Cours 9

Structures

Vincent Lepetit

vincent.lepetit@epfl.ch

Les structures

Notion de structure de données

`int`, `float`, `char`... et les pointeurs sont des types ***scalaires***: ils permettent de stocker une valeur et une seule par variable.

Les tableaux sont une ***structure de données***.

Ils regroupent plusieurs valeurs de même type, chacune étant repérée par un indice.

En langage C, il existe une autre structure de données, qu'on nomme ***structure***.

Les structures permettent de regrouper plusieurs valeurs pouvant être de types différents.

Les ***éléments d'une structure*** sont appelés ***champs***, et sont repérés ***par leur nom***.

Les structures sont une version primitive des objets des langages objets, une notion très importante en programmation.

Déclaration d'une structure

Voici un exemple de déclaration de structure:

```
struct Chat
{
  float poids;
  char sexe; // 'M' ou 'F'
  int numero_tatouage;
};
```

Il s'agit d'un *nouveau type* appelé `Chat`:

Ce type `Chat` peut maintenant être utilisé pour déclarer des variables:

```
Chat felix, garfield;
```

→ Chacune des variables `felix` et `garfield` contiendra un flottant (`poids`), un caractère (`sexe`) et un entier (`numero_tatouage`);

Déclaration d'une structure

`struct` est un mot-clé indiquant la déclaration d'une structure

`Chat` est le nom de la structure.

```
struct Chat
{
  float poids;
  char sexe;
  int numero_tatouage;
};
```

`poids` est un des champs de la structure, de type `float`.

La liste des champs de la structure est délimitée par des accolades `{` et `}`.

L'accolade fermante `}` est suivie d'un point-virgule `;` (contrairement à l'accolade fermante du corps d'une fonction, d'une boucle ou d'un `if`).

Déclaration d'une structure

```
struct Chat
{
  float poids;
  char sexe;
  int numero_tatouage;
};
```

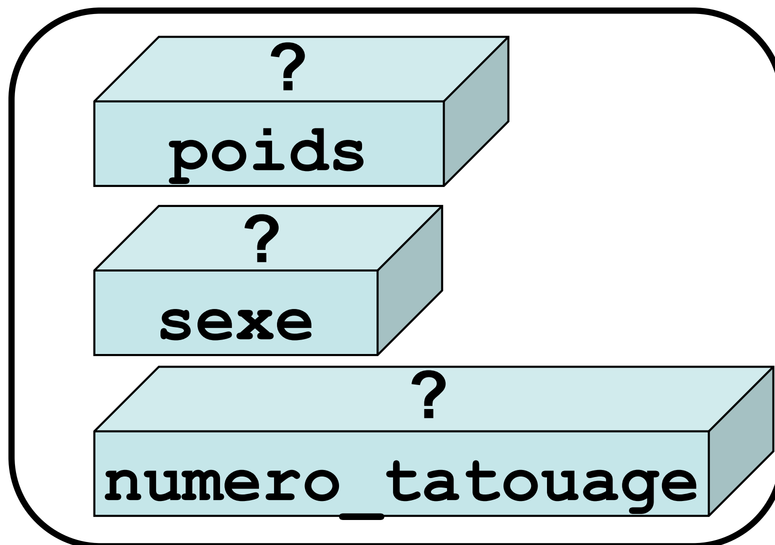
***Champs* de la structure**

Déclaration d'une variable de type structure

`Chat` est le nom de la structure.

`felix` est le nom de la variable.

```
Chat felix;
```



Utilisation des champs d'une structure

Chaque ***champ*** d'une structure peut être manipulé comme une variable:

```
felix.numero_tatouage = 65328;
felix.sexe = 'M';
```

Par exemple:

```
felix.numero_tatouage = 65328;
```

Un point `.` sépare le nom de la variable du nom du champ

`felix`: nom de la variable de type `Chat`

`numero_tatouage`: nom d'un champ de la structure `Chat`

`felix.numero_tatouage` se manipule alors comme toute variable de type `int`.

Exemple

```
#include <iostream>
using namespace std;

struct Chat
{
  float poids;
  char sexe; // 'M' ou 'F'
  int numero_tatouage;
};

int main(int argc, char ** argv)
{
  Chat felix;

  felix.numero_tatouage = 65328;
  felix.sexe = 'M';

  cout << "Entrez le poids du chat de tatouage "
       << felix.numero_tatouage << endl;
  cin >> felix.poids;

  cout << "Le chat " << felix.numero_tatouage
       << " pèse " << felix.poids << endl;

  return 0;
}
```

Déclaration de la structure `Chat` (struct Chat { … };)

Déclaration d'une variable `felix` de type `Chat` (Chat felix;)

Manipulation des champs de la variable `felix` (de felix.numero_tatouage = 65328; à return 0;)

Utilisation globale d'une structure

Il est possible d'affecter à une variable de type structure le contenu d'une autre variable de même type.

Par exemple, si `chat` et `felix` sont de type `Chat`:

```
Chat chat, felix;
```

on peut écrire:

```
chat = felix;
```

C'est la seule utilisation globale d'une structure autorisée par le C.

Attention

On ne peut pas faire:

```
cin >> chat;
```

La lecture au clavier d'une structure doit se faire champ par champ. Il faut faire:

```
cin >> chat.poids;
cin >> chat.sexe;
cin >> chat.numero_tatouage;
```

Initialisation d'une structure lors de sa déclaration

Par exemple:

```
Chat felix = {2.5, 'M', 65328};
```

Les valeurs sont alors copiées champ par champ.

Cet exemple revient donc à faire:

```
Chat felix;

felix.poids = 2.5;
felix.sexe = 'M';
felix.numero_tatouage = 65328;
```

Imbrication de structures de données

Les champs des structures que nous venons de voir étaient de type scalaire (ici `char`, `int` et `float`).

→ un champ peut être d'un type quelconque: *pointeur, tableau, structure…*

Structure comportant des tableaux

Soit les déclarations suivantes:

```
struct Chat
{
  char nom[100];
  float poids;
  char sexe;
  int numero_tatouage;
};
Chat felix, garfield;
```

Les deux variables `felix` et `garfield` ont maintenant un champ supplémentaire: `nom`, un tableau de caractères (donc une chaîne de caractères).

La notation

```
  felix.nom[3]
```

désigne le quatrième caractère du `nom` de `felix`.

Déclaration d'une structure

`struct` est un mot-clé indiquant la déclaration d'une structure

`Chat` est le nom de la structure.

```
struct Chat
{
  float poids;
  char sexe;
  int numero_tatouage;
};
```

La liste des champs de la structure est délimitée par des accolades `{` et `}`.

Déclaration d'une structure

```
struct Chat
{
  float poids;
  char sexe;
  int numero_tatouage;
};
```

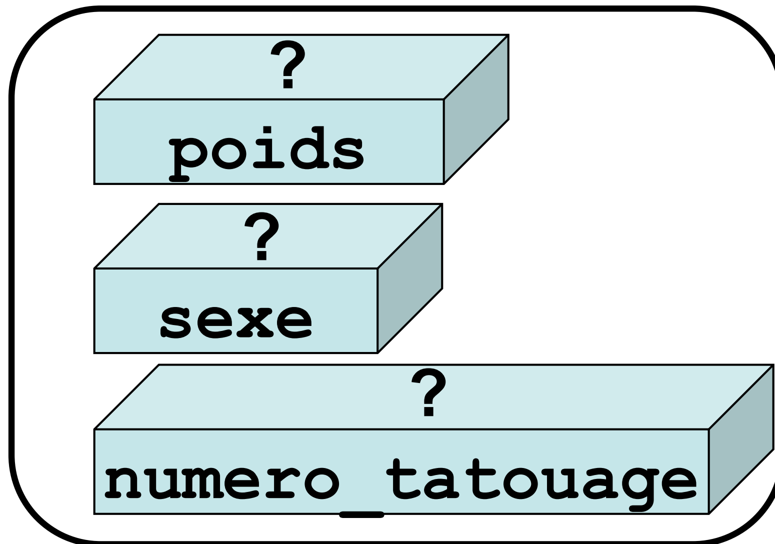
***Champs* de la structure**

Déclaration d'une variable de type structure

`Chat` est le nom de la structure.

`felix` est le nom de la variable.

```
Chat felix;
```



Déclaration d'une variable de type structure **en C**

En C, il fallait répéter le mot-clé `struct` pour déclarer une variable:

```
struct Chat felix;
```

au lieu de

```
Chat felix;
```

en C++

Utilisation des champs d'une structure

Chaque ***champ*** d'une structure peut être manipulé comme une variable:

```
felix.numero_tatouage = 65328;
felix.sexe = 'M';
```

Par exemple:

Un point . sépare le nom de la variable du nom du champ

```
felix.numero_tatouage = 65328;
```

`felix`: nom de la variable de type `Chat`

`numero_tatouage`: nom d'un champ de la structure `Chat`

`felix.numero_tatouage` se manipule alors comme toute variable de type `int`.

Imbrication de structures de données

Les champs des structures que nous venons de voir étaient de type scalaire (ici `char`, `int` et `float`).

→ un champ peut être d'un type quelconque: *pointeur, tableau, structure…*

Structure comportant des tableaux

Soit les déclarations suivantes:

```
struct Chat
{
  char nom[100];
  float poids;
  char sexe;
  int numero_tatouage;
};
Chat felix, garfield;
```

Les deux variables `felix` et `garfield` ont maintenant un champ supplémentaire: `nom`, un tableau de caractères (donc une chaîne de caractères).

La notation

```
  felix.nom[3]
```

désigne le quatrième caractère du `nom` de `felix`.

Tableaux de structures

On peut aussi déclarer des tableaux de structures.

Exemple de déclaration:

```
Chat elevage[50];
```

`elevage` est un tableau de 50 chats:

`elevage[2]`

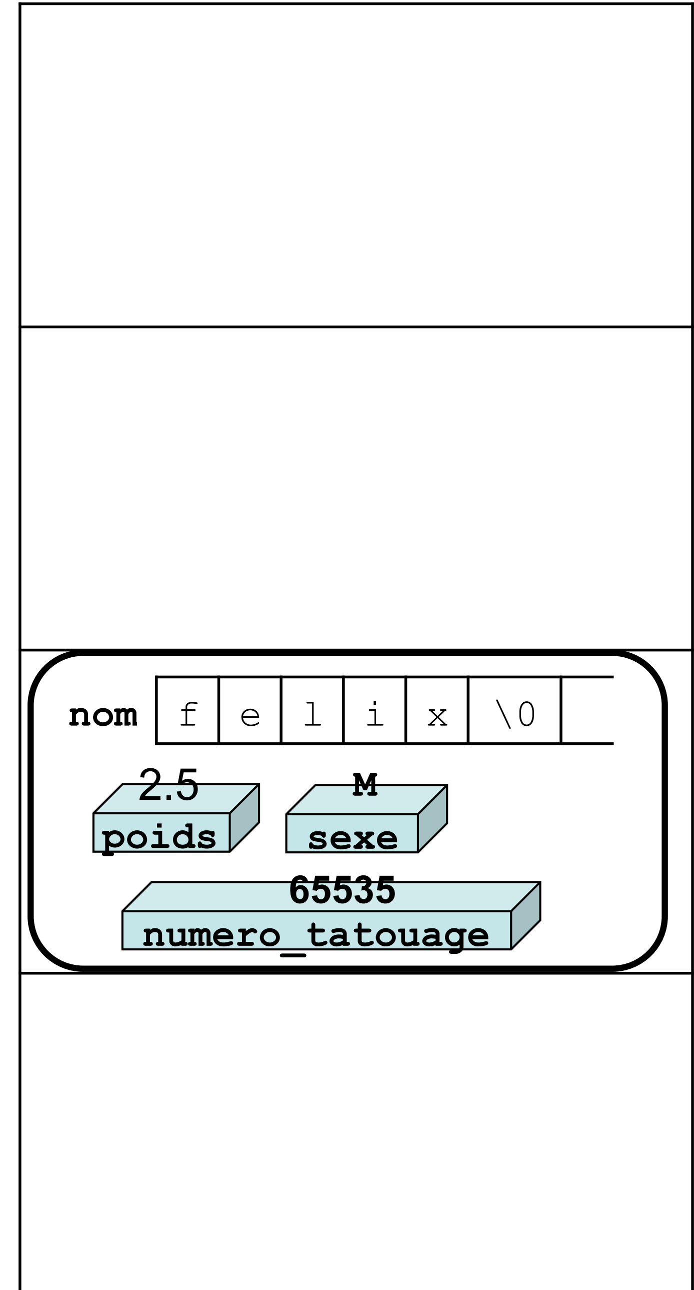
désigne le 3ème chat du tableau `elevage`.

`elevage[2].nom`

désigne le nom du 3ème chat du tableau `elevage`.

`elevage[2].nom[3]`

désigne la quatrième lettre du nom du 3ème chat du tableau `elevage`.



Structures contenant d'autres structures

Exemple:

soit une structure permettant de représenter une date:

```
struct Date
{
  int jour, mois, annee;
};
```

→ on peut utiliser cette structure dans une autre structure:

```
struct Chat
{
  char nom[100];
  float poids;
  char sexe;
  Date date_de_naissance;
  int numero_tatouage;
};
```

```
garfield.date_de_naissance.annee = 2005;
```

permet d'affecter 2005 à l'année de la date de naissance de `garfield`.

Exemples de structures

```
struct Date
{
  int jour, mois, annee;
};
```

```
struct Personne
{
  char nom[30], prenom[30];
  Date date_de_naissance;
};
```

```
struct Point
{
  float x, y;
};
```

```
struct Particule
{
  float position[3], vitesse[3];
  float masse, charge;
};
```

```
struct Etudiant
{
  char nom[100], prenom[100];
  char section[5];
  float note_bonus,
        note_examen,
        note_semestre;
};
```

```
struct Liste_Etudiants
{
  Etudiant * etudiants;
  int nb_etudiants;
  float moyenne_bonus,
        moyenne_examen,
        moyenne_semestre;
};
```

A quoi servent les structures ?

Elles simplifient la conception et l'écriture des programmes.

Elles permettent de représenter une entité complexe comme un chat, une personne, un produit, une date... et de faciliter sa manipulation en regroupant les données.

En C++, ce concept a été étendu pour donner les classes, qui regroupent à la fois les données et des fonctions pour manipuler ces données. C'est ce que vous verrez au deuxième semestre.

Exemple de l'intérêt des structures

Pour définir un ensemble de personnes avec un tableau de structures:

```
struct Chat
{
  char nom[100];
  float poids;
  char sexe;
  Date date_de_naissance;
  int numero_tatouage;
};
Chat elevage[50];
```

A COMPARER avec la solution sans structure, avec un tableau par champ:

```
char nom[50][100];
float poids[50];
char sexe[50];
int jour_de_naissance[50], mois_de_naissance[50],
    annee_de_naissance[50];
```

etc...

qui est plus pénible à manipuler.

Exemple d'utilisation

```
struct Chat
{
  char nom[100];
  float poids;
  char sexe;
  Date date_de_naissance;
  int numero_tatouage;
};
Chat elevage[50];
(...)
for(int i = 0; i < 50; i++)
{
  cout << "Entrez le nom du chat " << i + 1 << endl;
  cin >> elevage[i].nom;
  cout << "Entrez son poids" << endl;
  cin >> elevage[i].poids;
}

for(int i = 0; i < 50; i++)
  cout << elevage[i].nom << " pèse " << elevage[i].poids << endl;
```

Structures et fonctions

Structure en paramètre d'une fonction

```
struct Chat
{
  char nom[100];
  float poids;
};

void affiche(Chat c)
{
  cout << c.nom << " pese " << c.poids << " kg." << endl;
}

int main(int argc, char ** argv)
{
  Chat chat1;
```

(...on remplit les champs de `chat1`*...)*

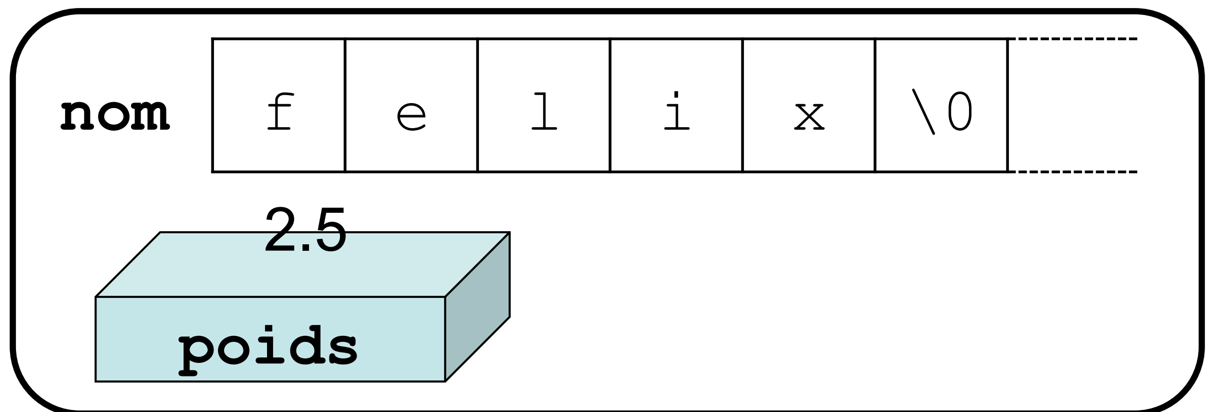
```
  affiche(chat1);

  return 0;
}
```

Pas-à-pas

```
void affiche(Chat c)
{
  cout << c.nom << " pese " << c.poids << " kg." << endl;
}
```

```
Chat chat1;
...
affiche(chat1);
```

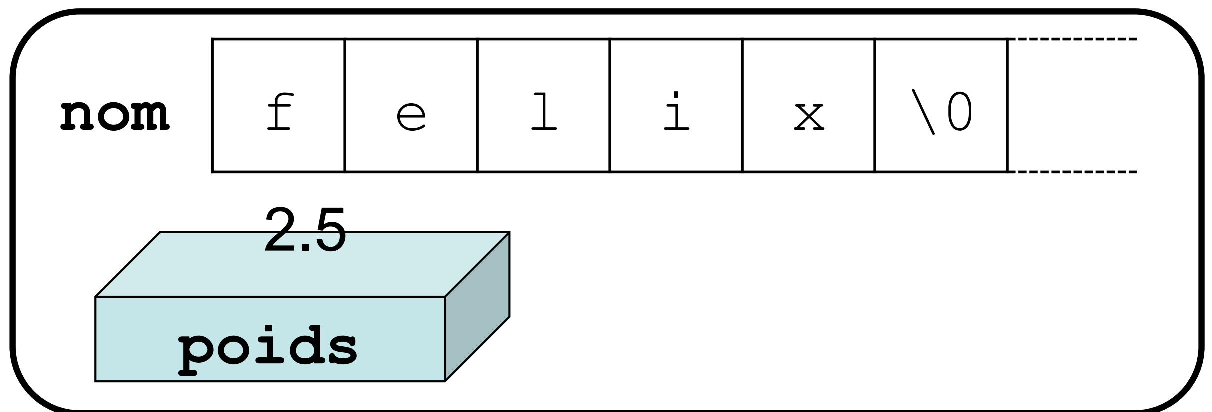
supposons qu'on ait rempli les champs de `chat1` avant d'appeler la fonction `affiche`.

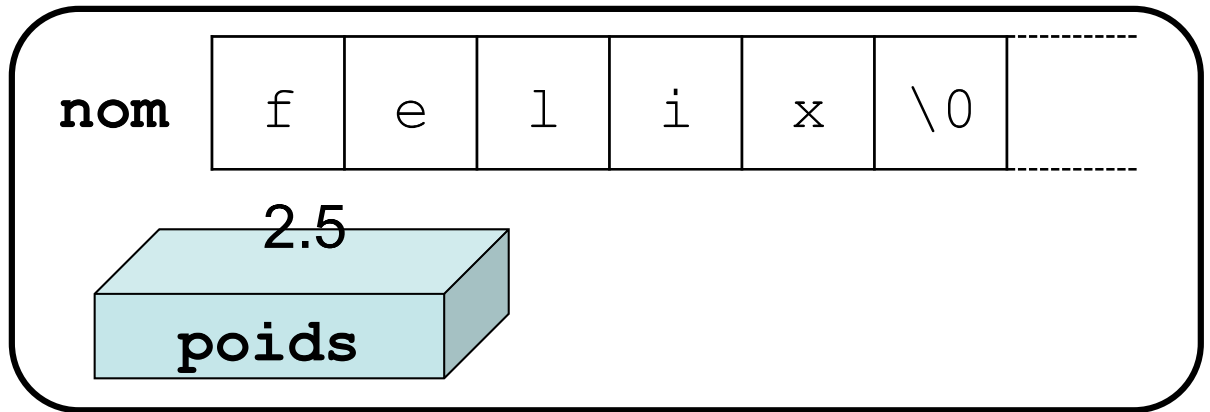


Pas-à-pas

```
void affiche(Chat c)
{
  cout << c.nom << " pese " << c.poids << " kg." << endl;
}
```

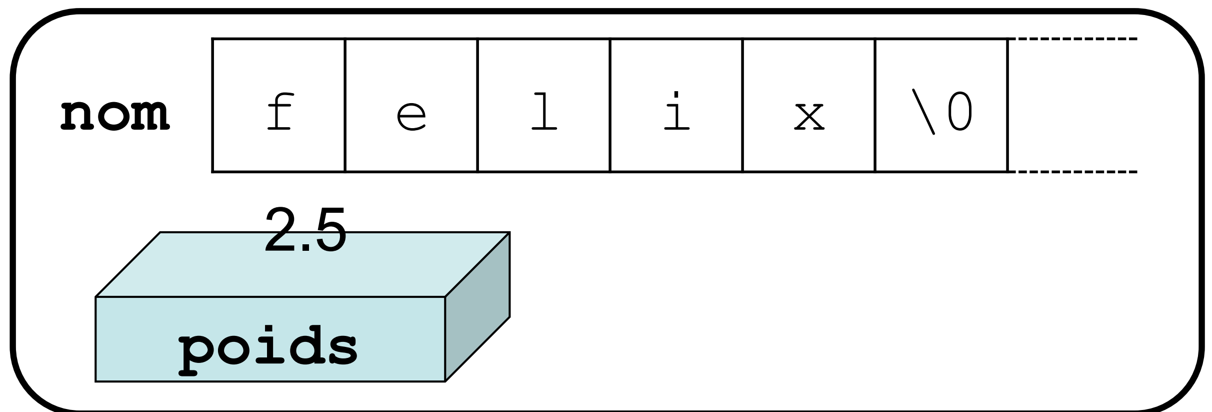
```
Chat chat1;
...
affiche(chat1);
```

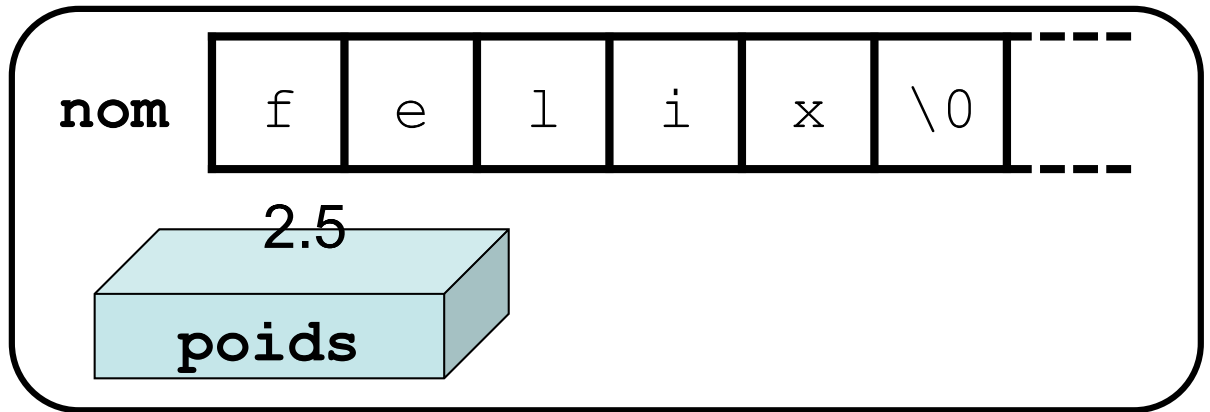





```
void affiche(Chat c)
{
  cout << c.nom << " pese " << c.poids << " kg." << endl;
}
```

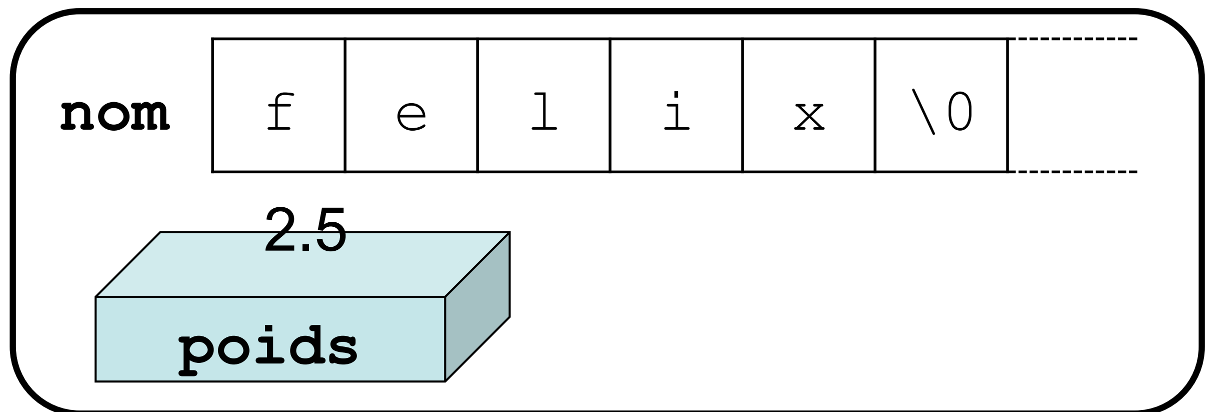
```
Chat chat1;
...
affiche(chat1);
```





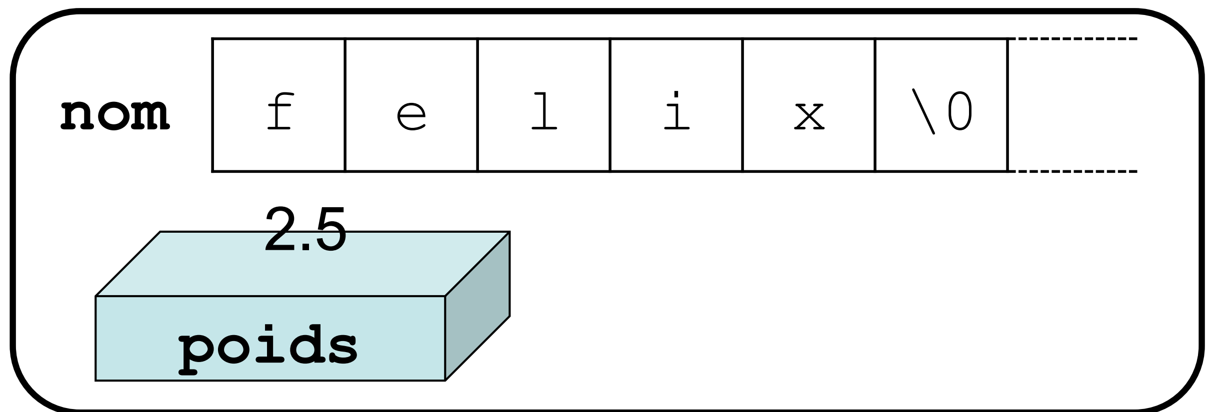

```
void affiche(Chat c)
{
  cout << c.nom << " pese " << c.poids << " kg." << endl;
}
```

```
Chat chat1;
...
affiche(chat1);
```



```
void affiche(Chat c)
{
  cout << c.nom << " pese " << c.poids << " kg." << endl;
}
```

```
Chat chat1;
...
affiche(chat1);
```



chat1

Passage par valeur d'une structure

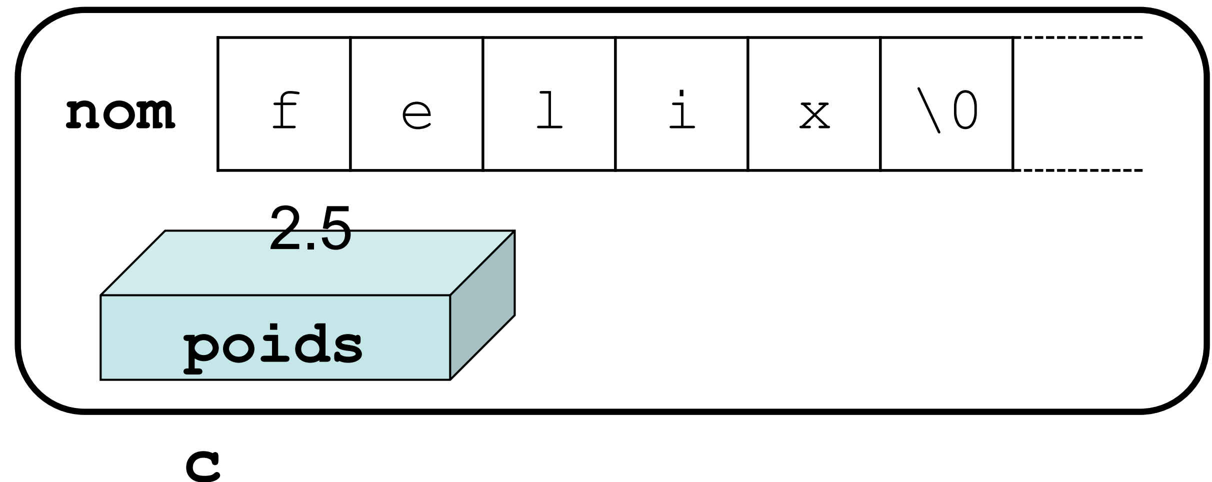
Le passage d'une structure par valeur a deux inconvénients:

- la fonction ne peut pas modifier les champs de la structure;
- le passage de paramètre est coûteux si la structure comporte beaucoup de données.

En règle générale, on évite le passage d'une structure par valeur.

A l'appel de la fonction `affiche`:

```
void affiche(Chat c)
{
  ...
}

  affiche(chat1);
```



c

- le champ `nom` de `chat1` est copié dans le champ `nom` du paramètre `c`;
- le champ `poids` de `chat1` est copié dans le champ `poids` du paramètre `c`.

Passage par adresse d'une structure

La fonction suivante utilise un passage par adresse.
On ne peut pas écrire simplement `*c.nom`, il faut écrire `(*c).nom`:

```
void affiche(Chat * c)
{
  cout << (*c).nom << " pese " << (*c).poids << " kg." << endl;
}

...

 affiche(&chat1);
```

→ `c` est maintenant de type pointeur sur `Chat`.

→ Uniquement l'ADRESSE de `chat1` est copié dans `c`.

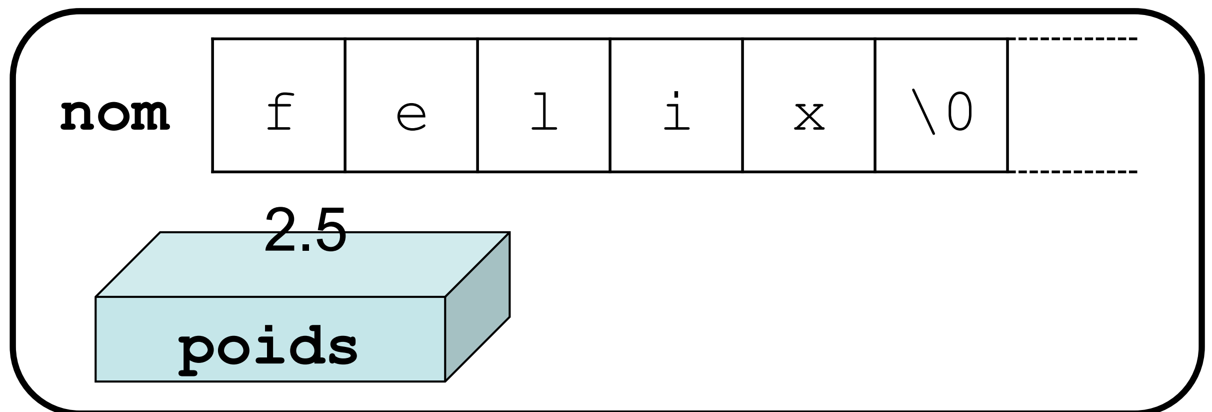
→ `c` pointe sur `chat1`, et `*c` désigne donc `chat1`.

→ `(*c).nom` désigne donc `nom` de `chat1`.

Pas-à-pas

```
void affiche(Chat * c)
{
  cout << (*c).nom << " pese " << (*c).poids << " kg." << endl;
}
```

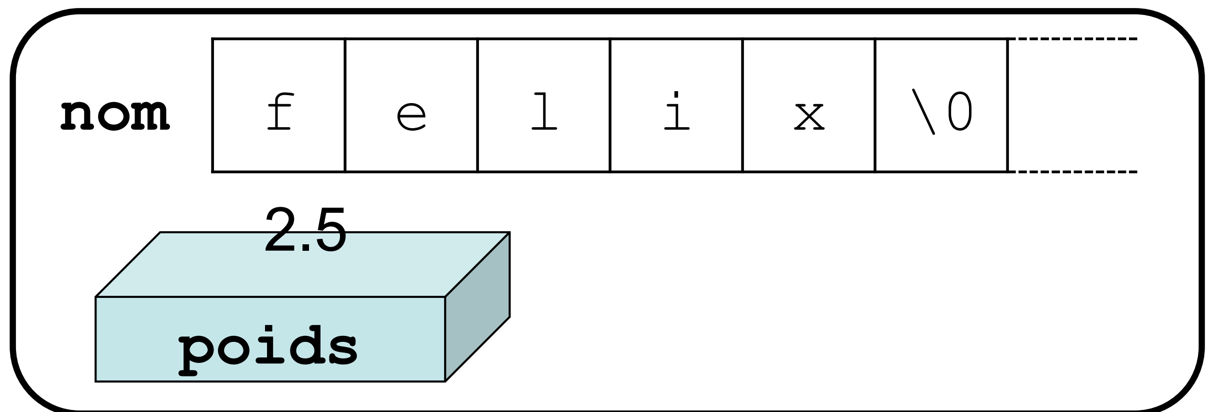
```
affiche(&chat1);
```



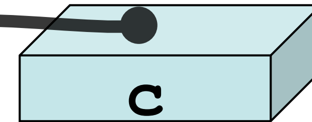
Pas-à-pas

```
void affiche(Chat * c)
{
  cout << (*c).nom << " pese " << (*c).poids << " kg." << endl;
}
```

```
affiche(&chat1);
```

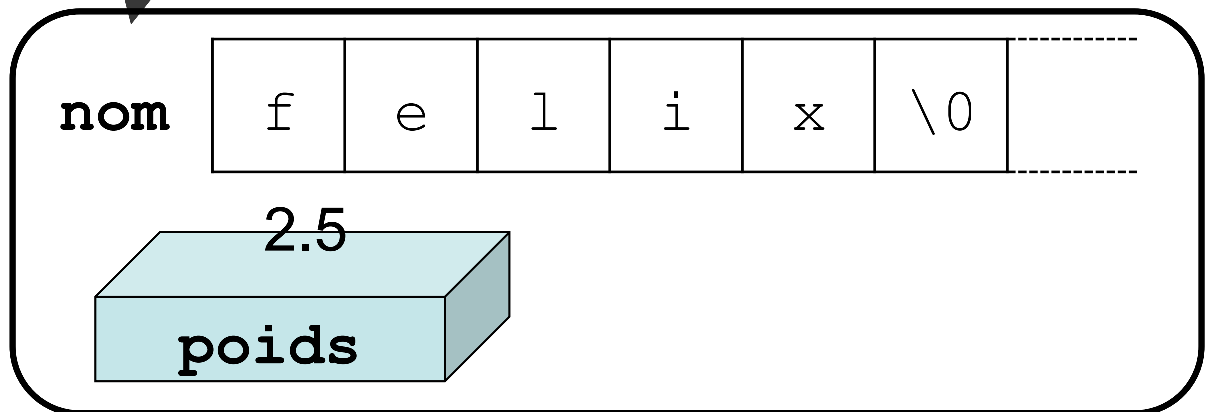


Pas-à-pas

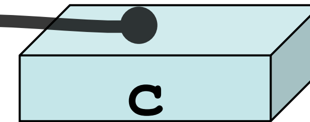



```
void affiche(Chat * c)
{
  cout << (*c).nom << " pese " << (*c).poids << " kg." << endl;
}
```

```
affiche(&chat1);
```

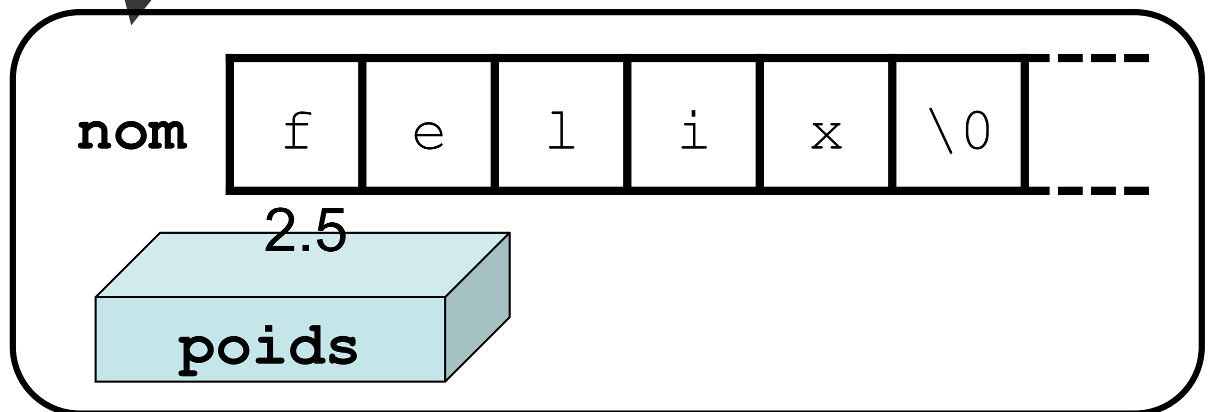


Pas-à-pas




```
void affiche(Chat * c)
{
  cout << (*c).nom << " pese " << (*c).poids << " kg." << endl;
}
```

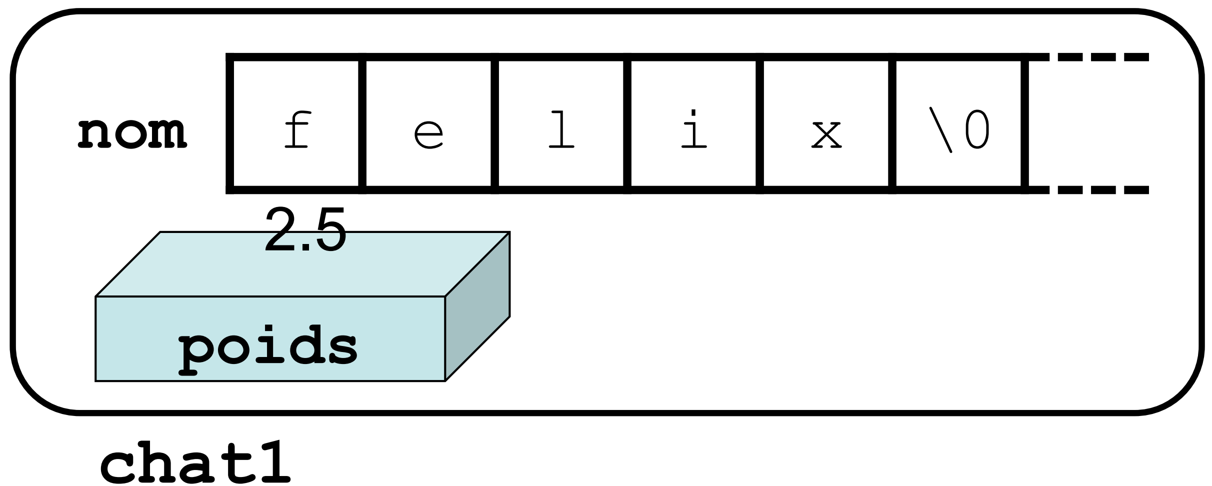
```
affiche(&chat1);
```



Pas-à-pas

```
void affiche(Chat * c)
{
  cout << (*c).nom << " pese " << (*c).poids << " kg." << endl;
}
```

```
affiche(&chat1);
```



L'opérateur ->

La notation

`(*c).nom`

n'est pas très lisible.

Il existe un opérateur, noté `->` (un signe moins `-`, un signe supérieur `>`), qui permet d'écrire des expressions équivalentes mais plus lisibles:

`c->nom`

`c->nom` est équivalent à `(*c).nom`

Opérateur ->

```
void affiche(Chat * c)
{
  cout << (*c).nom << " pese " << (*c).poids << " kg." << endl;
}
```

devient:

```
void affiche(Chat * c)
{
  cout << c->nom << " pese " << c->poids << " kg." << endl;
}
```

Dorénavant, utilisez **->**

`.` ou `->` ?

A gauche de **`.`** on trouve forcément une *variable* de type structure:

```
Chat c;
c.poids = 2.5;
```

A gauche de **`->`** on trouve forcément un *pointeur* sur structure:

```
Chat * pc = &c;
pc->poids = 2.5;
```

Messages d'erreur

- A la compilation de:
```
Chat c;
c->poids = 2.5; // !!
```
on obtient l'erreur:

```
error: base operand of `->' has non-pointer type `Chat'
```

`c` n'est pas un pointeur et appliquer l'opérateur `->` sur `c` n'a donc pas de sens.

- A la compilation de:
```
Chat * pc;
pc.poids = 2.5; // !!
```
on obtient l'erreur:

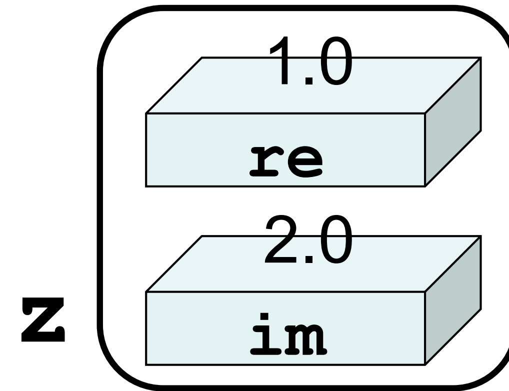
```
error: request for member `poids' in `pc', which is of non-class
type `Chat*'
```

L'opérateur `.` s'applique sur une structure, pas un pointeur comme `pc` est ici.

Fonctions de manipulation de complexes

Définissons une structure `Complexe`, permettant de stocker un nombre complexe:

```
struct Complexe
{
  float re;  // partie reelle
  float im;  // partie imaginaire
};
```



Nous allons écrire quelques fonctions permettant la manipulation de nombres complexes:

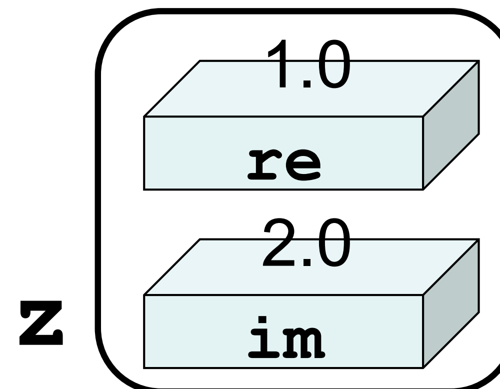
```
afficher_comp
lire_comp
add_comp
mult_comp
```

etc...

Fonctions d'initialisation

Pour créer une variable de type `Complexe` à partir de ses parties réelle et imaginaire, on pourrait faire:

```
Complexe z;
z.re = 1;
z.im = 2;
```



Nous allons voir comment mettre ces instructions dans une fonction.

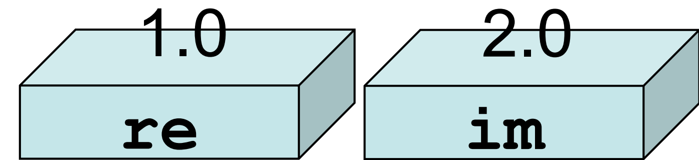
Structure en résultat d’une fonction: Fonction d'initialisation (première version)

```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```

Pas-à-pas

```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```

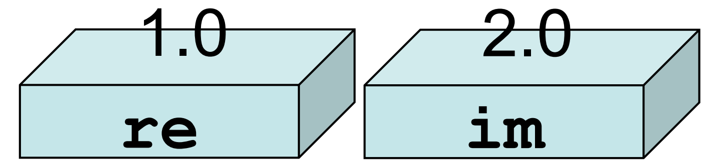
Pas-à-pas

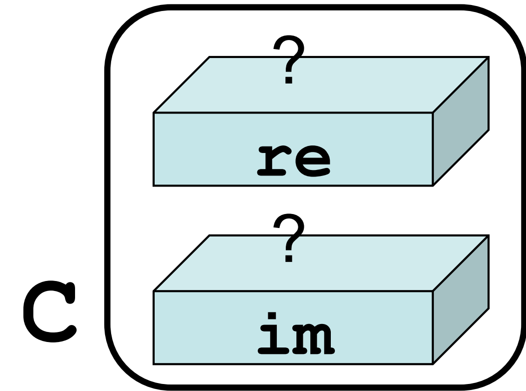



```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```

Pas-à-pas

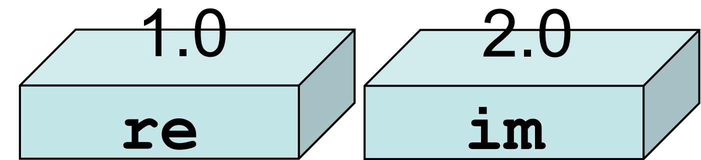
```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```

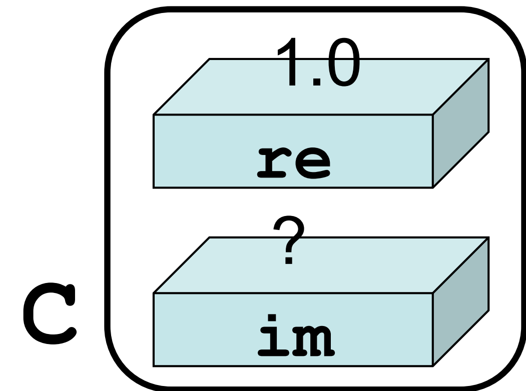




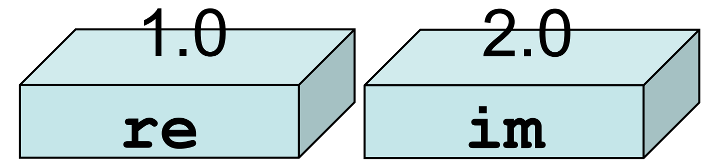
Pas-à-pas

```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```

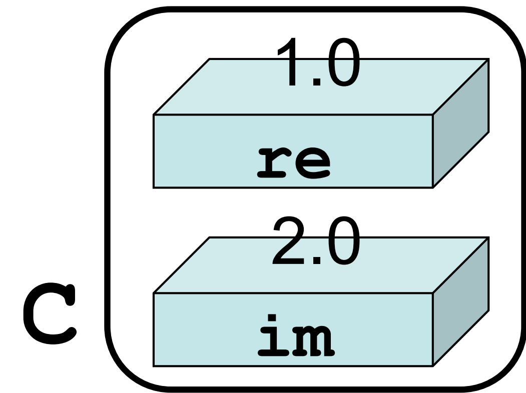




Pas-à-pas

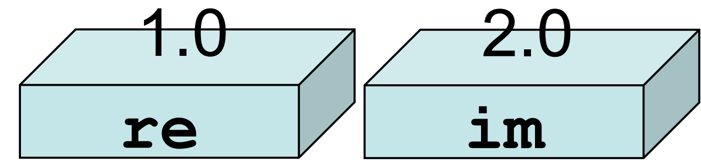


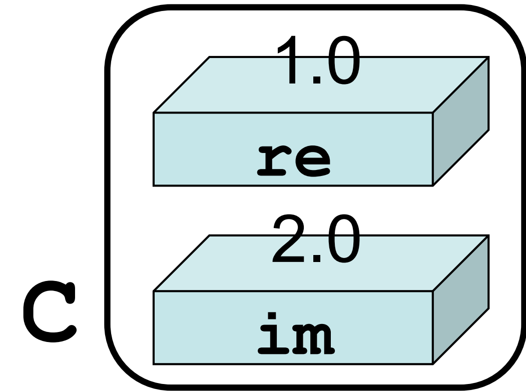

```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```



Pas-à-pas

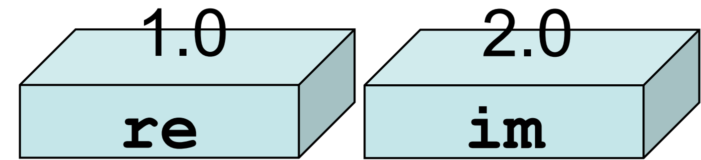
```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```

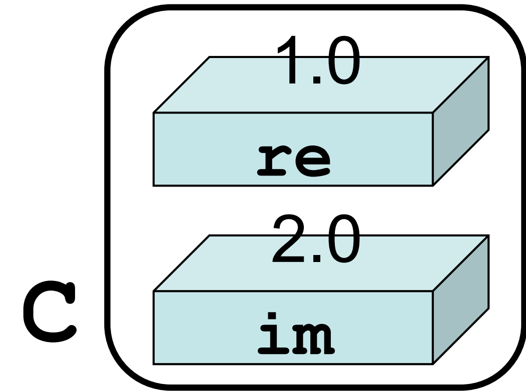




Pas-à-pas

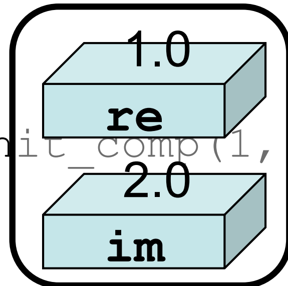
```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```

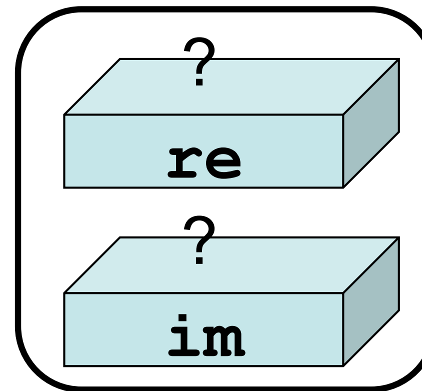




Pas-à-pas

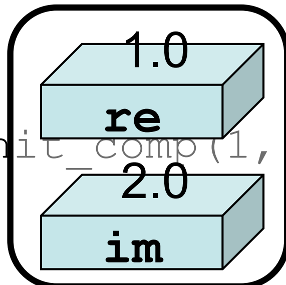
```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```

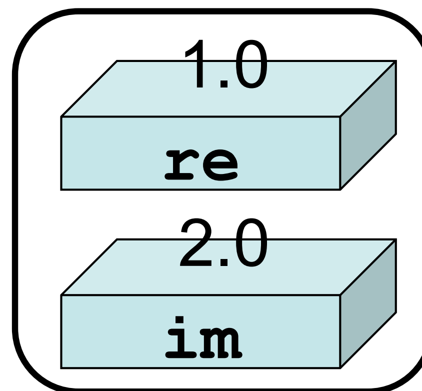




Pas-à-pas

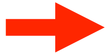
```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```

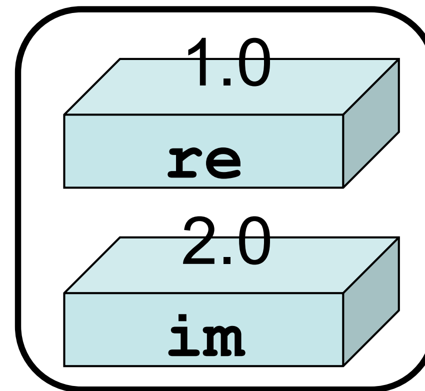




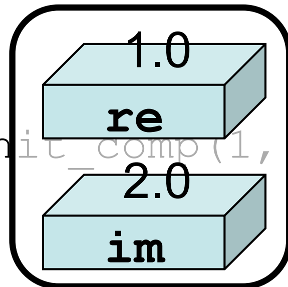
Pas-à-pas

```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```





```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}
...
Complexe z = init_comp(1, 2);
```



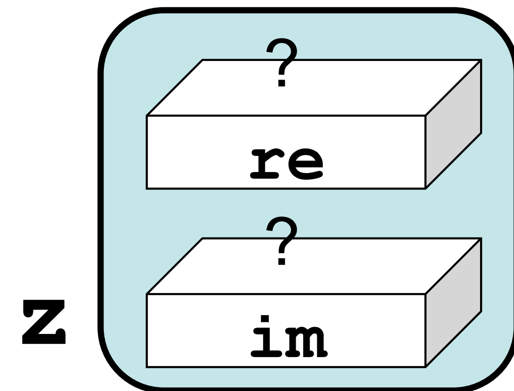
Attention, les champs du résultat de `init_comp` sont copiés dans ceux de `z`, ce qui peut être coûteux comme pour le passage par valeur d'une structure.

Fonction d'initialisation
Deuxième version

```
void init_comp(Complexe * c, float re, float im)
{
  c->re = re;
  c->im = im;
}
...
Complexe z;
init_comp(&z, 1, 2);
```

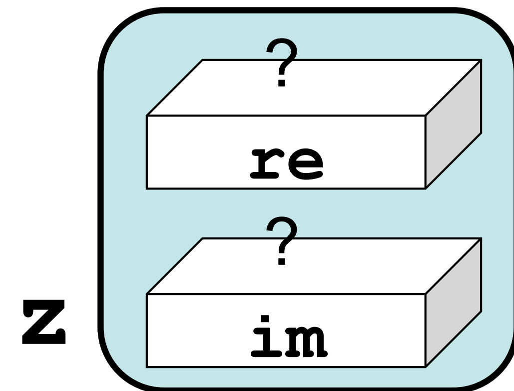
Pas-à-pas

```
void init_comp(Complexe * c, float re, float im)
{
  c->re = re;
  c->im = im;
}
...
Complexe z;
init_comp(&z, 1, 2);
```



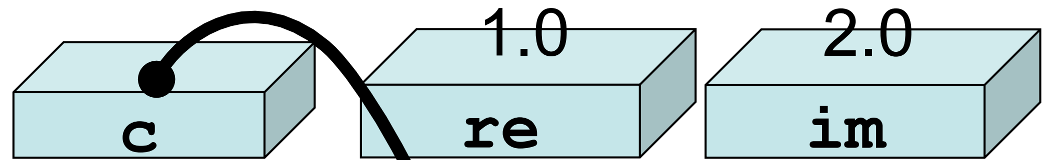
Pas-à-pas

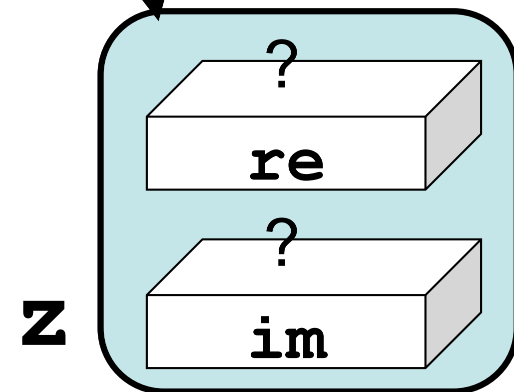
```
void init_comp(Complexe * c, float re, float im)
{
  c->re = re;
  c->im = im;
}
...
Complexe z;
init_comp(&z, 1, 2);
```



Pas-à-pas

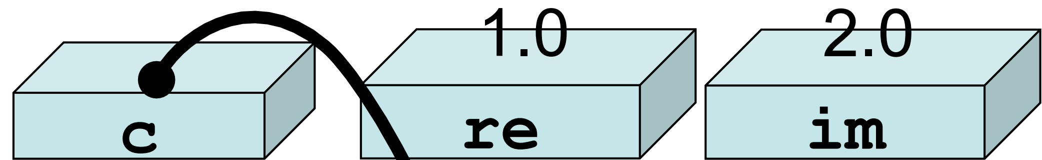
```
void init_comp(Complexe * c, float re, float im)
{
  c->re = re;
  c->im = im;
}
...
Complexe z;
init_comp(&z, 1, 2);
```

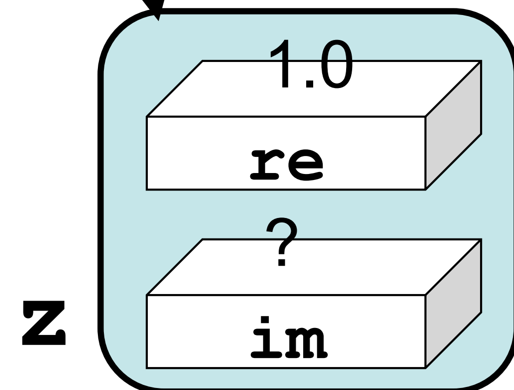




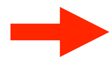
Pas-à-pas

```
void init_comp(Complexe * c, float re, float im)
{
  c->re = re;
  c->im = im;
}
...
Complexe z;
init_comp(&z, 1, 2);
```

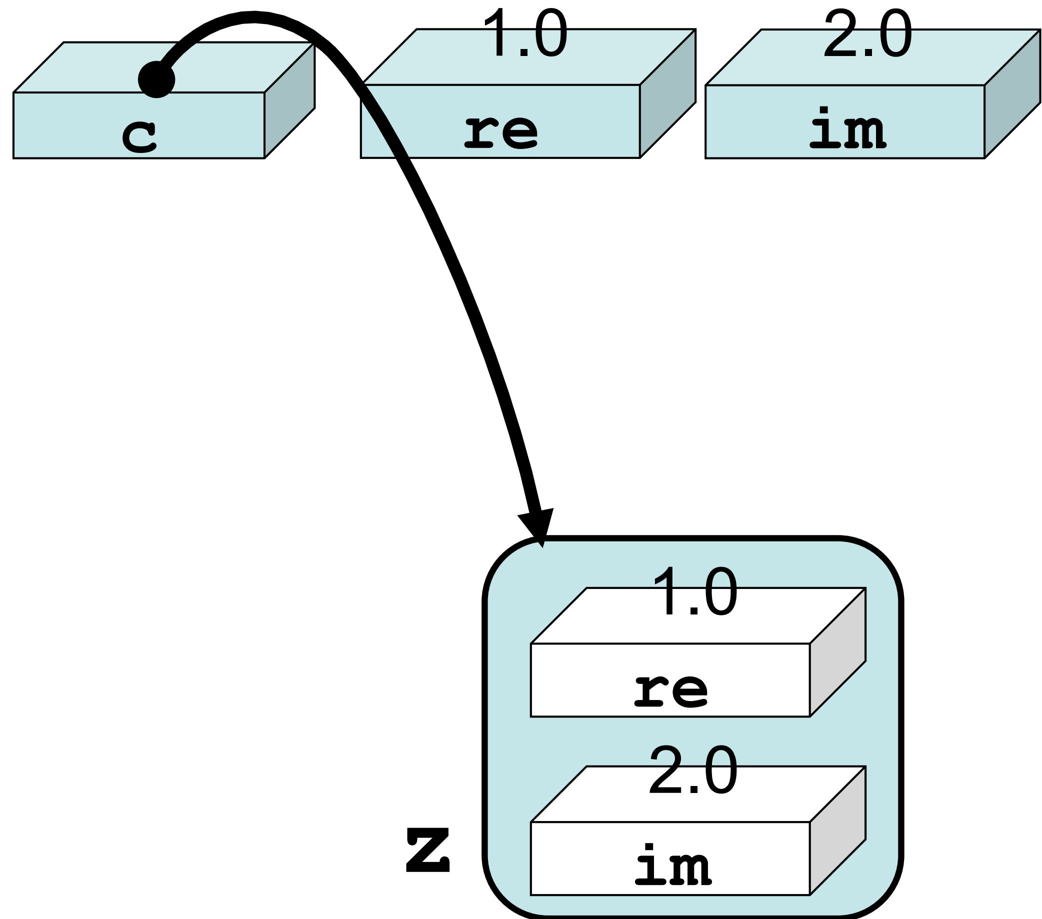




Pas-à-pas

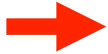


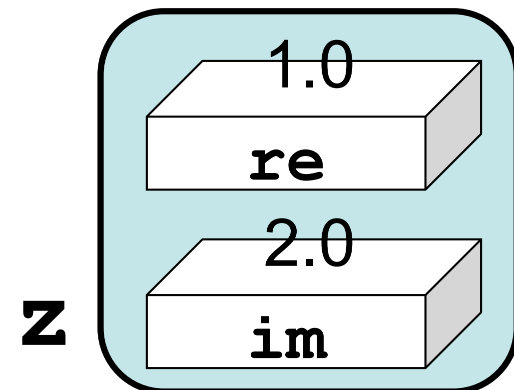
```
void init_comp(Complexe * c, float re, float im)
{
  c->re = re;
  c->im = im;
}
...
Complexe z;
init_comp(&z, 1, 2);
```



Pas-à-pas

```
void init_comp(Complexe * c, float re, float im)
{
  c->re = re;
  c->im = im;
}
...
Complexe z;
init_comp(&z, 1, 2);
```





Fonction d'initialisation Troisième version

```
Complexe * init_comp(float re, float im)
{
  Complexe * res = new Complexe;

  res->re = re;
  res->im = im;

  return res;
}
...
Complexe * z = init_comp(1, 2);
```

`z` est maintenant un pointeur sur `Complexe`.

Nous verrons plus en détail cette autre forme dans la suite.

Fonction `comp_0`

Appelons `comp_0` la fonction qui retourne le complexe 0. On peut l'écrire:

```
Complexe comp_0(void)
{
  Complexe C;

  C.re = 0;
  C.im = 0;

  return C;
}
```

Fonction `comp_0`

Une meilleure façon d'écrire la fonction `comp_0` est de ré-utiliser la fonction `init_comp`:

```
Complexe comp_0(void)
{
  Complexe C;

  C = init_comp(0, 0);

  return C;
}
```

ou, plus directement:

```
Complexe comp_0(void)
{
  return init_comp(0, 0);
}
```

Pas-à-pas

```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}

Complexe comp_0(void)
{
  return init_comp(0, 0);
}
...
```

```
Complexe z0 = comp_0();
```

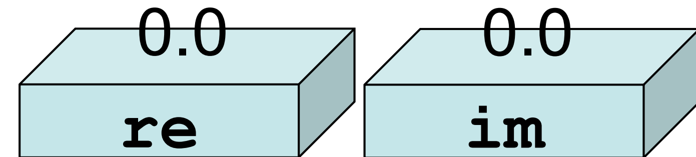
Pas-à-pas

```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}

Complexe comp_0(void)
{
  return init_comp(0, 0);
}
...

Complexe z0 = comp_0();
```

Pas-à-pas

```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}

Complexe comp_0(void)
{
  return init_comp(0, 0);
}
...

Complexe z0 = comp_0();
```

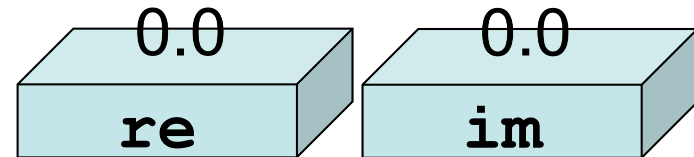
Pas-à-pas

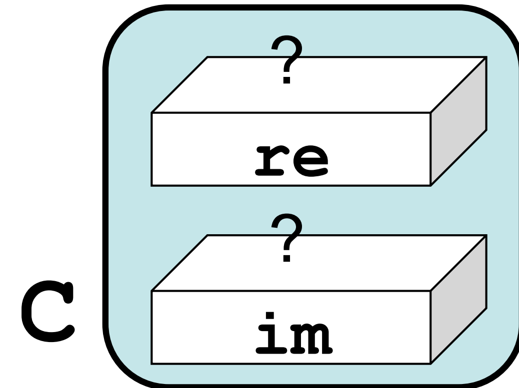



```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}

Complexe comp_0(void)
{
  return init_comp(0, 0);
}
...

Complexe z0 = comp_0();
```

Pas-à-pas

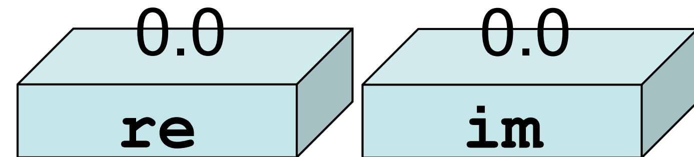
```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}

Complexe comp_0(void)
{
  return init_comp(0, 0);
}
...

Complexe z0 = comp_0();
```

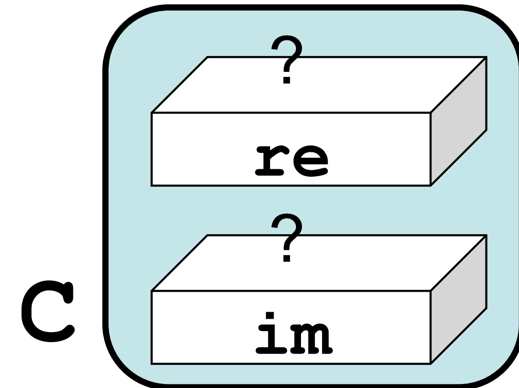




Pas-à-pas

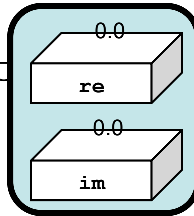
```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}

Complexe comp_0(void)
{
  return init_comp(0, 0);
}
...

Complexe z0 = comp_0();
```





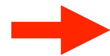
Pas-à-pas

```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}

Complexe comp_0(void)
{
  return init      , 0);
}
...

Complexe z0 = comp_0();
```

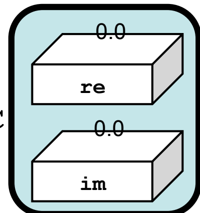


Pas-à-pas

```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}

Complexe comp_0(void)
{
  return init_comp(0, 0);
}
...
```

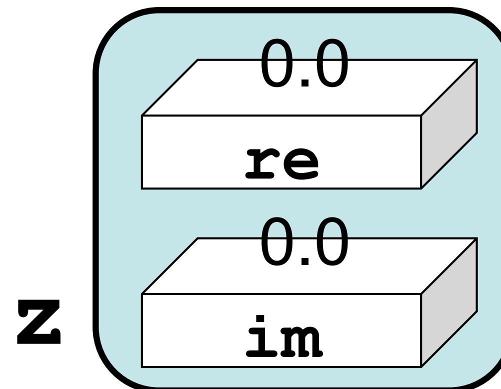


```
Complexe z0 = c
```



Pas-à-pas

```
Complexe init_comp(float re, float im)
{
  Complexe C;

  C.re = re;
  C.im = im;

  return C;
}

Complexe comp_0(void)
{
  return init_comp(0, 0);
}
...

Complexe z0 = comp_0();
```



Fonctions d'initialisation

Fonction qui retourne le complexe 1:

```
Complexe comp_1(void)
{
  return init_comp(1, 0);
}
```

Fonction qui retourne le complexe i:

```
Complexe comp_i(void)
{
  return init_comp(0, 1);
}
```

Fonction qui convertit un `float` en complexe:

```
Complexe comp_r(float r)
{
  return init_comp(r, 0);
}
```

Fonction d'affichage

```
void affiche_comp(Complexe C)
{
  cout << C.re << " + i " << C.im << endl;
}
```

ou, en passant un pointeur sur la structure:

```
void affiche_comp(Complexe * C)
{
  cout << C->re << " + i " << C->im << endl;
}
```

Fonction de lecture

```
Complexe lire_comp(void)
{
  Complexe C;

  cout << "Entrez la partie reelle:" << endl;
  cin >> C.re;
  cout << "Entrez la partie imaginaire" << endl;
  cin >> C.im;

  return C;
}
...
Complexe z = lire_comp();
```

Fonction de lecture

Il vaut mieux utiliser la fonction `init_comp()`:

```
Complexe lire_comp(void)
{
  float re, im;

  cout << "Entrez la partie reelle:" << endl;
  cin >> re;
  cout << "Entrez la partie imaginaire" << endl;
  cin >> im;

  return init_comp(re, im);
}
...
Complexe z = lire_comp();
```

Fonction de lecture
Deuxième version

```
void lire_comp(Complexe * C)
{
  float re, im;

  cout << "Entrez la partie reelle:" << endl;
  cin >> re;
  cout << "Entrez la partie imaginaire" << endl;
  cin >> im;

  init_comp(C, re, im);
}
...
Complexe z;
lire_comp(&z);
```

Fonction Addition

En retournant un résultat:

```
Complexe add_comp(Complexe C1, Complexe C2)
{
  Complexe Res;

  Res.re = C1.re + C2.re;
  Res.im = C1.im + C2.im;

  return Res;
}
```

Fonction Addition

Comme pour la fonction `comp_0()`, on peut utiliser la fonction `init_comp()` pour écrire la fonction `add_comp()` :

```
Complexe add_comp(Complexe C1, Complexe C2)
{
  return init_comp(C1.re + C2.re, C1.im + C2.im);
}
```

Fonction Addition

On peut aussi choisir d'écrire la fonction `add_comp` avec l'en-tête suivant:

```
void add_comp(Complexe * Res, Complexe C1, Complexe C2)
```

La fonction s'écrit alors:

```
void add_comp(Complexe * Res, Complexe C1, Complexe C2)
{
  Res->re = C1.re + C2.re;
  Res->im = C1.im + C2.im;
}
```

ou

```
void add_comp(Complexe * Res, Complexe C1, Complexe C2)
{
  init_comp(Res, C1.re + C2.re, C1.im + C2.im);
}
```

Autres fonctions

Multiplication de deux complexes: $(a+bi)(a'+b'i) = aa' - bb' + i\,(ab' + a'b)$

```
Complexe mult_comp(Complexe C1, Complexe C2)
{
  return init_comp(C1.re * C2.re - C1.im * C2.im,
                   C1.im * C2.re + C1.re * C2.im);
}
```

Multiplication d'un complexe par un réel:

```
Complexe mult_scal_comp(float s, Complexe C)
{
  return init_comp(s * C.re, s * C.im);
}
```

Suite

Utiliser les fonctions précédentes pour afficher les premiers termes de la suite:

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + p \end{cases}$$

où p est un nombre complexe qui sera demandé à l'utilisateur:

```
Complexe z = comp0();
Complexe p = lire_complexe();

for(int i = 0; i < 10; i++)
{
  z = add_comp(mult_comp(z, z), p);
  afficher_comp(z);
}
```

Avec une classe C++, on aurait pu écrire:

```
z = z * z + p;
cout << z << endl;
```

en utilisant ce qui s'appelle la surcharge d'opérateurs.

Allocation dynamique de structures

Exemple

```
Complexe * init_comp(float re, float im)
{
  Complexe * res = new Complexe;

  res->re = re;
  res->im = im;

  return res;
}

...

Complexe * z = init_comp(1, 2);
```

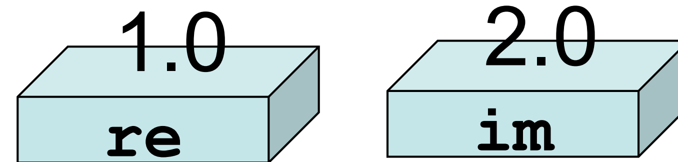
Il faut libérer la structure allouée quand elle n'est plus utile au programme en faisant:

```
delete z;
```

Pas-à-pas

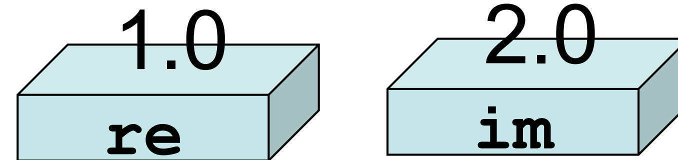
```
Complexe * init_comp(float re, float im)
{
  Complexe * res = new Complexe;

  res->re = re;
  res->im = im;

  return res;
}

...

Complexe * z = init_comp(1, 2);
```

Pas-à-pas



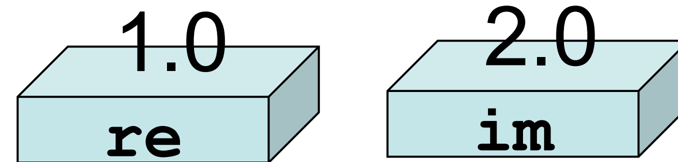

```
Complexe * init_comp(float re, float im)
{
  Complexe * res = new Complexe;

  res->re = re;
  res->im = im;

  return res;
}

...

Complexe * z = init_comp(1, 2);
```

Pas-à-pas

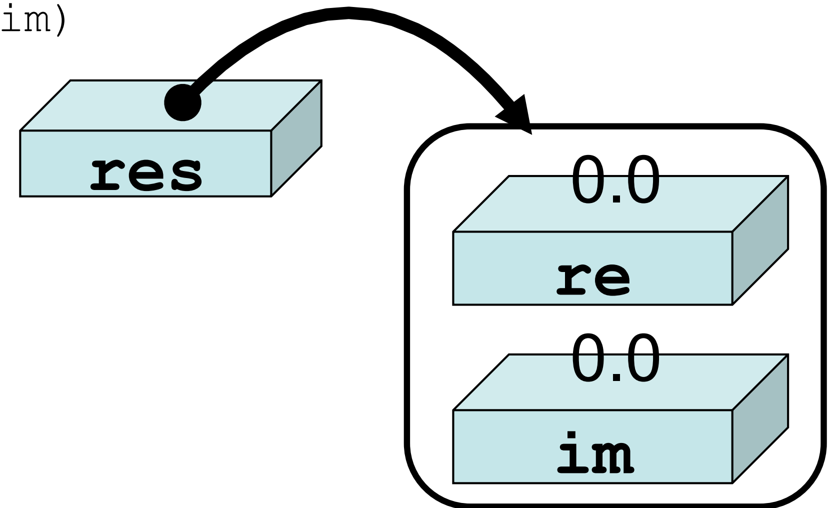



```
Complexe * init_comp(float re, float im)
{
    Complexe * res = new Complexe;

    res->re = re;
    res->im = im;

    return res;
}

...

Complexe * z = init_comp(1, 2);
```

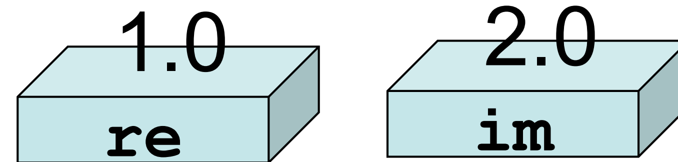
Pas-à-pas



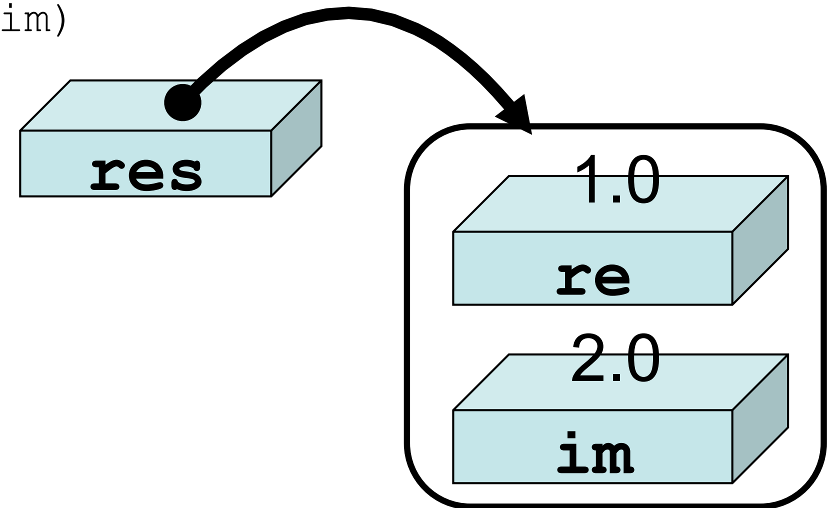

```
Complexe * init_comp(float re, float im)
{
  Complexe * res = new Complexe;

  res->re = re;
  res->im = im;

  return res;
}

...

Complexe * z = init_comp(1, 2);
```



Pas-à-pas

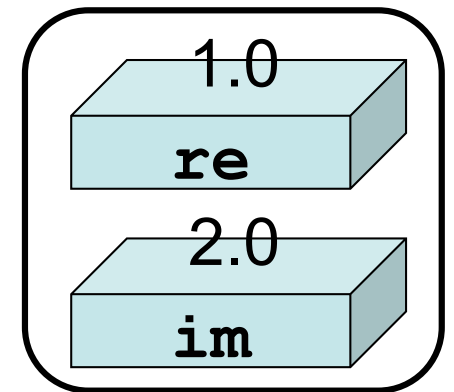


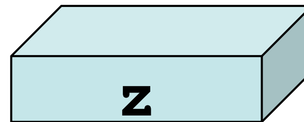

```
Complexe * init_comp(float re, float im)
{
  Complexe * res = new Complexe;

  res->re = re;
  res->im = im;

  return res;
}

...

Complexe * z = init_comp(1, 2);
```



Pas-à-pas

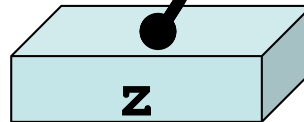
```
Complexe * init_comp(float re, float im)
{
  Complexe * res = new Complexe;

  res->re = re;
  res->im = im;

  return res;
}

...

Complexe * z = init_comp(1, 2);
```

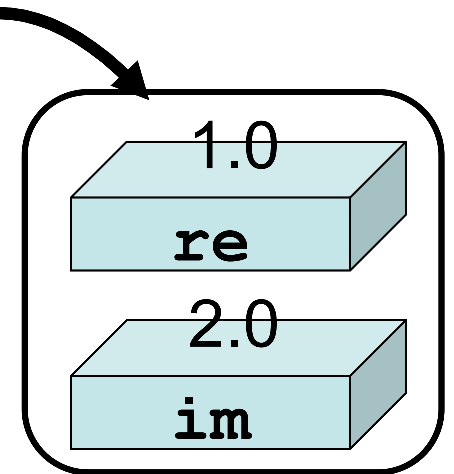




Pas-à-pas

```
Complexe * init_comp(float re, float im)
{
  Complexe * res = new Complexe;

  res->re = re;
  res->im = im;

  return res;
}

...

Complexe * z = init_comp(1, 2);
```





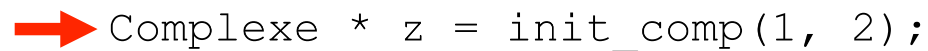
Version incorrecte

```
Complexe * init_comp(float re, float im)
{
  Complexe res;

  res.re = re;
  res.im = im;

  return &res;
}

...

Complexe * z = init_comp(1, 2);
```

Cette version compile, mais ne fonctionne pas.

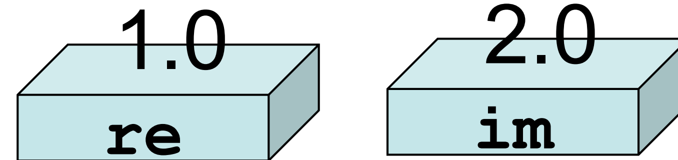
Version incorrecte
Pas-à-pas

```
Complexe * init_comp(float re, float im)
{
  Complexe res;

  res.re = re;
  res.im = im;

  return &res;
}

...
```



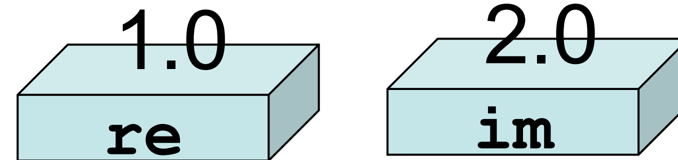
```
Complexe * z = init_comp(1, 2);
```

Version incorrecte pas-à-pas

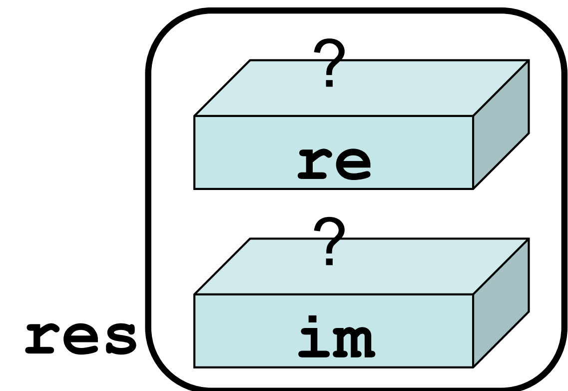



```
Complexe * init_comp(float re, float im)
{
  Complexe res;

  res.re = re;
  res.im = im;

  return &res;
}

...

Complexe * z = init_comp(1, 2);
```

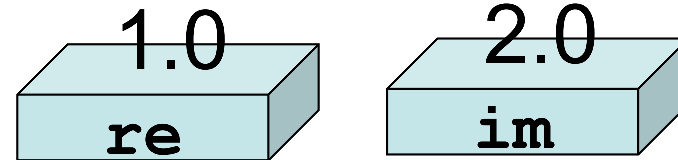
Version incorrecte pas-à-pas



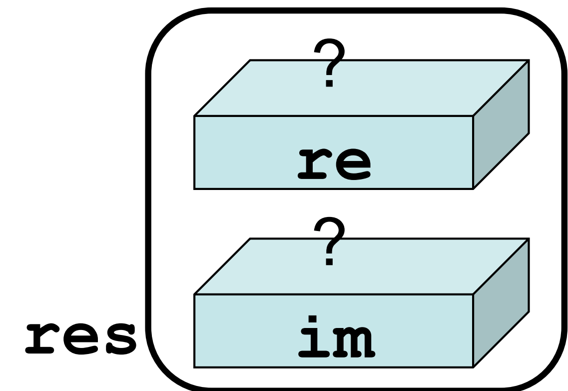

```
Complexe * init_comp(float re, float im)
{
  Complexe res;

  res.re = re;
  res.im = im;

  return &res;
}

...

Complexe * z = init_comp(1, 2);
```



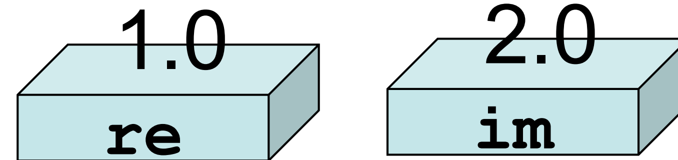
Version incorrecte pas-à-pas



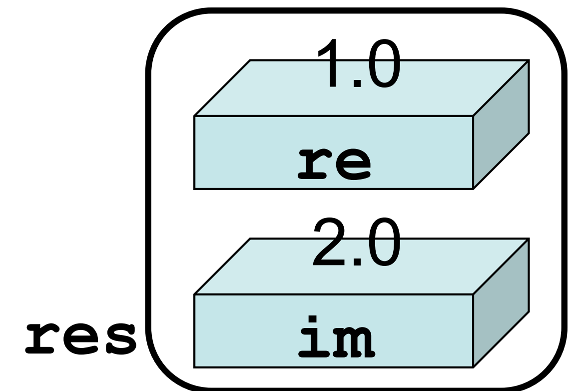

```
Complexe * init_comp(float re, float im)
{
  Complexe res;

  res.re = re;
  res.im = im;

  return &res;
}

...

Complexe * z = init_comp(1, 2);
```



Version incorrecte pas-à-pas

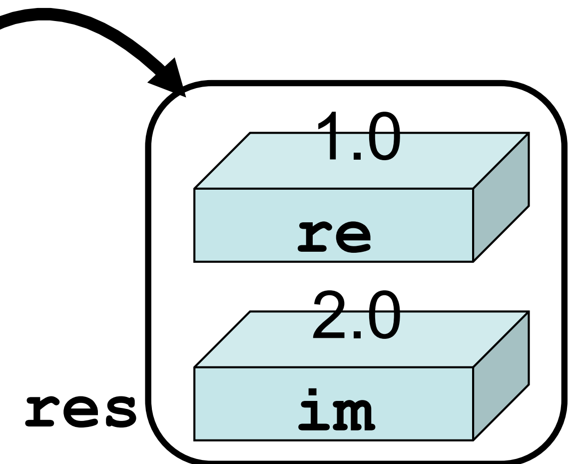


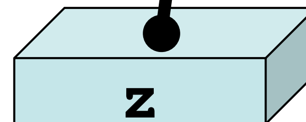

```
Complexe * init_comp(float re, float im)
{
  Complexe res;

  res.re = re;
  res.im = im;

  return &res;
}

...

Complexe * z = init_comp(1, 2);
```



Version incorrecte pas-à-pas

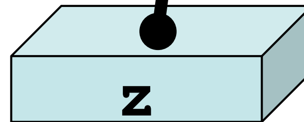
```
Complexe * init_comp(float re, float im)
{
  Complexe res;

  res.re = re;
  res.im = im;

  return &res;
}

...

Complexe * z = init_comp(1, 2);
```





Version incorrecte pas-à-pas

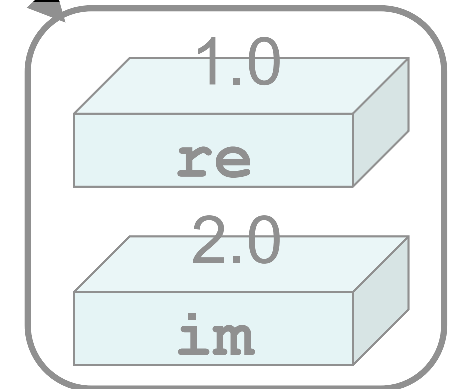
```
Complexe * init_comp(float re, float im)
{
  Complexe res;

  res.re = re;
  res.im = im;

  return &res;
}

...

Complexe * z = init_comp(1, 2);
```

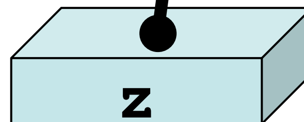


Version incorrecte pas-à-pas

Mais `res` est ici une variable locale, qui est détruite quand on sort de la fonction `init_comp`.
Plus précisément, la partie de la mémoire qu'occupait `init_comp` va sans doute être réutilisée dans la suite du programme pour stocker d'autres valeurs.

```
Complexe * init_comp(float re, float im)
{
  Complexe res;

  res.re = re;
  res.im = im;

  return &res;
}

...

Complexe * z = init_comp(1, 2);
```





Attention à ne jamais retourner un pointeur sur une variable locale !

Allocation dynamique d'un tableau de structures

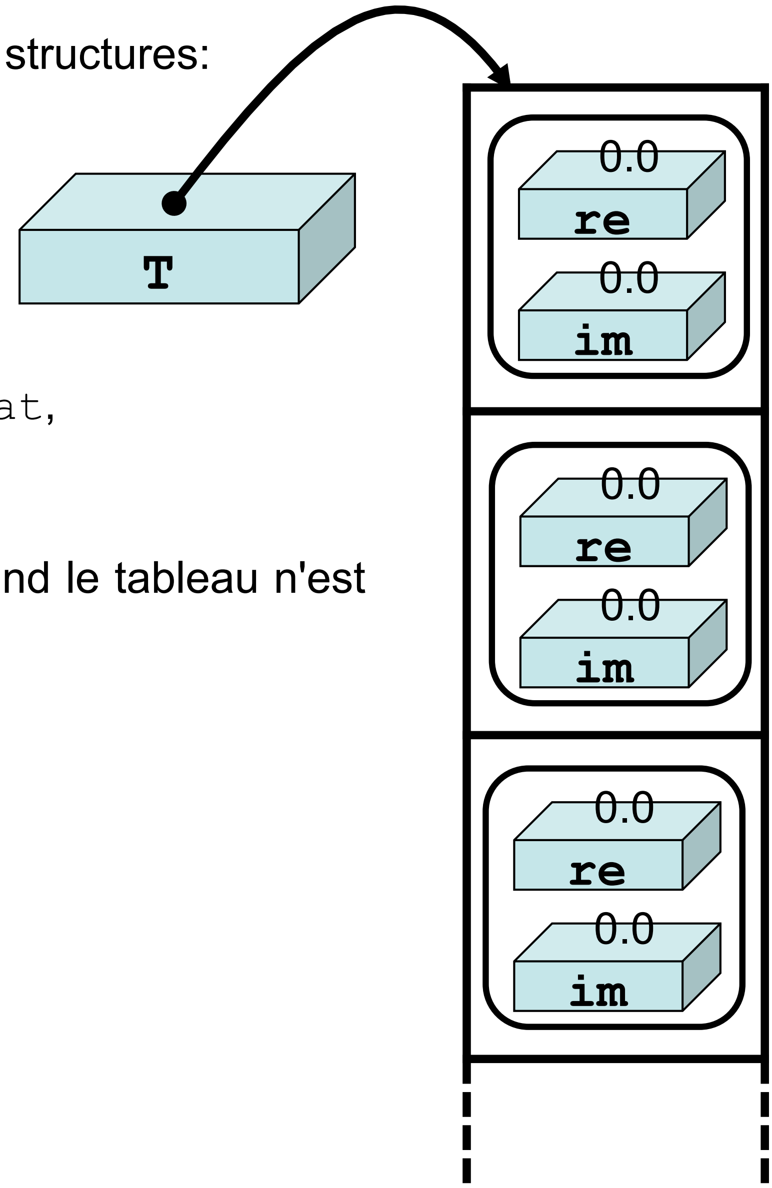
On peut aussi allouer dynamiquement un tableau de structures:

```
Complexe * T;

T = new Complexe[10];
```

alloue de la place en mémoire pour 10 structures `Chat`,
et `elevage` pointe sur la première structure.

Attention à ne pas oublier de libérer la mémoire quand le tableau n'est plus utile:

```
delete [] T;
```

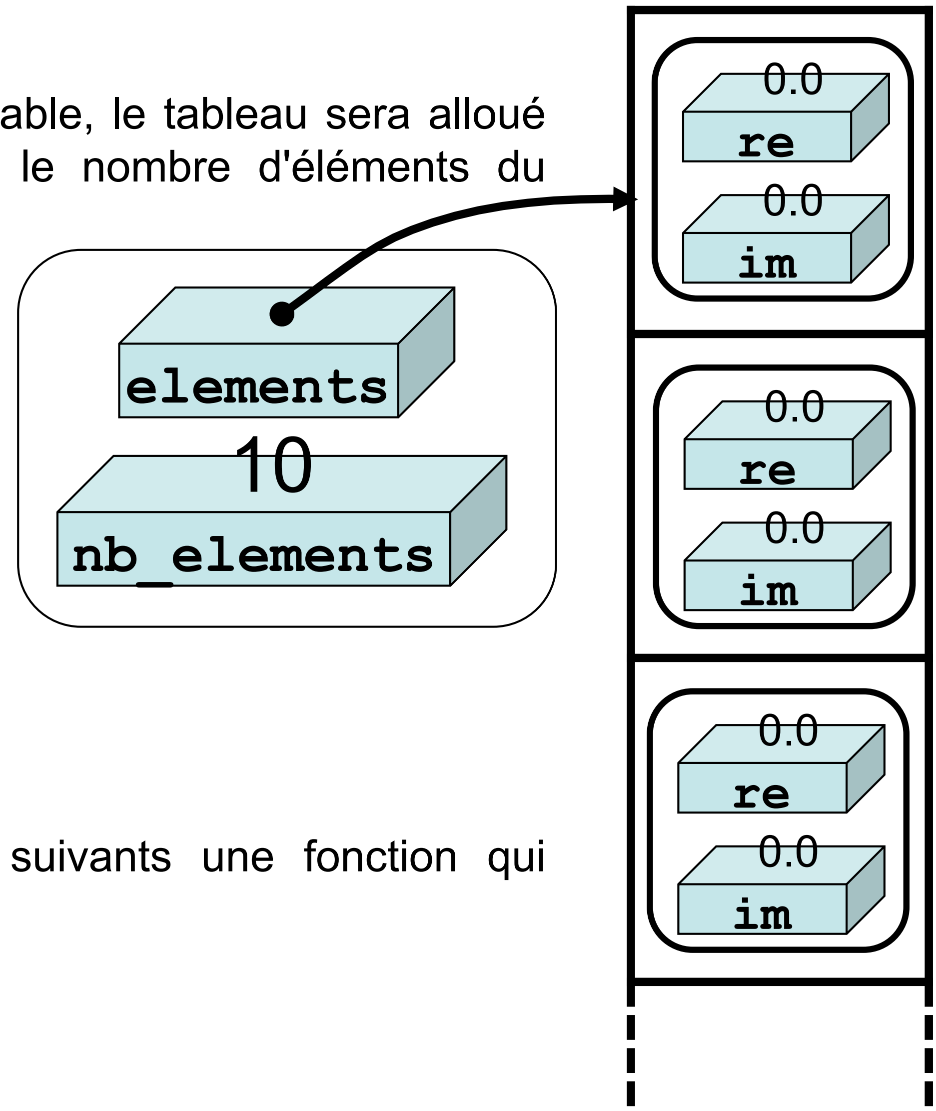


Structure `Tableau_Complexes`

On souhaite maintenant définir une structure contenant un tableau dont les éléments sont de type `Complexe`.

Pour permettre un nombre d'éléments variable, le tableau sera alloué dynamiquement, et la structure contiendra le nombre d'éléments du tableau:

```
struct Tableau_Complexes
{
  Complexe * elements;
  int nb_elements;
};
```



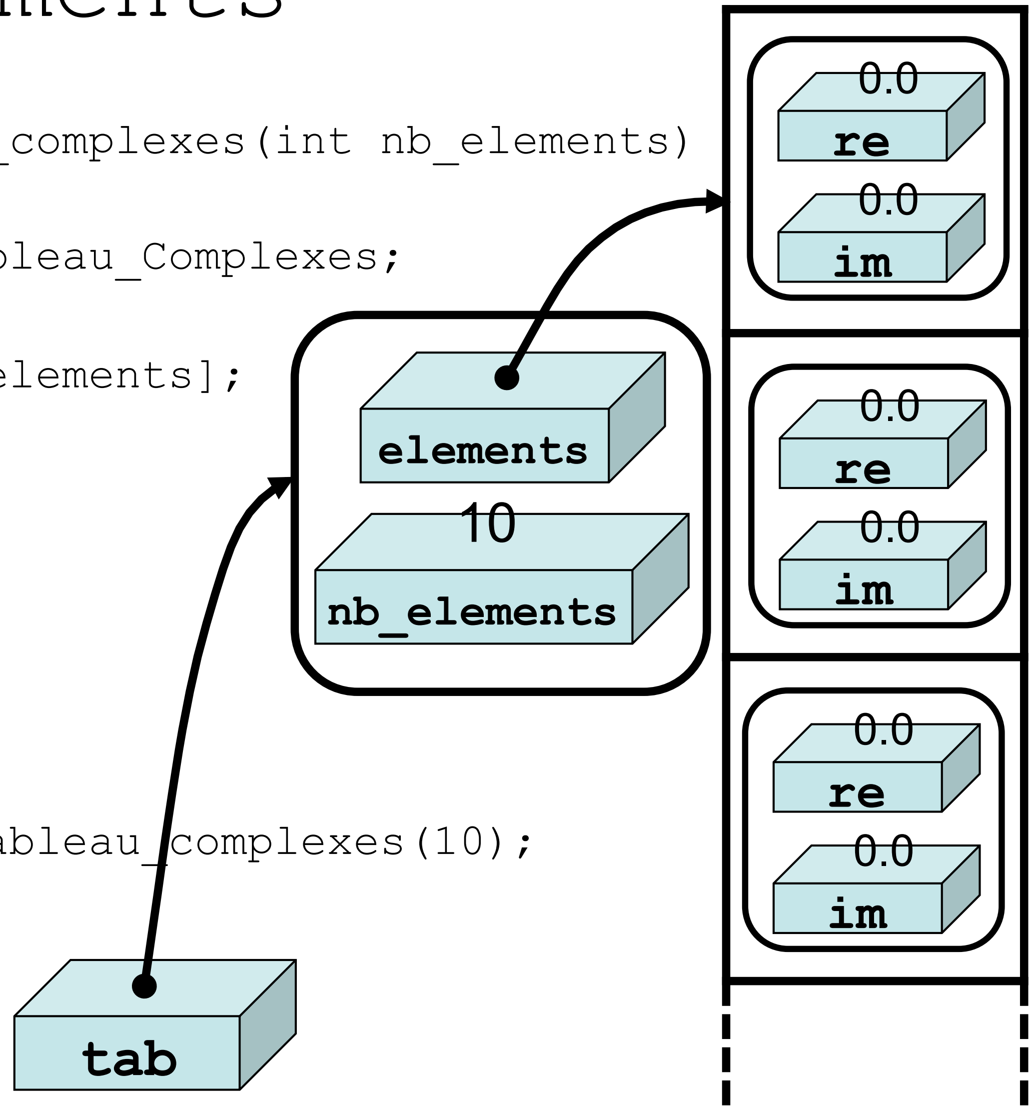
Nous allons voir dans les transparents suivants une fonction qui permet d'allouer une telle structure.

Fonction qui alloue une structure `Tableau_Complexes` et son champs `elements`

```
Tableau_Complexes * alloue_tableau_complexes(int nb_elements)
{
  Tableau_Complexes * res = new Tableau_Complexes;

  res->elements = new Complexe[nb_elements];
  res->nb_elements = nb_elements;

  return res;
}
```

Fonction qui s'utilise de la façon suivante:

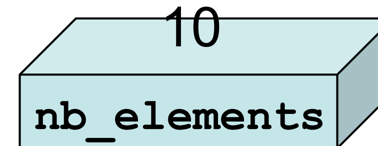
```
Tableau_Complexes * tab = alloue_tableau_complexes(10);
```



Pas-à-pas

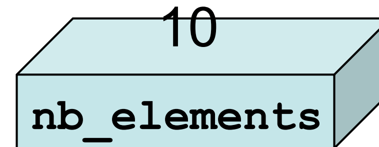
```
Tableau_Complexes * alloue_tableau_complexes(int nb_elements)
{
  Tableau_Complexes * res = new Tableau_Complexes;

  res->elements = new Complexe[nb_elements];
  res->nb_elements = nb_elements;

  return res;
}

...

Tableau_Complexes * tab = alloue_tableau_complexes(10);
```

Pas-à-pas

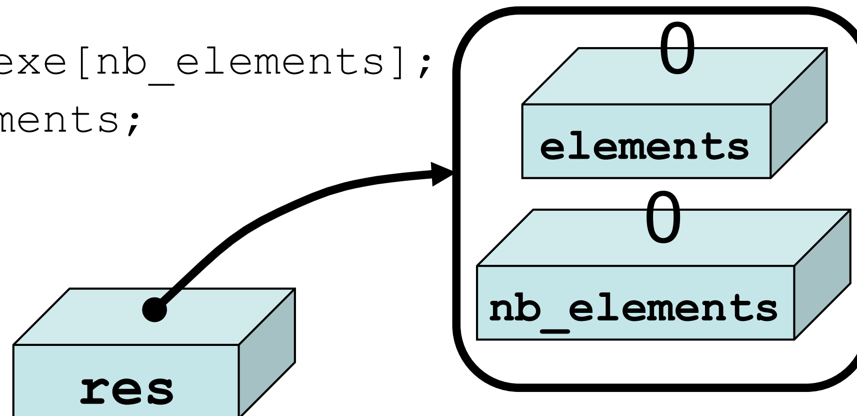



```
Tableau_Complexes * alloue_tableau_complexes(int nb_elements)
{
  Tableau_Complexes * res = new Tableau_Complexes;

  res->elements = new Complexe[nb_elements];
  res->nb_elements = nb_elements;

  return res;
}

…

Tableau_Complexes * tab = alloue_tableau_complexes(10);
```

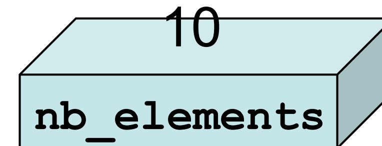
Pas-à-pas



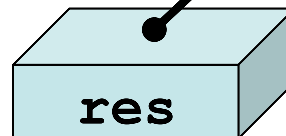

```
Tableau_Complexes * alloue_tableau_complexes(int nb_elements)
{
  Tableau_Complexes * res = new Tableau_Complexes;

  res->elements = new Complexe[nb_elements];
  res->nb_elements = nb_elements;

  return res;
}

…

Tableau_Complexes * tab = alloue_tableau_complexes(10);
```

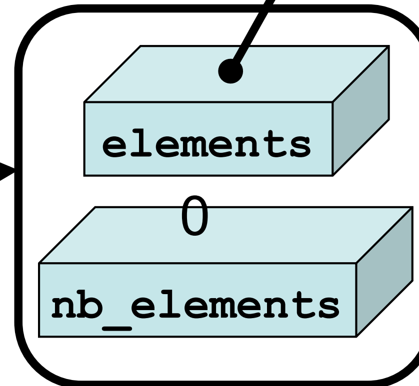


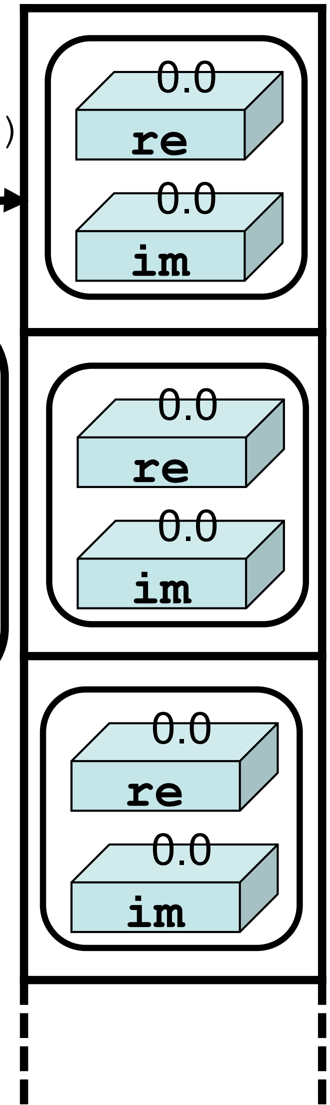
Pas-à-pas



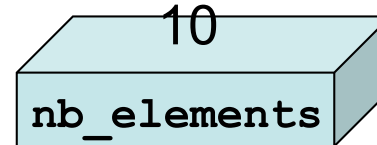

```
Tableau_Complexes * alloue_tableau_complexes(int nb_elements)
{
  Tableau_Complexes * res = new Tableau_Complexes;

  res->elements = new Complexe[nb_elements];
  res->nb_elements = nb_elements;

  return res;
}

...

Tableau_Complexes * tab = alloue_tableau_complexes(10);
```



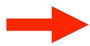


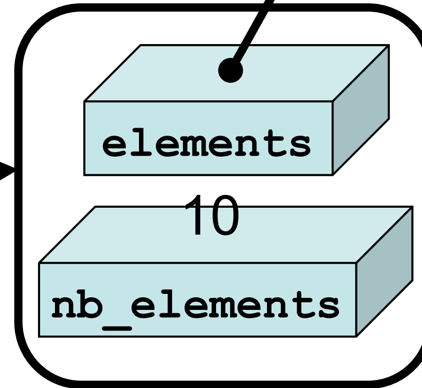


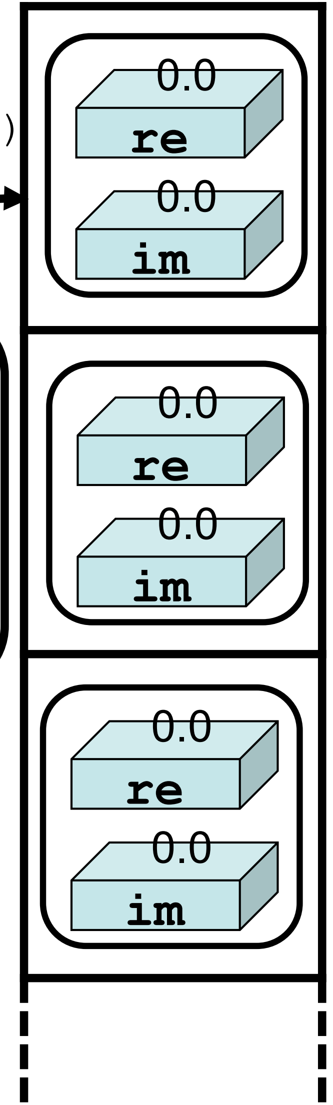
Pas-à-pas

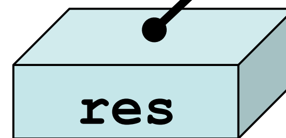



```
Tableau_Complexes * alloue_tableau_complexes(int nb_elements)
{
  Tableau_Complexes * res = new Tableau_Complexes;

  res->elements = new Complexe[nb_elements];
  res->nb_elements = nb_elements;

  return res;
}
```





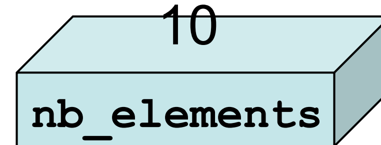




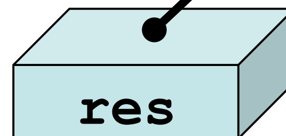
...

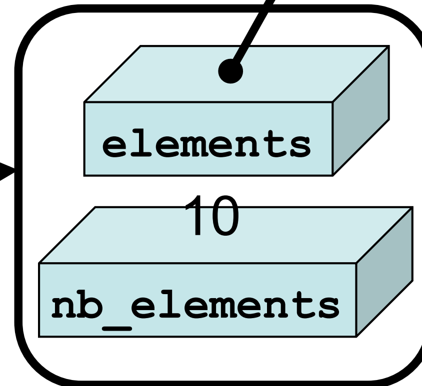
```
Tableau_Complexes * tab = alloue_tableau_complexes(10);
```

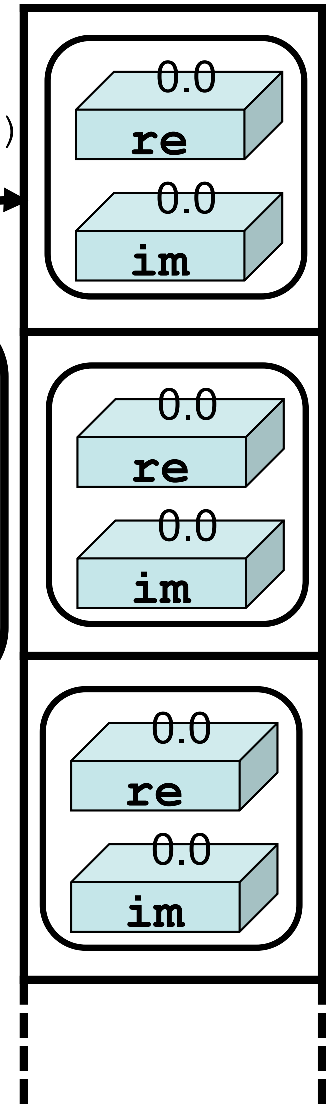
Pas-à-pas




```
Tableau_Complexes * alloue_tableau_complexes(int nb_elements)
{
  Tableau_Complexes * res = new Tableau_Complexes;

  res->elements = new Complexe[nb_elements];
  res->nb_elements = nb_elements;

  return res;
}

...

Tableau_Complexes * tab = alloue_tableau_complexes(10);
```

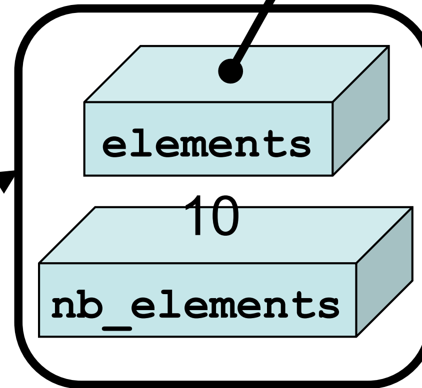


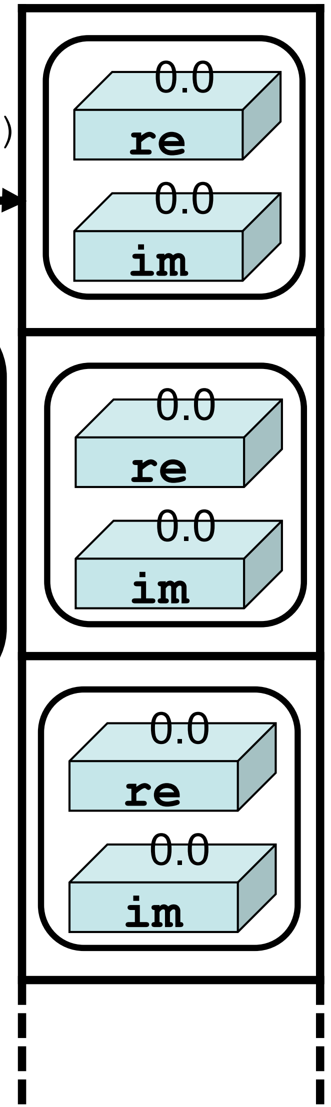


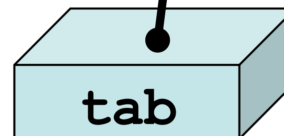


Pas-à-pas

```
Tableau_Complexes * alloue_tableau_complexes(int nb_elements)
{
  Tableau_Complexes * res = new Tableau_Complexes;

  res->elements = new Complexe[nb_elements];
  res->nb_elements = nb_elements;

  return res;
}

...

Tableau_Complexes * tab = alloue_tableau_complexes(10);
```







Piège 1

IL NE FAUT PAS FAIRE:

```
Tableau_Complexes * alloue_tableau_complexes(int nb_elements)
{
  Tableau_Complexes res;

  res.elements = new Complexe[nb_elements];
  res.nb_elements = nb_elements;

  return &res;
}
```

Cette fonction est acceptée par le compilateur mais elle n'est pas correcte.
Il s'agit de la même erreur que pour la version incorrecte de la fonction `init_comp`:
la fonction retourne un pointeur sur la variable locale `res`.

Piège 2

IL NE FAUT PAS FAIRE:

```
Tableau_Complexes * alloue_tableau_complexes(int nb_elements)
{
  Tableau_Complexes * res;

  res->elements = new Complexe[nb_elements];
  res->nb_elements = nb_elements;

  return res;
}
```

Cette fonction est aussi acceptée par le compilateur, elle n'est pas correcte non plus. Elle ne fait pas l'allocation dynamique de la structure `Tableau_Complexes`.

Piège 2

IL NE FAUT PAS FAIRE:

```
Tableau_Complexes * alloue_tableau_complexes(int nb_elements)
{
  Tableau_Complexes * res;

  res->elements = new Complexe[nb_elements];
  res->nb_elements = nb_elements;

  return res;
}
```

Quand les instructions

`res->elements =... ou res->nb_elements = ...`

sont exécutées, deux choses peuvent survenir:

- si `res` pointe sur une zone mémoire qui n'appartient pas au programme, une de ces instructions vont provoquer un `Segmentation fault`;

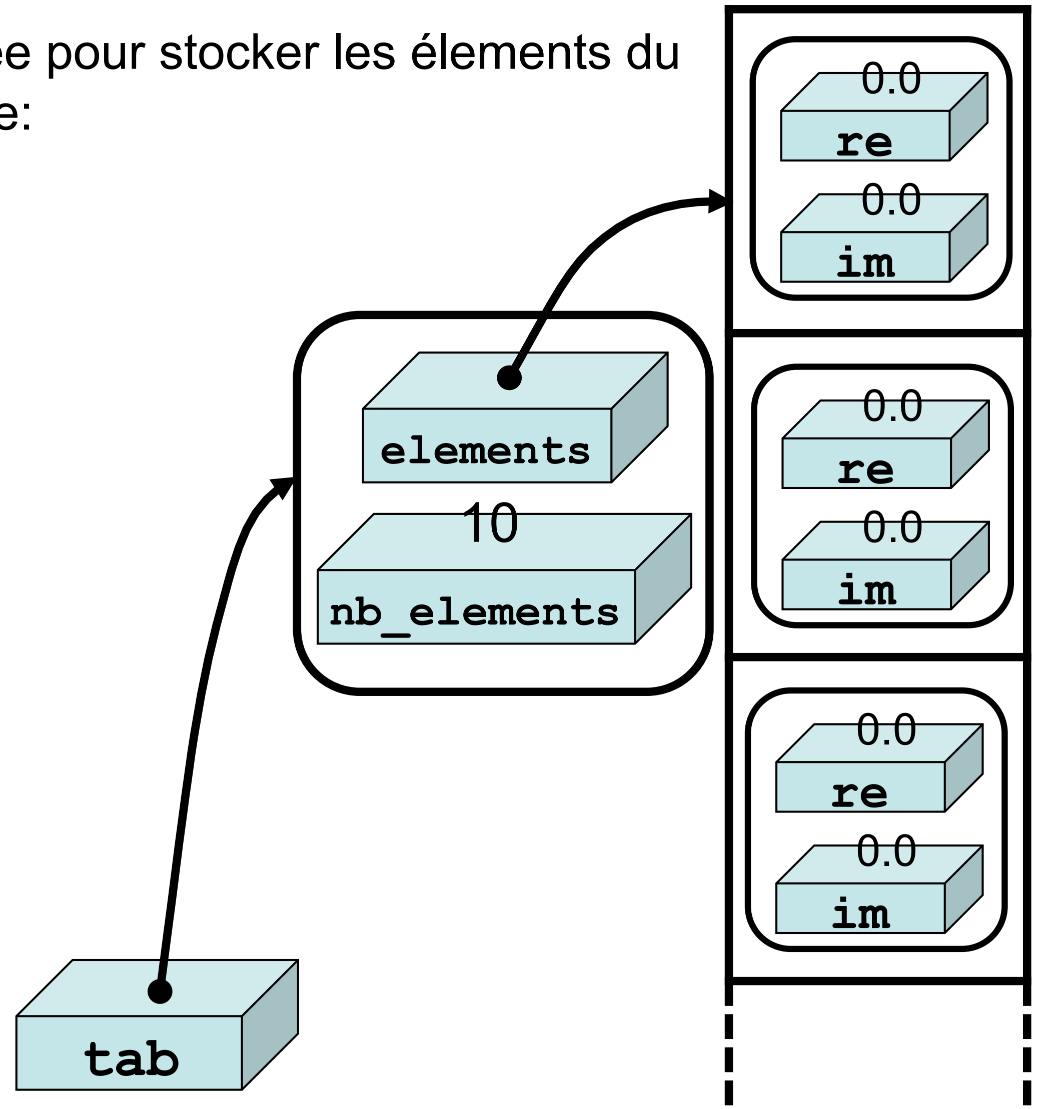
- sinon `res` pointe sur une zone mémoire appartenant au programme, et ces instructions vont écraser des données du programme.

Libération d'une structure

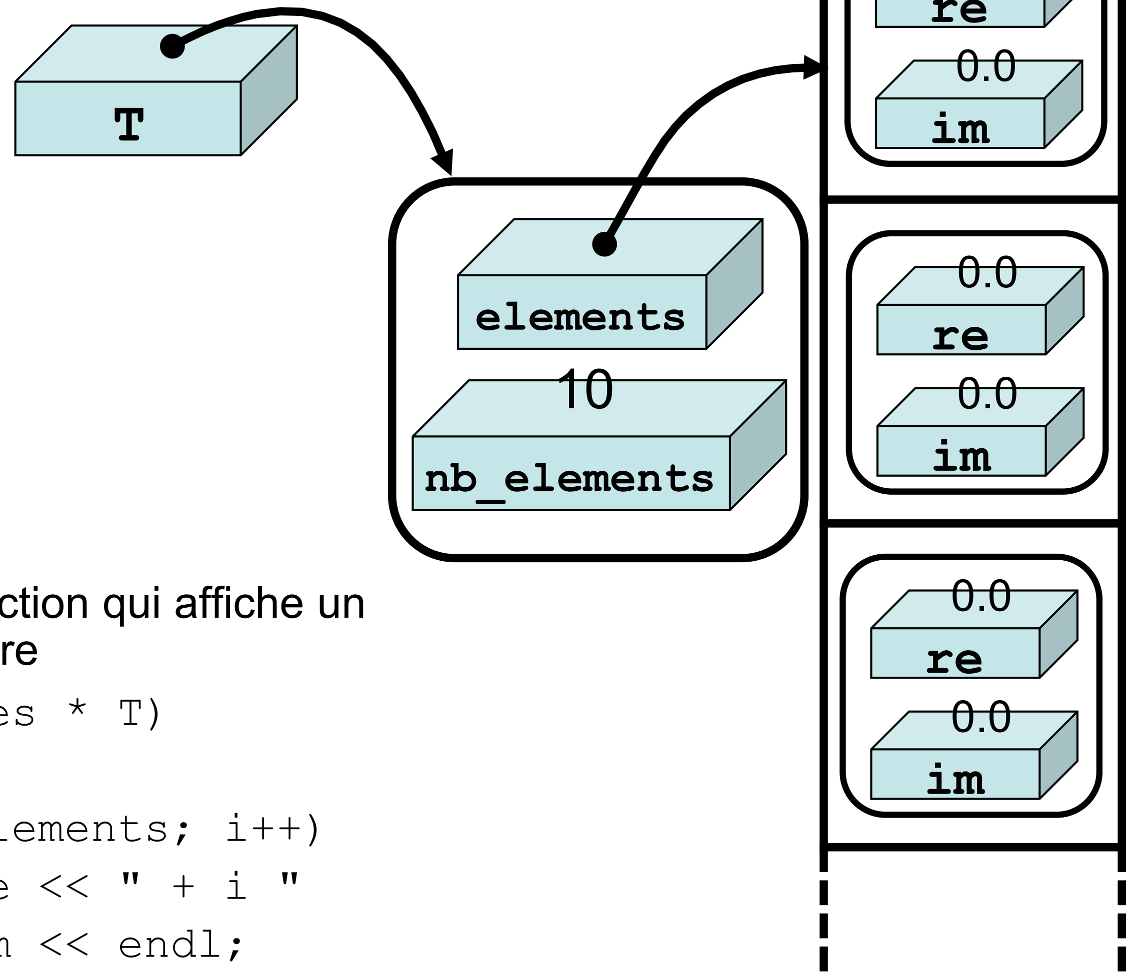
`Tableau_Complexes`

Il faut d'abord libérer la mémoire allouée pour stocker les élements du tableau, et ensuite la structure elle-même:

```
delete [] tab->elements;
delete tab;
```



affiche Tableau_Complexes



On souhaite maintenant écrire une fonction qui affiche un tableau de complexes passé en paramètre

```
void affiche(Tableau_Complexes * T)
{
  for(int i = 0; i < T->nb_elements; i++)
    cout << T->elements[i].re << " + i "
         << T->elements[i].im << endl;
}

...

affiche(tab);
```

. ou -> ?

A gauche de `->` on trouve forcément un *pointeur* sur structure.

A gauche de `.` on trouve forcément une *variable* de type structure:

```
void affiche(Tableau_Complexes * T)
{
  for(int i = 0; i < T->nb_elements; i++)
    cout << T->elements[i].re << " + i "
         << T->elements[i].im << endl;
}
```

`T` est un pointeur

`T->elements` est un pointeur MAIS:

`T->elements[i]` est une *variable* de type structure.

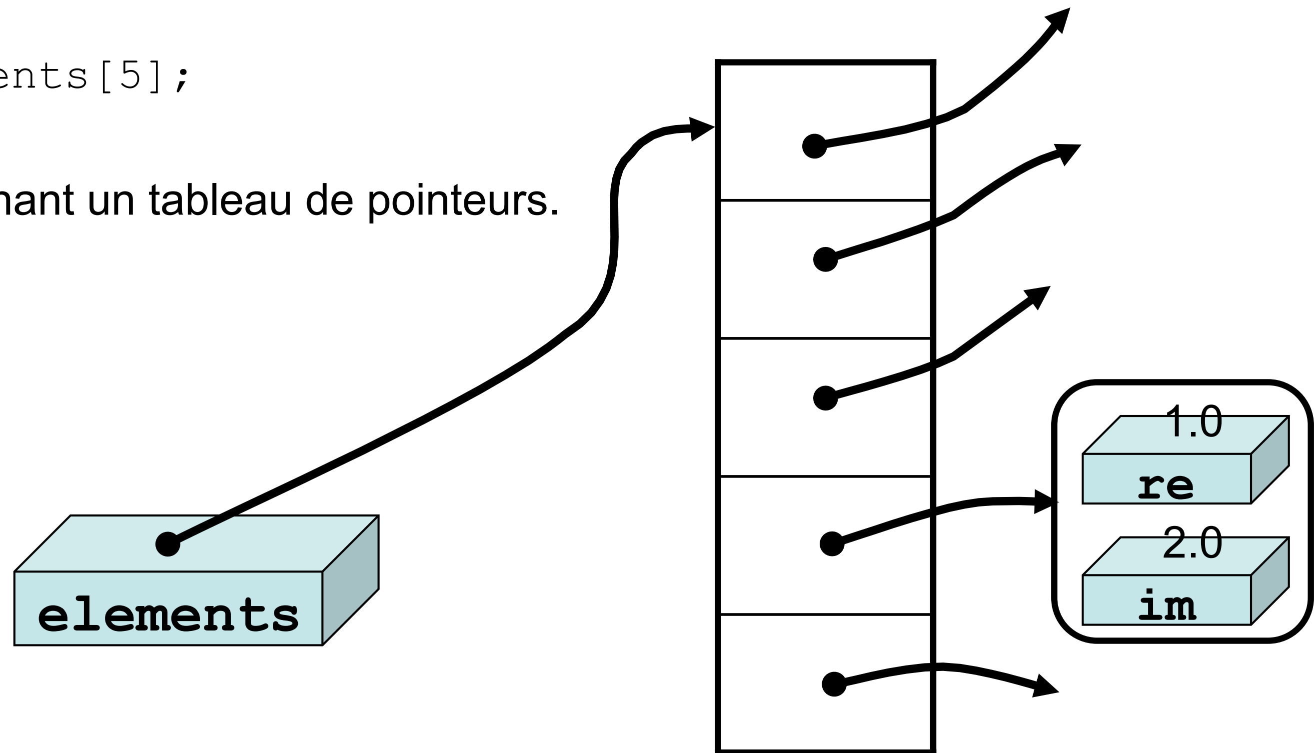
MAIS ATTENTION !!! (voir transparent suivant...)

. ou -> ?

Attention, si on fait:

```
Complexes * elements[5];
```

`elements` est maintenant un tableau de pointeurs.



`elements[3]` est maintenant un pointeur sur une structure `Complexe`.

Pour afficher le complexe correspondant, il faut faire:

```
cout << elements[3]->re << " + i " << elements[3]->im << endl;
```

Utiliser la fonction `affiche_comp` dans la fonction `affiche`

On a déjà écrit une fonction qui affiche un complexe:

```
void affiche_comp(Complexe * C)
{
  cout << C->re << " + i " << C->im << endl;
}
```

Il vaut mieux l'utiliser pour écrire la fonction `affiche`:

```
void affiche(Tableau_Complexes * T)
{
  for(int i = 0; i < T->nb_elements; i++)
    affiche_comp(&(T->elements[i]));
}
```

ou mieux:

```
void affiche(Tableau_Complexes * T)
{
  for(int i = 0; i < T->nb_elements; i++)
    affiche_comp(T->elements + i);
}
```