Utilisation des systèmes UNIX

Benjamin Negrevergne

PSL University – Paris Dauphine – Équipes *MILES*

Gestion mémoire

Au niveau matériel, il n'y a pas

- de variables, de nom de variable ou de types
  (i.e. pas d'entiers, pas de flottants, pas de chaîne de caractères ou de tableaux)
- de "zone vide" ou "zone écrite"
- de frontières qu'il serait possible positionner arbitrairement

Gestion mémoire

Au niveau matériel, il n'y a pas

- de variables, de nom de variable ou de types
  (i.e. pas d'entiers, pas de flottants, pas de chaîne de caractères ou de tableaux)
- de "zone vide" ou "zone écrite"
- de frontières qu'il serait possible positionner arbitrairement

Il y a seulement

- des bits (0/1) regroupés en octets (bytes)
  - chaque octet a un adresse mémoire
  - on peut modifier la valeur d'un ou plusieurs octets avec LOAD/STORE
- de grosses contraintes de performance
  dues aux problèmes d'alignement mémoire et aux mémoires caches

Gestion mémoire (2)

il faut:

- réserver un nombre suffisant d'octet pour chaque variable
- éviter le chevauchement des valeurs en mémoire (dans le temps ou dans l'espace)
- éviter de trop fragmenter la mémoire
- libérer la mémoire devenue inutile

Gestion mémoire (2)

il faut:

- réserver un nombre suffisant d'octet pour chaque variable
- éviter le chevauchement des valeurs en mémoire (dans le temps ou dans l'espace)
- éviter de trop fragmenter la mémoire
- libérer la mémoire devenue inutile

▶ Différentes *stratégies d'allocation* possible

- Allocation statique (variables globales)
- Allocation sur la pile (variables locales)
- Allocation dynamique (allocation avec malloc)
- ...

Idée: faire cohabiter différentes stratégies pour bénéficier des avantages de chaque stratégie

Pile/Tas

Plusieurs régions qui vont être gérées différemment:

- Allocation dans le segment données du programme
  - \+ Pas de fragmentation
  - − Pas d'allocation dynamique (uniquement pour les variables statiques)

Pile/Tas

Plusieurs régions qui vont être gérées différemment:

- Allocation dans le segment données du programme
  - \+ Pas de fragmentation
  - − Pas d'allocation dynamique (uniquement pour les variables statiques)
- Allocation dans la **pile**
  - \+ Allocation dynamique possible
  - − Allocations/désallocations nécessairement dans l'ordre LIFO
  - \+ Pas de fragmentation

Pile/Tas

Plusieurs régions qui vont être gérées différemment:

- Allocation dans le segment données du programme
  - \+ Pas de fragmentation
  - – Pas d'allocation dynamique (uniquement pour les variables statiques)
- Allocation dans la **pile**
  - \+ Allocation dynamique possible
  - – Allocations/désallocations nécessairement dans l'ordre LIFO
  - \+ Pas de fragmentation
- Allocation sur le **tas**
  - \+ Allocation dynamique dans n'importe quel ordre
  - – Lent
  - – Fragmentation

Pile: 

Tas: 

Pile/Tas (2)

Deux régions de l'espace des adresses utilisées pour stocker les données nécessaires à l'exécution du programme.

Extrait de `/proc/<pid>/maps`

```
5603b8e22000-5603b8e43000 rw-p 00000000 00:00 0                          [heap]
7ffda2e9e000-7ffda2ebf000 rw-p 00000000 00:00 0                          [stack]
```

Taille pile/tas:

- 1.3 Go pour le tas.
- 132 Ko pour la pile.

Pile/Tas (2)

Deux régions de l'espace des adresses utilisées pour stocker les données nécessaires à l'exécution du programme.

Extrait de `/proc/<pid>/maps`

```
5603b8e22000-5603b8e43000 rw-p 00000000 00:00 0                          [heap]
7ffda2e9e000-7ffda2ebf000 rw-p 00000000 00:00 0                          [stack]
```

Taille pile/tas:

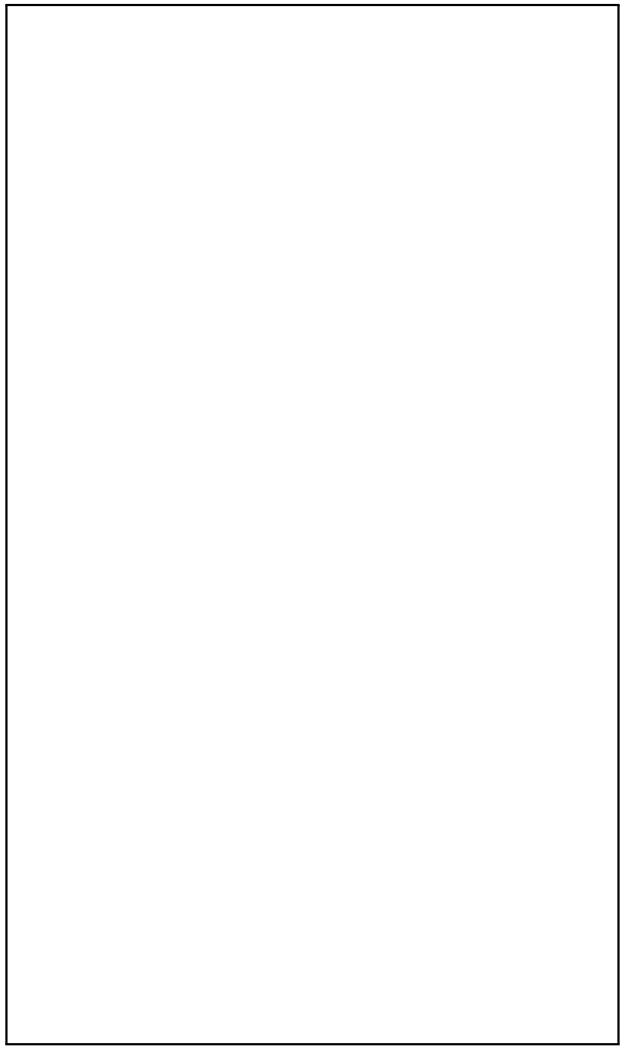
- 1.3 Go pour le tas.
- 132 Ko pour la pile.

```
void f(int a){
 int b = 2;
 int *c = malloc(2*sizeof(int));
 printf("%p, %p, %p, %p\n",
           &a, &b, &c, c);
}
```

```
&a: 0x7ffda2ebcc0c -> pile
&b: 0x7ffda2ebcc1c -> pile
&c: 0x7ffda2ebcc10 -> pile
 c: 0x5603b8e222a0 -> tas
```

Pile/Tas

```
1 &a: 0x7ffda2ebcc0c -> pile @12
2 &b: 0x7ffda2ebcc1c -> pile @28
3 &c: 0x7ffda2ebcc10 -> pile @16
4  c: 0x5603b8e222a0 -> tas
```



(note: schéma à compléter)

Pile (stack)

- Utilisée pour les variables locales et les arguments de fonctions.
- Allocation et libération implicite (gérée par le **compilateur**)
  - La mémoire est allouée au début de l'exécution de la fonction
  - La mémoire est libérée à la fin de l'exécution de la fonction
- Taille de la mémoire à allouer est déterminée à partir du types des variables
- Impossible de prolonger la durée de vie d'une variable au delà du contexte d'exécution de la fonction
- Pas de fuite mémoire possible
- Pas de fragmentation mémoire

Tas (heap)

- Allocation/libération explicite (gérée **par le programmeur** via `malloc` et `free`)
- Taille de la mémoire à allouer est déterminée par le programmeur
- Possibilité de prolonger la durée de vie d'une allocation arbitrairement
- La fragmentation est inévitable
- ❦possibilité de fuite mémoire

Tas (heap)

- Allocation/libération explicite (gérée **par le programmeur** via `malloc` et `free`)
- Taille de la mémoire à allouer est déterminée par le programmeur
- Possibilité de prolonger la durée de vie d'une allocation arbitrairement
- La fragmentation est inévitable
- ❦possibilité de fuite mémoire

▶ Préférer les allocations sur la pile lorsque c'est possible!

```
1 readbuf(...){
2   char *buf = malloc(sizeof(char)*BUF_SIZE); //inutile
3   read(fd, buf, BUF_SIZE);
4   printf("buf : %s\n", buf);
5   free(buf);
6 }
```

Pile et contexte d'exécution d'une fonction

```
1 int power(int x, int n){
2     int res;
3     if ( n == 0)
4        res = 1;
5     else
6        res = x * power(x, n - 1);
7     return res;
8 }
```

Pile et contexte d'exécution d'une fonction

```
1 int power(int x, int n){
2     int res;
3     if ( n == 0)
4        res = 1;
5     else
6        res = x * power(x, n - 1);
7     return res;
8 }
```

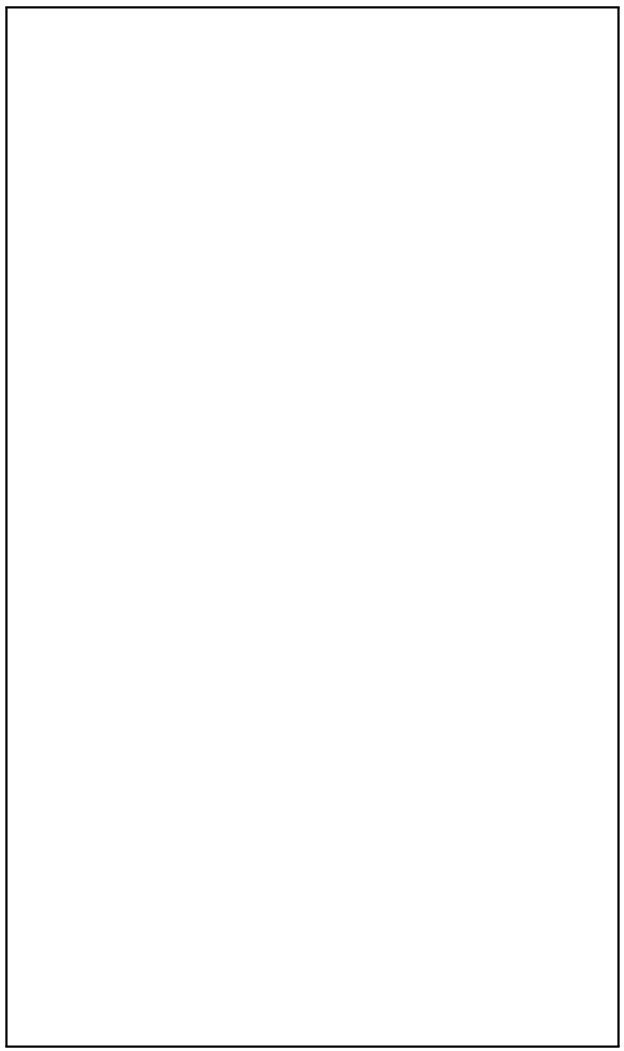
Combien de mémoire est nécessaire pour exécuter `power(2,3)`? (sans optimisation)

- 4 octet par `int`
- 3 entiers pour chaque appel de la fonction `power`
- 4 appels récursifs

▶ 48 octets a réserver!

Exécution récursive avec allocation sur la pile

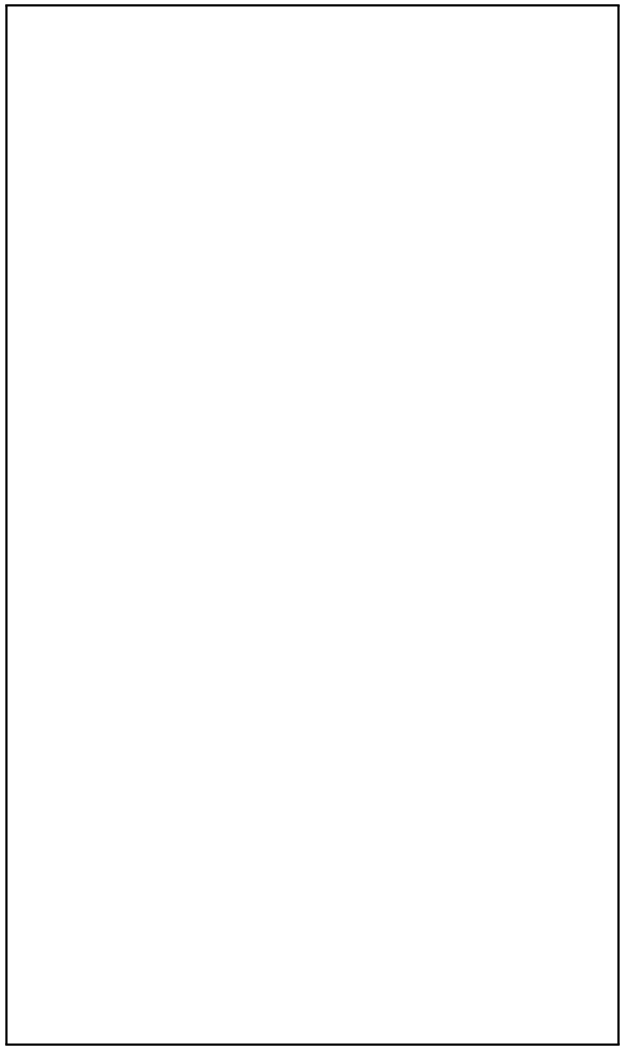
- `p(2,3)`
  x = 2, n = 3, res = ?



(note: schéma à compléter)

Exécution récursive avec allocation sur la pile

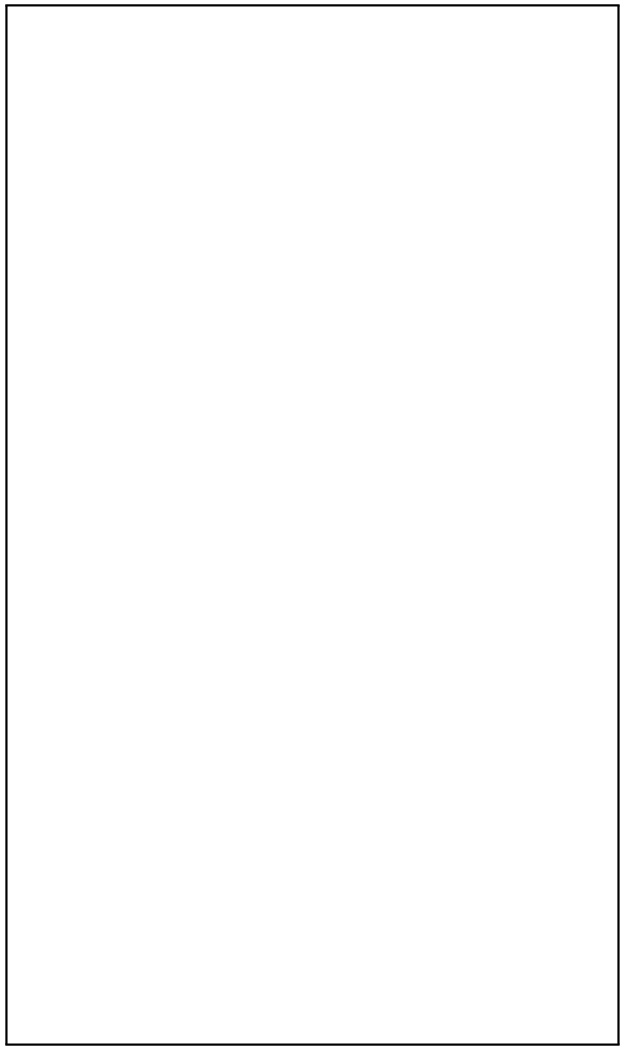
- `p(2,3)`
  x = 2, n = 3, res = ?
- `p(2,2)`
  x = 2, n = 2, res = ?



(note: schéma à compléter)

Exécution récursive avec allocation sur la pile

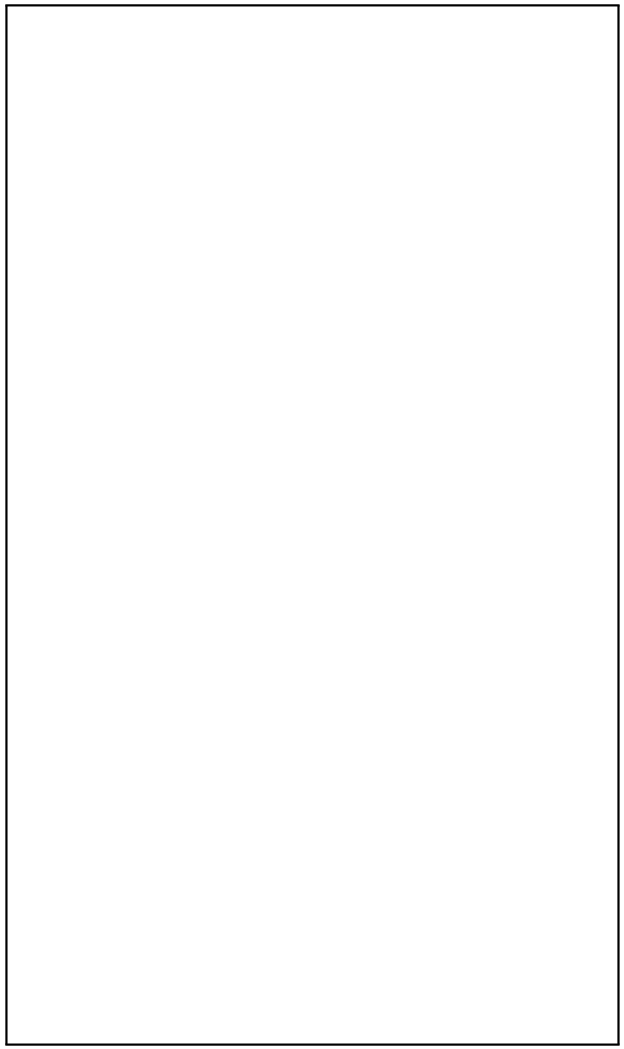
- `p(2,3)`
  x = 2, n = 3, res = ?
- `p(2,2)`
  x = 2, n = 2, res = ?
- `p(2,1)`
  x = 2, n = 1, res = ?



(note: schéma à compléter)

Exécution récursive avec allocation sur la pile

- `p(2,3)`
  x = 2, n = 3, res = ?
- `p(2,2)`
  x = 2, n = 2, res = ?
- `p(2,1)`
  x = 2, n = 1, res = ?
- `p(2,0)`
  x = 2, n = 0, res = 1



(note: schéma à compléter)

Pile et contexte d'exécution d'une fonction

La pile d'une fonction est déterminée par deux registres **BP** et **SP**

- **BP**: **Base Stack Pointer**:
  ▶ début de la pile de la fonction en cours d'exécution
- **SP**: **Stack Pointer**:
  ▶ fin de la pile de la fonction en cours d'exécution

Pile et contexte d'exécution d'une fonction

La pile d'une fonction est déterminée par deux registres **BP** et **SP**

- **BP**: **Base Stack Pointer**:
  ▶ début de la pile de la fonction en cours d'exécution
- **SP**: **Stack Pointer**:
  ▶ fin de la pile de la fonction en cours d'exécution

- L'espace entre BP et SP l'espace réservé pour la pile de la fonction
- Toutes les variables locales sont référencées de manière relative à BP
- Le SP et le BP de la fonction appelante sont restauré à la fin de l'exécution de la fonction

Allocation sur la pile

Code C

```c
void f(){
  int a = 2;
  int b = 3;
  int c = 4;
  int d = a + b;
  printf("%d %d %d\n", a, b, c, d);
}
```

Code Assembleur (simplifié)

```
        pushq   %rbp            // sauvegarde le BP de la fonction appelante,
        movq    %rsp, %rbp      // le SP devient le BP
        subq    $16, %rsp       // réserve 4x sizeof(int)
        movl    $2, -4(%rbp)    // écrit 2 dans a
        movl    $3, -8(%rbp)    // écrit 3 dans b
        movl    $4, -12(%rbp)   // écrit 4 dans c
        movl    -4(%rbp), %edx  // copie a dans le registre EDX
        movl    -8(%rbp), %eax  // copie b dans le registre EAX
        addl    %edx, %eax      // ajoute le contenu de EDX et EAX
        movl    %eax, -16(%rbp) // copie le résultat dans d
        ...
        call    printf@PLT
```

Erreur classique

```
1  int *f(){
2   int a = 2;
3   int b = 3;
4   int res = a + b;
5   return &res;
6  }
7  int main(){
8    int *r = f();
9    printf("%p %d\n");
10 }
```

Résultat ?

Erreur classique

```
1  int *f(){
2   int a = 2;
3   int b = 3;
4   int res = a + b;
5   return &res;
6  }
7  int main(){
8    int *r = f();
9    printf("%p %d\n");
10 }
```

Résultat ?

- Non défini!
  - r pointe vers une adresse dans la pile de f
  - la pile de f est libérée a la fin de l'exécution de `f`!

À l'exécution

```c
#include <stdio.h>

int *f(){
 int a = 2;
 int b = 3;
 int res = a + b;
 return &res;
}

int g(){
  printf("g écrase la pile\n");
}

int main(){
  int *r = f();
  // g();
  printf("Res: %d\n", *r);
}
```

- Sans l'appel à g() dans main()

```
$./a.out
Res: 5
```

- Avec l'appel à g()

```
$./a.out
g écrase la pile
Res 0
```

Comment détecter ce bug

```
1 $ gcc demo-local-var.c
2 demo_local_var.c: In function 'f':
3 demo_local_var.c:7:9: warning: function returns address of local variable [-
      Wreturn-local-addr]
4     7 |  return &res;
5       |         ^~~~
```

Comment détecter ce bug

```
$ gcc demo-local-var.c
demo_local_var.c: In function 'f':
demo_local_var.c:7:9: warning: function returns address of local variable [-
    Wreturn-local-addr]
   7 |  return &res;
     |         ^~~~
```

NE PAS IGNORER LES WARNINGS

Autre erreur classique

```
1 void broken(){
2   char T[4];
3   for(int i = 0; i <= 4; i++)
4     T[i] = 0;
5 }
6
7 int main(int argc, char **argv){
8   broken();
9 }
```

Résultat ?

Autre erreur classique

```
1 void broken(){
2   char T[4];
3   for(int i = 0; i <= 4; i++)
4     T[i] = 0;
5 }
6
7 int main(int argc, char **argv){
8   broken();
9 }
```

Résultat ?

- boucle infinie

Comment détecter ce bug

Avec un debuger

```
1 demo gdb 1
```

Autre erreur classique (2)

```
1  void broken(){
2    char *src = "abcd";
3    int len = strlen(src);
4    char *dest = malloc(len * sizeof(char));
5
6    for(int i = 0; i < len; i++)
7      dest[i] = src[i];
8
9    printf("%s",dest);
10
11
12 }
13
14 int main(int argc, char **argv){
15   broken();
16 }
```

Résultat ?

Autre erreur classique (2)

```
1  void broken(){
2    char *src = "abcd";
3    int len = strlen(src);
4    char *dest = malloc(len * sizeof(char));
5
6    for(int i = 0; i < len; i++)
7      dest[i] = src[i];
8
9    printf("%s",dest);
10
11
12 }
13
14 int main(int argc, char **argv){
15   broken();
16 }
```

Résultat ?

- probablement abcd

Comment détecter ce bug

Avec un debuger

```
1 demo valgrind
```

Limites de la pile

Sur la **pile**:

- La mémoire allouée pour les variable de pile est libérée systématiquement à la fin de l'exécution de l'exécution de la fonction
  - Les désallocations se font **toujours** dans l'ordre inverse des allocations
  - L'intégralité de l'espace libre se trouve toujours en haut de la pile
- L'espace requis pour chaque variable est fixe, et connu à la compilation

Limites de la pile

Sur la **pile**:

- La mémoire allouée pour les variable de pile est libérée systématiquement à la fin de l'exécution de l'exécution de la fonction
  - Les désallocations se font **toujours** dans l'ordre inverse des allocations
  - L'intégralité de l'espace libre se trouve toujours en haut de la pile
- L'espace requis pour chaque variable est fixe, et connu à la compilation

▶ Le compilateur s'occupe du **placement des variables** dans la pile

Limites de la pile

Sur la **pile**:

- La mémoire allouée pour les variable de pile est libérée systématiquement à la fin de l'exécution de l'exécution de la fonction
  - Les désallocations se font **toujours** dans l'ordre inverse des allocations
  - L'intégralité de l'espace libre se trouve toujours en haut de la pile
- L'espace requis pour chaque variable est fixe, et connu à la compilation

▶ Le compilateur s'occupe du **placement des variables** dans la pile

Que faire lorsque:

- On souhaite préserver le contenu d'une variable ou d'un tableau au délà de la portée d'une fonction?
- La taille de la mémoire nécessaire pour une variable/un tableau est inconnue à la compilation?

Limite de la pile N°1: allocation persistantes

```
1
2  #define STUDENT_COUNT 20
3
4  int *all_grades(){
5     int grades[STUDENT_COUNT] = {0};
6     return grades;
7  }
8
9  int main(){
10
11    int *g = all_grades();
12
13    for (int i = 0; i < STUDENT_COUNT; i++)
14      printf("grade: %d\n", grades[i]);
15
16 }
```

Limite de la pile N°1: allocation persistantes

```
1
2  #define STUDENT_COUNT 20
3
4  int *all_grades(){
5     int grades[STUDENT_COUNT] = {0};
6     return grades;
7  }
8
9  int main(){
10
11    int *g = all_grades();
12
13    for (int i = 0; i < STUDENT_COUNT; i++)
14      printf("grade: %d\n", grades[i]);
15
16 }
```

Non! L'espace réservé pour le tableau `grades` (dans `all_grades`) est libéré à la fin de l'exécution de la fonction `all_grades`

Allocation persistantes sur le tas

```
1
2  #define STUDENT_COUNT 20
3
4  int *all_grades(){
5     int * grades = malloc(sizeof(int) * STUDENT_COUNT);
6     return grades;
7  }
8
9  int main(){
10
11    int *g = all_grades();
12
13    for (int i = 0; i < STUDENT_COUNT; i++)
14      printf("grade: %d\n", grades[i]);
15
16    free(g);
17
18 }
```

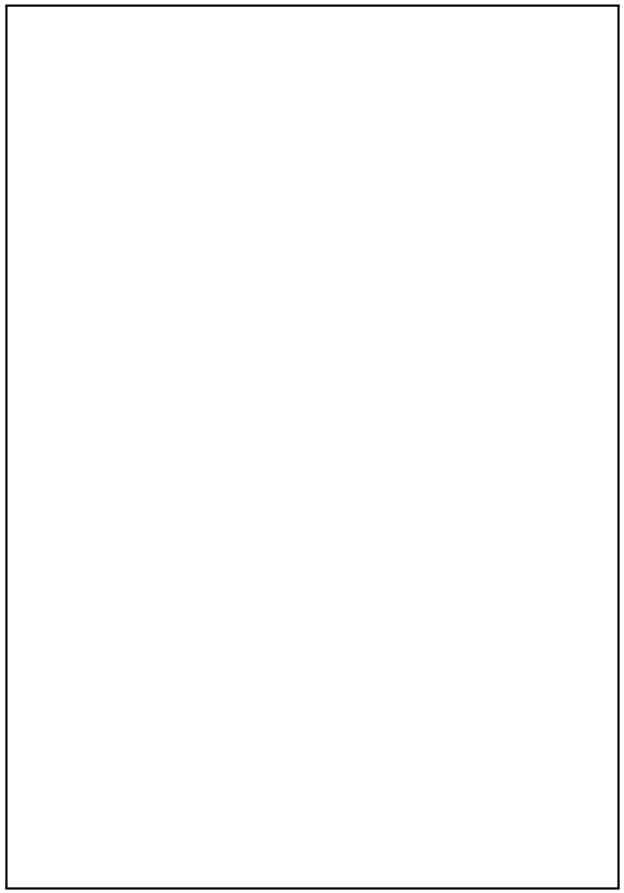
Allocation persistantes sur le tas

```
1
2  #define STUDENT_COUNT 20
3
4  int *all_grades(){
5     int * grades = malloc(sizeof(int) * STUDENT_COUNT);
6     return grades;
7  }
8
9  int main(){
10
11    int *g = all_grades();
12
13    for (int i = 0; i < STUDENT_COUNT; i++)
14      printf("grade: %d\n", grades[i]);
15
16    free(g);
17
18 }
```

Ok. L'espace réservé pour le tableau `grades` (dans `all_grades`) est maintenu jusqu'à l'appel de la fonction `free`.

Limite de la pile N°2: allocation dynamique

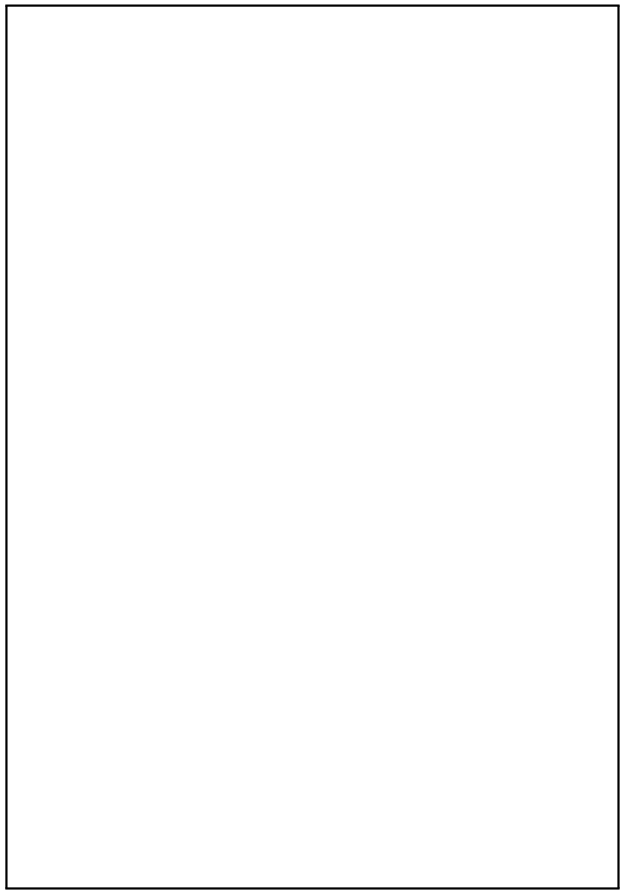
```
1
2  #include <stdio.h> //printf,scanf
3  #include <stdlib.h> //malloc
4
5  int main(){
6    int student_count;
7    printf("Nombre d'étudiants?: \n");
8    scanf("%d", &student_count);
9
10   int grades[student_count]; //allocation
         pile?
11   int dummy;
12   /* do something */
13 }
```



(note: schéma à compléter)

Limite de la pile N°2: allocation dynamique

```
1
2  #include <stdio.h> //printf,scanf
3  #include <stdlib.h> //malloc
4
5  int main(){
6    int student_count;
7    printf("Nombre d'étudiants?: \n");
8    scanf("%d", &student_count);
9
10   int grades[student_count]; //allocation
           pile?
11   int dummy;
12   /* do something */
13 }
```



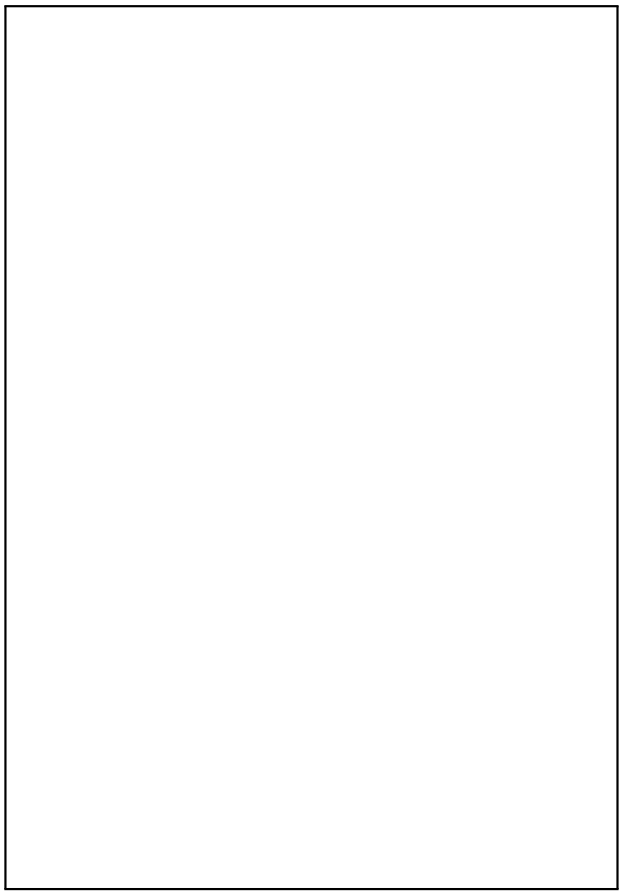
(note: schéma à compléter)

Non: `student_count` est variable (inconnu à la compilation)

▶ Le compilateur ne peut pas gérer le placement des variables de piles si la taille de `grades` n'est pas connue

Allocation dynamique sur le tas

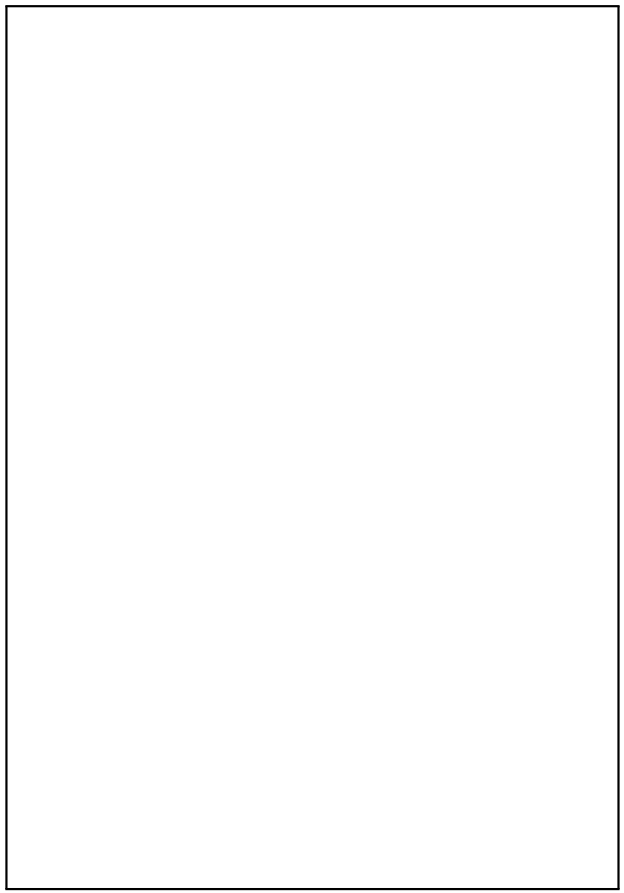
```
1
2  #include <stdio.h> //printf,scanf
3  #include <stdlib.h> //malloc
4
5  int main(){
6    int student_count;
7    printf("Nombre d'étudiants?: \n");
8    scanf("%d", &student_count);
9    int *grades;
10   grades = malloc(
11      sizeof(int) * student_count);
12   int dummy;
13   /* do something */
14   free(grades);
15 }
```



(note: schéma à compléter)

Allocation dynamique sur le tas

```
1
2  #include <stdio.h> //printf,scanf
3  #include <stdlib.h> //malloc
4
5  int main(){
6    int student_count;
7    printf("Nombre d'étudiants?: \n");
8    scanf("%d", &student_count);
9    int *grades;
10   grades = malloc(
11      sizeof(int) * student_count);
12   int dummy;
13   /* do something */
14   free(grades);
15 }
```



(note: schéma à compléter)

Ok. grades est un pointeur (taille fixe sur la pile)

Le tas!

Le **tas** est un (grand) espace de mémoire initialement vide, attribué au démarrage du processus. Réservé pour les allocation **dynamiques**.

`malloc` et `free` sont les deux fonction utilisées pour faire des allocation sur le tas.
Elles permettent de garder la trace des blocs libres/occupés à mesure des allocations.

`malloc`

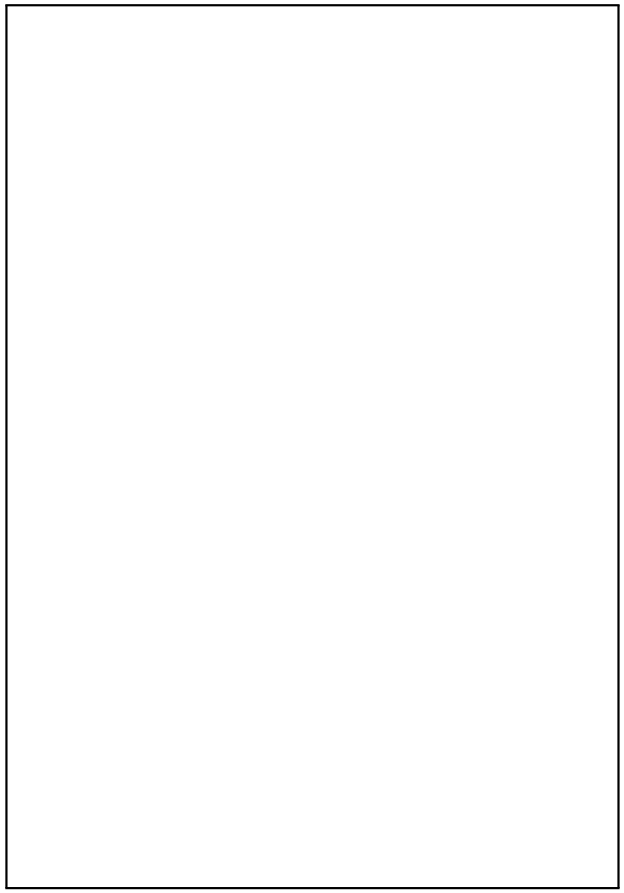
- parcours l'espace à la recherche d'un bloc disponible de taille suffisante, et le retourne à l'utilisateur

`free`

- libère le bloc alloué et le marque comme à nouveau disponible

Stratégie d'allocation

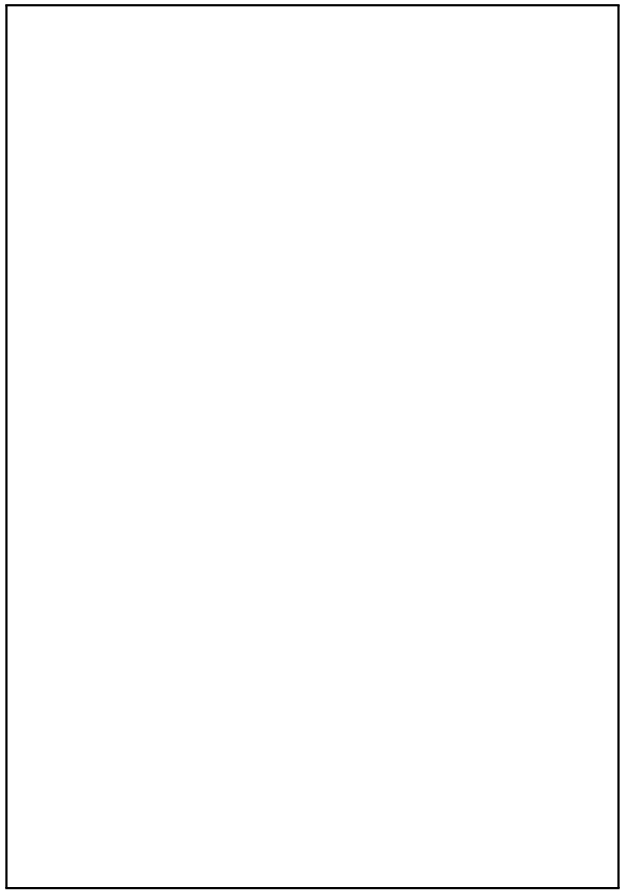
```
1 char *p1 = malloc(2);
2 char *p2 = malloc(5);
3 char *p3 = malloc(2);
4 char *p4 = malloc(4);
5 char *p5 = malloc(5);
6 free(p2);
7 free(p4);
8 malloc(4); //où ça va?
```



(note: schéma à compléter)

Stratégie d'allocation

```
1 char *p1 = malloc(2);
2 char *p2 = malloc(5);
3 char *p3 = malloc(2);
4 char *p4 = malloc(4);
5 char *p5 = malloc(5);
6 free(p2);
7 free(p4);
8 malloc(4); //où ça va?
```



(note: schéma à compléter)

- Nécessité d'une stratégie d'allocation pour limiter la **fragmentation**
  - first fit
  - best fit
  - ...

Chaînage des blocs

Descripteur de bloc, placés dans la mémoire:

```
1  typedef struct block {
2    int size;
3    struct block_t *next;
4  } block_t;
5
6  block_t *first_free;
```

Chaînage des blocs

Descripteur de bloc, placés dans la mémoire:

```
1  typedef struct block {
2    int size;
3    struct block_t *next;
4  } block_t;
5
6  block_t *first_free;
```

Les blocs vide sont chaînés entre eux à l'aide du pointeur `next`:

- Le tas est initialisé avec un seul grand bloc vide
- Lors d'un appel à malloc, la liste des blocs vide est parcourue pour trouver un bloc de taille suffisante
- Le pointeur vers l'espace utile (juste après le descripteur) est retourné à l'utilisateur)
- Le bloc est sorti de la liste des blocs vide
- Lors d'un appel a `free` le bloc est réintégrer dans la liste des blocs vides.