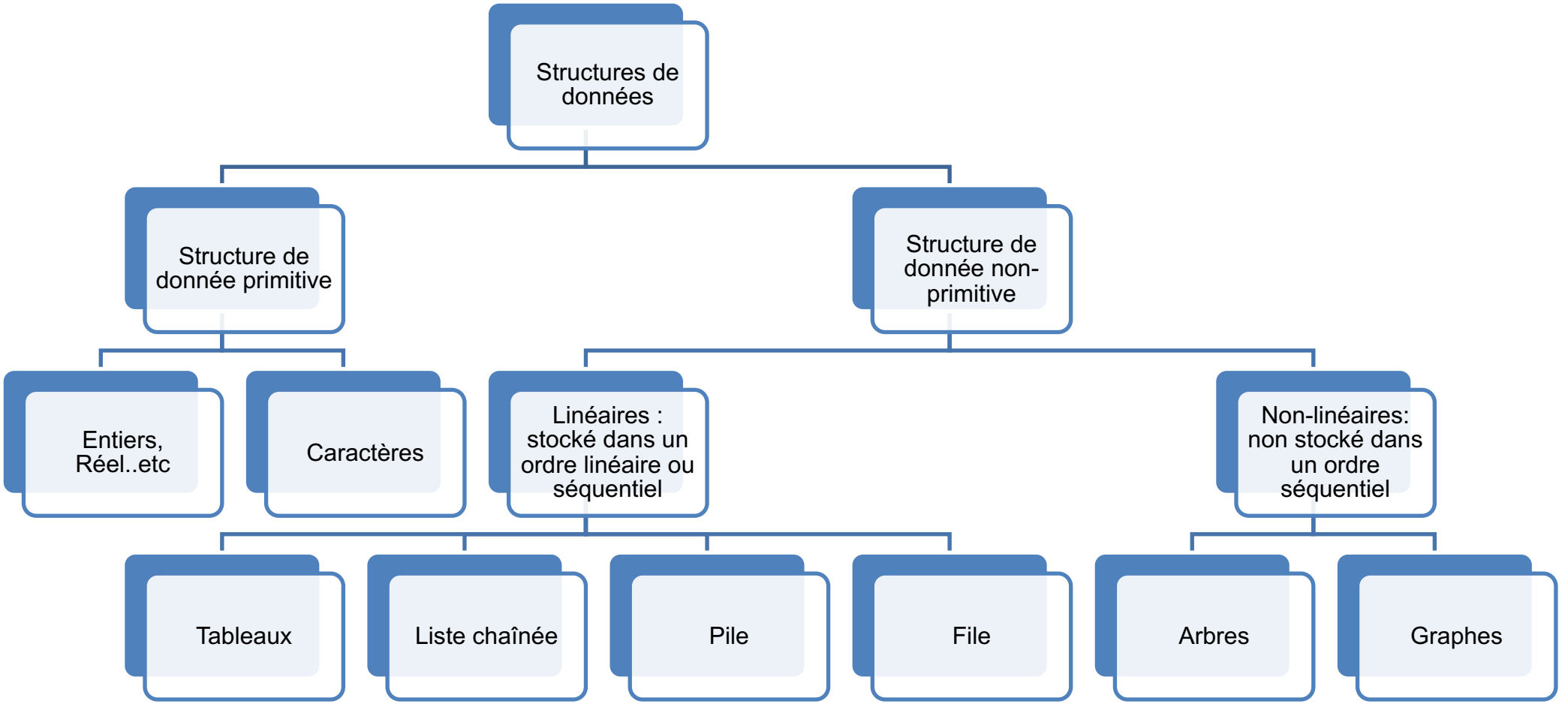
Course descriptions : semester 2, Data Science and Big Data..

Programmation & Algorithmique II

CM 3 : Gestion de la mémoire

CLASSIFICATION DES STRUCTURES DE DONNÉES

- **Structures de données primitives et non primitives**



PLAN

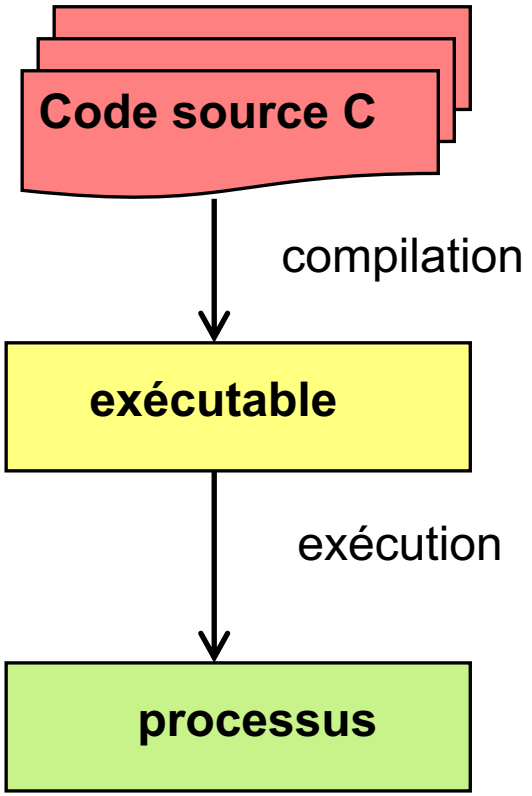
- Disposition de la mémoire
- Allocation et désallocation de mémoire
- Exemples
- Fuites de mémoire

PLAN

- **Disposition de la mémoire**
- Allocation et désallocation de mémoire
- Exemples
- Fuites de mémoire

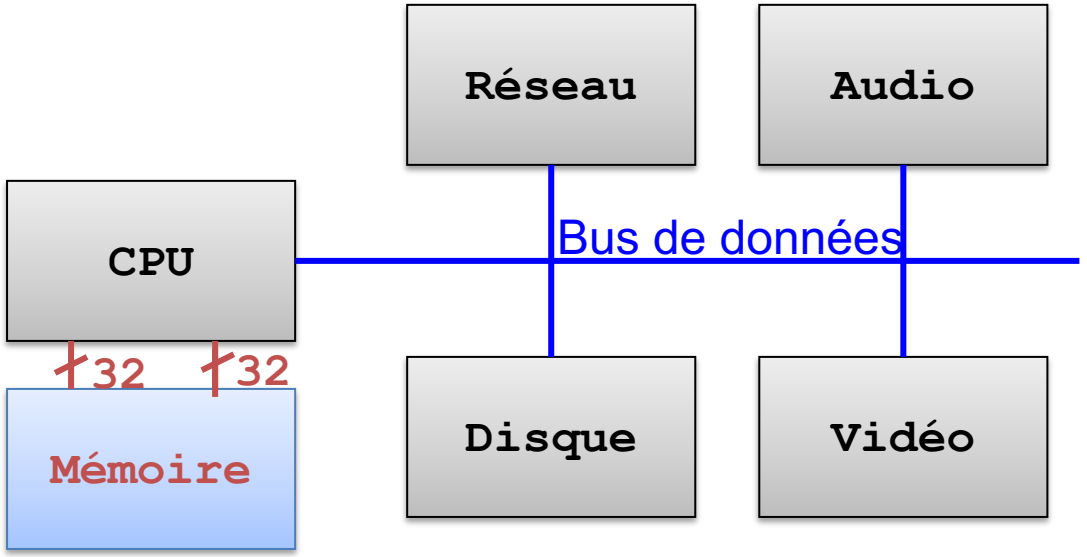
DISPOSITION DE LA MÉMOIRE

- **Code source C**
  - Instructions C organisées en fonctions
  - Stocké sous forme de collection de fichiers (.c et .h)
- **Module exécutable**
  - Image binaire générée par le compilateur
  - Stocké sous forme de fichier (par exemple, a.out)
- **Processus**
  - Instance d'un programme en cours d'exécution
    - Avec son propre espace d'adressage en mémoire
    - Avec son propre identifiant et son état d'exécution
  - Géré par le système d'exploitation



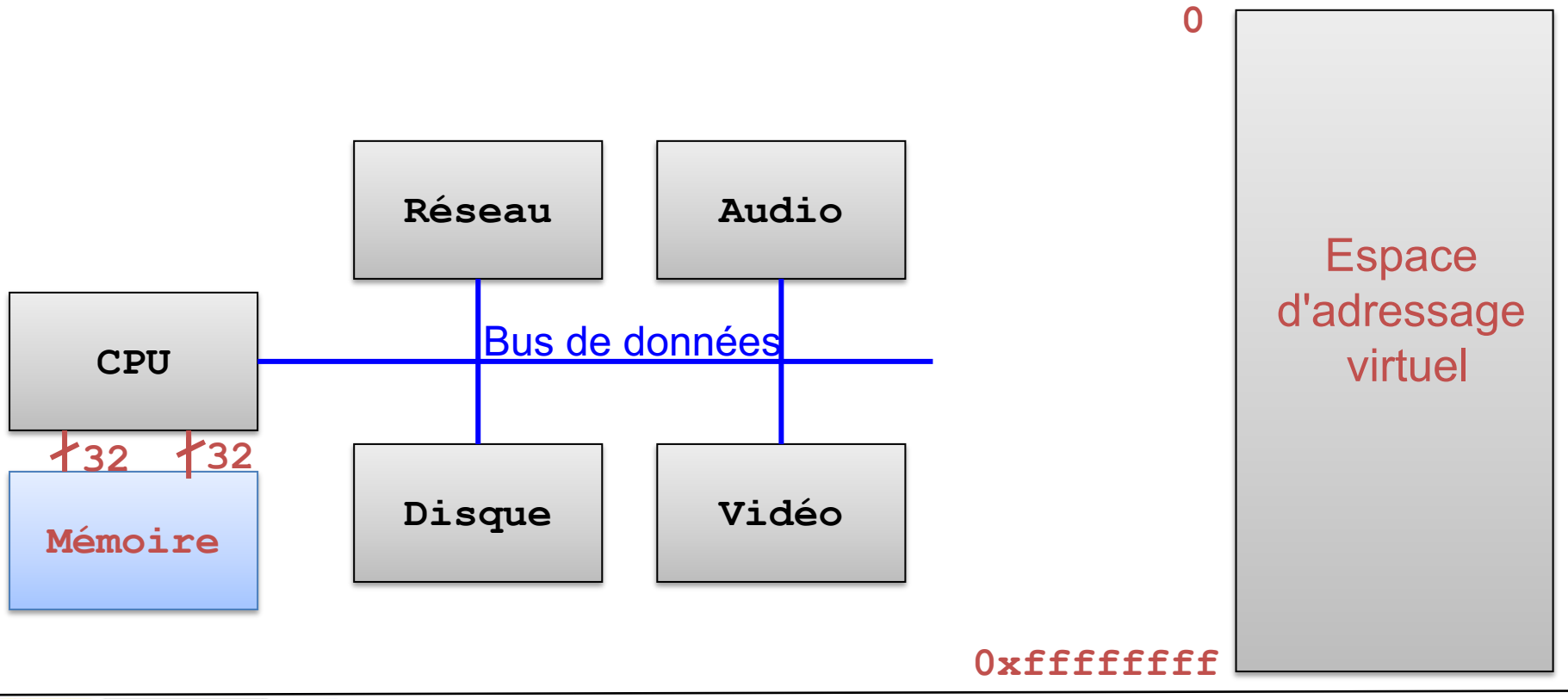
DISPOSITION DE LA MÉMOIRE

- **Qu'est-ce que la mémoire principale?**
  - Stockage des variables, des données, du code, etc.
  - Peut être partagé entre de nombreux processus



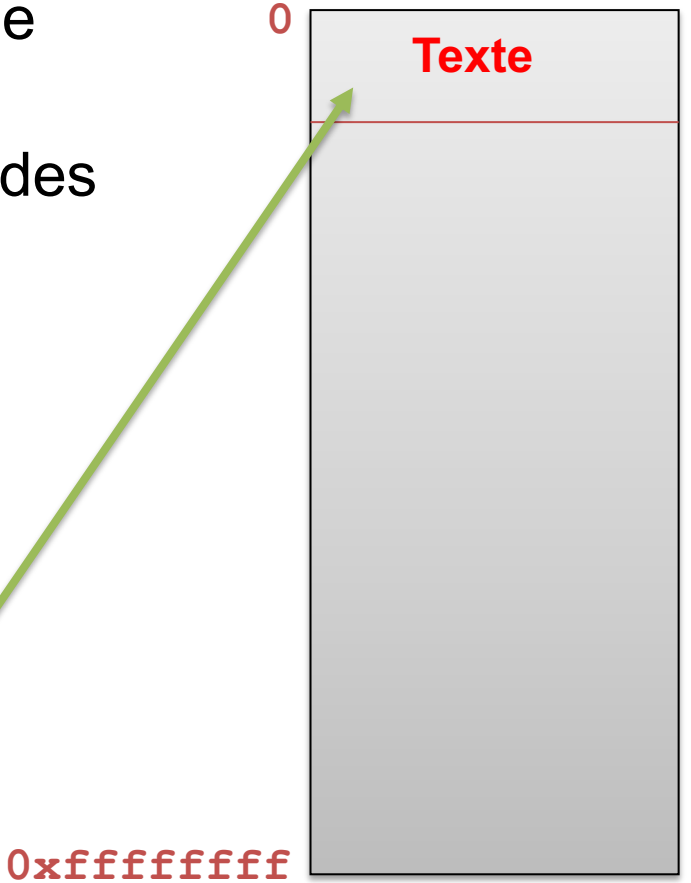
DISPOSITION DE LA MÉMOIRE

- **Qu'est-ce que la mémoire principale?**
  - Espace mémoire adressable contigu pour un seul processus
  - Peut être échangé dans la mémoire physique à partir du disque dans les pages
  - Supposons que chaque processus possède sa propre mémoire contiguë



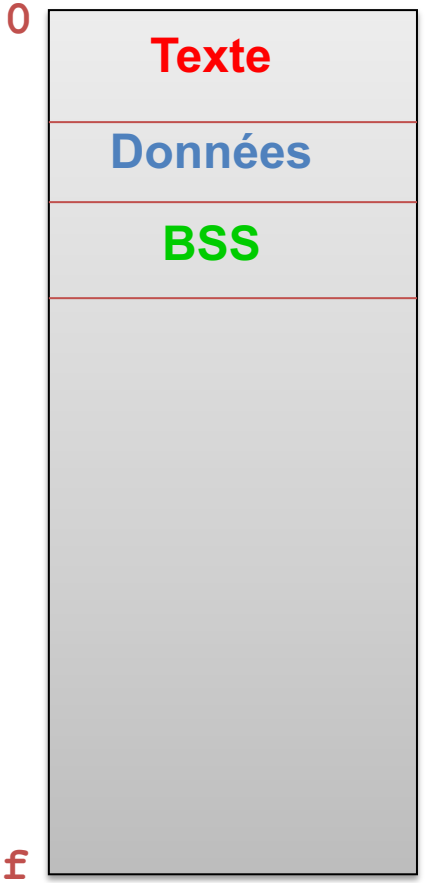
DISPOSITION DE LA MÉMOIRE

- **Quoi stocker: code et constantes**
  - Code exécutable et données constantes
    - Programme binaire et toutes les bibliothèques partagées qu'il charge
    - Nécessaire pour le système d'exploitation pour lire les commandes
  - Le système d'exploitation sait tout à l'avance
    - Il sait la quantité d'espace nécessaire
    - Il sait le contenu de la mémoire
  - Connu par le nom de segment de « Texte »



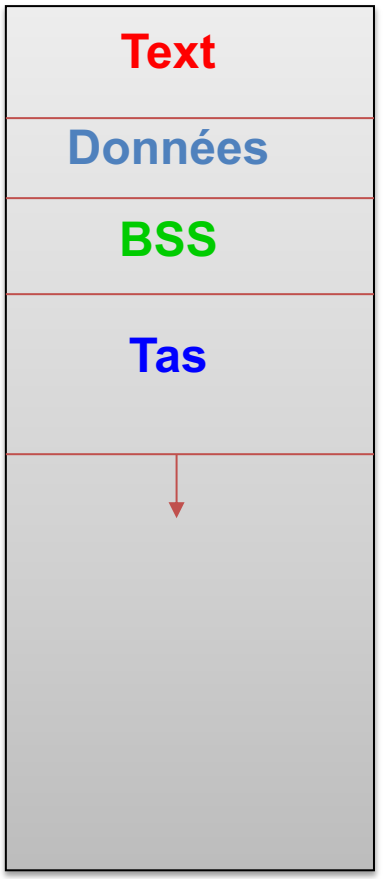
DISPOSITION DE LA MÉMOIRE

- **Quoi stocker: données «statiques»**
  - Variables qui existent pour l'ensemble du programme
    - Variables globales et variables locales «statiques»
    - La quantité d'espace nécessaire est connue à l'avance
  - **Données**: initialisées dans le code
    - Valeur initiale spécifiée par le programmeur
      - Par exemple, « int x = 97; »
    - La mémoire est initialisée avec cette valeur
  - **BSS**: non initialisé dans le code
    - BSS signifie en anglais «Block Started by Symbol» bloc démarré par un symbole.
    - Valeur initiale non spécifiée
      - Par exemple, « int x; »
    - Toute la mémoire est initialisée à 0 (sur la plupart des systèmes d'exploitation)



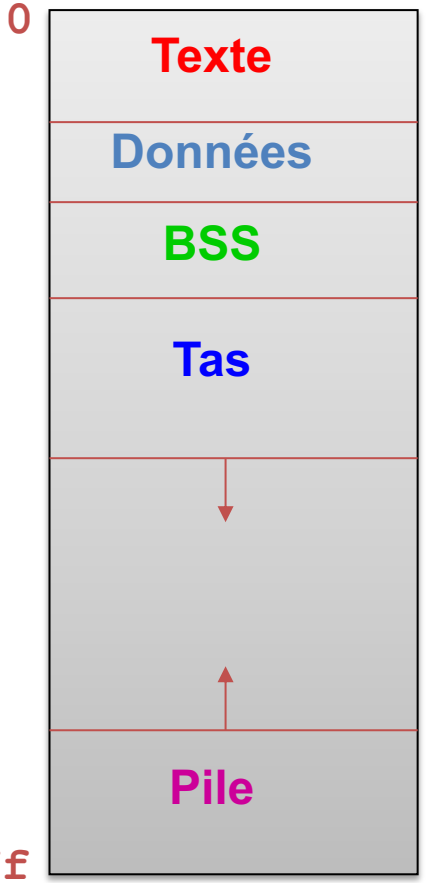
DISPOSITION DE LA MÉMOIRE

- **Quoi stocker: mémoire dynamique**
  - Mémoire allouée pendant l'exécution du programme
    - Par exemple, allouer à l'aide de la fonction malloc ()
    - Et désallouer à l'aide de la fonction free ()
  - Le système d'exploitation ne sait rien à l'avance
    - Ne sait pas la quantité d'espace
    - Ne sait pas le contenu
  - Alors, il faut laisser de la place pour grandir
    - Connu par le nom de «**Tas**» (en anglais '**Heap**')
    - Exemple détaillé en quelques diapositives



DISPOSITION DE LA MÉMOIRE

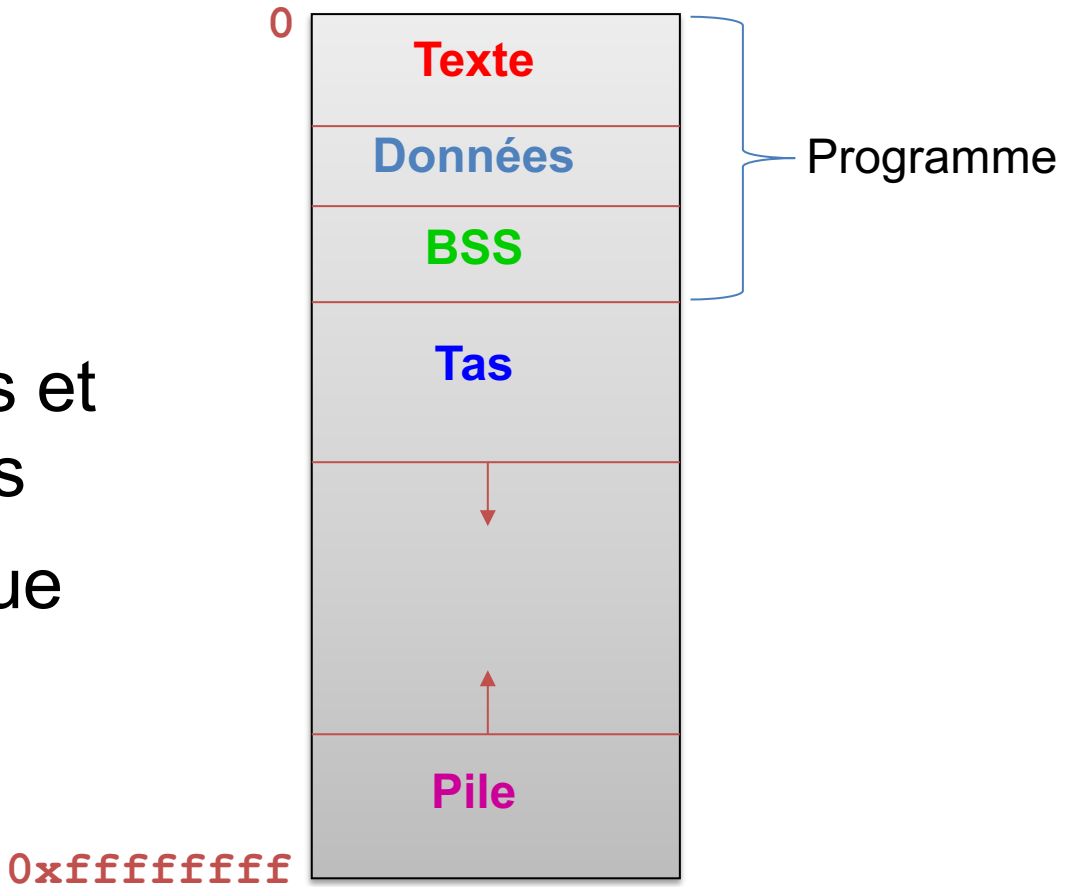
- **Quoi stocker: variables temporaires**
  - Mémoire temporaire pendant la durée de vie d'une fonction ou d'un bloc
    - Stockage des paramètres de fonction et des variables locales
  - Besoin de prendre en charge les appels de fonction imbriqués
    - Une fonction en appelle une autre, et ainsi de suite
    - Stocker les variables de la fonction d'appel
    - Savoir où retourner une fois terminé
  - Donc, il faut laisser de la place pour grandir
    - Connu par le nom de «**Pile**»
    - Empiler dans la pile lorsque la nouvelle fonction est appelée
    - Dépiler de la pile une fois l'execution de la fonction fini



C' est différent de la structure de donnée type pile parce que c'est un espace mémoire géré par le OS mais le mécanisme de fonctionnement est le même

DISPOSITION DE LA MÉMOIRE

- **Résumé**
  - **Texte**: code, données constantes
  - **Données**: variables globales et statiques initialisées
  - **BSS**: variables globales et statiques non initialisées
  - **Tas**: mémoire dynamique
  - **Pile**: variables locales

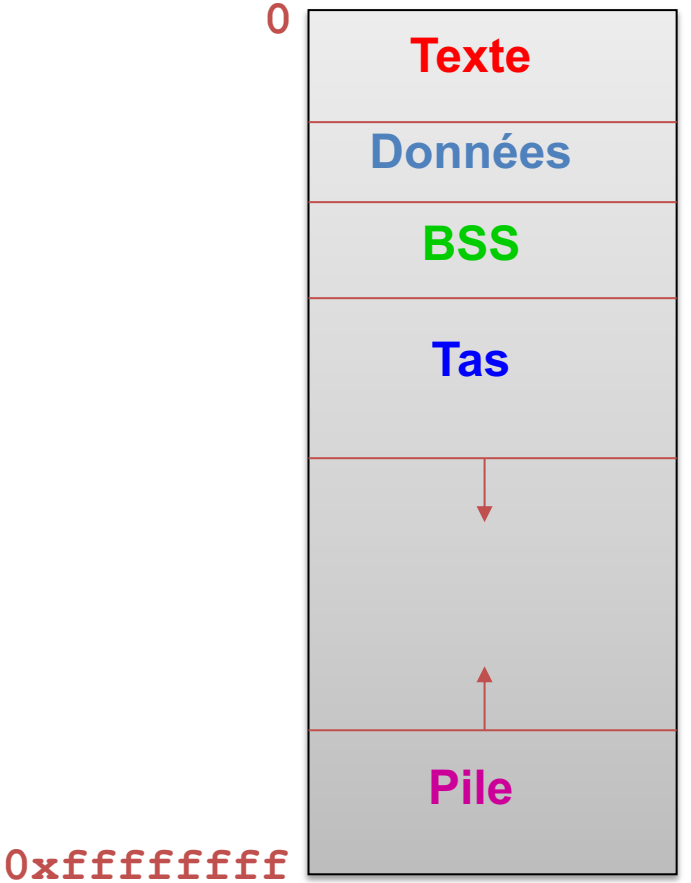


DISPOSITION DE LA MÉMOIRE

- **Exemple**

```
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

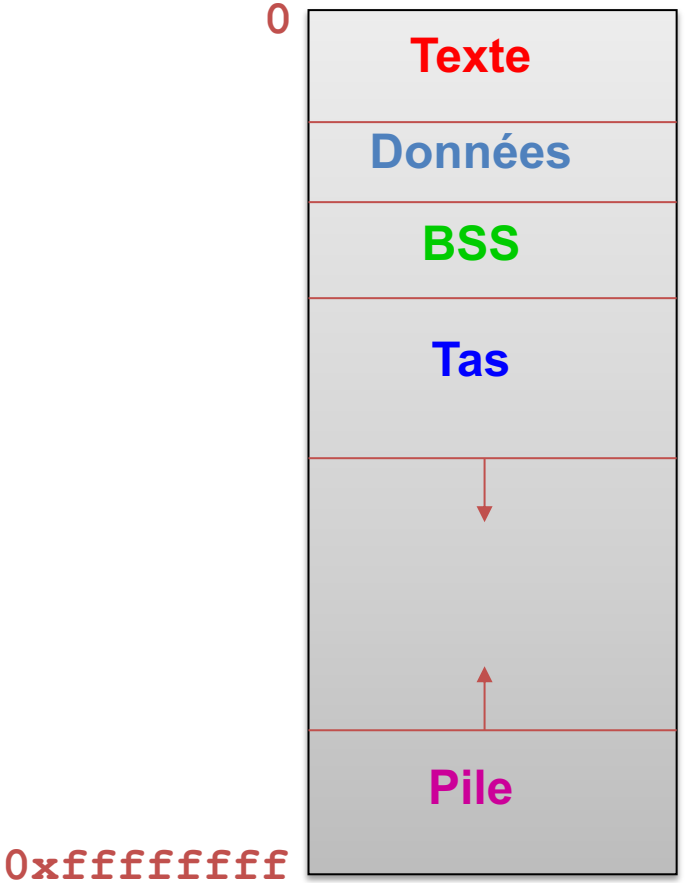


DISPOSITION DE LA MÉMOIRE

- **Exemple: Texte**

```
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

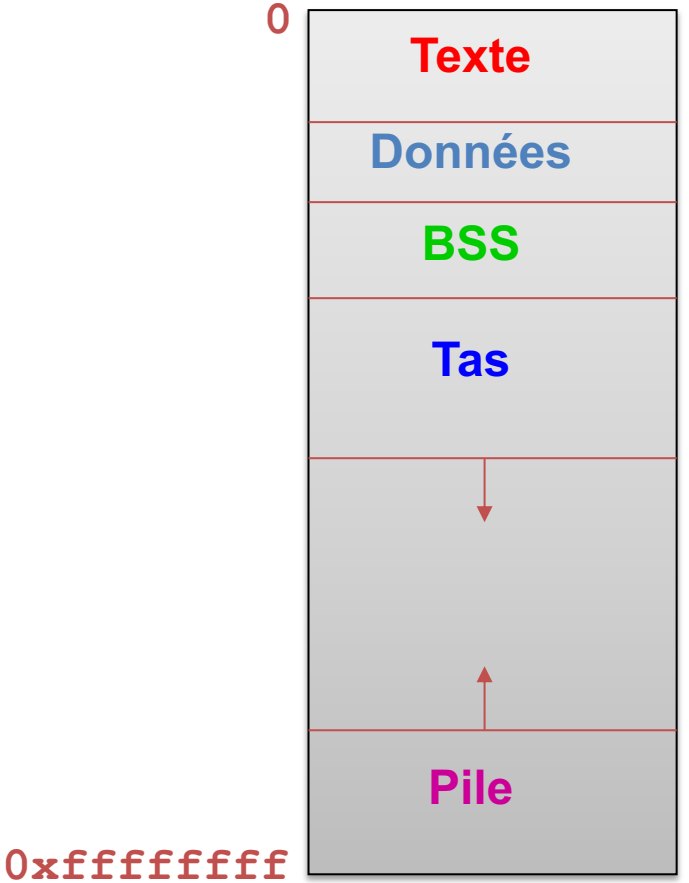


DISPOSITION DE LA MÉMOIRE

- **Exemple : Données**

```
char* string = "hello";
int iSize;

char* f(void)
{
        char* p;
        iSize = 8;
        p = malloc(iSize);
        return p;
}
```

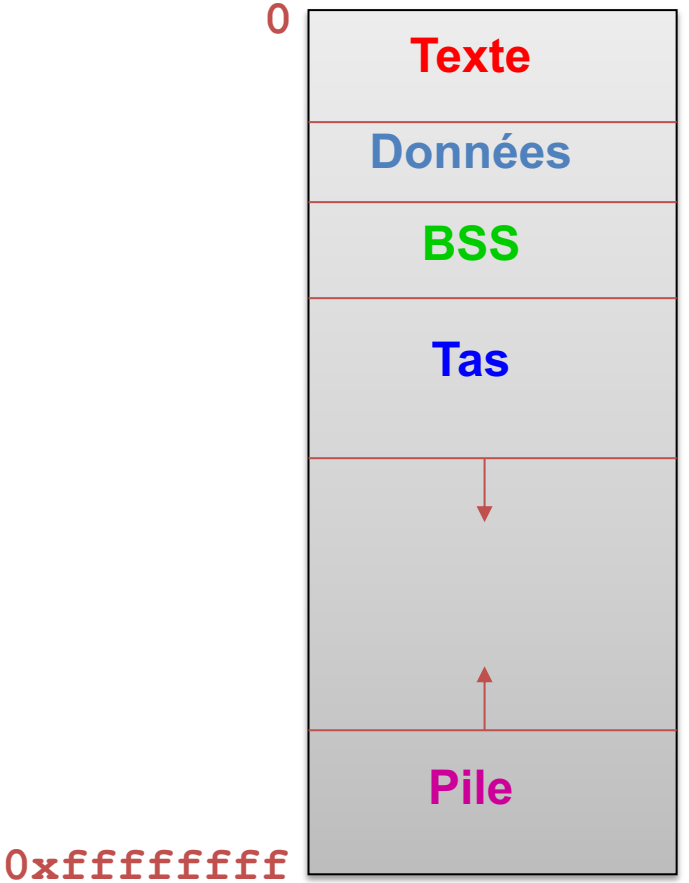


DISPOSITION DE LA MÉMOIRE

- **Exemple: BSS**

```
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

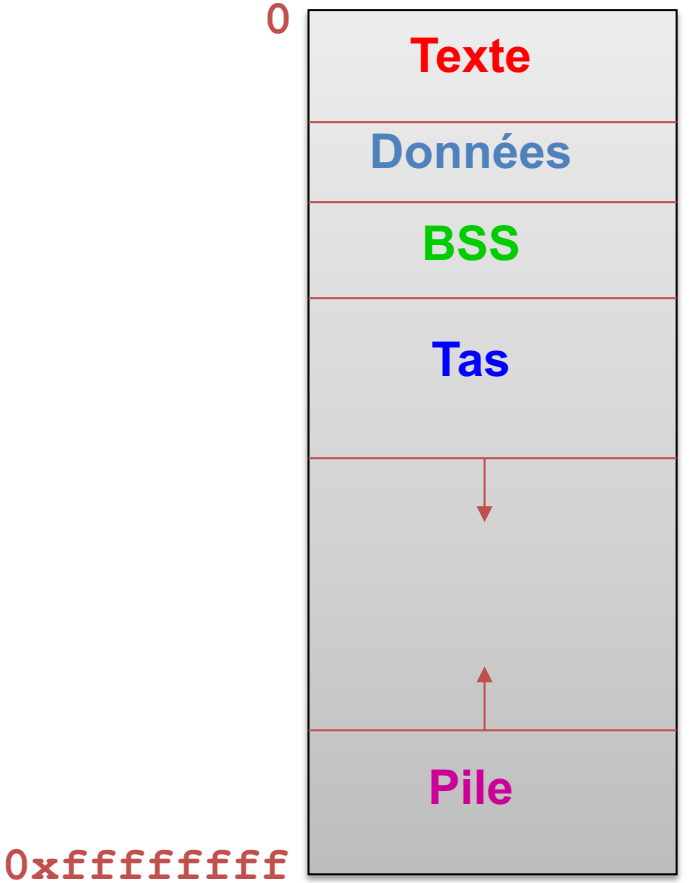


DISPOSITION DE LA MÉMOIRE

- **Exemple: Tas**

```
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

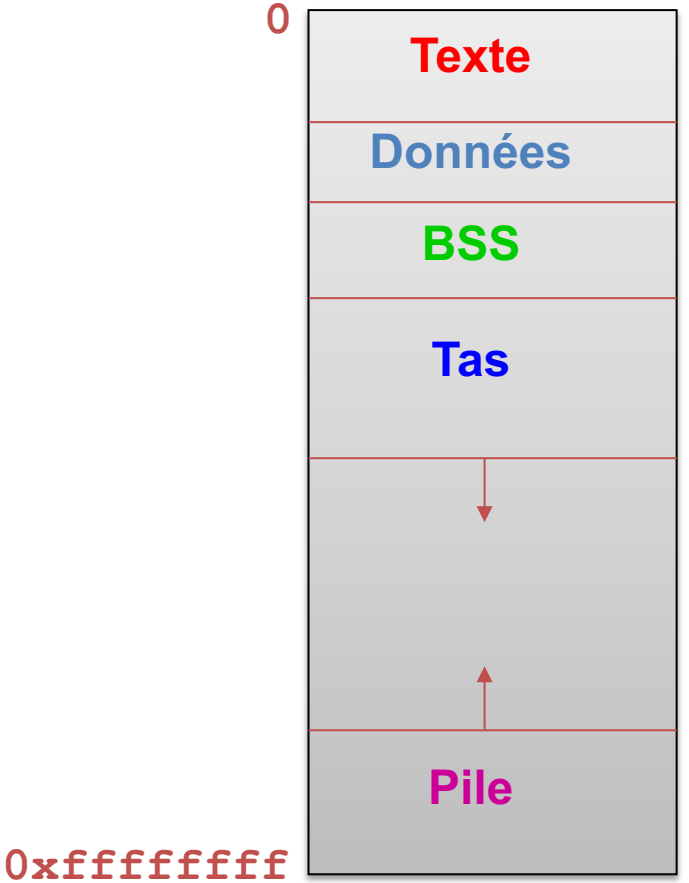


DISPOSITION DE LA MÉMOIRE

- **Exemple: Pile**

```
char* string = "hello";
int iSize;

char* f(void)
{
        char* p;
        iSize = 8;
        p = malloc(iSize);
        return p;
}
```

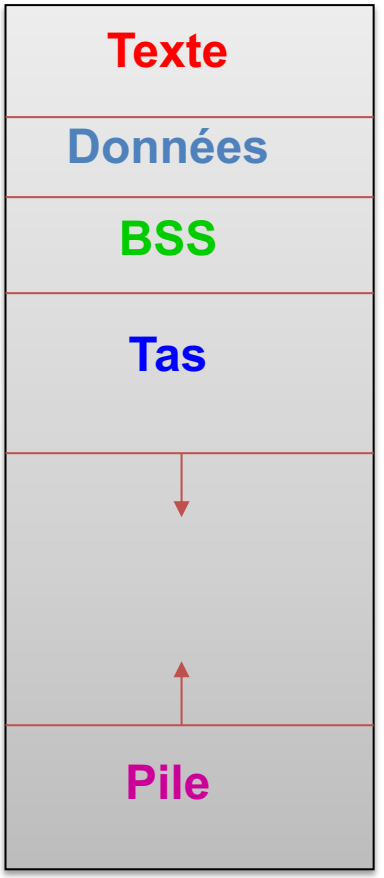


PLAN

- Disposition de la mémoire
- **Allocation et désallocation de mémoire**
- Exemples
- Fuites de mémoire

ALLOCATION ET DÉSALLOCATION DE MÉMOIRE

- **Comment et quand la mémoire est-elle allouée?**
  - Variables globales et statiques: démarrage du programme
  - Variables locales: appel de fonction
  - Mémoire dynamique: `malloc()`
- **Comment la mémoire est-elle désallouée?**
  - Variables globales et statiques: fin du programme
  - Variables locales: retour de fonction
  - Mémoire dynamique: `free()`
- **Toute la mémoire est libérée à la fin du programme**
  - Veut mieux libérer de la mémoire allouée de toute façon.



ALLOCATION ET DÉSALLOCATION DE MÉMOIRE

```
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

Données: "hello" au démarrage

BSS: 0 au démarrage

Pile: à l'appel de fonction

Tas: 8 octets à malloc

ALLOCATION ET DÉSALLOCATION DE MÉMOIRE

```
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

Disponible jusqu'à la résiliation

Disponible jusqu'à la résiliation/termination

Désallouer au retour de f

Désallouer à l'aide de free()

ALLOCATION ET DÉSALLOCATION DE MÉMOIRE

- **Généralités**
  - Tout processus, c'est-a-dire tout fichier exécutable (a.out en C) doit demander au système d'exploitation de lui réserver de la place pour stocker les différents objets du programme: code, variables, tableaux, etc.

- **Cette place peut être réservée**
  - Avant l'exécution (***Allocation statique***) ou
  - Pendant l'exécution (***Allocation dynamique***)

- **Il est important de comprendre que :**
  - L'ensemble de la mémoire réservée à un processus est contiguë
  - Cet espace est divisé en fonction de la nature des objets stockés
  - Les données du programme sont stockées à des endroits différents (programme, pile, tas).

ALLOCATION ET DÉSALLOCATION DE MÉMOIRE

- **Stratégies d'allocation de mémoire**
  - Dans le langage C,
    - On a le choix parmi les différentes méthodes d'allocation.
    - L'allocation dynamique impose la mise en place de bonnes pratiques afin d'éviter la mauvaise manipulation mémoire.
  - Dans des langage plus évolués comme Java,
    - L'allocation dynamique est transparente pour le programmeur et sa gestion est assurée par un « ramasse-miettes » (Garbage collector en anglais) qui garantit une allocation et une libération de mémoire optimales.

- **Allocation statistique**
  - Permet d'assurer que l' espace de mémoire utilisé est connu avant le lancement du programme.
  - Par exemple
    - `int i, j`; // Deux octets par (total 2) variables entières, deux octets pour le variables «i» et «j» seront alloués .
    - `float a[5], f`; // Quatre octets par (au total 6) variables réels. 20 octets au tableau f [5 éléments de réel, c'est-à-dire 5 × 4] et quatre octets pour la variable «f» seront alloués.

Occupation de la mémoire de type de données primitif

Type	Taille de stockage	Plage de valeurs
char	1 octet	-128 à 127 ou 0 à 255
unsigned char	1 octet	0 à 255
signed char	1 octet	-128 à 127
int	2 ou 4 octets	-32,768 à 32,767 ou -2,147,483,648 à 2,147,483,647
unsigned int	2 ou 4 octets	0 à 65,535 ou 0 à 4,294,967,295
short	2 octets	-32,768 à 32,767
unsigned short	2 octets	0 à 65,535
long	8 octets or (4octets pour 32 bit OS)	-92233720368547758O8 à 9223372036854775807
unsigned long	8 octets	0 to 18446744073709551615
float	4 octets	1.2E-38 to 3.4E+38
long double	10 octets	3.4E-4932 to 1.1E+4932

ALLOCATION ET DÉSALLOCATION DE MÉMOIRE

- **Les variables locales ont des valeurs non définies**

```
int count;
```

- **La mémoire allouée par `malloc()` a des valeurs indéfinies**

```
char* p = (char *) malloc(8);
```

- **Doit s'initialiser explicitement si vous voulez une valeur initiale particulière**

```
int count = 0;
p[0] = '\0';
```

- **Les variables globales et statiques sont initialisées à 0 par défaut**

```
static int count = 0;
```

est le même que

```
static int count;
```

C'est un mauvais style de dépendre de ça

ALLOCATION ET DÉSALLOCATION DE MÉMOIRE

- **Les types**
  - `void*:` pointeur générique vers n'importe quel type (peut être converti en d'autres types)
  - `size_t:` type entier non signé renvoyé par `sizeof ()`
- `void* malloc(size_t size)`
  - Renvoie un pointeur vers l'espace de taille **`size`**
  - … Ou **`NULL`** si la demande ne peut être satisfaite
    - Par exemple, `int* x = (int*) malloc(sizeof(int));`
- `void* calloc(size_t nobj, size_t size)`
  - Renvoie un pointeur vers l'espace pour un tableau d'objets `nobj` de taille `size`
  - … Ou `NULL` si la demande ne peut être satisfaite
  - Les octets sont initialisés à 0
- `void free(void* p)`
  - Désallouer l'espace pointé par le pointeur `p`
  - Le pointeur `p` doit être un pointeur vers l'espace précédemment alloué
  - Ne rien faire si `p` est `NULL`

`free(NULL)` : Ne fait rien et est donc valide.

`free()` : Appliqué à une zone non allouée (ou déjà libérée) crée un comportement instable pour le programme.

ALLOCATION ET DÉSALLOCATION DE MÉMOIRE

- `void * realloc (void * ptr, size_t size)`
  - «Augmente» le tampon (espace mémoire) alloué
  - Déplace / copie les données si l'ancien espace est insuffisant
  - … Ou `NULL` si la demande ne peut être satisfaite
- `void* alloca(size_t size)`
  - Non garanti d'exister (pas dans aucune norme officielle)
  - Alloue de l'espace sur l'espace de la pile locale
  - Espace libéré automatiquement lorsque la fonction fini
  - Particulièrement utile pour:

L'utilisation d'alloca gaspille très peu d'espace et est très rapide.

```
int calc(int numItems) {
        int items[numItems];
        int *items = alloca(numItems * sizeof(int));
}
```

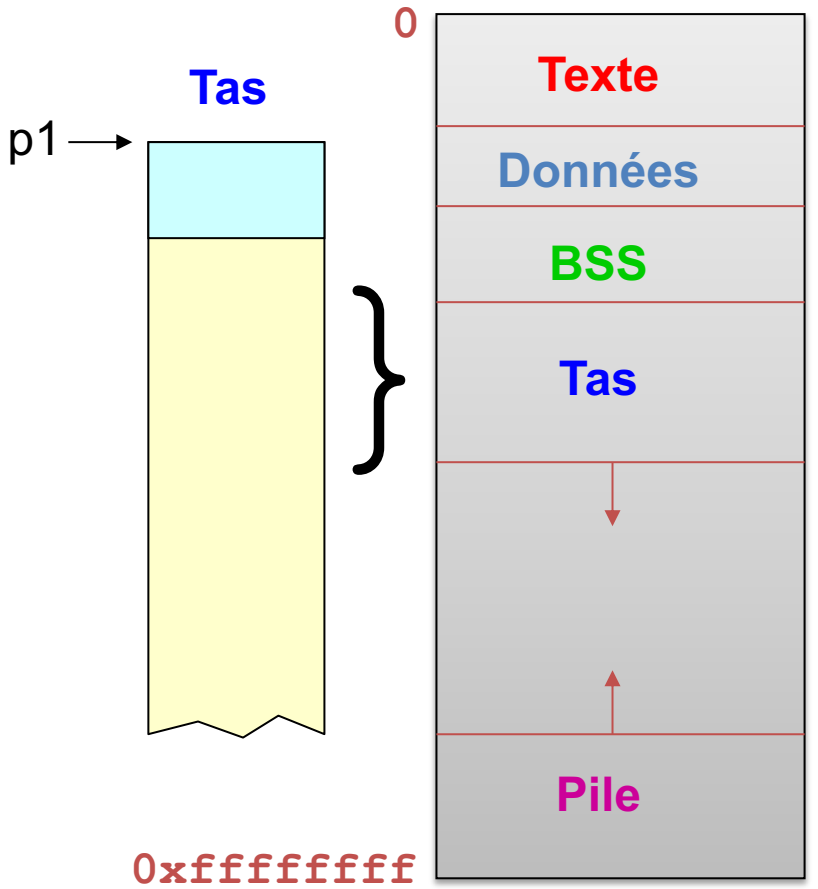
PLAN

- Disposition de la mémoire
- Allocation et désallocation de mémoire
- **Exemples**
- Fuites de mémoire

EXEMPLES

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

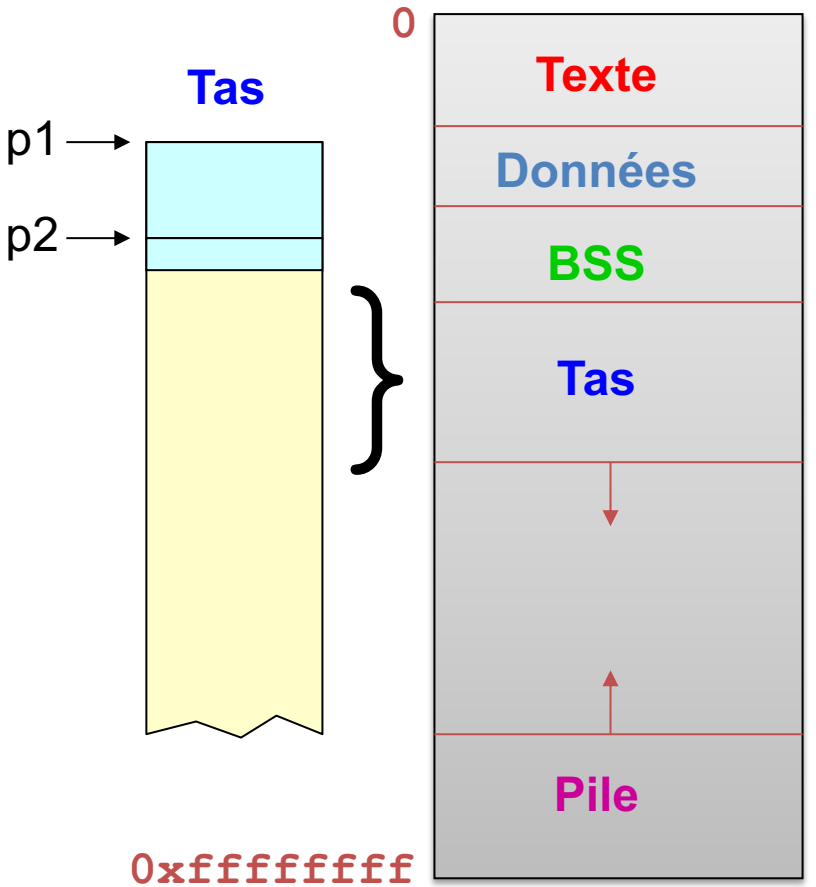
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



EXEMPLES

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

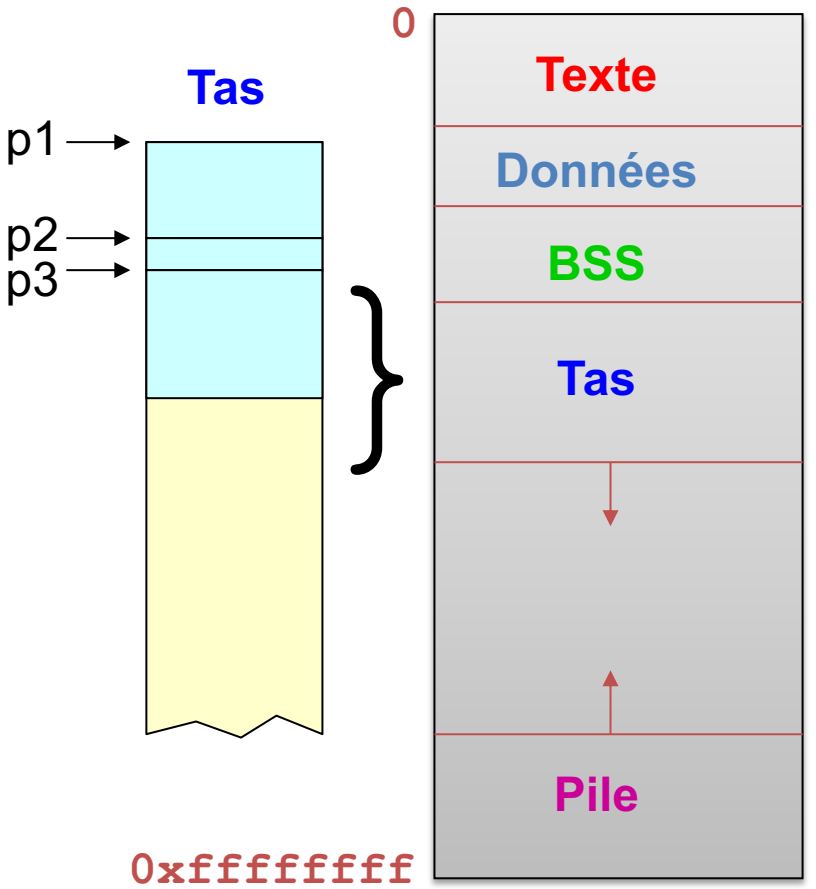
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



EXEMPLES

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

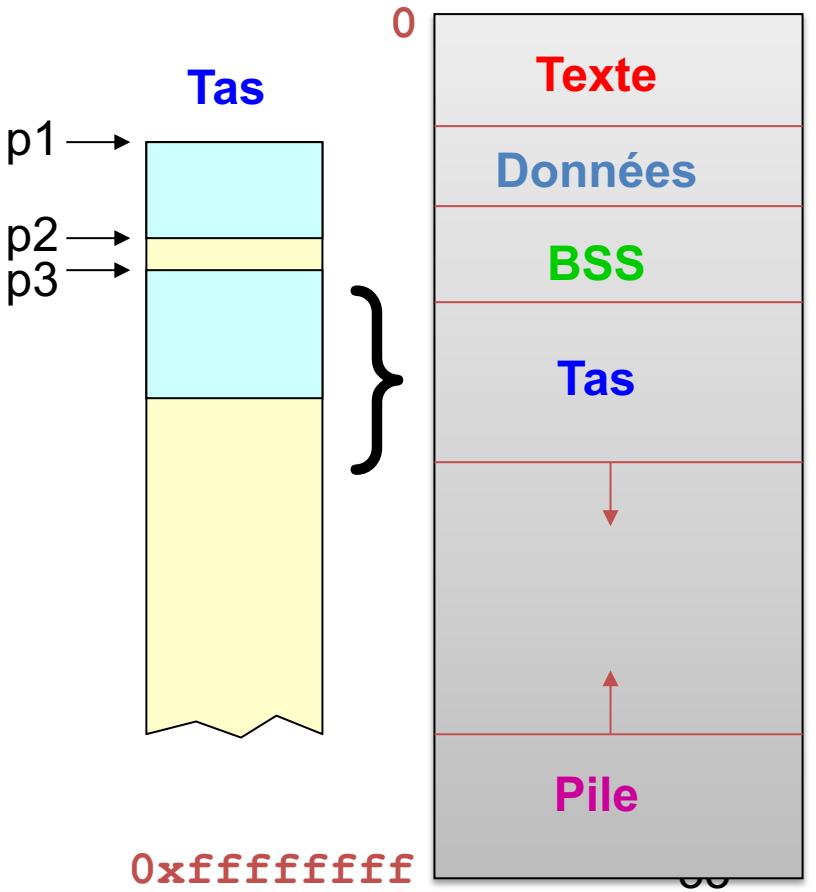
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



EXEMPLES

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

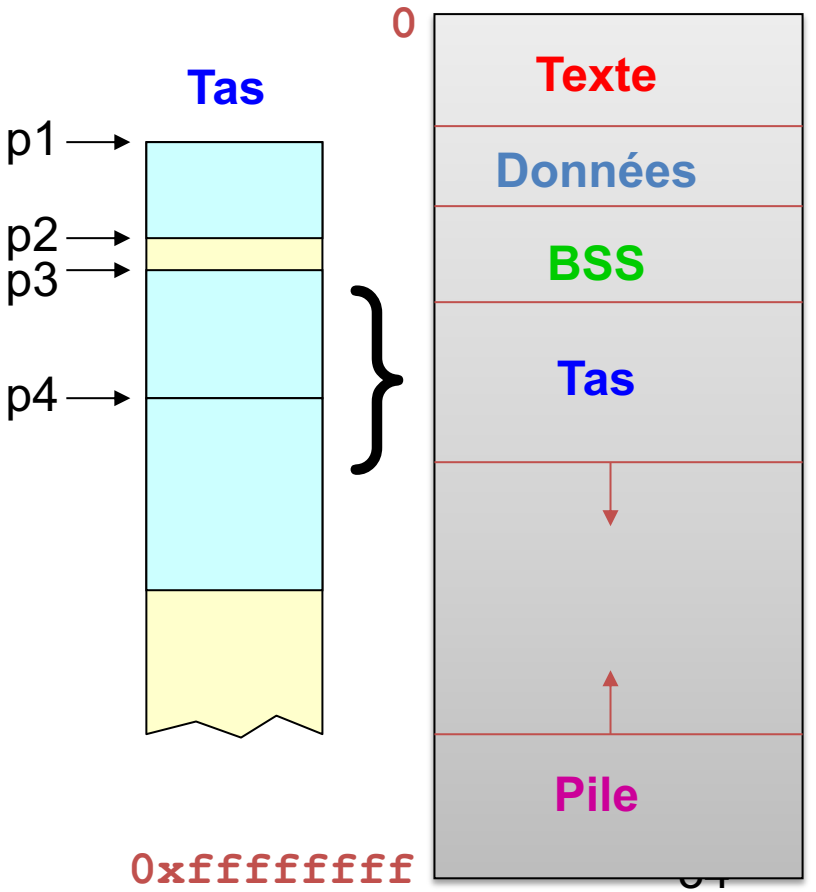
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



EXEMPLES

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

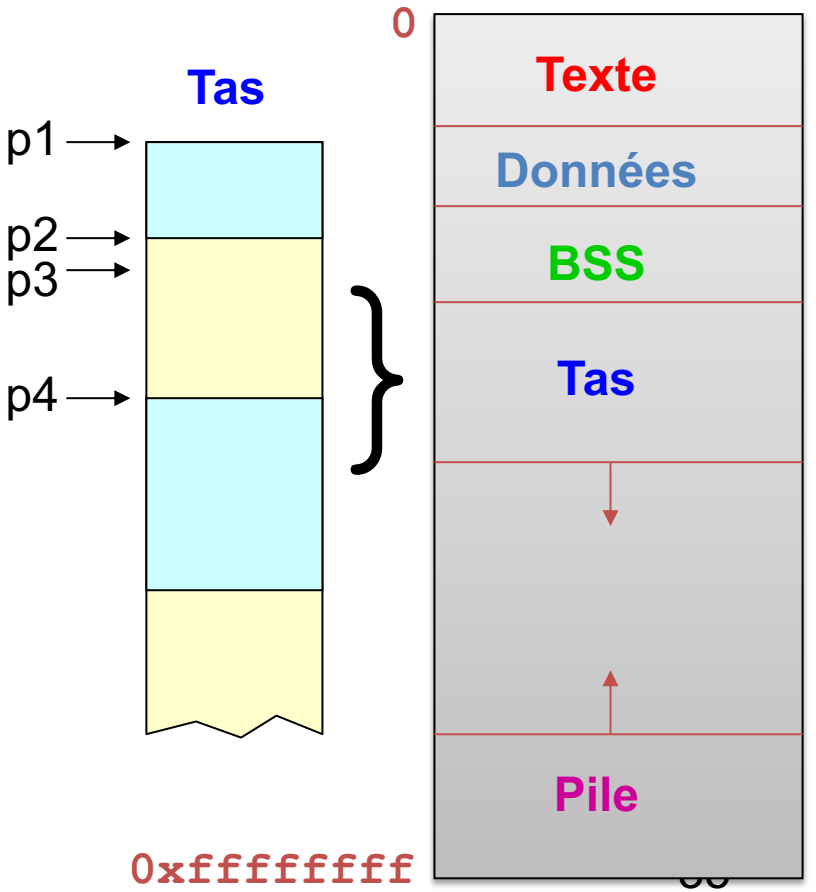
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



EXEMPLES

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

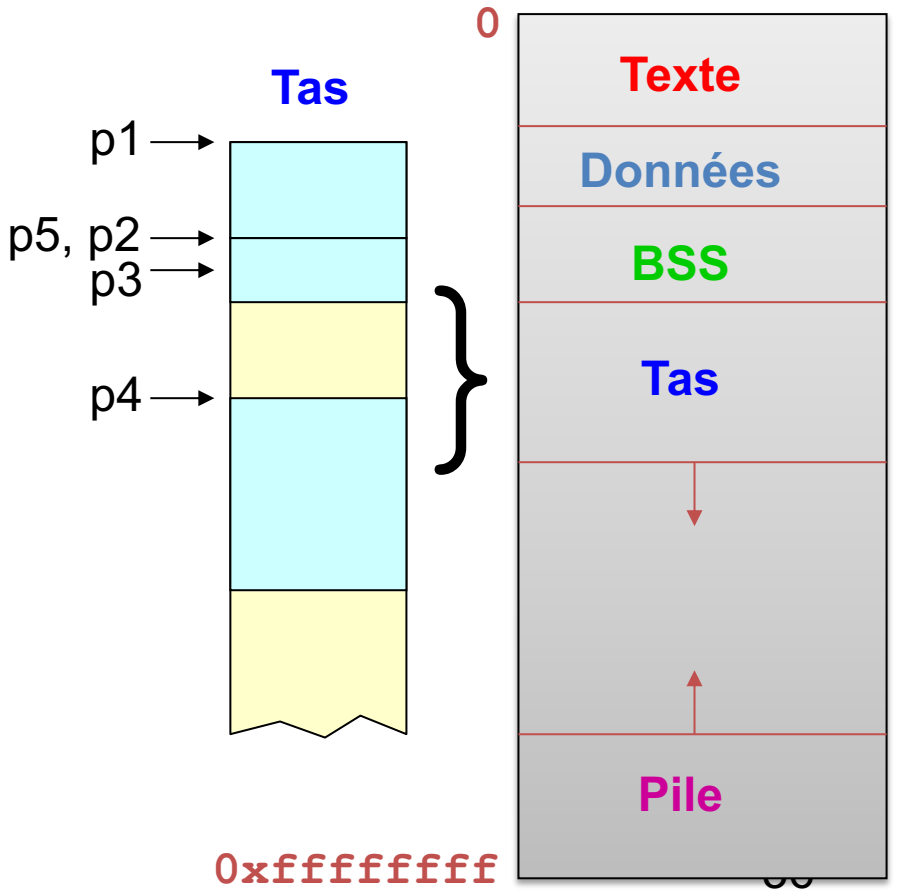
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



EXEMPLES

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

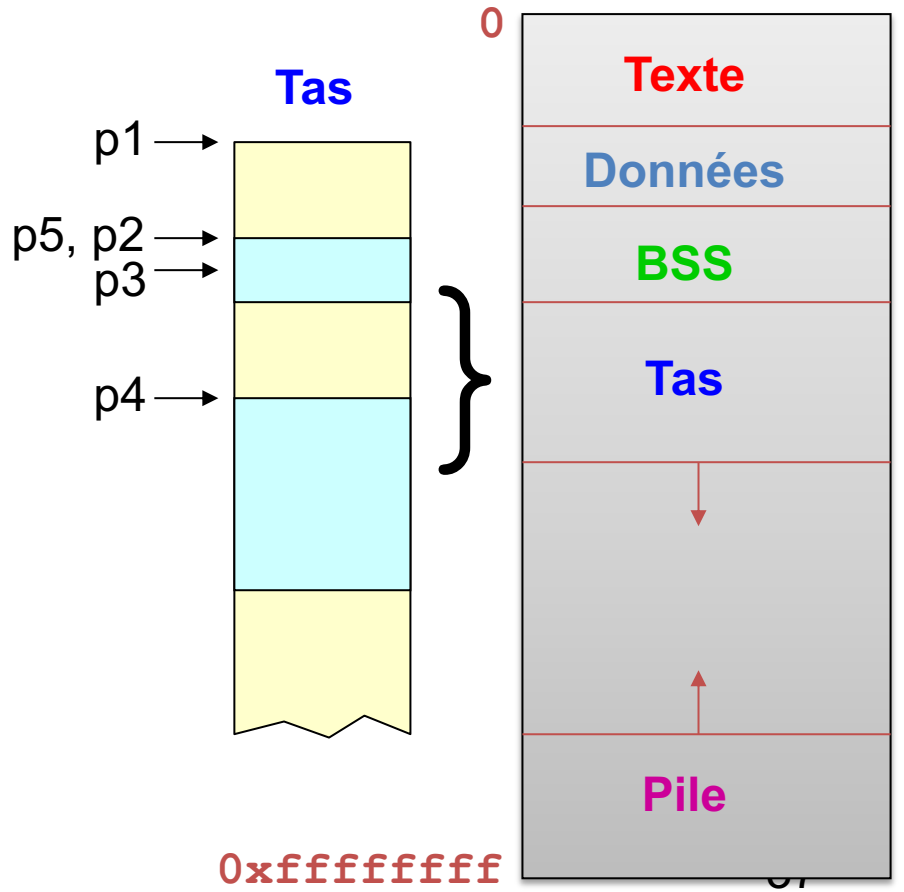
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



EXEMPLES

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

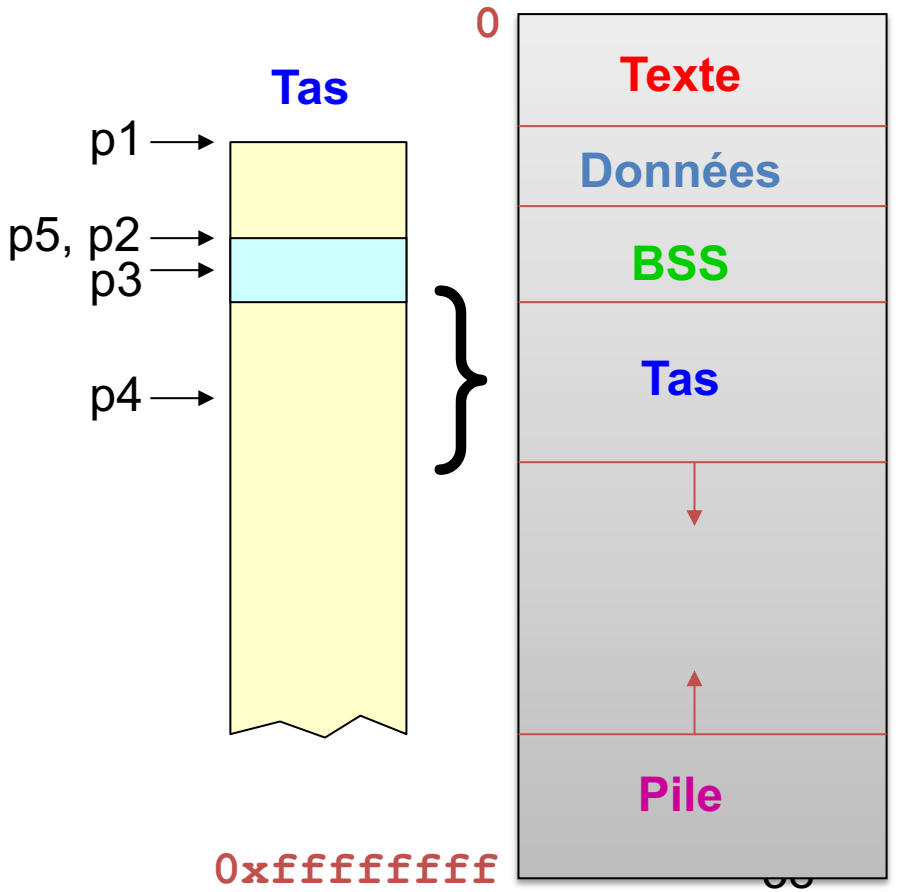
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



EXEMPLES

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

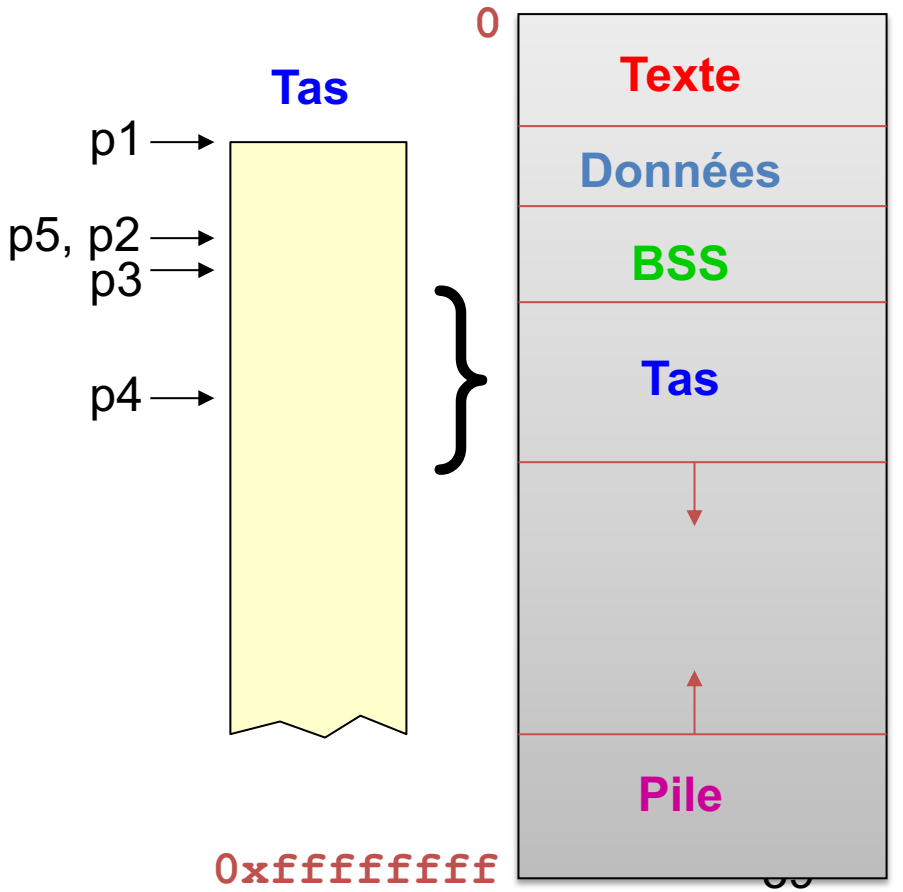
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



EXEMPLES

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



EXEMPLES

- **Exemples de tâches de programmation**
  - Lire du texte composé de plusieurs lignes (se terminant par «\n»)
  - Imprimer le texte dans le sens inverse dès le dernier caractère de la dernier ligne jusqu'au premier caractère de la première ligne.
- **Supposons que vous ne connaissiez pas à l'avance la taille de ligne maximale**
  - Pourrait choisir une taille de ligne maximale à l'avance
  - Par exemple, #define MAXCHAR 200
  - Mais que se passe-t-il si vous avez besoin de lire et de stocker une ligne plus longue?
- **Et vous ne connaissez pas le nombre de lignes à l'avance**
  - Pourrait choisir un nombre maximum de lignes à l'avance
  - Par exemple, #define MAXLINE 10000
  - Mais que se passe-t-il si l'entrée a plus de lignes que cela?
- **Mieux vaut (ré) allouer plus de mémoire au fur et à mesure**

EXEMPLES

- **Impression de caractères dans l'ordre inverse : en utilisant l'allocation statique**

```
#define MAXCHAR 1000

int main(void) {
  char a[MAXCHAR];
  int i, c;
  for (i=0; i<MAXCHAR && ((c=getchar()) != '#'); i++)
     a[i] = c;

  while (i > 0)
     putchar(a[--i]);
}
```

```
Console  Shell
> clang-7 -pthread -lm -o main main.c
> ./main
Bonjour
Comment allez-vous?
Au revoir
#

riover uA
?suov-zella tnemmoC
ruojnoB>
```

a

c=	B	o	n	j	o	u	r	\n	C	o	m	…	\n	…
i =	0	1	2	3	4	5	6	7	8	9	10	…	38	..1000

EXEMPLES

- **Impression de caractères dans l'ordre inverse : en utilisant l'allocation dynamique (malloc(),realloc(),free())**

```
#define INIT_SIZE 5

int main(void) {
  char* a;
  int i, c, size=INIT_SIZE;

  a = (char *) malloc(size * sizeof(char));
  for (i=0; ((c=getchar()) != '#'); i++) {
    if (i >= size) {
        size *= 2;
        a = (char *) realloc(a, size * sizeof(char));
    }
    a[i] = c;
  }
  while (i > 0)
     putchar(a[--i]);
  free(a);
  return 0;
}
```

```
Console  Shell
> clang-7 -pthread -lm -o main main.c
> ./main
Bonjour
Comment allez-vous?
Au revoir
#

riover uA
?suov-zella tnemmoC
ruojnoB>
```

Allocation mémoire	i=0	Size=5
Bonjour\n	7	10
Comment allez-vous?\n	10, 20, 27	20, 40
Au revoir\n	38	40

EXEMPLES

- **Impression de caractères dans l'ordre inverse : en utilisant l'allocation dynamique (malloc(),realloc(),free())**

```
#define INIT_SIZE 15

int main(void) {
  char* a;
  int i, c, size=INIT_SIZE;

  a = (char *) malloc(size * sizeof(char));
  for (i=0; ((c=getchar()) != '#'); i++) {
    if (i >= size) {
        size *= 4;
        a = (char *) realloc(a, size * sizeof(char));
    }
    a[i] = c;
  }
  while (i > 0)
     putchar(a[--i]);
  free(a);
  return 0;
}
```

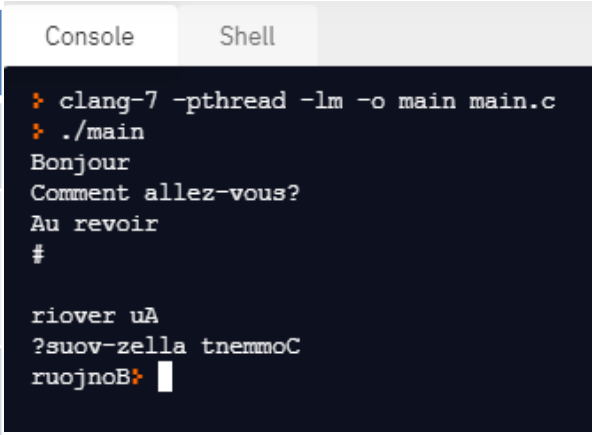
Question : Combien d'octet sont réservé dynamiquement dans la mémoire pour ce programme?

Choix:

A- 15
B- 38
C- 60

5 minutes

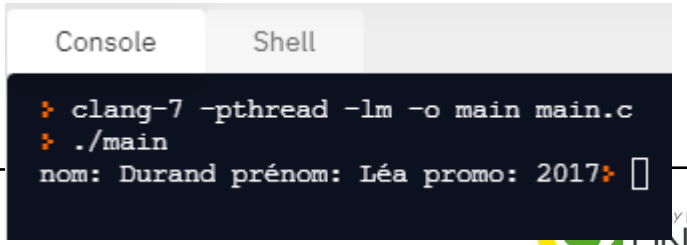
Allocation mémoire	i=0	Size=15
Bonjour\n		
Comment allez-vous?\n		
Au revoir\n		



EXEMPLES

- **Déclaration de structure étudiant : en utilisant l'allocation Statistique**

```c
#include <stdio.h>
// etudiant est un alias de struct étudiant
typedef struct etudiant etudiant;
struct etudiant {
  char nom[128];
  char prenom[128];
  int promo;
};
void changePromo(etudiant *e, int promo) {
(*e).promo = promo; // ou e->promo=promo
}
void printEtudiant(etudiant e){
  printf("nom: %s prénom: %s promo: %d", e.nom, e.prenom,
  e.promo);
}
int main (void)
{
  etudiant lea = { "Durand", "Léa", 2016 };
  changePromo(&lea, 2017);
  printEtudiant(lea);
  return 0;
}
```



EXEMPLES

- **Déclaration de structure étudiant : en utilisant l'allocation dynamique**

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct {
  char nom[128];
  char prenom[128];
  int promo;
} etudiant;
void affEtudiant(etudiant e) {
  printf("nom: %s prénom: %s promo: %d\n", e.nom,
  e.prenom, e.promo);
}
void lireEtudiant(etudiant *e) {
  printf("nom: ");
  scanf("%s", e->nom);
  printf("prénom: ");
  scanf("%s", e->prenom);
  printf("promo: ");
  scanf("%d", &e->promo);
}
```

EXEMPLES

- **Déclaration de structure étudiant : en utilisant l'allocation dynamique**

```
int main(void) {
   int n, i;
   printf("quel est le nombre d'étudiants à saisir : ");
   scanf("%d", &n);
   etudiant *listeEtudiants = (etudiant *) malloc (n *sizeof(etudiant));
   if (!listeEtudiants) {
          printf("Impossible d'allouer la mémoire:\n");
          return 1;
   }
   for (i=0; i<n; i++)
          lireEtudiant(listeEtudiants+i);
   if (n>0)
          printf("Vous avez saisi les étudiants suivants\n");
   else
          printf("Liste vide\n");
   for (i=0; i<n; i++)
          affEtudiant(listeEtudiants[i]);
   free(listeEtudiants);
   return 0;
}
```

```
Console  Shell
> clang-7 -pthread -lm -o main main.c
> ./main
quel est le nombre d'étudiants à saisir : 2
nom: swaileh
prénom: wassim
promo: 2020
nom: Rouen
prénom: France
promo: 2018
Vous avez saisi les étudiants suivants
nom: swaileh prénom: wassim promo: 2020
nom: Rouen prénom: France promo: 2018
>
```

PLAN

- Disposition de la mémoire
- Allocation et désallocation de mémoire
- Exemples
- **Fuites de mémoire**

- **Évitez les fuites de mémoire**
  - Les fuites de mémoire «perdent» les références à la mémoire dynamique.

```
int f(void)
{
    char* p;
    p = (char *) malloc(8 * sizeof(char));
    …
    return 0;
}

int main(void) {
    f();
    …
}
```

- **Évitez les pointeurs suspendus**
  - Des pointeurs suspendus pointent vers des données qui ne sont plus là.

```
char *f(void)
{
     char p[8];

     …
     return p;
}

int main(void) {
     char *res = f();
     …
}
```

RÉSUMÉ

- **Cinq types de mémoire pour les variables**
  - Texte: code, données constantes (données constantes en rodata sur les chapeaux)
  - Données: variables globales et statiques initialisées
  - BSS: variables globales et statiques non initialisées
  - Heap(tas): mémoire dynamique
  - Pile: variables locales
- **Il est important de comprendre les différences entre**
  - Allocation: espace alloué
  - Initialisation: valeur initiale, le cas échéant
  - Désallocation: espace récupéré
- **Comprendre l'allocation de mémoire est important**
  - Utiliser efficacement la mémoire
  - Évitez les «fuites de mémoire» dues aux pointeurs suspendus