lignes***[3] » son prototype, directement (à la place du «;») suivi du corps de la fonction entre accolades. Si le type de retour de la fonction est autre que `void`, alors le corps de la fonction doit avoir un retour (*i.e.* `return ...;`) correspondant à ce type dans toutes les situations rencontrées (tous les branchements du code). Ci-après, une forme générale d'une définition :

```
<type> [* [...]] nom_de_la_fonction(<void> | <type1 nom1>[, <type2 nom2>[...]]) {
   /* les instructions qui constituent le corps de la fonction */
   /* ne pas oublier un return lié au type dans toute situation */
}
```

- Quelques *qualifiers* et leurs impacts ?

---

2. Définir ses particularités : sans être exhaustifs, citons `extern` contre `static`, exportable au sein d'une bibliothèque ou non, ou encore *inlinable* (cf. `inline`) ou non (les deux derniers points sont compilateur-spécifiques).

3. Si la fonction est correctement prototypée (déclarée) avant sa définition, remettre les *qualifiers* est optionnel, voire non-productif. Par contre, dans le cas de la définition de fonction, il est nécessaire de nommer les paramètres (en plus des types).

Notions de base – les fonctions (en C) 4/4

- Qu'est-ce qu'une fonction ?
- Déclarer ET définir une fonction ?
- Quelques *qualifiers* et leurs impacts ?

  Le principal couple de *qualifiers* à connaître est : `static` ou `extern`. Le premier indique que la fonction a une portée limitée au fichier (à partir de sa déclaration) alors que le second lui permet d'avoir une portée au delà du fichier (*i.e.*le fichier `.c`), évidemment si son prototype est donné (généralement dans le fichier *header* `.h` inclus via la directive de préprocesseur `#include`).

   La vidéo `https://youtu.be/Q_XliWuKsKw` peut aider à comprendre la structuration multi-fichiers d'un programme et les concepts de `static` et `extern`. Prenez le temps de la visualiser en entier.

Notions de base – Informations utiles 1/5

Type	Occupation mémoire	Plage de valeurs
char	1 octet	-128 à 127
unsigned char	1 octet	0 à 255
int	2 ou 4 octets	Selon l'architecture
unsigned int	2 ou 4 octets	Selon l'architecture
short	2 octets	-32768 à 32767
unsigned short	2 octets	0 à 65535
long	4 octets	-2147483648 à 2147483647
unsigned long	4 octets	0 à 4294967295
long long	8 octets	-2^{63} à $2^{63}-1$
unsigned long long	8 octets	0 à $2^{64}-1$
Type à virgule flottante		
float	4 octets	3.4×10^{-38} à 3.4×10^{38} (IEEE 754)
double	8 octets	1.7×10^{-308} à 1.7×10^{308} (IEEE 754)
long double	10 octets	3.4×10^{-4932} à 3.4×10^{4932} (IEEE 754)

Table – Types de données standards (C89).

En pratique, les types de données évoluent et changent en fonction de l'architecture. La norme C99 spécifie de nouveaux types dont la taille est fixe :

Nouveaux type dans `stdint.h` https://en.wikibooks.org/wiki/C_Programming/stdint.h

Notions de base – Informations utiles 2/5

Priorité	Opérateur	Description	Exemple
0	`()`	appel de fonction, associativité	`foo();` `a = (b + c) * d;`
	`[]`	indexation	`int tab[3];` `tab[0] = tab[1] = tab[2] = 0;`
	`.`	nommage d'un champ	`obj.cdr = NULL;`
	`->`	nommage indirect de champ	`pt->cdr = NULL;`
1	`!`	négation	`!a` est vraie si `a` est fausse
	`~`	complément à 1	`a & (~a)` → `0`
	`-`	opposé	
	`++`	incrémentation	`i++; ++i;`
	`--`	décrémentation	`i--; --i;`
	`&`	adresse	`int i, *pt;` `pt = &i;`
	`*`	valeur indirecte	`*pt = 0; /* donne i = 0 */`
	`(type_de_donnée)`	force le type (cast)	`int i = (int)1.5;`
	`sizeof`	taille en octets	`s = sizeof i;`
2	`*`	Multiplication	
	`/`	Division	
	`%`	Modulo	
3	`+`	Addition	
	`-`	Soustraction	
4	`<<`	Décalage à gauche	
	`>>`	Décalage à droite	

Table – Table des priorités des opérateurs C/C++ (1/2)

Notions de base – Informations utiles 3/5

5	<	Strictement inférieur	
	<=	Inférieur ou égal	
	>	Strictement supérieur	
	>=	Supérieur ou égal	
6	==	Egal	
	!=	Différent	
7	&	“et” binaire	
8	^	“ou” exclusif binaire	
9	|	“ou” binaire	
10	&&	“et” logique	
11	||	“ou” logique	
12	?:	conditionnelle	`c = (a < b) ? a : b;`
13	= *= /= %= += -= ^= &= <<= >>= |=	Affectations	
14	,	Séquence	

Table – Table des priorités des opérateurs C/C++ (2/2)

Notions de base – Informations utiles 4/5

Format	Type de donnée	
%d	Entier	`short, int`
%i	Entier	`short, int`
%u	Entier non signé	`unsigned short, unsigned int`
%zu	taille	`size_t`
%o	Entier octal	`short, int`
%x, %X	Entier hexadécimal	`short, int`
%l, %ld, %li, %lu, %lo, %lx, %lX	Entier long	`long`
%lld, %lli, %llu, %llo, %llx, %llX	Entier long 64 bits	`long long`
%c	Caractère ASCII	`char, unsigned char`
%f, %F, %g, %G	Flottant	`float, double`
%e, %E	Flottant (exponentiel)	`float, double`
%Lf, %LF, %Lg, %LG, %Le, %LE	long double	`long double`
%s	Chaîne de caractères	`char *`
%p	Pointeur	`(type) *`

Table – Formats pris en charge par la famille de fonctions `printf`

Attention ! L'abus d'utilisation de fonctions « `print` » est dangereux pour la suite de votre cursus. Il est important de faire strictement ce qui est attendu de vous, exemple : si on vous demande d'écrire la fonction `foo` qui calcule et retourne « *bidule* », il **n'est pas attendu** d'y mettre de l'affichage (*i.e.*du `print`).

Notions de base – Informations utiles 5/5

Séquence d'échappement	Action
\n	Nouvelle ligne (`new line`)
\t	Tabulation horizontale
\v	Tabulation verticale
\b	Retour d'un caractère arrière (backspace)
\r	Retour chariot (`carriage return`)
\f	Saut de page (`form feed`)
\a	Signal sonore (`alarm`)
\'	Affiche une apostrophe
\"	Affiche un guillemet
\\	Affiche un `Backslash`
\ddd	Affiche les codes ASCII en octale
\xddd	Affiche les codes ASCII en hexadécimale

Table – Séquences d'échappement dans les chaînes de caractères

Exercices de révision 1/6

Faire en sorte que la fonction swap **échange** les contenus de variables (*i.e.* conteneurs) passées en paramètre, puis modifier l'appel.

```c
#include <stdio.h>
void swap(int a, int b) {
  int c;
  c = a; /* c sauvegarde la valeur stockée dans a */
  a = b; /* a prend la valeur stockée dans b */
  b = c; /* b prend la valeur stockée dans c, soit le a initial */
}
int main(void) {
  int a = 7, b = 42;
  /* affichage de a et b avant l'échange */
  printf("a = %d, b = %d\n", a, b);
  /* échange (?) à l'aide de swap */
  swap(a, b);
  /* affichage de a et b après l'échange */
  printf("a = %d, b = %d\n", a, b);
  return 0;
}
```

Code source 2.4 – À corriger (swap.c)

Exercices de révision 2/6

Vous arrivez à quelque chose d'équivalent à ça concernant `swap` ?

```c
void swap(int * pa, int * pb) {
  int c;
  c = *pa; /* c sauvegarde la valeur stockée dans le conteneur pointée
          par pa (*pa est "ce vers quoi pointe pa") */
  *pa = *pb; /* le conteneur pointé par pa prend la valeur stockée
          dans le conteneur pointée par pb */
  *pb = c; /* le conteneur pointé par pb prend la valeur stockée dans c,
        soit le *pa initial */
}
```

Code source 2.5 – Nouvelle fonction (`swap`)

Comment appeler `swap` ? (c'est à dire par quoi remplacer `swap(a, b);` dans le `main` de `swap.c`)

Exercices de révision 3/6

Pour le code donné ci-après :

```
char x = 7, y = 42;
x = (x ^ y);
y = (x ^ y);
x = (x ^ y);
```

1. Représentez (dessinez) chaque contenu de variable en binaire sur 6 bits (*i.e.* pour chacune, avoir six cases et y mettre soit des 0 soit des 1).
2. Réalisez les opérations ci-dessus, ne pas oublier de mettre à jour les contenus de vos variables après chaque XOR.
3. En déduire quelle est la “fonction” de ces trois instructions utilisant le XOR.
4. Modifiez `swap.c` pour en avoir une dernière version capable de se passer de la variable temporaire `c`.

Exercices de révision 4/6

Pour le code donné ci-après :

```c
#include <stdio.h>
static void d2b(unsigned char n) {
  /* posez le calcul pour en comprendre le sens */
  unsigned char mask = (1 << (((sizeof n) << 3) - 1));
  for( ; mask ; mask >>= 1) {
    if(n & mask)
      putchar('1');
    else
      putchar('0');
  }
  putchar('\n');
}
int main(void) {
  d2b(214);
  return 0;
}
```

1. Calculez à la main la valeur initiale de `mask` (**Spoiler Alert**, la bonne réponse est $1 \times 10^2 + 2 \times 10^1 + 8 \times 10^0$).
2. Faire tourner l'exemple à la main (donc `d2b(214)`).
3. En déduire l'utilité de la fonction `d2b`.

Exercices de révision 5/6

Questions faciles

Écrire l'ensemble des fonctions[4] dans le même fichier C et les tester[5] dans le `main`.

1. Écrire la fonction dont le prototype est **`static int ivmax(int * t, int n);`** qui retourne **l'indice de la plus grande valeur** contenue dans le tableau `t` passé en argument.
2. Écrire la fonction dont le prototype est **`static float somme(float * t, int n);`** qui retourne **la somme** des `n` éléments du tableau `t` passé en argument.
3. Écrire la fonction dont le prototype est **`static float pp(float * t, int n);`** qui retourne **le plus petit** des `n` éléments du tableau `t` passé en argument.
4. Écrire la fonction récursive dont le prototype est **`static long long puissance_r(int x, unsigned int n);`** qui retourne x^n.
5. Écrire la fonction itérative dont le prototype est **`static long long puissance_i(int x, unsigned int n)`** qui retourne x^n.

---

4. De préférence déclarer en haut du fichier et définir plus bas, de manière à ce que la première fonction définie soit la fonction `main`.

5. N'attendez pas de tout écrire pour commencer à tester : pour chaque fonction, une fois écrite, ajouter les instructions dans le `main` qui permettent de la tester.

Exercices de révision 6/6

Allons un peu plus loin. Écrire l'ensemble des fonctions[6] dans le même fichier C et les tester[7] dans le `main`.

1. Écrire la fonction **`static void inv_i(int t[], int n);`** qui inverse l'ordre des éléments dans `t` (`n` étant le nombre d'éléments de `t`).
2. Écrire la fonction **`static void inv_s(char * t);`** qui inverse l'ordre des éléments dans la chaîne de caractères `t`.

   
   Pour information, nous considérons qu'une chaîne de caractères est bien formée si et seulement si le pointeur sur la chaîne n'est pas `NULL` et que la chaîne se termine par le caractère `'\0' == 0`.
3. Réécrire votre propre version de la fonction `strcpy`[8] en la déclarant **`static char * my_strcpy(char * dest, const char * src);`** elle copie la chaîne pointée par `src` dans la zone mémoire pointée par `dest`.
4. Réécrire votre propre version de la fonction `strdup`[9] en la déclarant **`static char * my_strdup(const char * s);`** elle renvoie un pointeur sur une nouvelle chaîne de caractères qui est dupliquée depuis `s`.

---

6. De préférence déclarer en haut du fichier et définir plus bas, de manière à ce que la première fonction définie soit la fonction `main`.
7. N'attendez pas de tout écrire pour commencer à tester : pour chaque fonction, une fois écrite, ajouter les instructions dans le `main` qui permettent de la tester.
8. Faire `man strcpy` dans votre terminal pour obtenir le manuel de la fonction.
9. Faire `man strdup` dans votre terminal pour obtenir le manuel de la fonction.

Travail de révision

- Écrire un programme (`main` + fonction(s)) pour le calcul **rapide**[10] de la puissance : x^n avec x et n entiers.
- Écrire en `C` une implémentation du Crible d'Ératosthène. Ce programme prendra en entrée `N`, en argument de l'exécutable. `N` est considéré comme la borne supérieure des nombres premiers à trouver. Le programme doit afficher (à l'aide de `printf`) tous les nombres premiers trouvés (`N` peut en faire partie s'il est premier, il sera ainsi le dernier à être affiché). Merci de vous inspirer de l'algorithme décrit sur la page correspondante sur Wikipédia :

  « L'algorithme procède par élimination : il s'agit de supprimer d'une table des entiers de 2 à N tous les multiples d'un entier. En supprimant tous les multiples, à la fin il ne restera que les entiers qui ne sont multiples d'aucun entier, et qui sont donc les nombres premiers.
  On commence par rayer les multiples de 2, puis à chaque fois on raye les multiples du plus petit entier restant.
  On peut s'arrêter lorsque le carré du plus petit entier restant est supérieur au plus grand entier restant, car dans ce cas, tous les non-premiers ont déjà été rayés précédemment.
  À la fin du processus, tous les entiers qui n'ont pas été rayés sont les nombres premiers inférieurs à N. »

L'ensemble du travail doit être réalisé dans un même dossier (ex. `PA_DM01`) qui contiendra un fichier C par question et un fichier `Makefile` qui se chargera de compiler les deux programmes.

La vidéo `https://youtu.be/EOFg3zr7DXY` vous aidera à écrire votre `Makefile`, prenez le temps de la visualiser en entier.

10. Ce n'est donc pas une des versions vues précédemment dans la première série d'exercices.

La rue de la RAM 1/17

```c
#include <stdio.h>
int main(void) {
  int i;
  char une_maison = 97;
  char * une_adresse = &une_maison;
  char des_maisons[] = {'A', 'A' + 1, 67, 'Z' - 22};
  char * un_ptr_libre = NULL;
  printf("Affichage des adresses :\n"
     "\t@une_maison = %p\n"  "\t@une_adresse = %p\n"
     "\t@des_maisons = %p\n" "\t@un_ptr_libre = %p\n",
     &une_maison, &une_adresse, &des_maisons, &un_ptr_libre);
  printf("Affichage des valeurs en hexa castées en entier :\n"
     "\tune_maison = %x\n"    "\tune_adresse = %llx\n"
     "\tdes_maisons = %llx\n" "\tun_ptr_libre = %llx\n",
     une_maison, (long long)une_adresse,
     (long long)des_maisons, (long long)un_ptr_libre);
  printf("Affichage des char comme caractères :\n"
     "\tune_maison = '%c'\n", une_maison);
  for(i = 0; i < (int)(sizeof des_maisons / sizeof * des_maisons); ++i)
    printf("\tdes_maisons[%d] = '%c'\n", i, des_maisons[i]);
  return 0;
}
```

Code source 3.1 – Commençons par un code affichant des adresses de conteneurs ainsi que leur contenu (`mem_basics.c`)

Le copier/coller est susceptible de générer des erreurs difficiles à repérer, veuillez bien indenter et lire attentivement la sortie du compilateur.

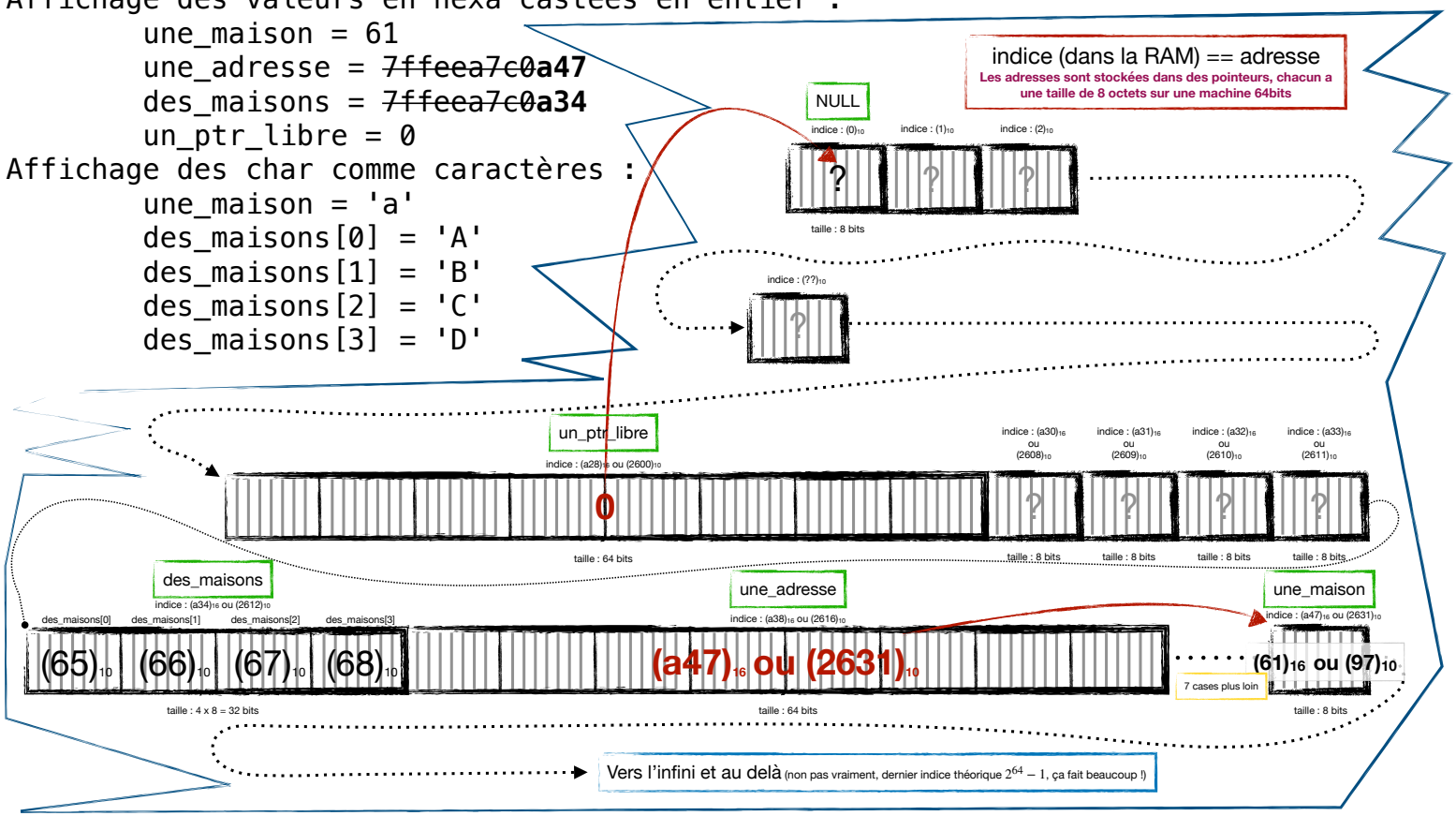
La rue de la RAM 2/17

Après compilation et exécution du code :

```
$ gcc -Wall -Wextra mem_basics.c -o mem_basics && ./mem_basics
Affichage des adresses :
        @une_maison = 0x7ffeea7c0a47
        @une_adresse = 0x7ffeea7c0a38
        @des_maisons = 0x7ffeea7c0a34
        @un_ptr_libre = 0x7ffeea7c0a28
Affichage des valeurs en hexa castées en entier :
        une_maison = 61
        une_adresse = 7ffeea7c0a47
        des_maisons = 7ffeea7c0a34
        un_ptr_libre = 0
Affichage des char comme caractères :
        une_maison = 'a'
        des_maisons[0] = 'A'
        des_maisons[1] = 'B'
        des_maisons[2] = 'C'
        des_maisons[3] = 'D'
```

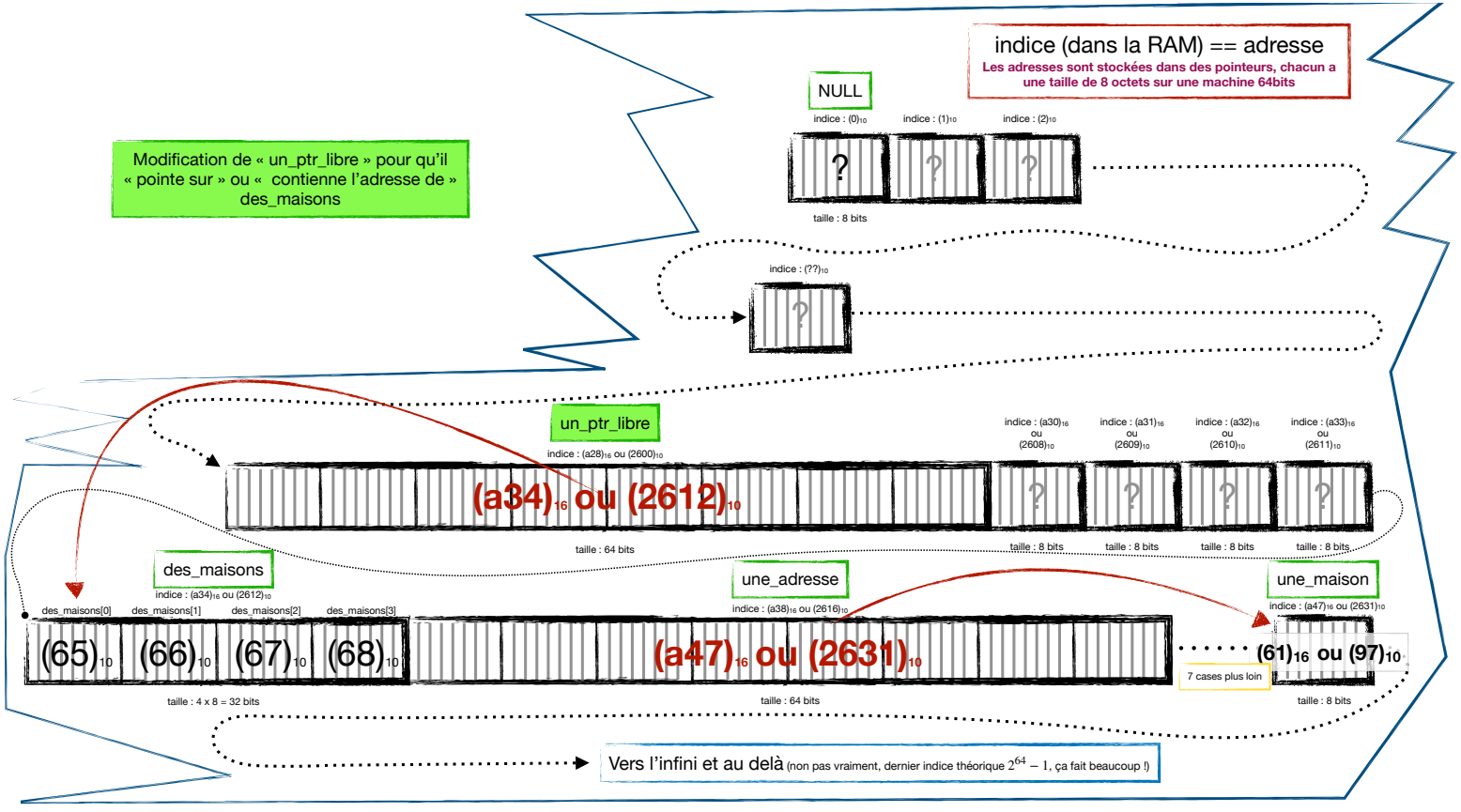
La rue de la RAM 3/17

```
$ gcc -Wall -Wextra mem_basics.c -o mem_basics && ./mem_basics
Affichage des adresses :
        @une_maison = 0x7ffeea7c0a47
        @une_adresse = 0x7ffeea7c0a38
        @des_maisons = 0x7ffeea7c0a34
        @un_ptr_libre = 0x7ffeea7c0a28
Affichage des valeurs en hexa castées en entier :
        une_maison = 61
        une_adresse = 7ffeea7c0a47
        des_maisons = 7ffeea7c0a34
        un_ptr_libre = 0
Affichage des char comme caractères :
        une_maison = 'a'
        des_maisons[0] = 'A'
        des_maisons[1] = 'B'
        des_maisons[2] = 'C'
        des_maisons[3] = 'D'
```



La rue de la RAM 4/17

Exercice :
en n'utilisant que `un_ptr_libre` et `des_maisons`, ajoutez une boucle affichant les caractères relatifs au contenu de `des_maisons`. La compilation ne doit provoquer aucun *warning*, même avec les options de compilation `-Wall -Wextra`



La rue de la RAM 5/17

Les pointeurs : comment lire la déclaration ?
`char *** ptr = NULL;`

- **ptr** est une variable de type pointeur, elle existe physiquement en RAM (c'est un conteneur), elle contient donc une valeur, ici cette valeur est `((void*)0)`
- **char *** ptr** nous indique que `ptr` est un pointeur de pointeur de pointeur vers un `char`
- **char **** ptr** nous indique que `*ptr` est **potentiellement** un pointeur de pointeur vers un pointeur de `char`. Néanmoins, attention, `*ptr` n'est pas lisible car `ptr` ne peut être déréférencé (« devenir ce vers quoi il pointe ») car il pointe sur la zone d'indice mémoire 0. Ceci est valable pour tous les cas listés ci-dessous
- **char *** ptr** nous indique que `**ptr` est **potentiellement** un pointeur vers un pointeur de pointeur de `char` (non lisible aussi dans le cas présent)
- **char *** ptr** nous indique que `***ptr` est **potentiellement** un `char` (non lisible aussi dans le cas présent)

La rue de la RAM 6/17

Dessinez cet exemple puis faites l'exercice dont l'énoncé est en commentaire dans le code.

```c
#include <stdio.h>
int main(void) {
  short a = 42, * b = &a, ** c = &b, *** d = &c;
  printf("a = %d, son adresse est %p\n", a, &a);
  printf("b = %p, son adresse est %p\n", b, &b);
  printf("c = %p, son adresse est %p\n", c, &c);
  printf("d = %p, son adresse est %p\n", d, &d);
  /* EXERCICE ! (FACILE) */
  /* en n'utilisant que printf et d, affichez
   * le contenu de chaque variable d, c, b, et a
   */
  /* printf("%p, %p, %p, %d\n", ..............);
  return 0;
}
```

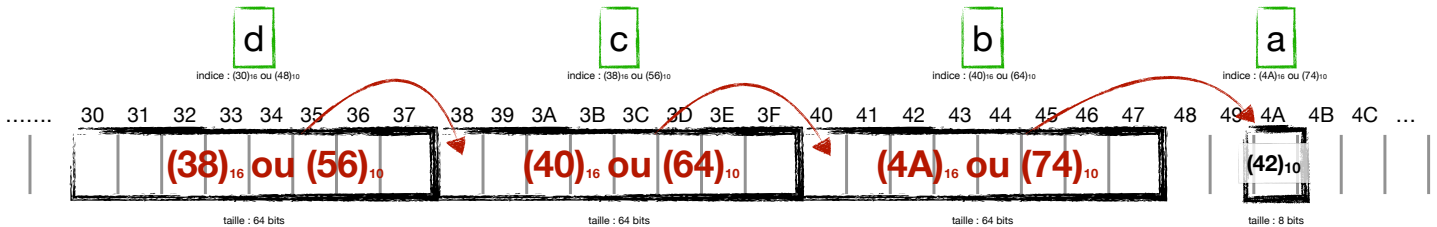
Code source 3.2 – Illustration d'utilisation de pointeur de pointeur de ... (mem_basics_02.c)

```
$ gcc -Wall -Wextra  mem_basics_02.c -o mem_basics_02 && ./mem_basics_02
a = 42, son adresse est 0x7ffee6fc8a4a
b = 0x7ffee6fc8a4a, son adresse est 0x7ffee6fc8a40
c = 0x7ffee6fc8a40, son adresse est 0x7ffee6fc8a38
d = 0x7ffee6fc8a38, son adresse est 0x7ffee6fc8a30
```

La rue de la RAM 7/17

État de la mémoire de l'exemple précédent :

```
$ gcc -Wall -Wextra  mem_basics_02.c -o mem_basics_02 && ./mem_basics_02
a = 42, son adresse est 0x7ffee6fc8a4a
b = 0x7ffee6fc8a4a, son adresse est 0x7ffee6fc8a40
c = 0x7ffee6fc8a40, son adresse est 0x7ffee6fc8a38
d = 0x7ffee6fc8a38, son adresse est 0x7ffee6fc8a30
```



La rue de la RAM 8/17

À faire :

Faire un schéma du contenu connu de mémoire liée à ce programme. Sans tester sur une machine, essayez de deviner quel affichage nous obtenons suite à son exécution. Enfin, testez sur machine pour vérifier votre proposition.

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
int main(void) {
  char str01[] = "Hello", str02[] = "world";
  char ** ptr = NULL;
  ptr = malloc(2 * sizeof *ptr);
  assert(ptr);
  ptr[0] = str01;
  ptr[1] = str02;
  printf("%c\n", *ptr[1]);
  printf("%c\n", (*ptr)[1]);
  free(ptr);
  return 0;
}
```

Code source 3.3 – Qu'affiche ce programme??? (`mem_basics_03.c`)

La rue de la RAM 9/17

Retour sur l'allocation dynamique de mémoire :

- Pour un pointeur **data** de n'importe quel type :
  `<type> * [*[*[...]]] data = NULL;`
- Demander à **malloc**[11] d'allouer une zone mémoire correspondant au nombre d'octets souhaités. Le nombre d'octets peut différer du nombre d'éléments qu'on imagine y mettre, par conséquent, pour **nb_elements** souhaités, il faudra allouer **nb_elements** $\times$ **la_taille_d_un_element**. Pour **nb_elements** donné, le *pattern* d'appel à **malloc** est toujours le même, le seul élément qui change est le nom du pointeur vers la nouvelle zone demandée :
  `data = malloc(nb_elements * sizeof *data);`
- Un test vérifiant que l'allocation est effective (résultat différent de `NULL`) est nécessaire. Pour faire simple, nous recommandons l'usage de **assert** (cf. le `man` et inclure `assert.h`) :
  `assert(data);`
- Utilisez votre mémoire allouée, sans jamais « perdre de vue » le pointeur vers le début de la zone.
- Libérez la mémoire allouée, à l'aide de **free**, quand vous n'en avez plus l'usage :
  `free(data);`
- Remettez le pointeur à `NULL` afin d'éviter de faire des « bêtises » avec :
  `data = NULL;`

11. Fonction de la bibliothèque standard **stdlib** (à inclure donc) qui alloue un nombre d'octets passés en argument et retourne un pointeur sans type (*i.e.* **void ***) vers la zone allouée. Voir **man malloc** pour lire en détail la documentation de cette fonction ainsi que celles des fonctions associées ou équivalentes telle que **calloc, realloc, free ...**

La rue de la RAM 10/17

Entraînement (1)

À faire ou à lire (solution dans le slide suivant) : créer une structure pour un point 3D [12], contenant trois champs flottants `x`, `y` et `z`. Proposer deux différentes fonctions pour remplir un tableau d'éléments de cette structure (`n`, le nombre d'éléments sera donné) avec des valeurs pseudo-aléatoires [13] comprises dans l'intervalle $[0, 1[$. L'une des fonctions de remplissage, **void initpoint3dv(point3d_t * p3darray, int n);**, remplit les **n** éléments de type **point3d_t**, l'autre, **void init3fv(float * triplefloatsarray, int n);**, remplit les $\mathbf{3} \times \mathbf{n}$ flottants du tableau. Pour deux tableaux statiques de **point3d_t**, de même taille et non initialisés, appeler pour chacun une fonction de remplissage différente. Comparer les deux tableaux à l'aide de la fonction `memcmp` (voir le *man*) et afficher le résultat de la comparaison.

---

12. Par exemple, cette structure pourra être nommée : `point3d_t`

13. Utilisez la fonction **rand()** de la `stdlib` (cf. **man 3 rand**). Nous utiliserons **srand** avec la même graine, avant chaque appel de la fonction de remplissage, afin d'avoir la même chaîne de nombres pseudo-aléatoires dans les deux cas.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct point3d point3d_t;
struct point3d {
  float x, y, z;
};
void initpoint3dv(point3d_t * p3darray, int n) {
  int i;
  for(i = 0; i < n; ++i) {
    p3darray[i].x = rand() / (RAND_MAX + 1.0f);
    p3darray[i].y = rand() / (RAND_MAX + 1.0f);
    p3darray[i].z = rand() / (RAND_MAX + 1.0f);
  }
}
void init3fv(float * triplefloatsarray, int n) {
  int i, _3n = 3 * n;
  for(i = 0; i < _3n; ++i)
    triplefloatsarray[i] = rand() / (RAND_MAX + 1.0f);
}
int main(void) {
  point3d_t A[100], B[100];
  srand(42); initpoint3dv(A, sizeof A / sizeof *A);
  srand(42); init3fv((float *)B, sizeof B / sizeof *B);
  if(memcmp(A, B, sizeof A) == 0)
    printf("même contenu\n");
  else
    printf("contenu différent\n");
  return 0;
}
```

Code source 3.4 – proposition de solution au slide précédent (`point3d.c`)

La rue de la RAM 11/17

Entraînement (2)

A faire en cours (TD à venir) : récupérer des lignes[14] de texte[15] sur l'entrée standard, les stocker dans un `char **` (le premier indice faisant référence au numéro de ligne), les réafficher en mettant les lettres en majuscule, sortir « proprement ».

14. Nous fixons une limite de 1024 caractères par ligne, le `'\0'` compris.
15. Je recommande l'utilisation de la fonction `fgets`.

La rue de la RAM 12/17

Lecture de code :

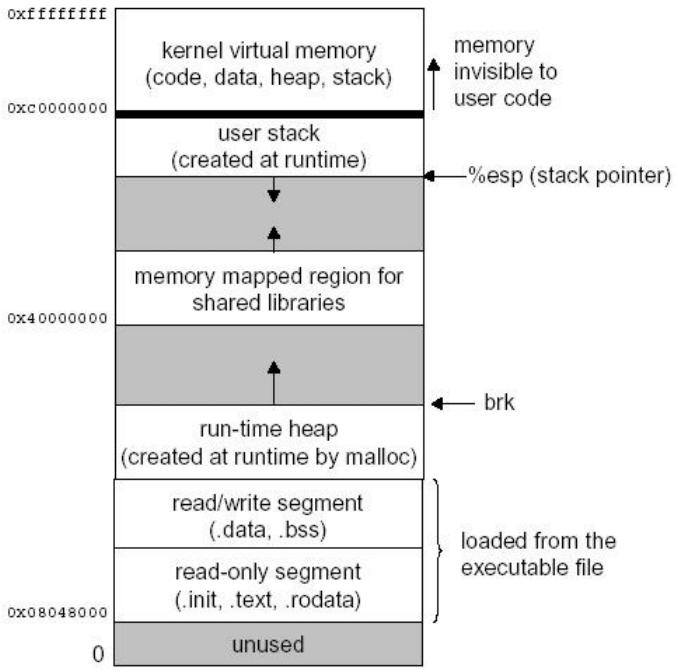
Que fait cette fonction ?

```
char * foo(char * d, const char * s) {
  char * retour = d;
  while( (*d++ = *s++) );
  return retour;
}
```

Pour vous entraîner à comprendre un code, testez-le. Ici, testez l'appel à **foo** en lui passant, comme premier argument, un pointeur vers un tableau de **char** suffisamment grand, et, comme second argument, une chaîne de caractères.
Proposez un **main** pour tester.

La rue de la RAM 13/17

Comment est utilisées la mémoire au sein d'un système d'exploitation ?



Cas d'un OS de type Unix (crédits Monjurul Karim – Source *Code based Buffer Overflow Detection Technology* – 2015)

.bss https://en.wikipedia.org/wiki/.bss
en français https://fr.wikipedia.org/wiki/Segment_BSS

La rue de la RAM 14/17

Expérimenter le placement en mémoire
Faire et tester sur un ou plusieurs OS, l'impression d'adresses :

- D'une variable locale *dynamique*, dans `main`;
- D'une variable locale *statique*, dans `main`;
- D'une variable locale *dynamique*, dans une fonction *externe* et *statique*;
- D'une variable locale *statique*, dans une fonction *externe* et *statique*;
- D'une allocation mémoire;
- D'une variable globale *externe*;
- D'une variable globale *statique*;
- D'une fonction *externe*;
- D'une fonction *statique*;
- D'une fonction de bibliothèque (par exemple `cos`);

Puis classer ces adresses.

La rue de la RAM 15/17

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h> /* pour cos */
static void st_print(char * what, const void * addr);
extern void foo(void);
static void bar(void);
extern int glo_var_e; /* déclaration */
int glo_var_e = 7;    /* définition  */
static int glo_var_s = 42;
int main(void) {
  int dy_var = 7;
  static int st_var = 42;
  const int const_var = 2;
  static const int st_const_var = 6;
  int * alloc = malloc(sizeof *alloc);
  st_print("variable locale dans main", &dy_var);
  st_print("variable locale statique dans main", &st_var);
  st_print("variable const dans main", &const_var);
  st_print("variable const statique dans main", &st_const_var);
  foo();
  bar();
  st_print("allocation avec malloc", alloc);
  st_print("variable globale externe", &glo_var_e);
  st_print("variable globale statique", &glo_var_s);
  st_print("fonction externe", foo);
  st_print("fonction statique", bar);
  st_print("fonction de bibliotheque (cos de la libm)", cos);
  free(alloc);
  return 0;
}
void st_print(char * what, const void * addr) {
  printf("@ d'une %50s\t: %16p (%llu)\n", what, addr, (unsigned long long)addr);
}
void foo(void) {
  int dy_var = 7;
  static int st_var = 42;
  st_print("variable locale dans foo (extern)", &dy_var);
  st_print("variable locale statique dans foo (extern)", &st_var);
}
void bar(void) {
  int dy_var = 7;
  static int st_var = 42;
  st_print("variable locale dans bar (static)", &dy_var);
  st_print("variable locale statique dans bar (static)", &st_var);
}
```

Code source 3.5 – Place en mémoire selon le type/variété (`memory_zones.c`)

La rue de la RAM 16/17

Darwin Kernel Version 20.2.0

```
$ gcc -Wall -Wextra -O0 memory_zones.c -o memory_zones -lm && ./memory_zones
@ d'une                          variable locale dans main        :   0x7ffee121da68 (140732675512936)
@ d'une                 variable locale statique dans main        :      0x10e9ea024 (4540244004)
@ d'une                           variable const dans main        :   0x7ffee121da64 (140732675512932)
@ d'une                  variable const statique dans main        :      0x10e9e5de0 (4540227040)
@ d'une                  variable locale dans foo (extern)        :   0x7ffee121da3c (140732675512892)
@ d'une         variable locale statique dans foo (extern)        :      0x10e9ea02c (4540244012)
@ d'une                  variable locale dans bar (static)        :   0x7ffee121da3c (140732675512892)
@ d'une         variable locale statique dans bar (static)        :      0x10e9ea030 (4540244016)
@ d'une                             allocation avec malloc        :   0x7faf7bc05960 (140391672207712)
@ d'une                           variable globale externe        :      0x10e9ea020 (4540244000)
@ d'une                          variable globale statique        :      0x10e9ea028 (4540244008)
@ d'une                                   fonction externe        :      0x10e9e5d20 (4540226848)
@ d'une                                  fonction statique        :      0x10e9e5d60 (4540226912)
@ d'une          fonction de bibliothèque (cos de la libm)        :   0x7fff2a468338 (140733902652216)
```

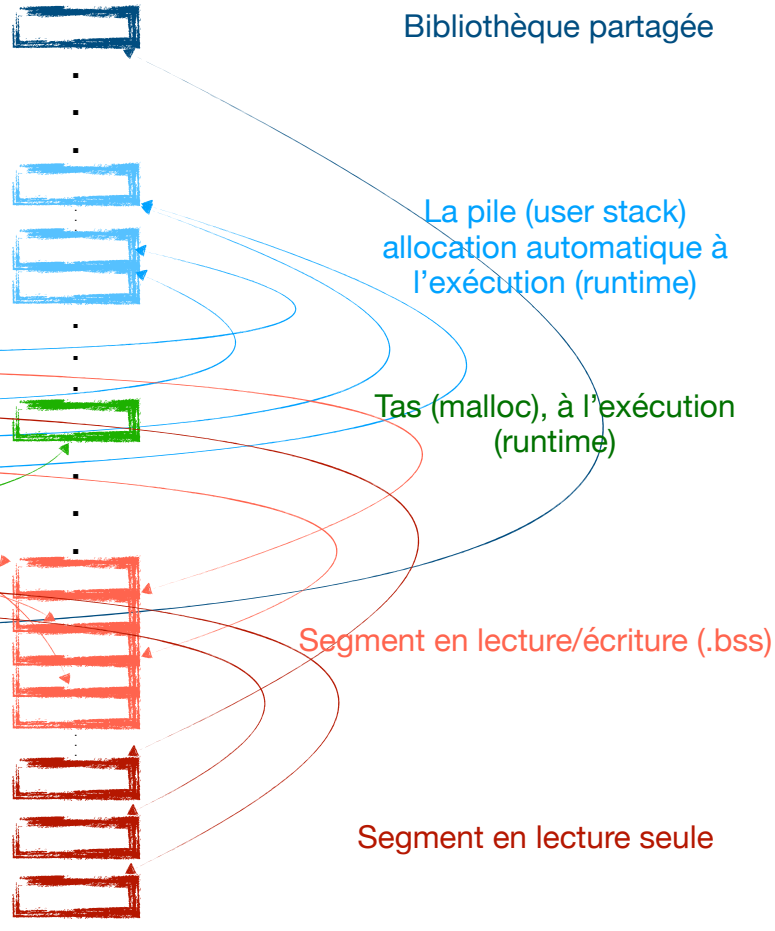
Linux 4.9

```
$ gcc -Wall -Wextra -O0 memory_zones.c -o memory_zones -lm && ./memory_zones
@ d'une                          variable locale dans main        :   0x7ffd7bbde8c8 (140726679496904)
@ d'une                 variable locale statique dans main        :   0x556104851018 (93875176017944)
@ d'une                           variable const dans main        :   0x7ffd7bbde8cc (140726679496908)
@ d'une                  variable const statique dans main        :   0x55610464fc1c (93875173915676)
@ d'une                  variable locale dans foo (extern)        :   0x7ffd7bbde8a4 (140726679496868)
@ d'une         variable locale statique dans foo (extern)        :   0x55610485101c (93875176017948)
@ d'une                  variable locale dans bar (static)        :   0x7ffd7bbde8a4 (140726679496868)
@ d'une         variable locale statique dans bar (static)        :   0x556104851020 (93875176017952)
@ d'une                             allocation avec malloc        :   0x556104852260 (93875176022624)
@ d'une                           variable globale externe        :   0x556104851010 (93875176017936)
@ d'une                          variable globale statique        :   0x556104851014 (93875176017940)
@ d'une                                   fonction externe        :   0x55610464f8f2 (93875173914866)
@ d'une                                  fonction statique        :   0x55610464f94d (93875173914957)
@ d'une          fonction de bibliotheque (cos de la libm)        :   0x7f1fcc2aaa60 (139774546061920)
```

La rue de la RAM 17/17

Visualisation des emplacements mémoire en fonction du type de donnée

```
                variable locale dans main:   0x7ffee121da68 (140732675512936)
      variable locale statique dans main:      0x10e9ea024 (4540244004)
                variable const dans main:   0x7ffee121da64 (140732675512932)
       variable const statique dans main:      0x10e9e5de0 (4540227040)
       variable locale dans foo (extern):   0x7ffee121da3c (140732675512892)
variable locale statique dans foo (extern):      0x10e9ea02c (4540244012)
       variable locale dans bar (static):   0x7ffee121da3c (140732675512892)
variable locale statique dans bar (static):      0x10e9ea030 (4540244016)
                  allocation avec malloc:   0x7faf7bc05960 (140391672207712)
       variable globale externe:              0x10e9ea020 (4540244000)
       variable globale statique:             0x10e9ea028 (4540244008)
                  fonction externe:           0x10e9e5d20 (4540226848)
                  fonction statique:          0x10e9e5d60 (4540226912)
fonction de bibliothèque (cos de la libm):   0x7fff2a468338 (140733902652216)
```



Licence **I**nformatique et **V**idéoludisme
Université Paris 8