Programmation Avancée

Cours 3 : Le modèle mémoire

Simon Forest

11 février 2021

Dans ce cours

- Comment un programme `C` est stocké en mémoire ?
- À quels types de mémoire un programme a accès ?
- Pourquoi faut-il faire des `malloc` et des `free` parfois ?
- C'est quoi une erreur de segmentation ?

Qu'est-ce que la « mémoire » ?

La « mémoire » d'un ordinateur est essentiellement constituée :

- de la **mémoire de stockage** (disque dur, clé USB, SSD, *etc.*)
  C'est une mémoire de grande taille (plusieurs dizaines de gigaoctets) mais à accès relativement lent (sauf le SSD).
- de la **mémoire vive** : RAM pour « Random Access Memory »

Mémoire de stockage

Ce type de mémoire stocke de façon fiable de **grandes quantités de données** (plusieurs gigaoctets) sur de **longues durées** (plusieurs années) même sans alimentation.

C'est la mémoire qui stocke les fichiers de l'ordinateur.

Exemples :

- disque dur
- clé USB
- SSD

Cependant, l'accès à cette mémoire est **lent** et en particulier inadapté à l'exécution des programmes.

Mémoire vive

Ce type de mémoire stocke des données **de taille modérée** (quelques gigaoctets) de façon fiable tant qu'elle est alimentée en énergie.

C'est la mémoire qui stocke **les données en cours d'utilisation** de l'ordinateurs (programmes, fichiers ouverts, *etc.*).

On parle en anglais de RAM (*Random Access Memory*) pour souligner que l'ensemble de cette mémoire s'accède rapidement (contrairement aux disques durs par ex.).

L'accès à cette mémoire est **rapide** et adapté à l'exécution des programmes.

Exécution d'un programme

Lorsque l'on lance l'exécution d'un programme, son contenu **passe de la mémoire de stockage à la mémoire vive** avant que ne commence son exécution.

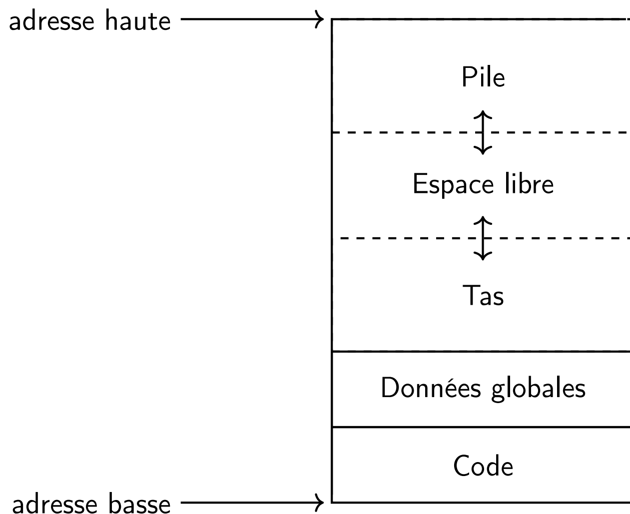
Le contenu chargé en mémoire est divisé en **segments** ayant des rôles différents.

On distingue 4 « zones » principales regroupant un ou plusieurs segments :

- la zone du code du programme
- la zone des variables globales
- la zone de la **pile**
- le zone du **tas**

Les différentes zones à charger sont pour la plupart décrites dans le fichier exécutable à charger.

Schéma



Sommaire

La zone du code

Données globales

La pile

Le tas

Les erreurs mémoire

Sommaire

La zone du code

Données globales

La pile

Le tas

Les erreurs mémoire

La zone du code

Cette zone est la où est stocké le code du programme sous la forme de code **assembleur** (séquence d'octets encodant des opérations du processeur).

Cette « zone » est constituée essentiellement de la section `.text` qui contient le code principal du programme.

Exemple

```c
int f(int x)
{
  return x+1;
}
int main()
{
  printf("%d",f(0));
  return 0;
}
```

On compile et on examine la section `text` du programme obtenu :

```
gcc main.c -o prog
objdump -d --section=.text
```

Exemple

```c
int f(int x)
{
  return x+1;
}
int main()
{
  printf("%d",f(0));
  return 0;
}
```

```
Déassemblage de la section .text :
...
0000000000400526 <f>:
400526:       55                    push   %rbp
400527:       48 89 e5              mov    %rsp,%rbp
40052a:       89 7d fc              mov    %edi,-0x4(%rbp)
40052d:       8b 45 fc              mov    -0x4(%rbp),%eax
400530:       83 c0 01              add    $0x1,%eax
400533:       5d                    pop    %rbp
400534:       c3                    retq
```

Exemple

```c
int f(int x)
{
  return x+1;
}
int main()
{
  printf("%d",f(0));
  return 0;
}
```

```
Déassemblage de la section .text :
...
0000000000400535 <main>:
400535:        55                            push   %rbp
400536:        48 89 e5                      mov    %rsp,%rbp
400539:        bf 00 00 00 00                mov    $0x0,%edi
40053e:        e8 e3 ff ff ff                callq  400526 <f>
...      ...                   ...
40054f:        e8 ac fe ff ff                callq  400400 <printf@plt>
...      ...                   ...
```

Sommaire

La zone du code

Données globales

La pile

Le tas

Les erreurs mémoire

Variables globales

Ce sont des variables qui existent en dehors des fonctions.

```c
int x; // variable globale x accessible partout
void f()
{
  x++;            // accès possible de x par f
  return;
}
int main()
{
  f();
  printf("%d",x); // accès possible de x par main
  return 0;
}
```

Variables globales

Cela contraste avec les **variables locales**, uniquement accessibles dans le bloc `{...}` où elles sont définies.

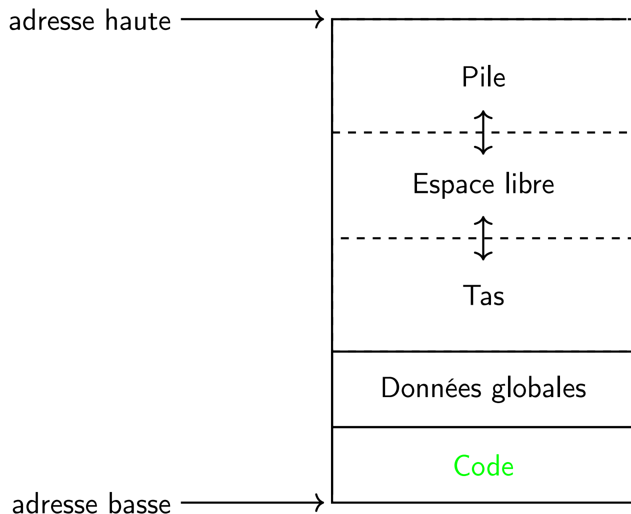
```
void f()
{
  int x; // x accessible uniquement dans f
  x++;
  return;
}
int main()
{
  printf("%d",x); // ERREUR: x n'existe pas dans main!
  return 0;
}
```

Variables globales

Cela contraste avec les **variables locales**, uniquement accessibles dans le bloc `{...}` où elles sont définies.

```
int main()
{
  for(int i = 0; i < 3; i++)
  {
    int x = i*i;
    printf("%d",x);
  }
  printf("%d",x); // ERREUR: x n'existe plus ici
  return 0;
}
```

Retour au schéma



Données globales

La zone des données globales sert à stocker... les **variables globales**.

Elle est constituée de plusieurs sections pour les différents types de variables globales :

- la section `.data` pour les variables globales initialisées
- la section `.bss` pour les variables globales non initialisées
- la section `.rodata` pour les constantes

Exemple

```c
int x = 0x11223344;
int y;
const int z = 0x55667788;
char message[6] = "Salut";
int main()
{
  printf("Bonjour %s!", "Simon");
  return 0;
}
```

Exemple

```c
int x = 0x11223344;
int y;
const int z = 0x55667788;
char message[6] = "Salut";
int main()
{
  printf("Bonjour %s!", "Simon");
  return 0;
}
```

```
gcc main.c -o prog
objdump -s --section=.data prog
```

```
Contenu de la section .data :
 601028 00000000 00000000 00000000 00000000  ................
 601038 44332211 53616c75 7400               D3".Salut.
```

Exemple

```c
int x = 0x11223344;
int y;
const int z = 0x55667788;
char message[6] = "Salut";
int main()
{
  printf("Bonjour %s!", "Simon");
  return 0;
}
```

```
gcc main.c -o prog
objdump -s --section=.rodata prog
```

```
Contenu de la section .rodata :
 4005d0 01000200 88776655 53696d6f 6e00426f  .....wfUSimon.Bo
 4005e0 6e6a6f75 72202573 2100               njour %s!.
```

Exemple

```c
int x = 0x11223344;
int y;
const int z = 0x55667788;
char message[6] = "Salut";
int main()
{
  printf("Bonjour %s!", "Simon");
  return 0;
}
```

```
gcc main.c -o prog
objdump -s --section=.bss prog
```

```
Section .bss vide
```

C'est normal, car il n'y a rien besoin de stocker pour `.bss`.

Variables `static`

Un autre type de variables est constitué par les variables déclarées `static`.

Ce sont des variables

- qui sont déclarées et initialisées à l'intérieur d'une fonction
- qui sont uniquement visibles pour cette fonction
- qui sont allouées à l'exécution du programme et détruites à son arrêt (comme les variables globales)

Elles sont aussi stockées dans la zones des données globales.

Exemple

```c
int f()
{
  int c = 0;
  c++;
  return c;
}
int main()
{
  for(int i = 0; i < 10; i++)
    printf("%d, ", f());
  return 0;
}
```

```
1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

Exemple

```c
int f()
{
  static int c = 0; // variable statique ici
  c++;
  return c;
}
int main()
{
  for(int i = 0; i < 10; i++)
    printf("%d, ", f());
  return 0;
}
```

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

Sommaire

La zone du code

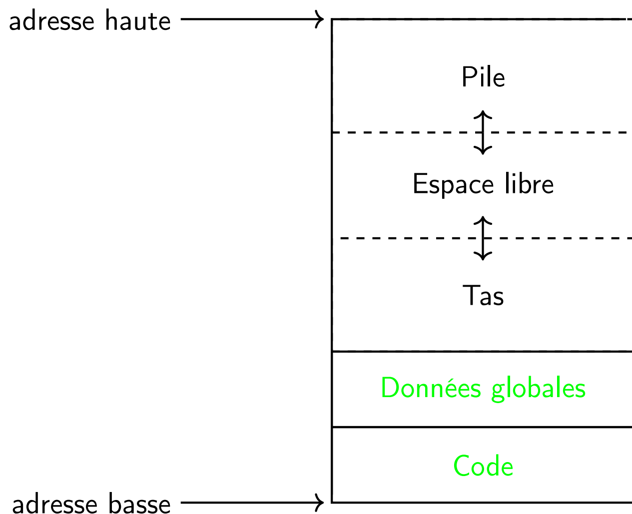
Données globales

La pile

Le tas

Les erreurs mémoire

Retour au schéma



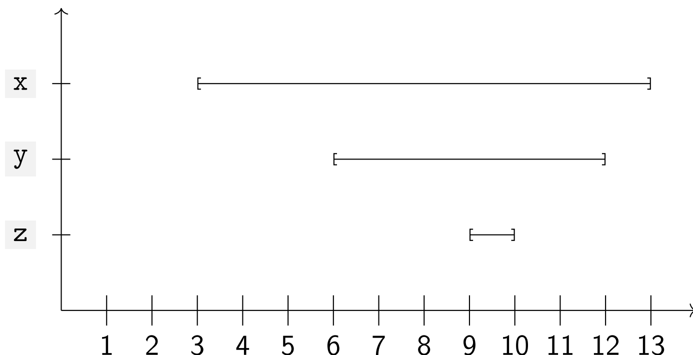
Variables locales

```c
int main()
{
  int x = 3;
  while(x > 0)
  {
    int y = x+1;
    if(y > 2)
    {
      int z = y+2;
    }
    x--;
  }
}
```

Variables locales

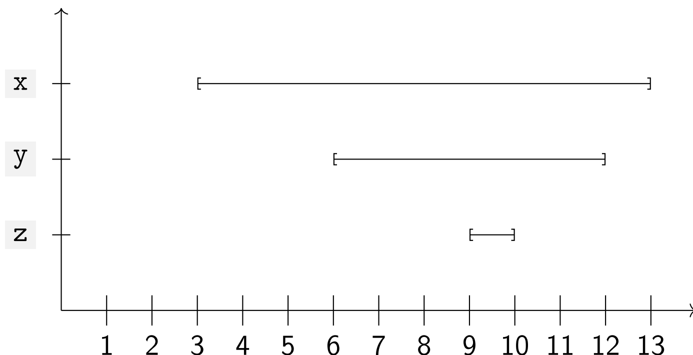
```c
int main()
{
  int x = 3;
  while(x > 0)
  {
    int y = x+1;
    if(y > 2)
    {
      int z = y+2;
    }
    x--;
  }
}
```

Périodes d'existence des variables



Variables locales

```
1  int main()
2  {
3    int x = 3;
4    while(x > 0)
5    {
6      int y = x+1;
7      if(y > 2)
8      {
9        int z = y+2;
10     }
11     x--;
12   }
13 }
```



Observation : dernière variable apparue, première disparue.

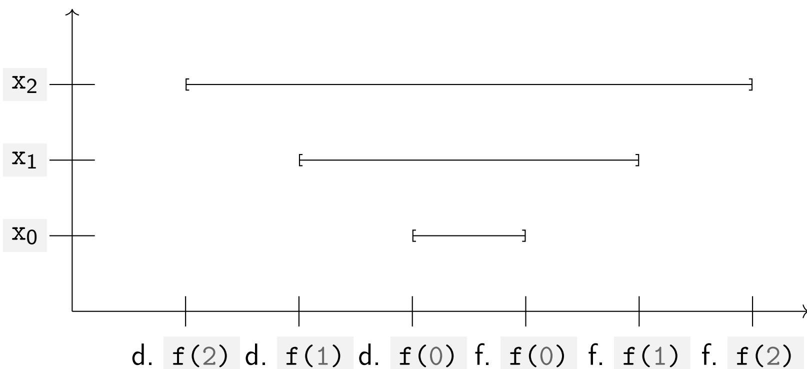
Variables locales : le cas récursif

```c
int f(int x)
{
  if(x == 0)
  {
    return 0;
  }
  return f(x-1) + x;
}
int main()
{
  f(2);
}
```

Variables locales : le cas récursif

```
1   int f(int x)
2   {
3     if(x == 0)
4     {
5       return 0;
6     }
7     return f(x-1) + x;
8   }
9   int main()
10  {
11    f(2);
12  }
```

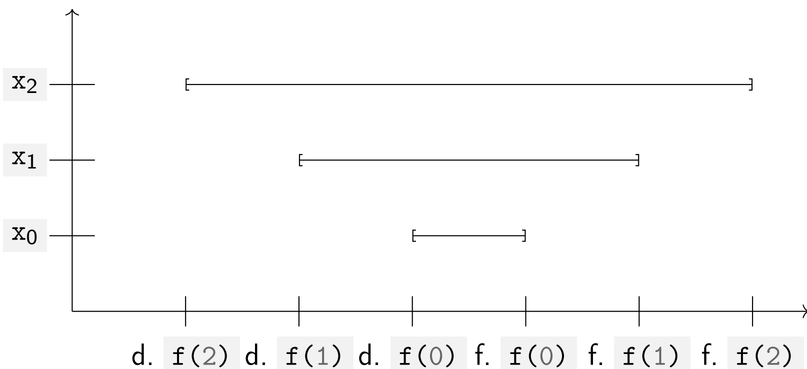
Périodes d'existence des variables



Variables locales : le cas récursif

```
1   int f(int x)
2   {
3     if(x == 0)
4     {
5       return 0;
6     }
7     return f(x-1) + x;
8   }
9   int main()
10  {
11    f(2);
12  }
```

Périodes d'existence des variables



Même observation : dernière variable apparue, première disparue.

Pile

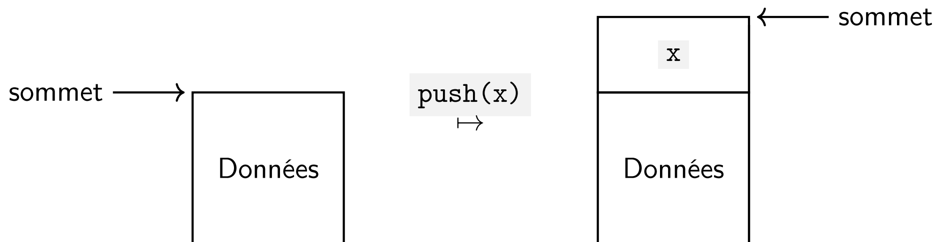
Une **pile** est une structure permettant de stocker des données et satisfaisant le principe **LIFO** (*Last In First Out*, ou Dernier Arrivé Premier Sorti).

Pile

Une **pile** est une structure permettant de stocker des données et satisfaisant le principe **LIFO** (*Last In First Out*, ou Dernier Arrivé Premier Sorti).

Une pile a donc habituellement deux opérations :

- une opération `push` pour ajouter des données au sommet de la pile

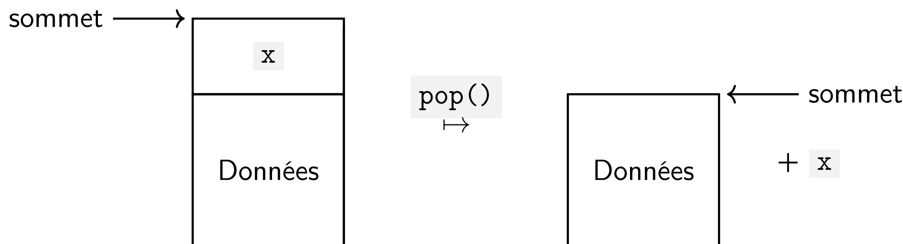


Pile

Une **pile** est une structure permettant de stocker des données et satisfaisant le principe **LIFO** (*Last In First Out*, ou Dernier Arrivé Premier Sorti).

Une pile a donc habituellement deux opérations :

- une opération `push` pour ajouter des données au sommet de la pile
- une opération `pop` pour libérer des données du sommet de la pile



Pile d'un programme

La mémoire d'un programme est constituée d'une pile.

Pile d'un programme

La mémoire d'un programme est constituée d'une pile.

Cette pile a la particularité d'avoir **le sommet en bas** : les éléments sont empilés et dépilés **par le bas**.

Pile d'un programme

La mémoire d'un programme est constituée d'une pile.

Cette pile a la particularité d'avoir **le sommet en bas** : les éléments sont empilés et dépilés **par le bas**.

L'adresse du sommet de la pile est stockée dans un registre appelé `%sp` (pour *Stack Pointer*).

Pile d'un programme

La mémoire d'un programme est constituée d'une pile.

Cette pile a la particularité d'avoir **le sommet en bas** : les éléments sont empilés et dépilés **par le bas**.

L'adresse du sommet de la pile est stockée dans un registre appelé `%sp` (pour *Stack Pointer*).

La valeur de ce registre change constamment pour allouer/désallouer de l'espace pour les variables locales.

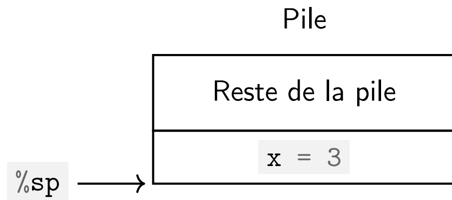
Exemple

```
1  int main()          ←
2  {
3    int x = 3;
4    while(x > 0)
5    {
6      int y = x+1;
7      if(y > 2)
8      {
9        int z = y+2;
10     }
11     x--;
12   }
13 }
```

Pile

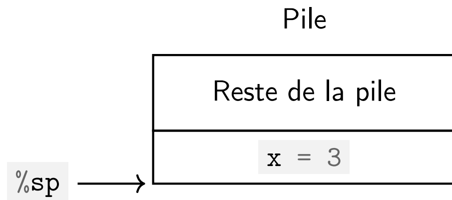
Exemple

```
1   int main()
2   {
3     int x = 3;        ←
4     while(x > 0)
5     {
6       int y = x+1;
7       if(y > 2)
8       {
9         int z = y+2;
10      }
11      x--;
12    }
13  }
```



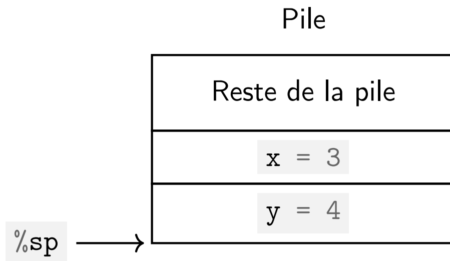
Exemple

```
1  int main()
2  {
3    int x = 3;
4    while(x > 0)      ←
5    {
6      int y = x+1;
7      if(y > 2)
8      {
9        int z = y+2;
10     }
11     x--;
12   }
13 }
```



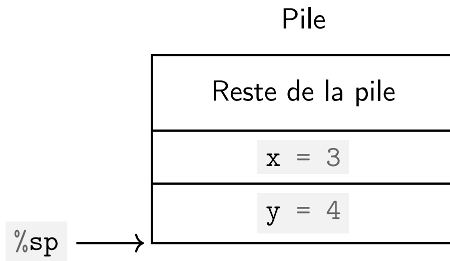
Exemple

```
1   int main()
2   {
3     int x = 3;
4     while(x > 0)
5     {
6       int y = x+1;    ←
7       if(y > 2)
8       {
9         int z = y+2;
10      }
11      x--;
12    }
13  }
```



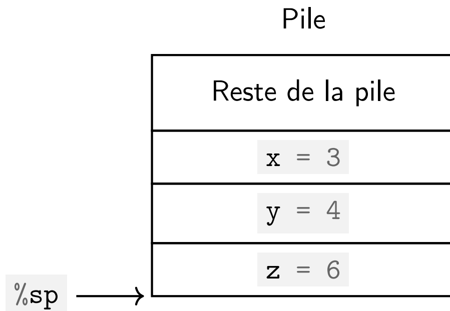
Exemple

```c
int main()
{
  int x = 3;
  while(x > 0)
  {
    int y = x+1;
    if(y > 2)        ←
    {
      int z = y+2;
    }
    x--;
  }
}
```



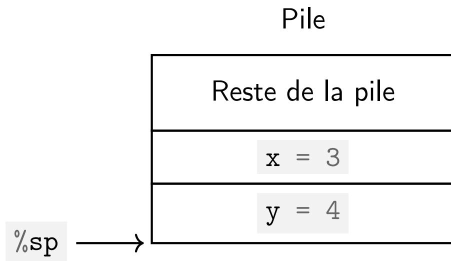
Exemple

```
1  int main()
2  {
3    int x = 3;
4    while(x > 0)
5    {
6      int y = x+1;
7      if(y > 2)
8      {
9        int z = y+2; ←
10     }
11     x--;
12   }
13 }
```



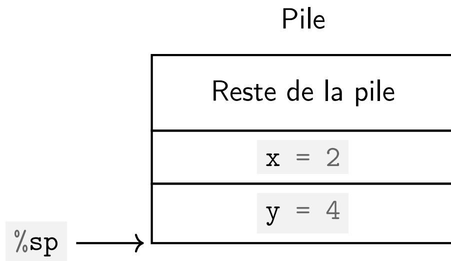
Exemple

```
1  int main()
2  {
3    int x = 3;
4    while(x > 0)
5    {
6      int y = x+1;
7      if(y > 2)
8      {
9        int z = y+2;
10     }              ←
11     x--;
12   }
13 }
```



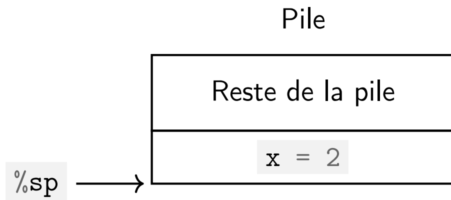
Exemple

```
1   int main()
2   {
3     int x = 3;
4     while(x > 0)
5     {
6       int y = x+1;
7       if(y > 2)
8       {
9         int z = y+2;
10      }
11      x--;            ←
12    }
13  }
```



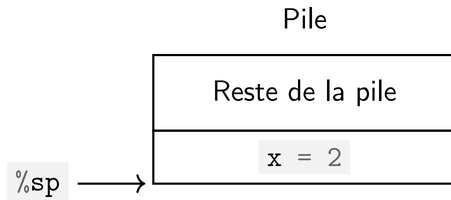
Exemple

```
1  int main()
2  {
3    int x = 3;
4    while(x > 0)
5    {
6      int y = x+1;
7      if(y > 2)
8      {
9        int z = y+2;
10     }
11     x--;
12   }                    ←
13 }
```



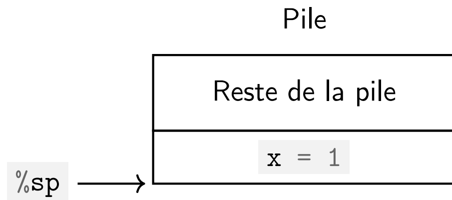
Exemple

```
1  int main()
2  {
3    int x = 3;
4    while(x > 0)      ←
5    {
6      int y = x+1;
7      if(y > 2)
8      {
9        int z = y+2;
10     }
11     x--;
12   }
13 }
```



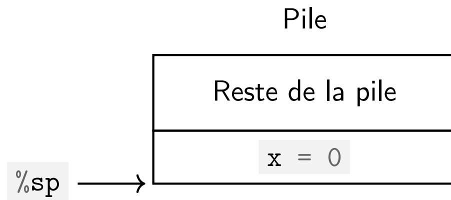
Exemple

```
1  int main()
2  {
3    int x = 3;
4    while(x > 0)      ←
5    {
6      int y = x+1;
7      if(y > 2)
8      {
9        int z = y+2;
10     }
11     x--;
12   }
13 }
```



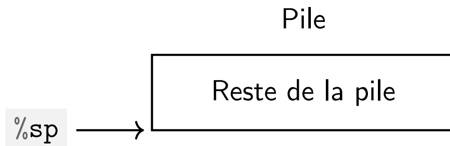
Exemple

```
1   int main()
2   {
3     int x = 3;
4     while(x > 0)       ←
5     {
6       int y = x+1;
7       if(y > 2)
8       {
9         int z = y+2;
10      }
11      x--;
12    }
13  }
```



Exemple

```c
int main()
{
  int x = 3;
  while(x > 0)
  {
    int y = x+1;
    if(y > 2)
    {
      int z = y+2;
    }
    x--;
  }
}                  ←
```



Appels de fonctions

Quand on retourne d'une fonction, comment connait-on l'adresse où l'on doit continuer ?

```
int f(int x)
{
  return x+1;
}
int main()
{
  printf("Premier:  %d\n",f(4));
  printf("Deuxième: %d\n",f(4));
}
```

Appels de fonctions

Quand on retourne d'une fonction, comment connait-on l'adresse où l'on doit continuer ?

```
int f(int x)
{
  return x+1;
}
int main()
{
  printf("Premier:  %d\n",f(4));
  printf("Deuxième: %d\n",f(4));
}
```

Appels de fonctions

Quand on retourne d'une fonction, comment connait-on l'adresse où l'on doit continuer ?

```
int f(int x)
{
  return x+1;                    ←
}
int main()
{
  printf("Premier:  %d\n",f(4));
  printf("Deuxième: %d\n",f(4));
}
```

Appels de fonctions

Quand on retourne d'une fonction, comment connait-on l'adresse où l'on doit continuer ?

```
int f(int x)
{
  return x+1;
}
int main()
{
  printf("Premier:  %d\n",f(4));  ← ??
  printf("Deuxième: %d\n",f(4));
}
```

Appels de fonctions

Quand on retourne d'une fonction, comment connait-on l'adresse où l'on doit continuer ?

```
int f(int x)
{
  return x+1;
}
int main()
{
  printf("Premier:  %d\n",f(4));
  printf("Deuxième: %d\n",f(4));  ← ??
}
```

Appels de fonctions

Quand on retourne d'une fonction, comment connait-on l'adresse où l'on doit continuer ?

```
int f(int x)
{
  return x+1;
}
int main()
{
  printf("Premier:  %d\n",f(4));
  printf("Deuxième: %d\n",f(4));
}
```

Aussi : où stocker le résultat de la fonction ?

Pile et appels

Juste avant qu'une fonction soit appelée, il se passe trois choses :

- de l'espace est réservé sur la pile pour le type de retour
- les arguments de la fonction sont empilés sur la pile
- une **adresse de retour** sur la pile pour savoir **où continuer** quand on `return`.

Pile et appels

Juste avant qu'une fonction soit appelée, il se passe trois choses :

- de l'espace est réservé sur la pile pour le type de retour
- les arguments de la fonction sont empilés sur la pile
- une **adresse de retour** sur la pile pour savoir **où continuer** quand on `return`.

Le code de la fonction s'exécute alors.

Pile et appels

Juste avant qu'une fonction soit appelée, il se passe trois choses :

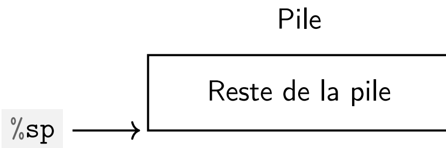
- de l'espace est réservé sur la pile pour le type de retour
- les arguments de la fonction sont empilés sur la pile
- une **adresse de retour** sur la pile pour savoir **où continuer** quand on `return`.

Le code de la fonction s'exécute alors.

À la fin de la fonction, on sait où placer le résultat, et où doit continuer l'exécution.

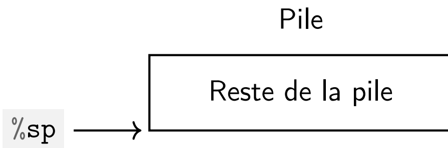
Exemple

```c
int f(int x, int y)
{
  printf("Bonjour!!!");
  return x + y;
}
int main()
{
  int z = f(3,4);
  printf("Valeur de z: %d",z);
}
```



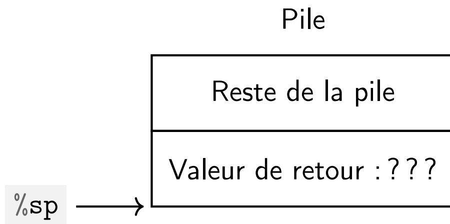
Exemple

```
1   int f(int x, int y)
2   {
3     printf("Bonjour!!!");
4     return x + y;
5   }
6   int main()
7   {
8     int z = f(3,4);                  ←
9     printf("Valeur de z: %d",z);
10  }
```



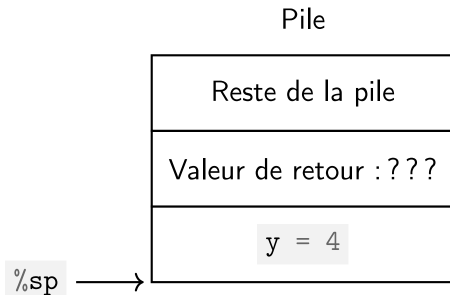
Exemple

```
1  int f(int x, int y)
2  {
3    printf("Bonjour!!!");
4    return x + y;
5  }
6  int main()
7  {
8    int z = f(3,4);                    ←
9    printf("Valeur de z: %d",z);
10 }
```



Exemple

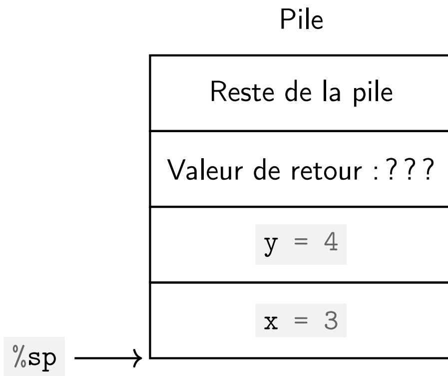
```c
int f(int x, int y)
{
  printf("Bonjour!!!");
  return x + y;
}
int main()
{
  int z = f(3,4);
  printf("Valeur de z: %d",z);
}
```



On empile les arguments dans l'ordre inverse.

Exemple

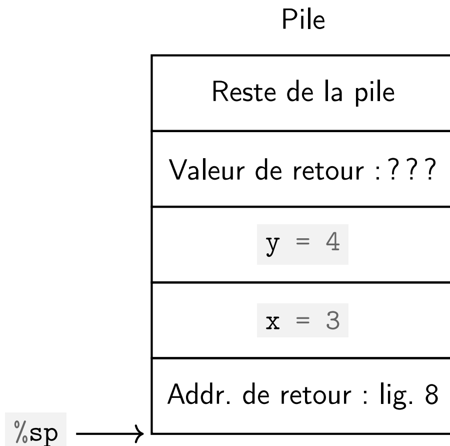
```
1  int f(int x, int y)
2  {
3    printf("Bonjour!!!");
4    return x + y;
5  }
6  int main()
7  {
8    int z = f(3,4);                    ←
9    printf("Valeur de z: %d",z);
10 }
```



On empile les arguments dans l'ordre inverse.

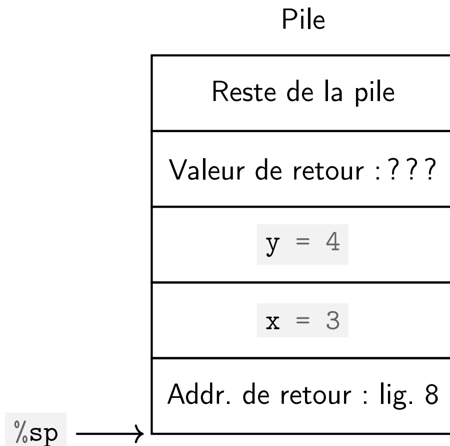
Exemple

```c
int f(int x, int y)
{
  printf("Bonjour!!!");
  return x + y;
}
int main()
{
  int z = f(3,4);                      ←
  printf("Valeur de z: %d",z);
}
```



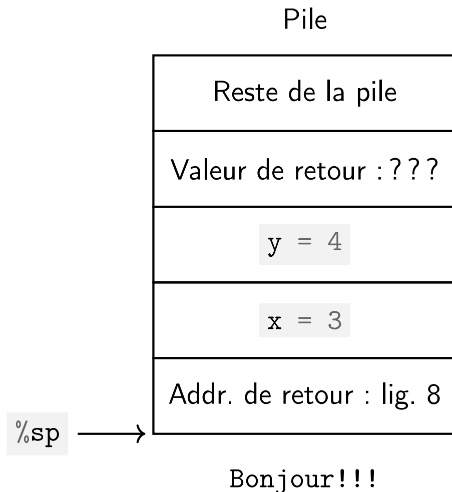
Exemple

```
1  int f(int x, int y)          ←
2  {
3    printf("Bonjour!!!");
4    return x + y;
5  }
6  int main()
7  {
8    int z = f(3,4);
9    printf("Valeur de z: %d",z);
10 }
```



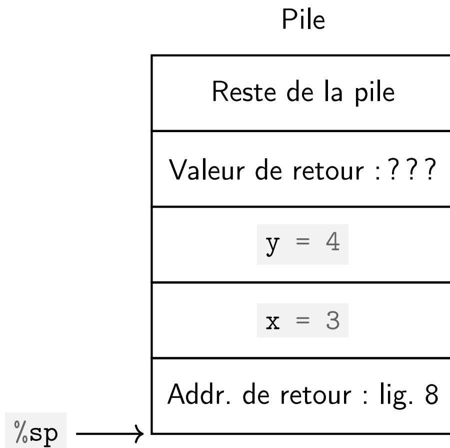
Exemple

```
1  int f(int x, int y)
2  {
3    printf("Bonjour!!!");        ←
4    return x + y;
5  }
6  int main()
7  {
8    int z = f(3,4);
9    printf("Valeur de z: %d",z);
10 }
```



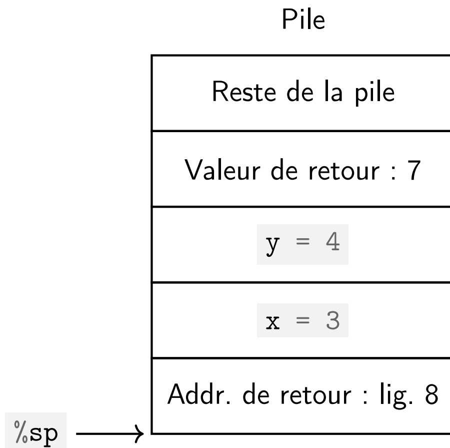
Exemple

```
1  int f(int x, int y)
2  {
3    printf("Bonjour!!!");
4    return x + y;                ←
5  }
6  int main()
7  {
8    int z = f(3,4);
9    printf("Valeur de z: %d",z);
10 }
```



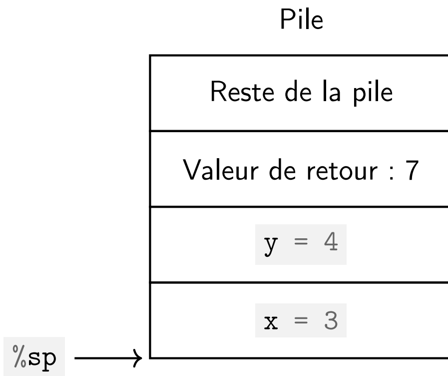
Exemple

```
1  int f(int x, int y)
2  {
3    printf("Bonjour!!!");
4    return x + y;                      ←
5  }
6  int main()
7  {
8    int z = f(3,4);
9    printf("Valeur de z: %d",z);
10 }
```



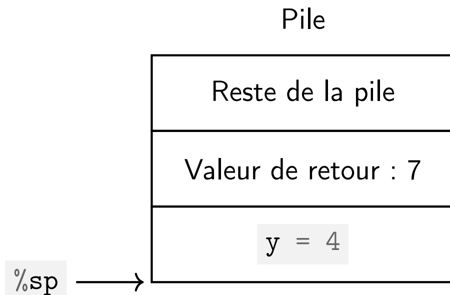
Exemple

```
int f(int x, int y)
{
  printf("Bonjour!!!");
  return x + y;
}
int main()
{
  int z = f(3,4);      ←
  printf("Valeur de z: %d",z);
}
```



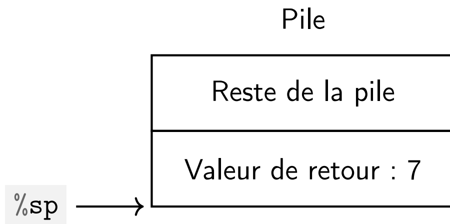
Exemple

```c
1   int f(int x, int y)
2   {
3     printf("Bonjour!!!");
4     return x + y;
5   }
6   int main()
7   {
8     int z = f(3,4);          ←
9     printf("Valeur de z: %d",z);
10  }
```



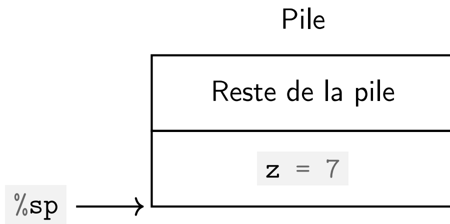
Exemple

```
1  int f(int x, int y)
2  {
3    printf("Bonjour!!!");
4    return x + y;
5  }
6  int main()
7  {
8    int z = f(3,4);          ←
9    printf("Valeur de z: %d",z);
10 }
```



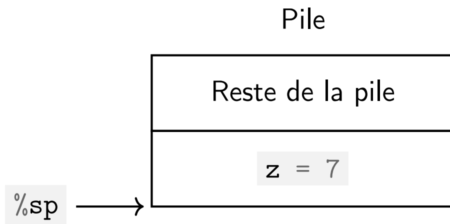
Exemple

```
1  int f(int x, int y)
2  {
3    printf("Bonjour!!!");
4    return x + y;
5  }
6  int main()
7  {
8    int z = f(3,4);        ←
9    printf("Valeur de z: %d",z);
10 }
```



Exemple

```c
1  int f(int x, int y)
2  {
3    printf("Bonjour!!!");
4    return x + y;
5  }
6  int main()
7  {
8    int z = f(3,4);
9    printf("Valeur de z: %d",z); ←
10 }
```




```
Valeur de z: 7
```

Sommaire

La zone du code

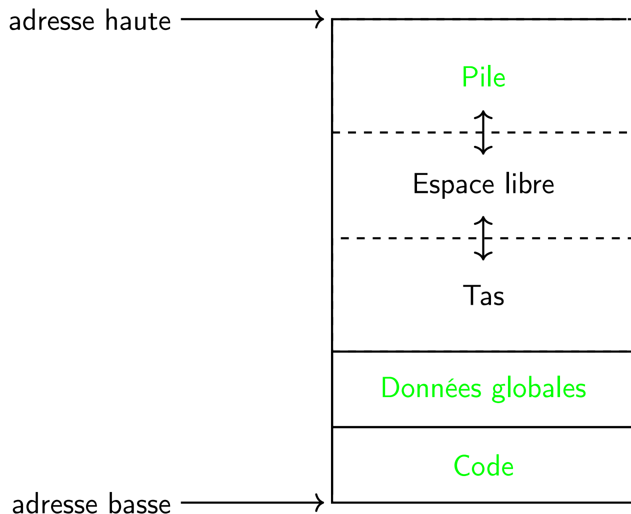
Données globales

La pile

Le tas

Les erreurs mémoire

Retour au schéma



Données de taille variable

Nous avons vu les **variables globales** qui permettent de stocker un ensemble de données de taille fixe. Exemple :

```
int x,y;
char z[100];
int f()
{
  // x,y,z utilisables
}
int main()
{
  // x,y,z utilisables
}
```

Ici, on a accès à $2 \times 4 + 100 = 108$ octets de façon globale dans le programme.

Données de taille variable

Les variables globales ne conviennent pas si l'on souhaite manipuler des données de taille variable.

```
int n;
int tab[100];
int main()
{
  scanf("%d",&n);
  for(int i = 0; i < n; i++)
  {
    scanf("%d",&tab[i]);
  }
  // action sur tab...
}
```

- si `n` < 100, alors `tab` gaspille $100 -$ `n` octets
- si `n` > 100, alors on ne sait pas traiter l'entrée.

Données de taille variable

À la place, on peut s'en sortir avec des **variables locales allouées sur la pile**.

```
int main()
{
  int n;
  scanf("%d",&n);
  int tab[n]; // la taille de tab dépend de n
              // qui vient juste d'être rentré
  for(int i = 0; i < n; i++)
  {
    scanf("%d",&tab[i]);
  }
  // action sur tab...
}
```

Données de taille variable

À la place, on peut s'en sortir avec des **variables locales allouées sur la pile**.

```
int main()
{
  int n;
  scanf("%d",&n);
  int tab[n]; // la taille de tab dépend de n
              // qui vient juste d'être rentré
  for(int i = 0; i < n; i++)
  {
    scanf("%d",&tab[i]);
  }
  // action sur tab...
}
```

Cependant, une fois que `tab` est déclaré avec sa taille, il n'est plus possible d'en modifier la taille !

Données de taille variable

Par exemple, on ne peut pas imaginer un programme qui proposerait de changer la taille des données une fois celles-ci rentrées :

```
int main()
{
  int n;
  scanf("%d",&n);
  int tab[n];
  // ...
  printf("Pour ajuster les données, entrer un nouveau nombre de données: ");
  int m;
  scanf("%d",&m);
  // on est bloqué ici, car la taille de tab ne peut plus être modifiée
}
```

Données de taille variable

Aussi, une variable locale sur la pile ne survit pas au bloc qui la contient :

```
int f()
{
  int n;
  scanf("%d",&n);
  int tab[n];
  // ... (opérations sur tab)
} // <- tab sera détruit ici
```

Dans cet exemple, toutes les opérations sur `tab` doivent se faire avant l'accolade finale, qui délimite le bloc où existe `tab`.

Solution

Ainsi, les données globales et allouées sur la pile ne permettent pas

- d'avoir des données dont la taille peut varier avec l'exécution,
- d'avoir des données qui survivent au bloc qui les a créées.

Solution

Ainsi, les données globales et allouées sur la pile ne permettent pas

- d'avoir des données dont la taille peut varier avec l'exécution,
- d'avoir des données qui survivent au bloc qui les a créées.

Pour manier des données avec des deux propriétés, on utilise la dernière zone mémoire à évoquer : le **tas**.

Le tas

Le **tas** est une zone mémoire qui peut stocker des données

- de taille non connue à l'avance
- qui ne disparaissent pas quand on quitte le bloc qui les a créées.

Le tas

Le **tas** est une zone mémoire qui peut stocker des données

- de taille non connue à l'avance
- qui ne disparaissent pas quand on quitte le bloc qui les a créées.

Par défaut, l'accès aux octets du tas est **interdit**.

Le tas

Le **tas** est une zone mémoire qui peut stocker des données

- de taille non connue à l'avance
- qui ne disparaissent pas quand on quitte le bloc qui les a créées.

Par défaut, l'accès aux octets du tas est **interdit**.

Pour y avoir accès, il faut en faire la requête au système d'exploitation par la fonction `malloc` (`#include <stdlib.h>`).

Le tas

Le **tas** est une zone mémoire qui peut stocker des données

- de taille non connue à l'avance
- qui ne disparaissent pas quand on quitte le bloc qui les a créées.

Par défaut, l'accès aux octets du tas est **interdit**.

Pour y avoir accès, il faut en faire la requête au système d'exploitation par la fonction `malloc` (`#include <stdlib.h>`).

Une fois que l'on a fini d'utiliser la zone mémoire donnée par `malloc`, il faut la libérer avec la fonction `free`.

Rappels sur les pointeurs

Les fonctions `malloc` et `free` travaillent avec des **adresses mémoire**.

Rappels sur les pointeurs

Les fonctions `malloc` et `free` travaillent avec des **adresses mémoire**.

En `C`, ce sont les **pointeurs** qui permettent des adresses.

Rappels sur les pointeurs

Les fonctions `malloc` et `free` travaillent avec des **adresses mémoire**.

En C, ce sont les **pointeurs** qui permettent des adresses.

Le type des pointeurs pour le type `t` s'obtient en ajoutant une `*` devant `t` : `t*`.

Rappels sur les pointeurs

Les fonctions `malloc` et `free` travaillent avec des **adresses mémoire**.

En `C`, ce sont les **pointeurs** qui permettent des adresses.

Le type des pointeurs pour le type `t` s'obtient en ajoutant une `*` devant `t` : `t*`.

Exemples :

`char` : type des caractères	⇝	`char*` : type des pointeurs sur `char`
`int` : type des entiers 32-bits	⇝	`int*` : type des pointeurs sur `int`

etc.

Rappels sur les pointeurs

Étant donnée une variable `x` de type `t`, pour avoir son adresse, on utilise l'opérateur `&`. On obtient alors une valeur de type `t*`.

Rappels sur les pointeurs

Étant donnée une variable `x` de type `t`, pour avoir son adresse, on utilise l'opérateur `&`. On obtient alors une valeur de type `t*`.

Exemple :

```
int main()
{
  int x = 3;
  int* ptr = &x; // on stocke l'adresse de x dans ptr
  // ...
  return 0;
}
```

Rappels sur les pointeurs

Étant donnée un pointeur `ptr` de type `t*`, on utilise l'opérateur `*` pour accéder aux données contenues à cette adresse. On obtient alors une valeur de type `t`.

Rappels sur les pointeurs

Étant donnée un pointeur `ptr` de type `t*`, on utilise l'opérateur `*` pour accéder aux données contenues à cette adresse. On obtient alors une valeur de type `t`.

Exemple :

```c
int main()
{
  int *ptr = ...; // on obtient d'une façon quelconque un pointeur ici
  int x = *ptr;  // accès au contenu à l'adresse par l'opérateur *
}
```

La fonction `malloc`

La fonction `malloc` prend un argument, qui est le nombre d'octets souhaités.

```
malloc(4); // demande 4 octets
malloc(53); // demande 53 octets
```

La fonction `malloc`

La fonction `malloc` prend un argument, qui est le nombre d'octets souhaités.

```
malloc(4); // demande 4 octets
malloc(53); // demande 53 octets
```

Le plus souvent, on veut un nombre d'octets qui correspond à la taille d'un type ou d'une structure. Cette taille est donnée par `sizeof`.

```
malloc(sizeof(int));           // demande le nombre d'octets d'un int
malloc(sizeof(struct point)); // demande le nombre d'octets
                               // de la structure point
```

La fonction `malloc`

La fonction `malloc` prend un argument, qui est le nombre d'octets souhaités.

```
malloc(4); // demande 4 octets
malloc(53); // demande 53 octets
```

Le plus souvent, on veut un nombre d'octets qui correspond à la taille d'un type ou d'une structure. Cette taille est donnée par `sizeof`.

```
malloc(sizeof(int));          // demande le nombre d'octets d'un int
malloc(sizeof(struct point)); // demande le nombre d'octets
                              // de la structure point
```

Ainsi, le plus souvent, on écrira des appels à `malloc` de la forme

```
t* ptr = malloc(sizeof(t));
```

où `t` est un type.

La fonction `malloc`

Si l'allocation n'a pas pu se faire, `malloc` renvoie `NULL`. C'est une erreur plutôt improbable si l'espace disponible dans la mémoire vive est élevé.

```c
int* ptr = malloc(sizeof(int));
if(ptr == NULL)
{
  // traitement d'une erreur d'allocation
}
```

La fonction `malloc`

Si l'allocation n'a pas pu se faire, `malloc` renvoie `NULL`. C'est une erreur plutôt improbable si l'espace disponible dans la mémoire vive est élevé.

```
int* ptr = malloc(sizeof(int));
if(ptr == NULL)
{
  // traitement d'une erreur d'allocation
}
```

S'il n'y a pas eu d'erreur, on peut utiliser l'espace affecté en utilisant l'opérateur `*` :

```
*ptr = 3; // on affecte 3 à l'int à l'adresse ptr
```

La fonction `malloc`

Si l'allocation n'a pas pu se faire, `malloc` renvoie `NULL`. C'est une erreur plutôt improbable si l'espace disponible dans la mémoire vive est élevé.

```
int* ptr = malloc(sizeof(int));
if(ptr == NULL)
{
  // traitement d'une erreur d'allocation
}
```

S'il n'y a pas eu d'erreur, on peut utiliser l'espace affecté en utilisant l'opérateur `*` :

```
*ptr = 3; // on affecte 3 à l'int à l'adresse ptr
```

Bien distinguer avec les affectations sans `*` :

```
ptr = 3; // l'adresse stockée dans ptr est maintenant 3
         // l'ancienne adresse est perdue
```

La fonction `free`

Une fois que l'on a terminé d'utiliser la mémoire allouée par `malloc`, on la libère en appelant `free` :

```c
char* ptr = malloc(sizeof(char));
// ...
free(ptr);
```

Si on ne le fait pas, les zones mémoires inutilisées vont continuer de prendre de la place dans le système : on parle de **fuites de mémoire**.

Les fuites de mémoire induisent une consommation excessive de RAM et peuvent aller jusqu'à bloquer le système.

Exemple

```c
int main()
{
  int* ptr;
  while(1)
  {
    ptr = malloc(sizeof(int));
  }
  return 0;
}
```

- à chaque tour, 4 octets sont alloués sans être libérés ;
- cela va progressivement saturer la RAM et bloquer le système.

Rappels : pointeurs et structures

Avec les structures, l'utilisation de `*` nécessite des parenthèses pour accéder aux attributs.

```
typedef struct
{
  int x,y;
} point;
int main()
{
  point* ptr = malloc(sizeof(point));
  (*ptr).x = 3;
  (*ptr).y = 4;
  return 0;
}
```

C'est un peu lourd.

Rappels : pointeurs et structures

À la place, on dispose d'un opérateur `->` qui permet d'écrire les choses de façon plus concise.

```c
typedef struct
{
  int x,y;
} point;
int main()
{
  point* ptr = malloc(sizeof(point));
  ptr->x = 3; // équivalent à: (*ptr).x = 3;
  ptr->y = 4; // équivalent à: (*ptr).y = 4;
  return 0;
}
```

Tableaux dynamiques

Comme `malloc` alloue des octets consécutifs, on peut l'utiliser pour créer des tableaux dynamiques.

Pour cela, il suffit d'allouer un **multiple** de la taille de la structure considérée.

```
typedef struct { int x,y; } point;

int *ptr_int   = malloc(5 * sizeof(int));   // un tableau de 5 int
int *ptr_point = malloc(7 * sizeof(point)); // un tableau de 7 point
```

On peut alors accéder aux cases comme pour un tableau statique : `ptr_int[3]`, `ptr_point[2]`, *etc.*

Rappel : arithmétique des pointeurs

En fait, la syntaxe `ptr[n]` est du **sucre syntaxique** pour l'expression `*(ptr + n)`.

Rappel : arithmétique des pointeurs

En fait, la syntaxe `ptr[n]` est du **sucre syntaxique** pour l'expression `*(ptr + n)`.

```
int *ptr = malloc(3 * sizeof(int));
ptr[2]   = 42;
```

Ici, on a `ptr[2]` qui est équivalent à `*(ptr + 2)`.

Rappel : arithmétique des pointeurs

En fait, la syntaxe `ptr[n]` est du **sucre syntaxique** pour l'expression `*(ptr + n)`.

```
int *ptr = malloc(3 * sizeof(int));
ptr[2]   = 42;
```

Ici, on a `ptr[2]` qui est équivalent à `*(ptr + 2)`.

Pour permettre la bonne indexation, les expressions `ptr + n` sont interprétées de façon spéciale : `ptr + 2` **n'est pas**

l'adresse du 2ème octet après `ptr`

mais plutôt

l'adresse du (`2 * sizeof(int)`)-ème octet après `ptr`.

Exemple

```c
int *ptr = malloc(3 * sizeof(int));
```

Exemple

```c
int *ptr = malloc(3 * sizeof(int));
```

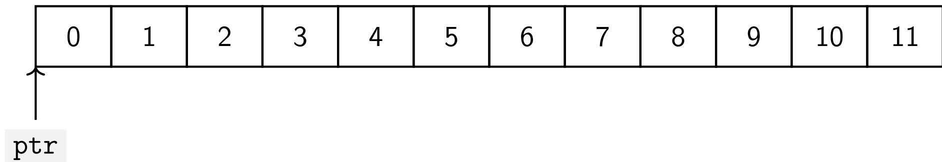
Rappel : `sizeof(int)` est égal à 4. Donc 12 octets alloués.

Exemple

```c
int *ptr = malloc(3 * sizeof(int));
```

Rappel : `sizeof(int)` est égal à 4. Donc 12 octets alloués.

Dans le tas :

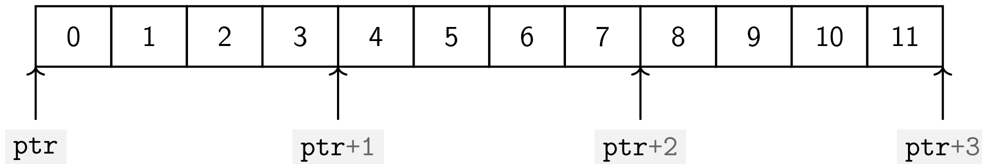


Exemple

```
int *ptr = malloc(3 * sizeof(int));
```

Rappel : `sizeof(int)` est égal à 4. Donc 12 octets alloués.

Dans le tas :

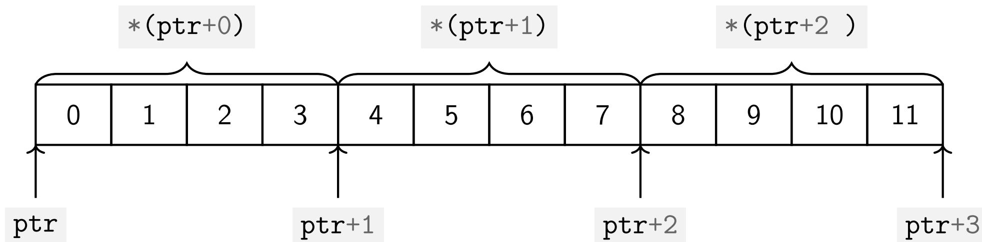


Exemple

```c
int *ptr = malloc(3 * sizeof(int));
```

Rappel : `sizeof(int)` est égal à 4. Donc 12 octets alloués.

Dans le tas :

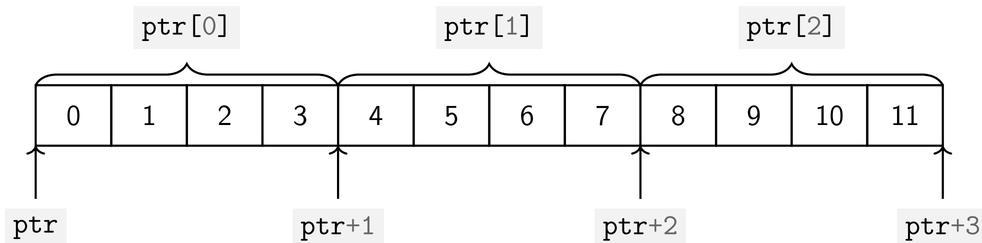


Exemple

```
int *ptr = malloc(3 * sizeof(int));
```

Rappel : `sizeof(int)` est égal à 4. Donc 12 octets alloués.

Dans le tas :



Quelques variantes : `calloc`

Pour utiliser des tableaux dynamiques, on fait souvent la même chose :

- on alloue avec `malloc` un multiple du `sizeof` du type
- on initialise les données à 0 avec une boucle ou `memset` (`#include <stdlib.h>`).

Quelques variantes : `calloc`

Pour utiliser des tableaux dynamiques, on fait souvent la même chose :

- on alloue avec `malloc` un multiple du `sizeof` du type
- on initialise les données à 0 avec une boucle ou `memset` (`#include <stdlib.h>`).

```
int* ptr = malloc(42 * sizeof(int));
for(int i = 0; i < 42; i++)
{
  ptr[i] = 0;
}
```

Quelques variantes : `calloc`

Pour utiliser des tableaux dynamiques, on fait souvent la même chose :

- on alloue avec `malloc` un multiple du `sizeof` du type
- on initialise les données à 0 avec une boucle ou `memset` (`#include <stdlib.h>`).

```
int* ptr = malloc(42 * sizeof(int));
memset(ptr, 0, 42 * sizeof(int));
```

Quelques variantes : `calloc`

Pour utiliser des tableaux dynamiques, on fait souvent la même chose :

- on alloue avec `malloc` un multiple du `sizeof` du type
- on initialise les données à 0 avec une boucle ou `memset` (`#include <stdlib.h>`).

```
int* ptr = malloc(42 * sizeof(int));
memset(ptr, 0, 42 * sizeof(int));
```

Ce n'est pas beaucoup de choses à écrire, mais on peut se tromper facilement.

Quelques variantes : `calloc`

Pour utiliser des tableaux dynamiques, on fait souvent la même chose :

- on alloue avec `malloc` un multiple du `sizeof` du type
- on initialise les données à 0 avec une boucle ou `memset` (`#include <stdlib.h>`).

```
int* ptr = malloc(42 * sizeof(int));
memset(ptr, 0, 42 * sizeof(int));
```

Ce n'est pas beaucoup de choses à écrire, mais on peut se tromper facilement.

À la place, on peut utiliser `calloc`, qui alloue un tableau d'éléments à la bonne taille et initialise à 0. On peut ainsi écrire simplement :

```
int* ptr = calloc(42, sizeof(int));
```

Quelques variantes : `realloc`

Quand on manipule des tableaux dynamiques, on veut souvent changer la taille du tableau en ajoutant ou supprimant des éléments.

Quelques variantes : `realloc`

Quand on manipule des tableaux dynamiques, on veut souvent changer la taille du tableau en ajoutant ou supprimant des éléments.

Pour cela, il faut :

- allouer un nouveau tableau avec la nouvelle taille par `malloc`,
- recopier l'ancien tableau dans le nouveau,
- libérer l'ancien avec `free`.

Quelques variantes : `realloc`

Quand on manipule des tableaux dynamiques, on veut souvent changer la taille du tableau en ajoutant ou supprimant des éléments.

```
point* ptr = malloc(42 * sizeof(point));
f(taille,ptr); // on utilise le tableau de taille 42
// on veut maintenant faire passer la taille à 21
// en conservant les 21 premières données
point* new_ptr = malloc(21 * sizeof(point));
for(int i = 0; i < 21; i++)
{
  new_ptr[i] = ptr[i]; // on recopie les anciennes données
}
free(ptr); // on libère l'ancien tableau
ptr = new_ptr; // on remplace le pointeur par la nouvelle adresse
```

C'est assez long et on peut facilement se tromper.

Quelques variantes : `realloc`

Quand on manipule des tableaux dynamiques, on veut souvent changer la taille du tableau en ajoutant ou supprimant des éléments.

`realloc` permet de faire toutes ces opérations simplement :

```
point* ptr = malloc(42 * sizeof(point));
f(taille,ptr); // on utilise le tableau de taille 42
// on veut maintenant faire passer la taille à 21
// en conservant les 21 premières données
ptr = realloc(ptr, 21 * sizeof(point));
// ^ tableau redimensionné à 21 et données recopiées
```

Sommaire

La zone du code

Données globales

La pile

Le tas

Les erreurs mémoire

Erreurs mémoire

Un programme `C` peut rencontrer différentes erreurs liées à la mémoire à l'exécution :

- l'**erreur de segmentation** (*segmentation fault*)
- le **débordement de pile** (*stack overflow*)
- la **corruption mémoire** (*memory corruption*)
- le **dépassement de tampon** (*stack buffer overflow*)

Erreur de segmentation

On peut avoir une erreur de segmentation quand on accède une adresse mémoire non autorisée.

```c
int *ptr = NULL;
*ptr = 0;
```

```
Erreur de segmentation (core dumped)
```

Erreur de segmentation

On peut avoir une erreur de segmentation quand on accède une adresse mémoire non autorisée.

```
int *ptr = malloc(30 * sizeof(int));
ptr[100000] = 3;
```

```
Erreur de segmentation (core dumped)
```

Erreur de segmentation

On peut avoir une erreur de segmentation quand on accède une adresse mémoire non autorisée.

Mais ce n'est pas toujours le cas !

```c
int *ptr = malloc(30 * sizeof(int));
ptr[30] = 3; // accès INVALIDE normalement
// mais probablement pas d'erreur
```

```
Le programme s'est terminé correctement.
```

Débordement de pile

La pile a une limite de taille relativement petite.

Comme les appels de fonctions consomment de l'espace sur la pile, on peut dépasser cette limite facilement avec des appels récursifs.

```
int f(int x)
{
  if(x == 0)
    return 0;
  return x + f(x-1);
}
// ...
f(10000000);
```

```
Erreur de segmentation (core dumped)
```

C'est une erreur de segmentation due à la pile.

Débordement de pile

La pile a une limite de taille relativement petite.

Comme les appels de fonctions consomment de l'espace sur la pile, on peut dépasser cette limite facilement avec des appels récursifs.

```
int f(int x)
{
  if(x == 0)
    return 0;
  return x + f(x-1);
}
// ...
f(10000000);
```

```
Erreur de segmentation (core dumped)
```

C'est une erreur de segmentation due à la pile. **Attention aux fonctions récursives !**

Corruption mémoire

En modifiant tableaux en dehors des index permis, on peut changer la valeur d'autres variables. On parle de **corruption mémoire**.

```c
char *ptr1 = malloc(10 * sizeof(char));
char *ptr2 = malloc(10 * sizeof(char));
memset(ptr2,'a',10);  // cases de ptr2 initialisées à 'a'
memset(ptr1,'b',100); // memset dangereux
printf("%c",ptr2[0]); // affiche 'b'
```

Ici, `ptr2[0]`, initialisé à `'a'`, a été modifié par le `memset` de `ptr1`.

Corruption mémoire

En modifiant tableaux en dehors des index permis, on peut changer la valeur d'autres variables. On parle de **corruption mémoire**.

Quand détectée, elle peut provoquer l'arrêt du programme.

```
int *ptr = malloc(30 * sizeof(int));
memset(ptr,0x77,1000); // memset dangereux mais qui s'exécute
free(ptr);             // programme arrêté ici
```

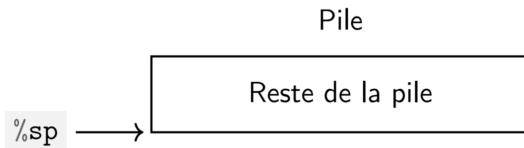
```
*** Error in `./prog': free(): invalid next size (fast): 0x126010 ***
======= Backtrace: =========
...
```

Ici, les variables internes qu'utilisent `malloc` et `free` ont été modifiées par le `memset`.

Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

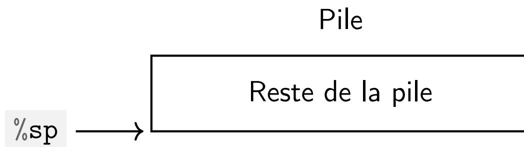
```
1  int f()
2  {
3    char tab[4];
4    memset(tab, 0, 30);
5    // ^ memset dangereux
6    printf("Vivant !\n");
7    return 1;
8  }
9  int main()                  ←
10 {
11   f();
12
13 }
```



Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

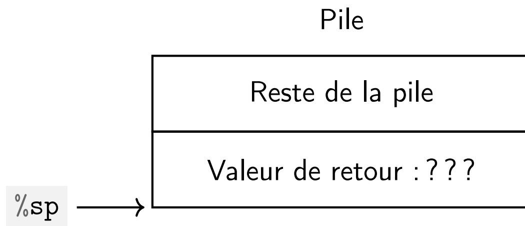
```
1  int f()
2  {
3    char tab[4];
4    memset(tab, 0, 30);
5    // ^ memset dangereux
6    printf("Vivant !\n");
7    return 1;
8  }
9  int main()
10 {
11   f();                      ←
12
13 }
```



Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

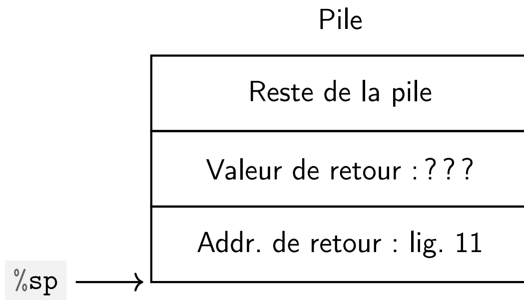
```c
1  int f()
2  {
3    char tab[4];
4    memset(tab, 0, 30);
5    // ^ memset dangereux
6    printf("Vivant !\n");
7    return 1;
8  }
9  int main()
10 {
11   f();                    ←
12
13 }
```



Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

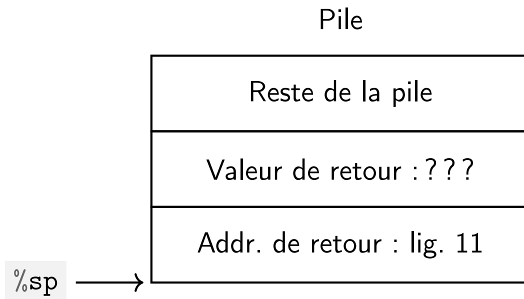
```
1   int f()
2   {
3     char tab[4];
4     memset(tab, 0, 30);
5     // ^ memset dangereux
6     printf("Vivant !\n");
7     return 1;
8   }
9   int main()
10  {
11    f();                    ←
12
13  }
```



Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

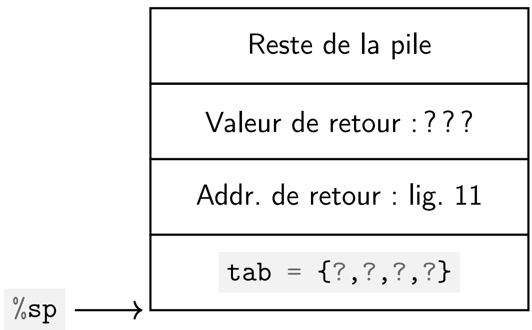
```c
int f()                          ←
{
  char tab[4];
  memset(tab, 0, 30);
  // ^ memset dangereux
  printf("Vivant !\n");
  return 1;
}
int main()
{
  f();

}
```



Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

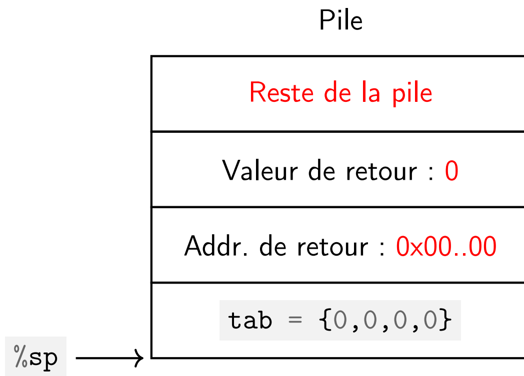
```c
int f()
{
  char tab[4];                 ←
  memset(tab, 0, 30);
  // ^ memset dangereux
  printf("Vivant !\n");
  return 1;
}
int main()
{
  f();

}
```



Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

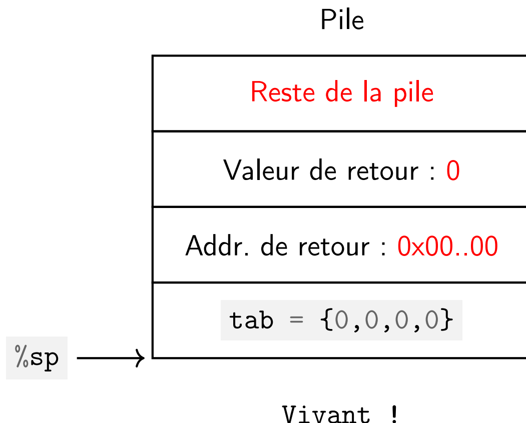
```c
int f()
{
  char tab[4];
  memset(tab, 0, 30);     ←
  // ^ memset dangereux
  printf("Vivant !\n");
  return 1;
}
int main()
{
  f();

}
```



Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

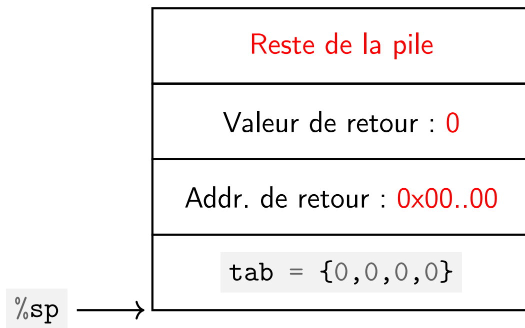
```c
int f()
{
  char tab[4];
  memset(tab, 0, 30);
  // ^ memset dangereux
  printf("Vivant !\n");   ←
  return 1;
}
int main()
{
  f();

}
```



Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

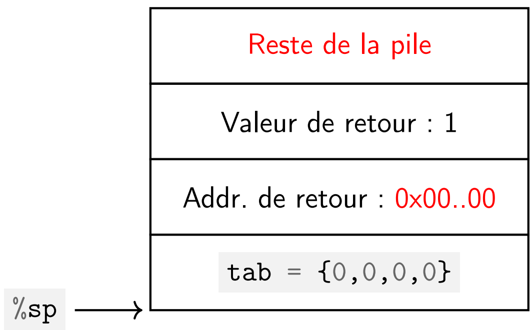
```
1  int f()
2  {
3    char tab[4];
4    memset(tab, 0, 30);
5    // ^ memset dangereux
6    printf("Vivant !\n");
7    return 1;                ←
8  }
9  int main()
10 {
11   f();
12
13 }
```



Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

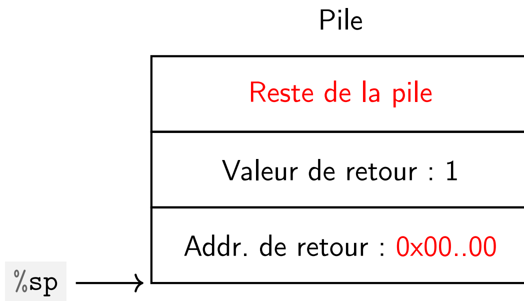
```c
int f()
{
  char tab[4];
  memset(tab, 0, 30);
  // ^ memset dangereux
  printf("Vivant !\n");
  return 1;            ←
}
int main()
{
  f();

}
```



Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

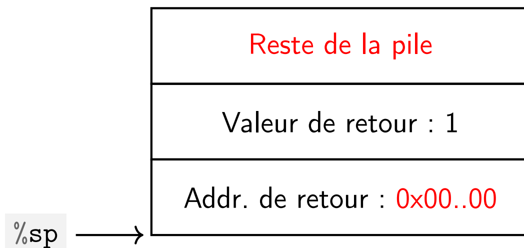
```c
int f()
{
  char tab[4];
  memset(tab, 0, 30);
  // ^ memset dangereux
  printf("Vivant !\n");
  return 1;          ←
}
int main()
{
  f();

}
```



Dépassement de tampon

En corrompant la mémoire de la pile, on peut altérer les adresses de retour des fonctions, ce qui arrête le programme quand détecté.

```c
int f()
{
  char tab[4];
  memset(tab, 0, 30);
  // ^ memset dangereux
  printf("Vivant !\n");
  return 1;
}
int main()
{
  f();

}
```




```
*** stack smashing detected ***:
./prog terminated
Abandon (core dumped)
```