Mémoire : La pile

La pile

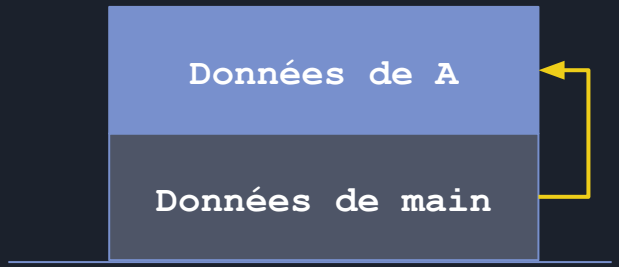
La **pile (stack)** est une **zone mémoire** où sont empilées les données nécessaires à l'exécution des **fonctions appelées**.

La pile

Au lancement programme, la fonction `main()` est exécutée. Les données nécessaires à l'exécution `main()` sont empilées dans une couche à la **racine de la pile**.

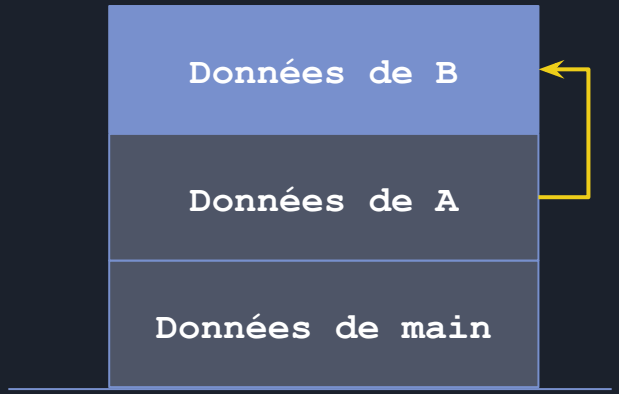
La pile

Si la fonction `main()` appelle une fonction `A()`, **une couche de données est ajoutée sur la pile** pour assurer l'exécution de `A()`.



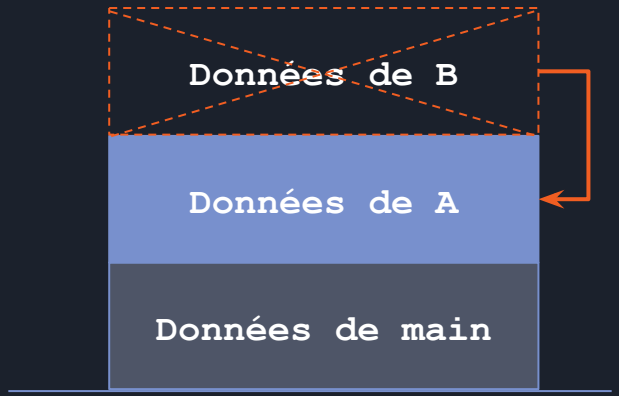
La pile

Si `A()` appelle la fonction `B()`, **une nouvelle couche s'empile** (et ainsi de suite).



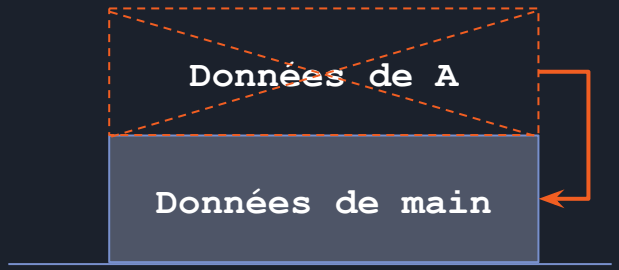
La pile

Dès qu'un appel de fonction se termine, la couche de données associée à cet appel **est supprimée de la pile**.



La pile

Et ainsi de suite.



Stack frame

Stack frame : couche de données empilé sur la pile pour un appel de fonction. Elle contient :

- La **position en code** de l'appel.
- Les **paramètres en entrée** de fonction.
- Les **variables locales**.

Stack frame

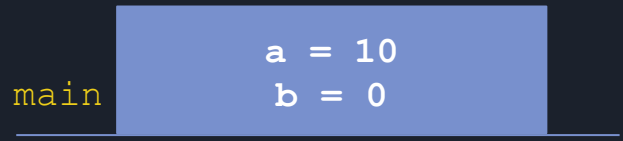
Considérons la fonction suivante :

```
void main() {
  int a = 10, b = 0;
  b = Increment(a);
}

int Increment(int val) {
  val++;
  return val;
}
```

Stack frame

La stack frame de `main()` est ajoutée à la pile :

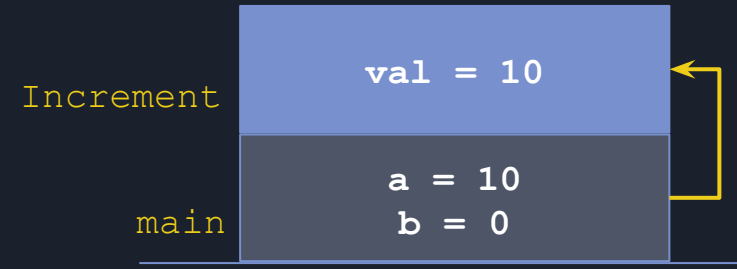
```
void main() {
  int a = 10, b = 0;
  b = Increment(a);
}

int Increment(int val) {
  val++;
  return val;
}
```



Stack frame

On **empile** la stack frame de `Increment()` :

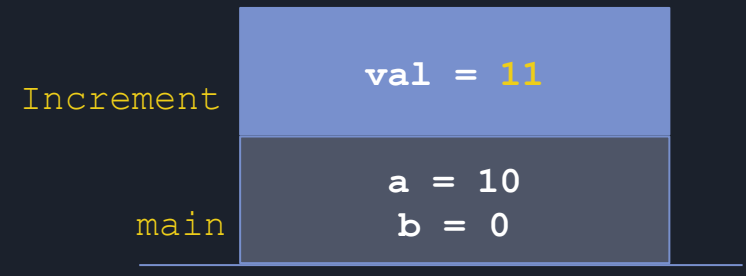
```
void main() {
  int a = 10, b = 0;
  b = Increment(a);
}

int Increment(int val) {
  val++;
  return val;
}
```



Stack frame

On modifie `val` dans la stack frame de `Increment()` :

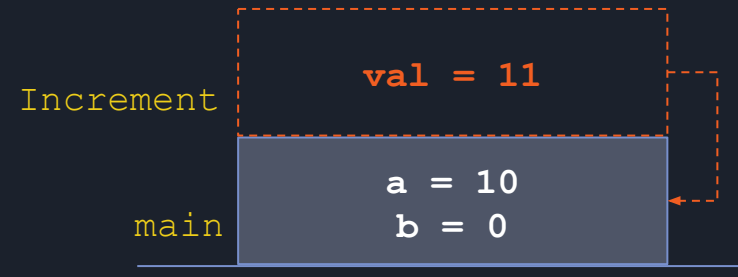
```
void main() {
  int a = 10, b = 0;
  b = Increment(a);
}

int Increment(int val) {
  val++;
  return val;
}
```



Stack frame

On **dépile** la frame de `Increment()` et on retourne la valeur de `val` dans la frame de `main()` :

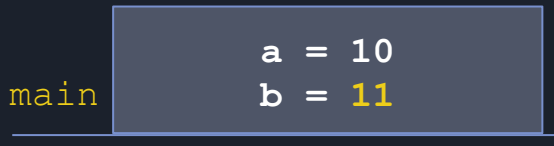
```
void main() {
  int a = 10, b = 0;
  b = Increment(a);
}

int Increment(int val) {
  val++;
  return val;
}
```



Stack frame

La **valeur de retour** de `Increment()` est **assignée** à `b` dans la frame de `main()` :

```
void main() {
  int a = 10, b = 0;
  b = Increment(a);
}

int Increment(int val) {
  val++;
  return val;
}
```



La pile

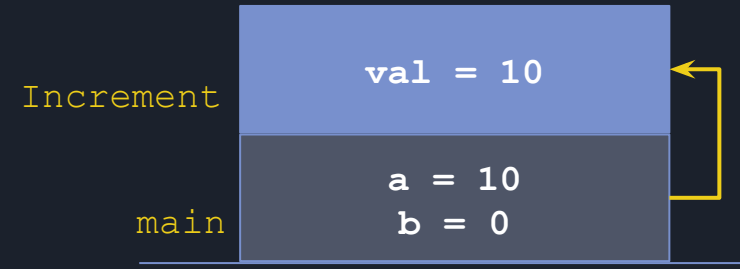
Lors d'un appel de fonction, les paramètres **copient** les valeurs des variables !

Ce sont **les valeurs, et non les variables**, qui sont passées en paramètres.

La pile

Ici, `a` et `val` sont **deux variables distinctes** contenant chacune la valeur `10`.

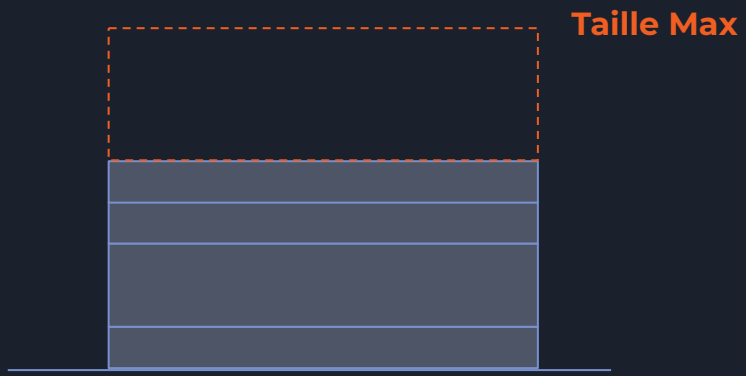
```
void main() {
  int a = 10, b = 0;
  b = Increment(a);
}

int Increment(int val) {
  val++;
  return val;
}
```



Taille de la pile

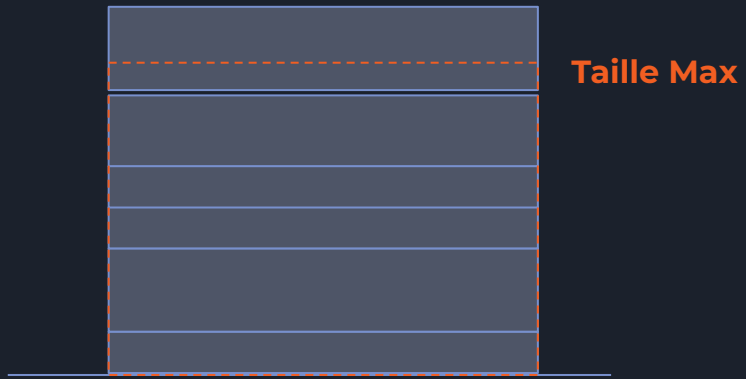
Les espaces mémoires alloués dans la pile sont contigus.

La pile a une **taille maximale** réservée.



Stack overflow

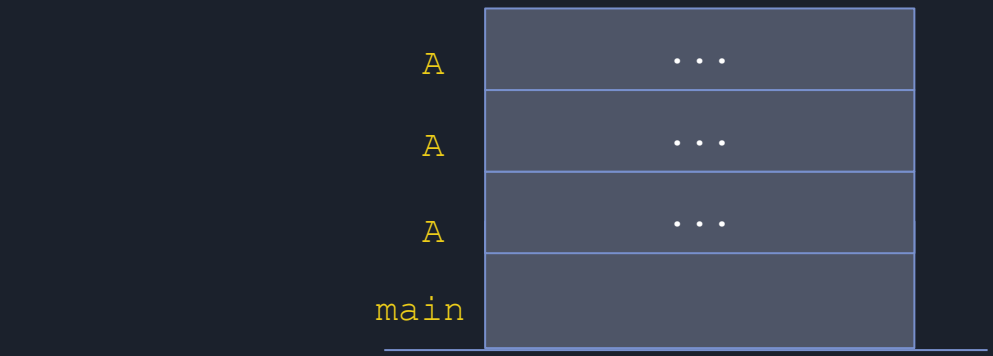
Si lors d'un appel de fonction, la taille maximale de la pile est dépassée, **le programme s'arrête** avec une erreur fatale. On parle de **stack overflow**.



Récursivité

Une fonction récursive est une fonction qui s'appelle elle-même.

```
void A(...) {
   [...]
   A(...)
   [...]
}
```



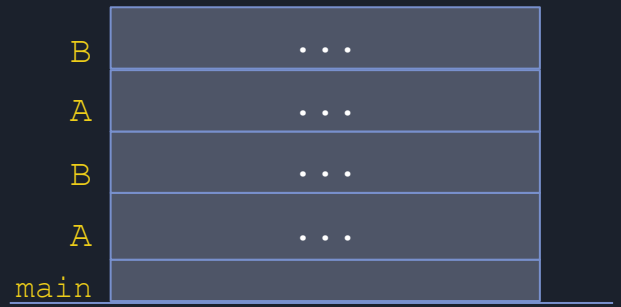
Récursivité

Ces deux fonctions produisent le même résultat :

```
int GetFactorial(int val) {
  int result = 1;
  for (int i = 2; i <= val; i++) {
    result *= i;
  }
  return result;
}

int GetFactorialRecursive( int val) {
  if (val > 1) {
    int result = GetFactorialRecursive(val - 1);
    return val * result;
  }
  return 1;
}
```

Récursivité indirecte

Un **algorithme récursif indirect** est composé d'un ensemble de fonctions qui s'appellent **en chaîne**.
Par exemple, `A()` appelle `B()` et `B()` appelle `A()`.



Récursivité

On cherchera à **limiter au maximum** la profondeur de récursivité des algorithmes.

Une fonction **récursive** :

- Peut toujours **être remplacée** par une boucle.
- Est **moins performante** qu'une boucle.
- Peut dans certains cas être **plus lisible ou plus pratique** qu'une boucle.

Récursivité

Pour remplacer une récursion “simples” par une boucle :

- Déterminer la **condition d’arrêt** de récursion,
- Déclarer une ou plusieurs **variables de résultats intermédiaires** au besoin,
- Déclarer une **variable de résultat final**,
- Implémenter une boucle correspondante mettant à jour l’ensemble des variables à chaque itération.

Certaines conversions nécessitent de garder l’ensemble les résultats intermédiaires.
Dans ce cas, on utilisera un tableau ou une collection.

Contraintes de la pile

La **durée de vie** d'une variable locale dépend de la **stack frame** dans laquelle elle existe.

Pour **échanger des données** entre stack frame, on multiplie les **copies de valeurs**.

La **taille des tableaux** dans la pile est **fixe**.

La **taille de la pile** est relativement **limitée**.

Il serait utile d'avoir une zone mémoire plus malléable que la pile ! Voir : Le tas !