Langage C/C++

Cours 3/5 : Création dynamique d'objets

Hubert Godfroy

27 novembre 2014

La dernière fois…

On a vu…

- Encombrement mémoire
- Types non structurés (`int`, `long`, `float`, …)
- Données structurées (`enum`, `struct`, tableau, …)
- Représentation en mémoire

La dernière fois…

On a vu…

- Encombrement mémoire
- Types non structurés (`int`, `long`, `float`, …)
- Données structurées (`enum`, `struct`, tableau, …)
- Représentation en mémoire

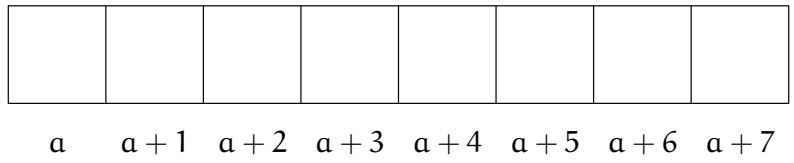
Problèmes soulevés

- Création de tableau de taille non connue à l'avance
- Définition de structures récursives

Plan

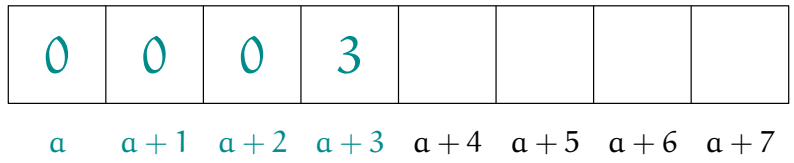
Plan

Nécessité d'un nouveau type



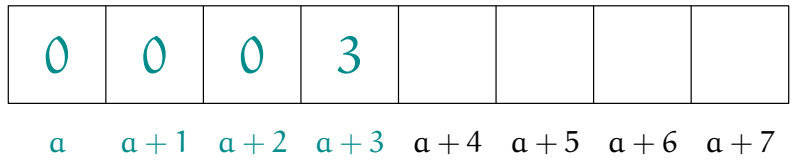
Nécessité d'un nouveau type

```
int i = 3
```



Nécessité d'un nouveau type

`int i = 3`



$\Rightarrow$ On accède à a avec `&i`.

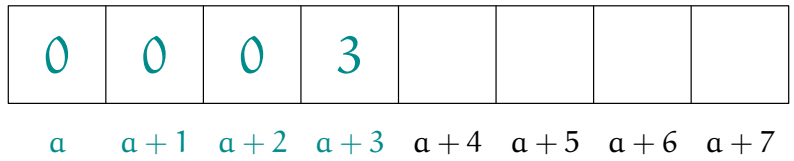
Exemple

```
int i = 3;
printf(``%p\n'', &i);
```

affiche 19283749249 (l'adresse en mémoire de i))

Nécessité d'un nouveau type

`int i = 3`



⇒ On accède à a avec `&i`.

Exemple

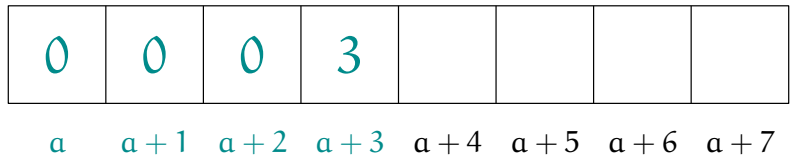
```
int i = 3;
printf(``%p\n'', &i);
```

affiche 19283749249 (l'adresse en mémoire de i))

⇝ Comment manipuler ces adresses (les stockées elle même en mémoire) ?

Nécessité d'un nouveau type

`int i = 3`



⇒ On accède à a avec `&i`.

Exemple

```
int i = 3;
printf(``%p\n'', &i);
```

affiche 19283749249 (l'adresse en mémoire de i))

⇝ Comment manipuler ces adresses (les stockées elle même en mémoire) ?

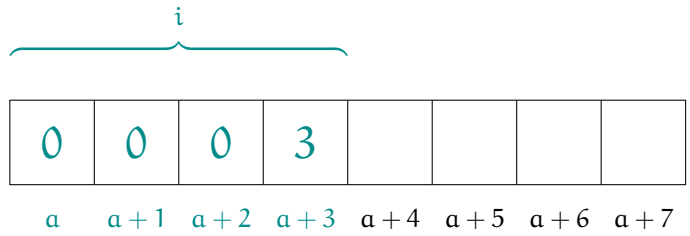
⇒ Un nouveau type : les pointeurs

Les pointeurs

- Si T est un type, on note T$*$ le type des pointeurs sur un type T.

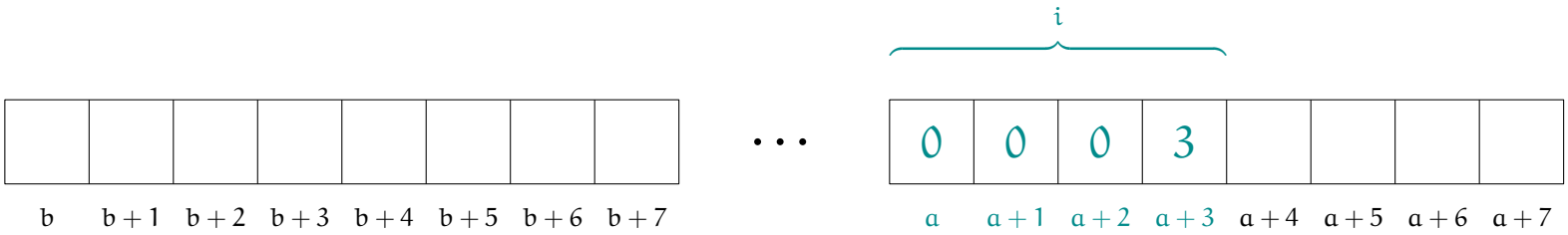
Les pointeurs

- Si T est un type, on note T$*$ le type des pointeurs sur un type T.



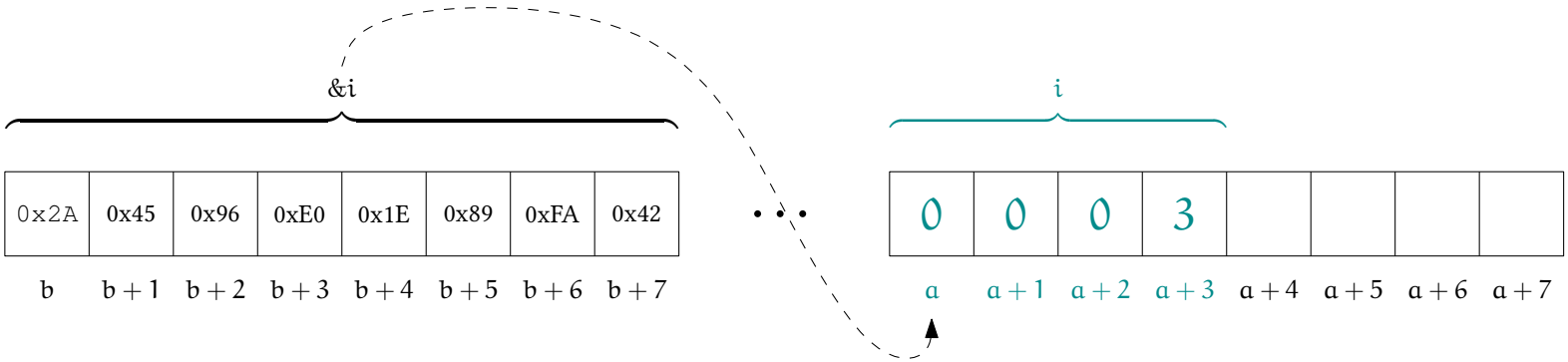
Les pointeurs

- Si T est un type, on note T$*$ le type des pointeurs sur un type T.



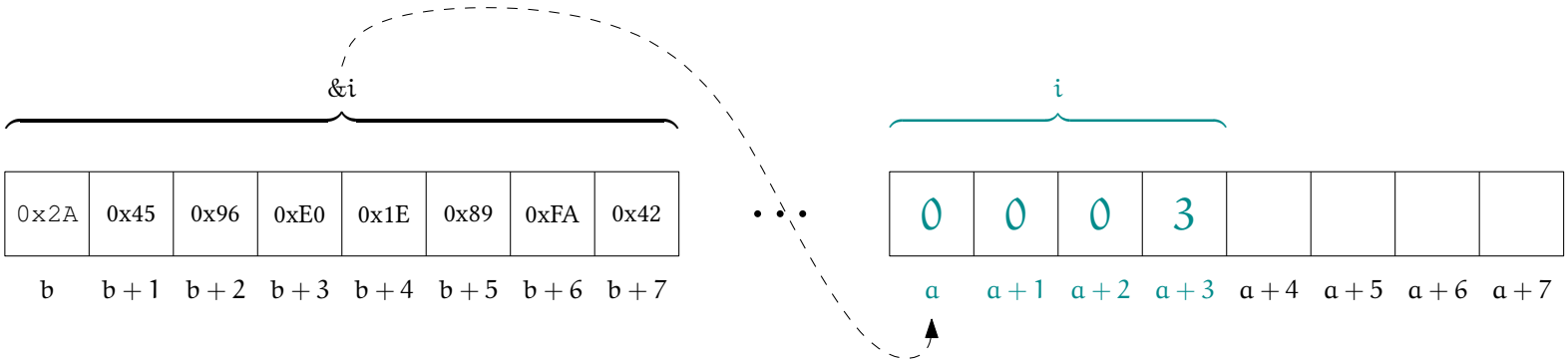
Les pointeurs

- Si T est un type, on note T$*$ le type des pointeurs sur un type T.



Les pointeurs

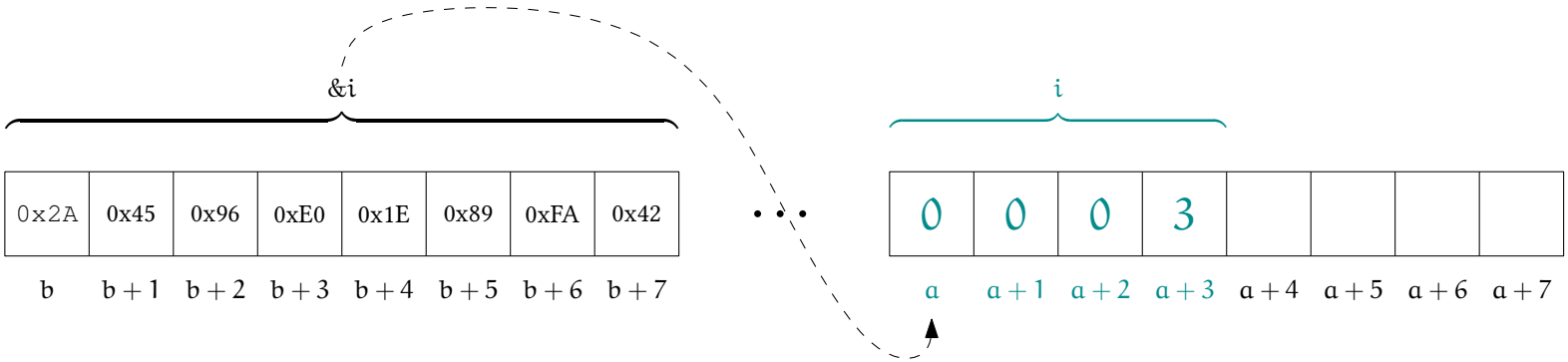
- Si T est un type, on note T$*$ le type des pointeurs sur un type T.



- ici : `int* p = &i`

Les pointeurs

- Si T est un type, on note T$*$ le type des pointeurs sur un type T.

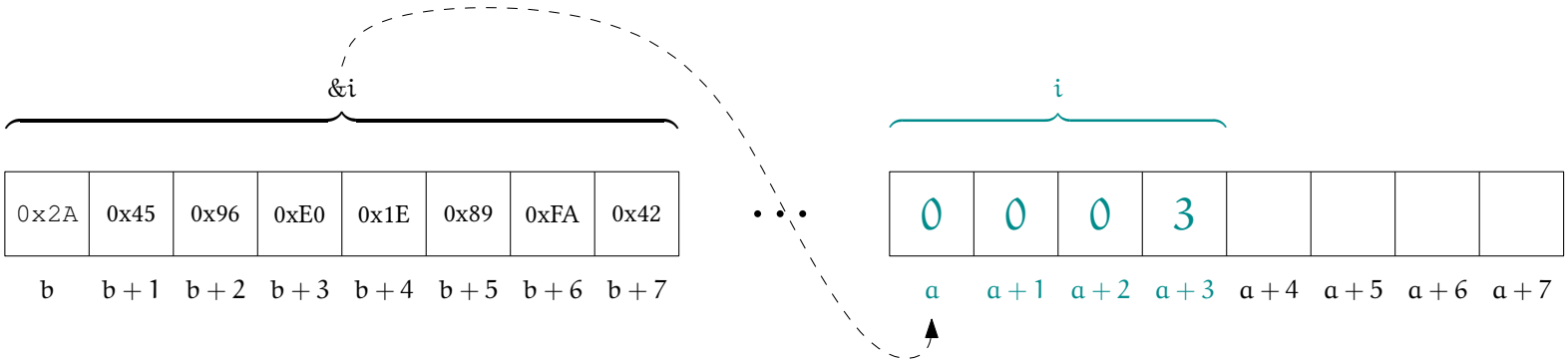


- ici : `int* p = &i`
- On peut retrouver `i` :

$$*p = *\&i = i$$

Les pointeurs

- Si T est un type, on note T$*$ le type des pointeurs sur un type T.



- ici : `int* p = &i`
- On peut retrouver `i` :

$$*p = *\&i = i$$

- La taille d'un pointeur ne varie pas, quelque soit le type pointé.

Pointeurs et structures

Si

```
struct type_struct{
  int champ;
  ...
}
```

et `p` de type `(struct type_struct)*`, on peux faire

```
p->champ
```

Pointeurs et tableaux

- Un tableau est le pointeur indiquant son premier élément.
- Si `p` est de type `int*`, `p[i]` est de type `int`.

Pointeurs non typés

- Un pointeur dont on ne sais pas sur quoi il point est de type `void*`.
- Il est interdit de faire un `*p` si `p` est de type `void*`.
- Il faut faire un cast sur un pointeur pour lui indiquer vers quel type il pointe : `(int*) p`

Exemple d'utilisation : structures récursives

- Cette structure est impossible

```
struct liste{
  int valeur;
  struct liste queue;
}
```

Exemple d'utilisation : structures récursives

- Cette structure est impossible

```
struct liste{
  int valeur;
  struct liste queue;
}
```

car

$$\texttt{sizeof(liste)} = \texttt{sizeof(int)} + \texttt{sizeof(liste)}$$

Exemple d'utilisation : structures récursives

- Cette structure est impossible

```
struct liste{
  int valeur;
  struct liste queue;
}
```

car

$$\texttt{sizeof(liste)} = \texttt{sizeof(int)} + \texttt{sizeof(liste)}$$

- Celle ci l'est

```
struct liste{
  int valeur;
  (struct liste)* queue;
}
```

Exemple d'utilisation : structures récursives

- Cette structure est impossible

```
struct liste{
  int valeur;
  struct liste queue;
}
```

car

$$\texttt{sizeof(liste)} = \texttt{sizeof(int)} + \texttt{sizeof(liste)}$$

- Celle ci l'est

```
struct liste{
  int valeur;
  (struct liste)* queue;
}
```

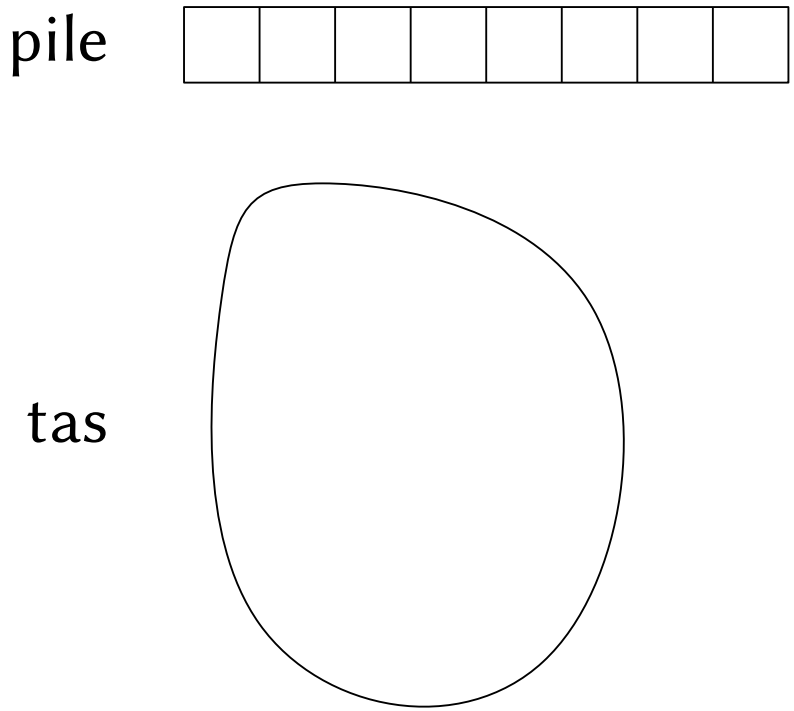
car la taille d'un pointeur est invariable.

$$\texttt{sizeof(liste)} = \texttt{sizeof(int)} + \texttt{sizeof(liste*)}$$

Plan

Pile/tas

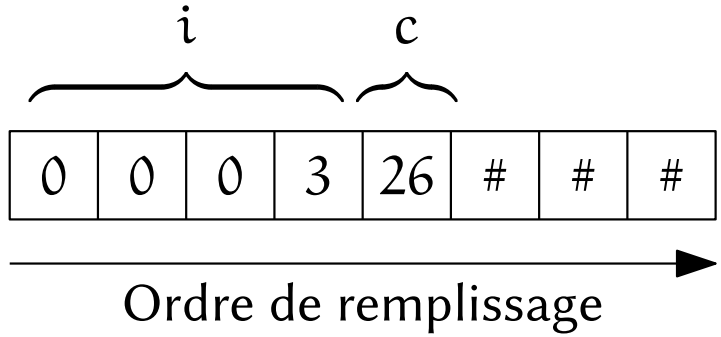
La mémoire est partitionnée en deux instances :



Utilisation de la pile

- Les variables y sont stockées

```
int i = 3;
char c = 26;
```

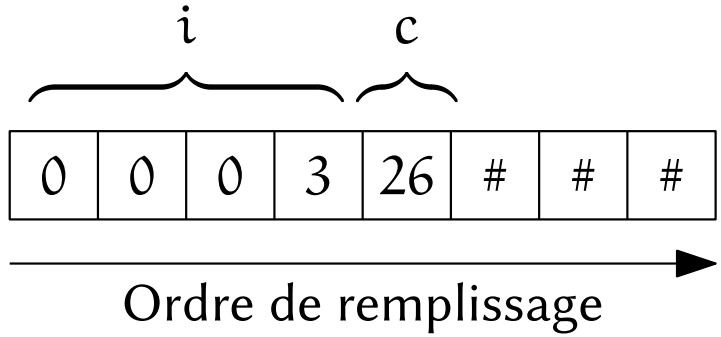


On dit que la déclaration de ces variables est statique.

Utilisation de la pile

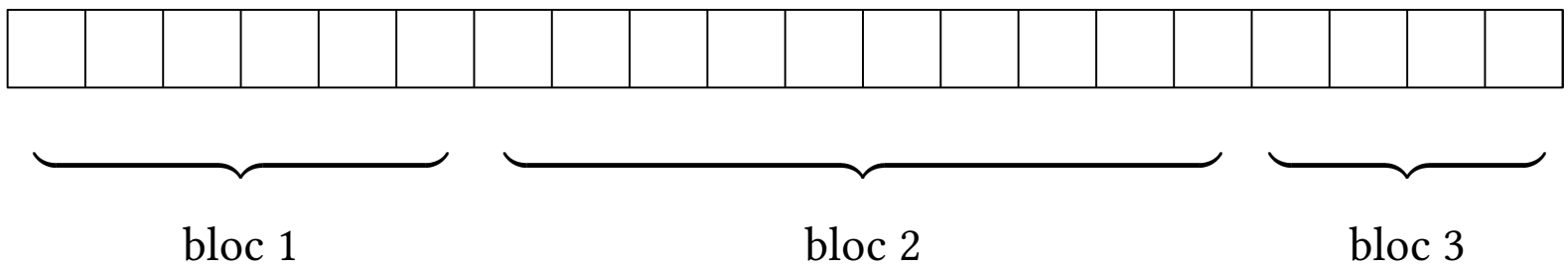
- Les variables y sont stockées

```
int i = 3;
char c = 26;
```



On dit que la déclaration de ces variables est statique.

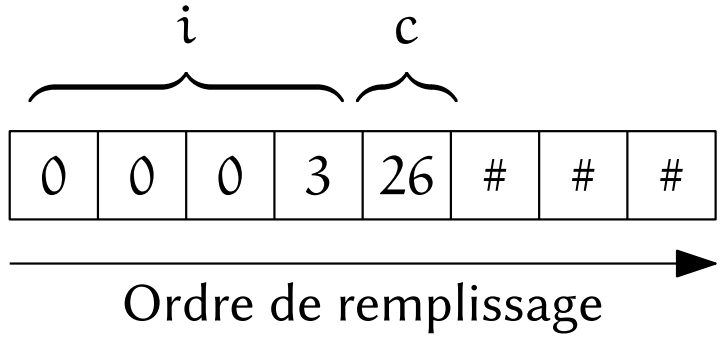
- La pile correspond aux blocs (code entre {})



Utilisation de la pile

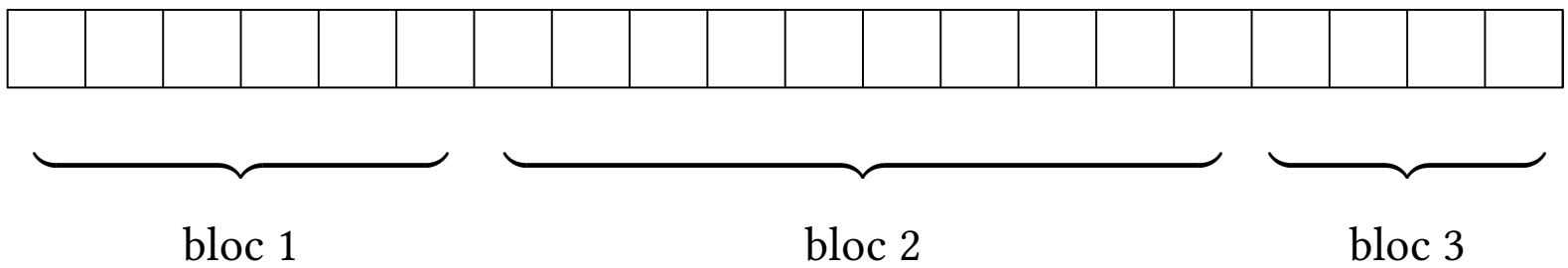
- Les variables y sont stockées

```
int i = 3;
char c = 26;
```



On dit que la déclaration de ces variables est statique.

- La pile correspond aux blocs (code entre {})



- Les valeurs d'un bloc sont effacées lorsque l'on sort de ce bloc

Cas des fonctions

- Les paramètres des fonctions sont copiés sur la pile
- Si une fonction est définie

```
int fonc(int i, char c){
   ...
}
```

Cas des fonctions

- Les paramètres des fonctions sont copiés sur la pile
- Si une fonction est définie

```
int fonc(int i, char c){
   ...
}
```

l'appel dans le code

```
...
fonc(2,'a');
...
```

Cas des fonctions

- Les paramètres des fonctions sont copiés sur la pile
- Si une fonction est définie

```
int fonc(int i, char c){
   ...
}
```

l'appel dans le code

```
...
fonc(2,'a');
...
```

est équivalent à

```
...
{
   int i = 2;
   char c = 'a';
   ...
}
...
```

Conclusions pratiques

- Une variable définie dans un bloc n'est pas accessible dans les blocs supérieurs.
- Les arguments passés en paramètres ne peuvent pas être changés.
- La taille des variables doit être connue lors de la compilation (`int t[2]` et pas `int t[var]`)

Plan

Allocation dynamique

- Contrairement aux variables statiques stockées sur la pile, les variables dynamiques le sont sur le tas.

Allocation dynamique

- Contrairement aux variables statiques stockées sur la pile, les variables dynamiques le sont sur le tas.
- Leur emplacement en mémoire est indépendante des blocs.

Allocation dynamique

- Contrairement aux variables statiques stockées sur la pile, les variables dynamiques le sont sur le tas.
- Leur emplacement en mémoire est indépendante des blocs.

$\Rightarrow$ Elle ne sont pas effacées automatiquement à la sortie d'un bloc.

Allocation dynamique

- Contrairement aux variables statiques stockées sur la pile, les variables dynamiques le sont sur le tas.
- Leur emplacement en mémoire est indépendante des blocs.

$\Rightarrow$ Elle ne sont pas effacées automatiquement à la sortie d'un bloc.

$\Rightarrow$ Leur destruction est effectuée manuellement.

Allocation dynamique

- Contrairement aux variables statiques stockées sur la pile, les variables dynamiques le sont sur le tas.
- Leur emplacement en mémoire est indépendante des blocs.

$\Rightarrow$ Elle ne sont pas effacées automatiquement à la sortie d'un bloc.

$\Rightarrow$ Leur destruction est effectuée manuellement.

- On utilise les fonctions `(void*)malloc(int)` et `(void)free(void*)` définies dans `stdlib.h`.

Schéma final

Un objet de taille 20 octets est alloué dans la mémoire :

Schéma final

Un objet de taille 20 octets est alloué dans la mémoire :