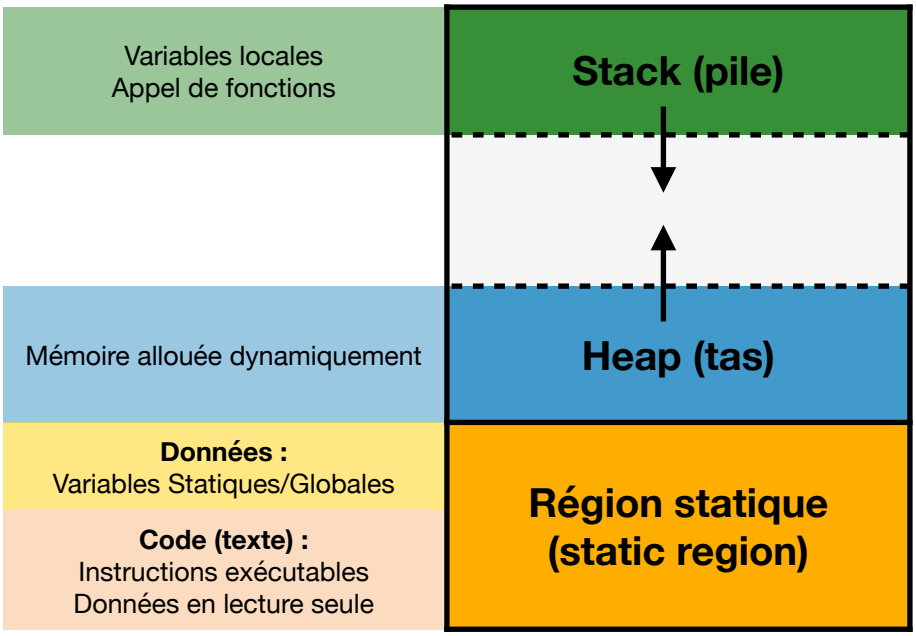
C++ (C) prend en charge la **gestion dynamique de la mémoire**, ce qui signifie que vous, en tant que programmeur, êtes responsable de l'**allocation** et de la **désallocation** de la mémoire.

La **gestion automatique de la mémoire** signifie que le langage de programmation **(Python, Java …)** automatise ce processus en effectuant l'**allocation** et la **désallocation** de la mémoire pour vous.

Notions de base sur le modèle de mémoire en C++
Chaque mot (ou bloc) de mémoire est généralement composé de deux, quatre ou huit octets, selon l'architecture matérielle. On peut faire référence à un bloc dans notre programme C++ en utilisant son adresse numérique. L'adresse du premier bloc est 0, tandis que l'adresse du dernier bloc dépend de la taille de la mémoire de votre ordinateur. La figure ci-dessous représente un bloc de mémoire.



En C++, la mémoire d'un programme est divisée (pour simplifier) en trois parties :

- **La région statique (static region)**, dans laquelle sont stockées les variables statiques et globales ainsi que les instructions du programme. Les variables statiques sont des variables qui restent utilisées tout au long de l'exécution d'un programme. La taille de la région statique ne change pas pendant l'exécution du programme C++. Cette région est en lecture seule afin que l'on ne puisse modifier dynamiquement les instructions exécutées ainsi que certaines chaînes de caractères
- **La pile (stack)**, dans laquelle sont stockées les trames de pile. Une nouvelle trame de pile est créée pour chaque appel de fonction. Une trame de pile est une trame de données qui contient les variables locales de la fonction correspondante et est détruite (éjectée) lorsque cette fonction a été exécutée.
- **Le tas (heap)**, dans lequel la mémoire allouée dynamiquement est stockée. Pour optimiser l'utilisation de la mémoire, le tas et la pile se développent généralement l'un vers l'autre, comme illustré dans la figure ci-dessus.

L'allocation dynamique

L'allocation dynamique est le mécanisme permettant de créer lors de l'exécution du programme des structures de données dans une zone mémoire appelée le **tas** (contrairement aux variables locales initialisées dans la **pile**). On parle d'allocation **dynamique** par opposition à l'allocation statique survenant lors de la compilation ; il est impossible par le compilateur (sauf certains cas) de deviner quand et comment les allocations dynamiques seront réalisées : ce n'est qu'à l'exécution que l'on peut le savoir.

On utilise l'allocation dynamique lorsque la quantité de données à stocker en mémoire dépend de conditions uniquement connues à l'exécution (données reçues de l'extérieur). Pour chaque allocation réalisée, il est nécessaire de prévoir une libération ultérieure afin d'éviter toute fuite de mémoire. Des fuites de mémoire conduisent un programme à utiliser de plus en plus de mémoire lors de son exécution jusqu'à son éventuel arrêt forcé par le noyau en cas de pénurie de mémoire.

Allocation et la désallocation de mémoire sur le tas (Heap) :

On peut effectuer la gestion de la mémoire en C++ à l'aide de deux opérateurs :

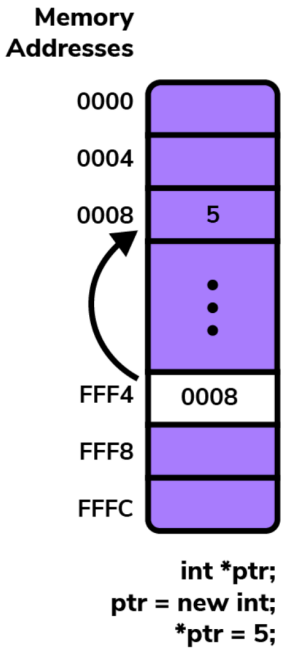
- opérateur **new** pour allouer un bloc de mémoire sur le tas
- opérateur **delete** pour désallouer un bloc de mémoire du tas

Exemple :

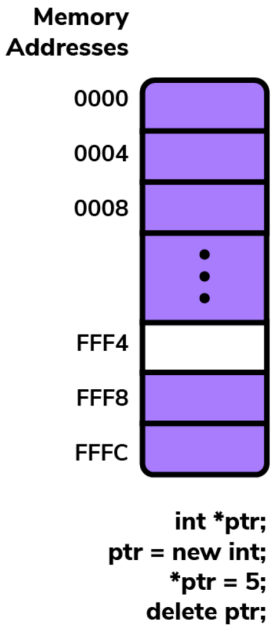
```
#include <iostream>
int main() {
   // un pointeur vers un entier
   int *ptr;

   // Alloue de la mémoire pour un entier
   ptr = new int;

   // Affecte une valeur à l'entier nouvellement alloué
   *ptr = 5;

   // Affiche la valeur de l'entier
   std::cout << "\nValeur int =" << *ptr;

   // Affiche l'emplacement mémoire, où l'entier est stocké
   std::cout << "\nint est sauvegarder à l'adresse =" << ptr << "\n";

   // Libère la mémoire réservée à l'entier (pour éviter les fuites de
mémoire)
   delete ptr;

   return 0;
}
```

- L'opérateur **new** réserve un emplacement mémoire qui peut stocker un entier C++ (i.e. 4 octets). Par la suite, il renvoie l'adresse mémoire nouvellement allouée.
- On crée un pointeur, **ptr**, pour stocker l'adresse mémoire renvoyée par l'opérateur **new**.

- On sauvegarde une valeur entière sur l'adresse mémoire nouvellement allouée en utilisant `*ptr=5`.



- On affiche l'adresse mémoire où l'entier est stocké et la valeur entière stockée à cet emplacement mémoire.
- Enfin, on libère le bloc de mémoire réservé par **`new`** à l'aide de l'opérateur **`delete`**.



Bugs courants de gestion de la mémoire

Il existe deux bugs de codage courants qu'on peut rencontrer avec la gestion dynamique de la mémoire : les fuites de mémoire et les erreurs de segmentation.

Les fuites de mémoire se produisent lorsque la mémoire n'est pas libérée, même après qu'elle ne soit plus nécessaire. Cela peut amener le programme à manquer de la mémoire maximale dont il dispose.

```
#include <iostream>

void fuite_de_memoire() {
   // Pointeur vers un entier
   int *ptr;
   // Alloue de la mémoire pour un entier
   ptr = new int;
   // La mémoire n'est pas désallouée ici (car la ligne suivante est
commentée)
   //delete ptr;
}
int main() {
   fuite_de_memoire();
   // Le pointeur (ptr) n'est plus accessible
   // mais la mémoire est toujours allouée pour un int.
   return 0;
}
```

Dans cet exemple de code, la fonction **`fuite_de_memoire()`** alloue de la mémoire, mais cette mémoire n'est pas désallouée. Une fois la fonction renvoyée, la mémoire allouée est toujours utilisée, même si elle n'est plus accessible.

L'erreur de segmentation est un autre bug de gestion de mémoire dynamique. Ce bug se produit lorsqu'un programme accède à un emplacement de mémoire qui ne lui est pas alloué dans l'espace d'adressage du programme. L'espace d'adressage fait référence à la région de mémoire où un programme est autorisé à allouer de la mémoire.

Le programme suivant génère l'erreur de segmentation dès qu'il n'a plus d'espace d'adressage :

```
#include <iostream>

void defaut_de_segmentation() {
   // Un pointeur vers un entier
   int *ptr;
   // Allocation de mémoire pour un entier.
   ptr = new int;
   while(true) {
       // Ce qui suit génère une erreur de segmentation
       // lorsque nous manquons d'espace d'adressage du
       *(ptr++) = 5;
   }
}

int main() {
   defaut_de_segmentation();
   return 0;
}
```