

IN 101 - Cours 08

4 novembre 2011



présenté par

Matthieu Finiasz

Structures en C

IN101 - 2011-2012

- ✘ Les structures sont une façon de définir de nouveaux types en C :
 - ✘ des **types composés** de plusieurs éléments d'autres types,
 - ✘ chaque sous élément **a un nom** → accessible avec **.**

```
1 struct complex {
2     double re;
3     double im;
4 };                               // attention au ;
```

- ✘ Les structures s'utilisent ensuite exactement comme les types natifs

```
1 struct complex complex_add(struct complex a,
2                             struct complex b) {
3     struct complex res;        // déclaration
4     res.re = a.re + b.re;     // accès à la partie re
5     res.im = a.im + b.im;
6     return res;               // valeur de retour
7 }
```

- ✘ Taper *struct* avant le type à chaque fois est un peu lourd
 - ✘ on utilise en général un `typedef` → crée un alias

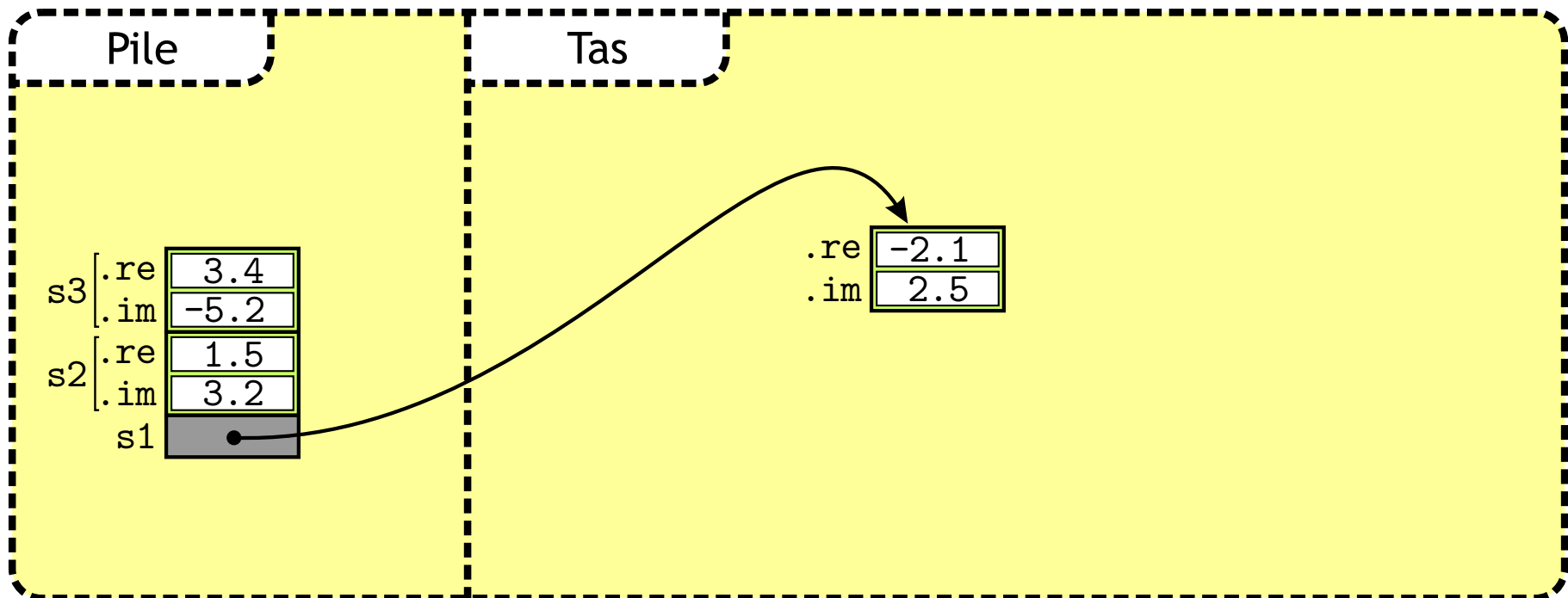
```
1 struct complex_st {  
2     double re;  
3     double im;  
4 };  
5 typedef struct complex_st complex;
```

- ✘ On peut aussi utiliser le `typedef` directement à la définition :

```
1 typedef struct {  
2     double re;  
3     double im;  
4 } complex;
```

- ✘ La structure s'utilise alors directement avec `complex res;`

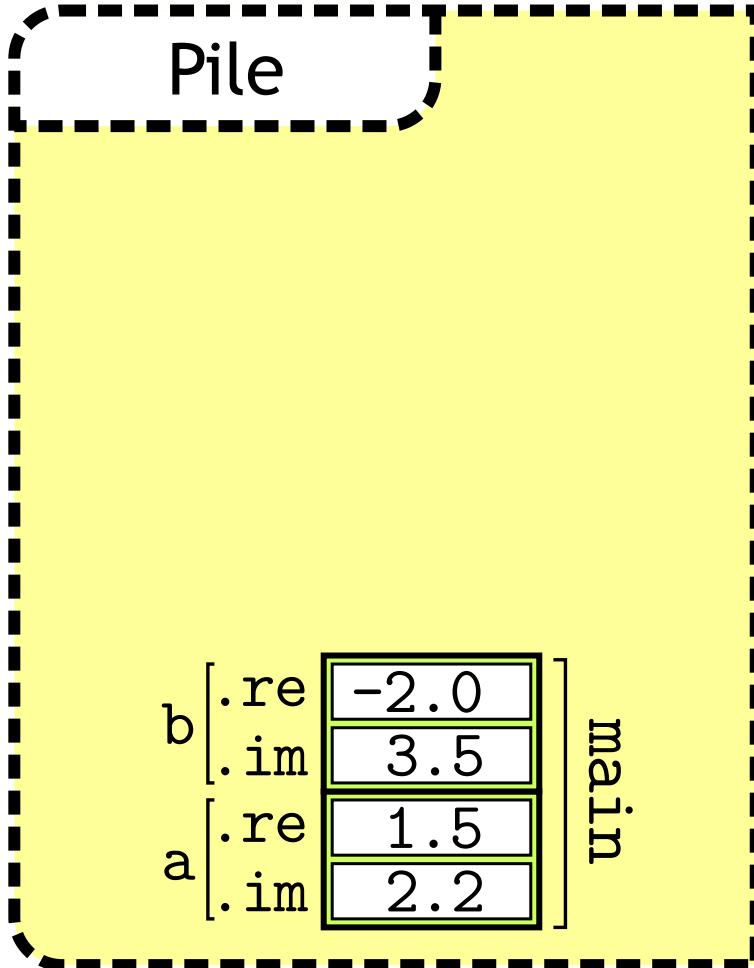
- ✘ Une structure est un “objet” composée de plusieurs parties
 - ✘ c’est la même chose en mémoire.
- ✘ On peut aussi utiliser `sizeof` sur une structure
 - renvoie le nombre d’octet nécessaire à son stockage (au moins la somme des `sizeof` de ses parties).



```
1 typedef struct {
2     unsigned int l,c;      // lignes/colonnes
3     double** coef;        // coefficients
4 } matrix;
5
6 typedef struct {
7     unsigned int epoch_sec;
8     int utc_offset;
9 } date;
10
11 typedef struct {
12     char* nom;
13     char* prenom;
14     int sexe;
15     int promo;
16 } eleve;
```

- ✘ Les structures sont gérées comme des types natifs :
 - ✘ elles sont donc **recopiées** quand on les passe en argument,
 - ✘ elles peuvent être des variables locales
 - libérées automatiquement en fin de fonction.
- ✘ C'est pratique, mais c'est assez cher
 - ✘ pour les petites structures copier n'est pas trop cher,
 - ✘ pour des grosses structures, mieux vaut utiliser des pointeurs.
- ✘ Attention, seul le **contenu de la structure** est copié, pas ce sur quoi elle pointe !

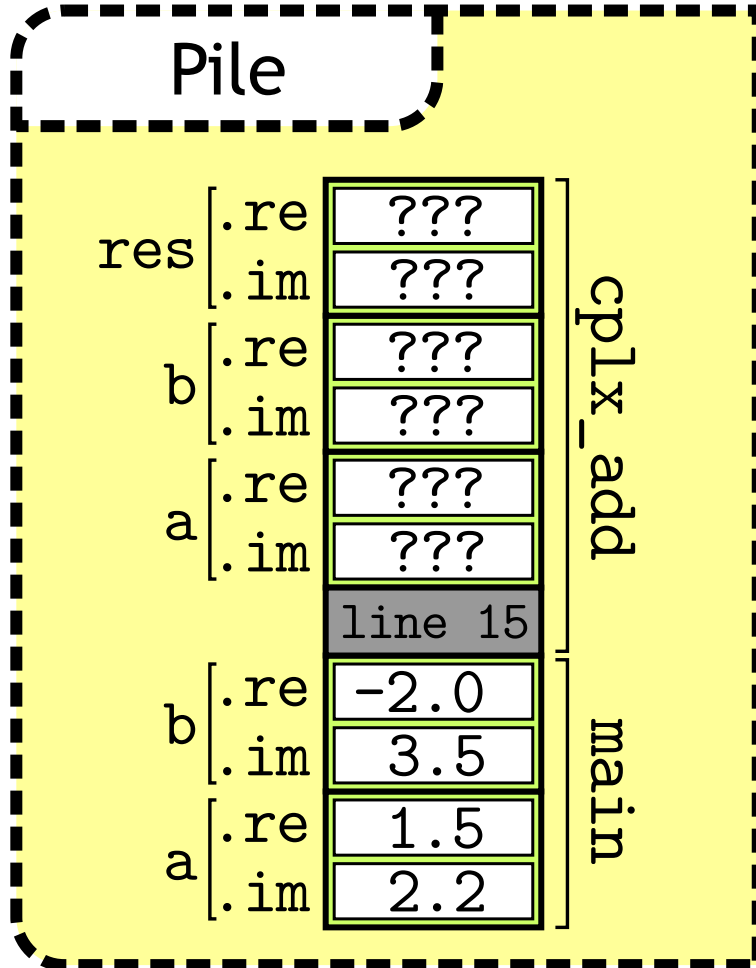
```
1 struct table {  
2     int size;  
3     int* tab;  
4 };
```



```

1 typedef struct {
2     double re, im;
3 } complex;
4
5 complex cplx_add(complex a,
6                 complex b) {
7     complex res;
8     res.re = a.re + b.re;
9     res.im = a.im + b.im;
10    return res;
11 }
12 int main() {
13     ► complex a = {1.5, 2.2};
14     complex b = {-2, 3.5};
15     a = cplx_add(a,b);
16     return 0;
17 }

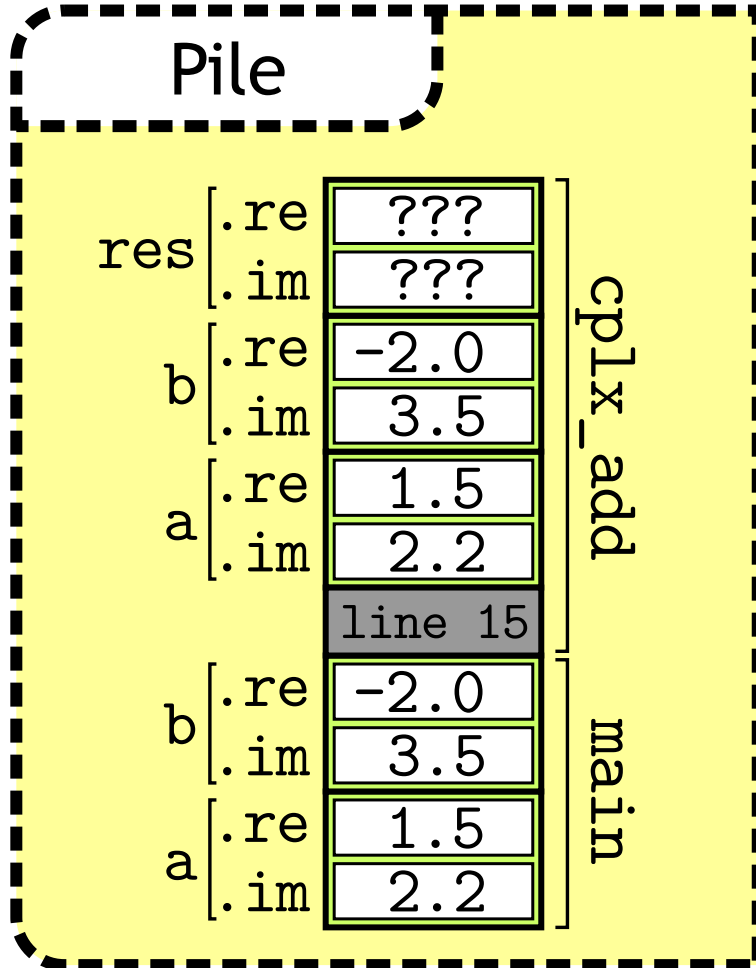
```

```

1 typedef struct {
2     double re, im;
3 } complex;
4
5 complex cplx_add(complex a,
6                 complex b) {
7     complex res;
8     res.re = a.re + b.re;
9     res.im = a.im + b.im;
10    return res;
11 }
12 int main() {
13     complex a = {1.5, 2.2};
14     complex b = {-2, 3.5};
15     ▶ a = cplx_add(a,b);
16     return 0;
17 }

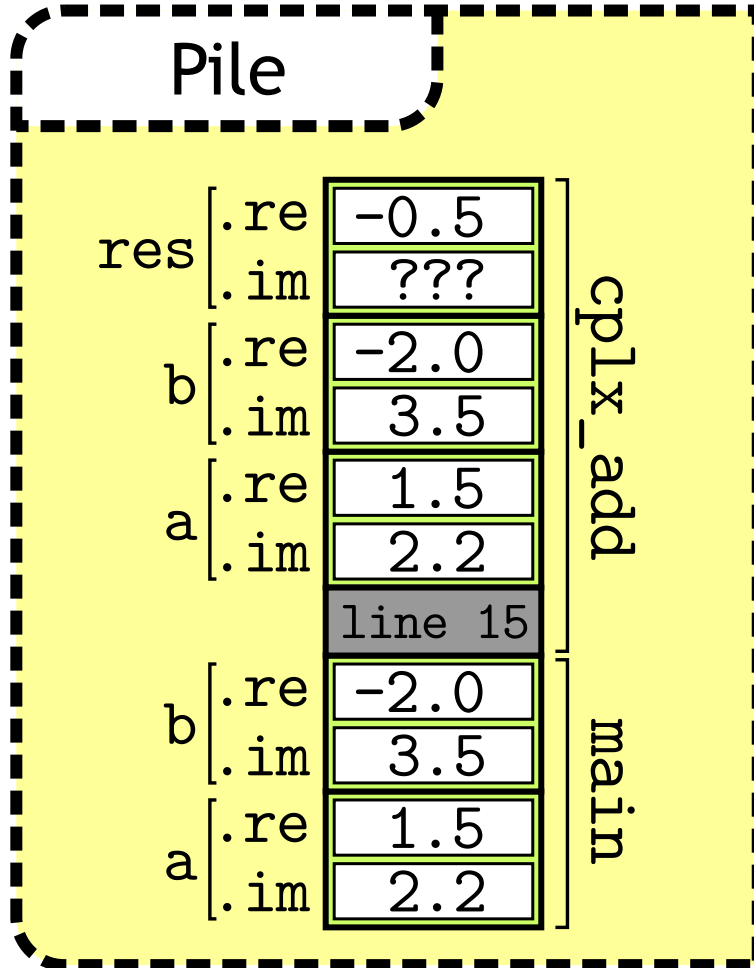
```



```

1 typedef struct {
2     double re, im;
3 } complex;
4
5 complex cplx_add(complex a,
6                 complex b) {
7     complex res;
8     res.re = a.re + b.re;
9     res.im = a.im + b.im;
10    return res;
11 }
12 int main() {
13     complex a = {1.5, 2.2};
14     complex b = {-2, 3.5};
15     a = cplx_add(a, b);
16     return 0;
17 }

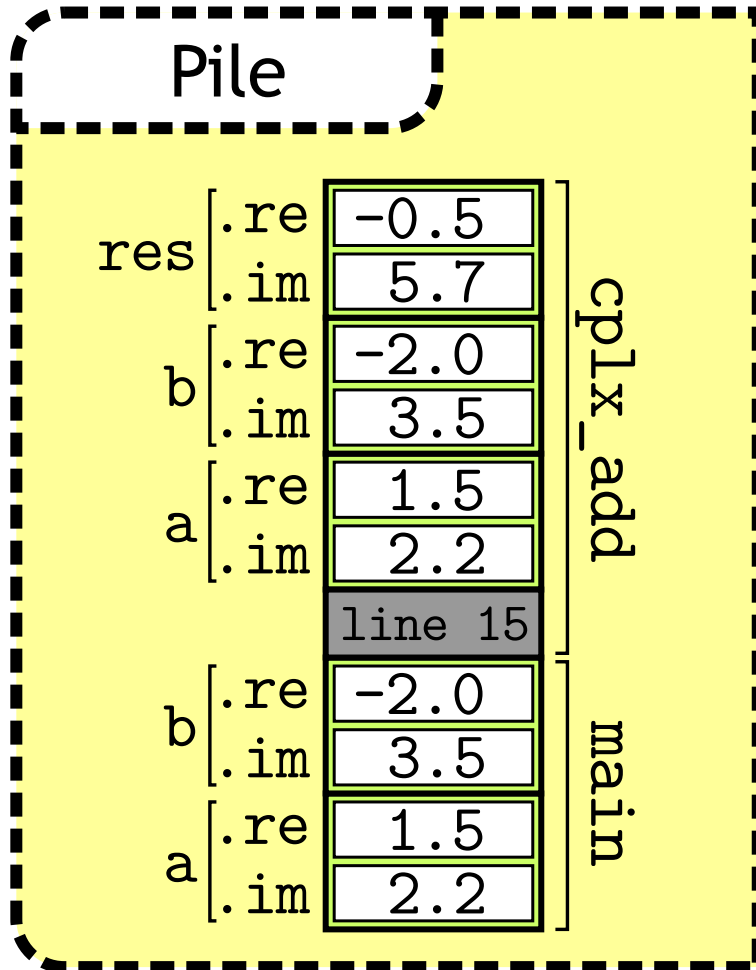
```



```

1 typedef struct {
2     double re, im;
3 } complex;
4
5 complex cplx_add(complex a,
6                 complex b) {
7     complex res;
8     ► res.re = a.re + b.re;
9     res.im = a.im + b.im;
10    return res;
11 }
12 int main() {
13     complex a = {1.5, 2.2};
14     complex b = {-2, 3.5};
15     a = cplx_add(a,b);
16     return 0;
17 }

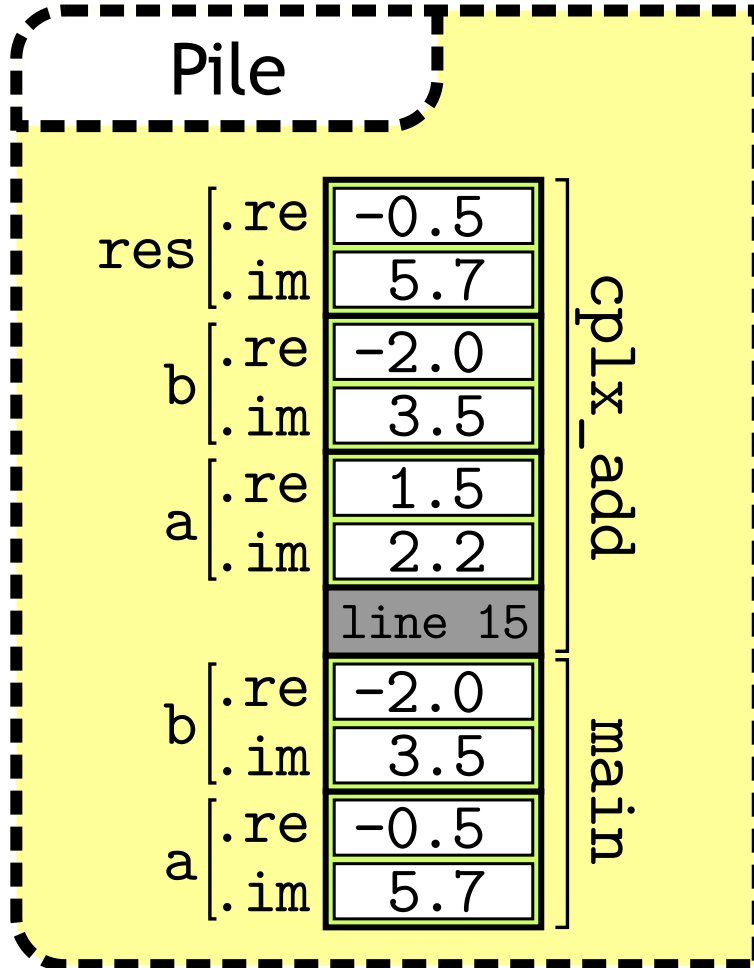
```



```

1 typedef struct {
2     double re, im;
3 } complex;
4
5 complex cplx_add(complex a,
6                 complex b) {
7     complex res;
8     res.re = a.re + b.re;
9     ► res.im = a.im + b.im;
10    return res;
11 }
12 int main() {
13     complex a = {1.5, 2.2};
14     complex b = {-2, 3.5};
15     a = cplx_add(a,b);
16     return 0;
17 }

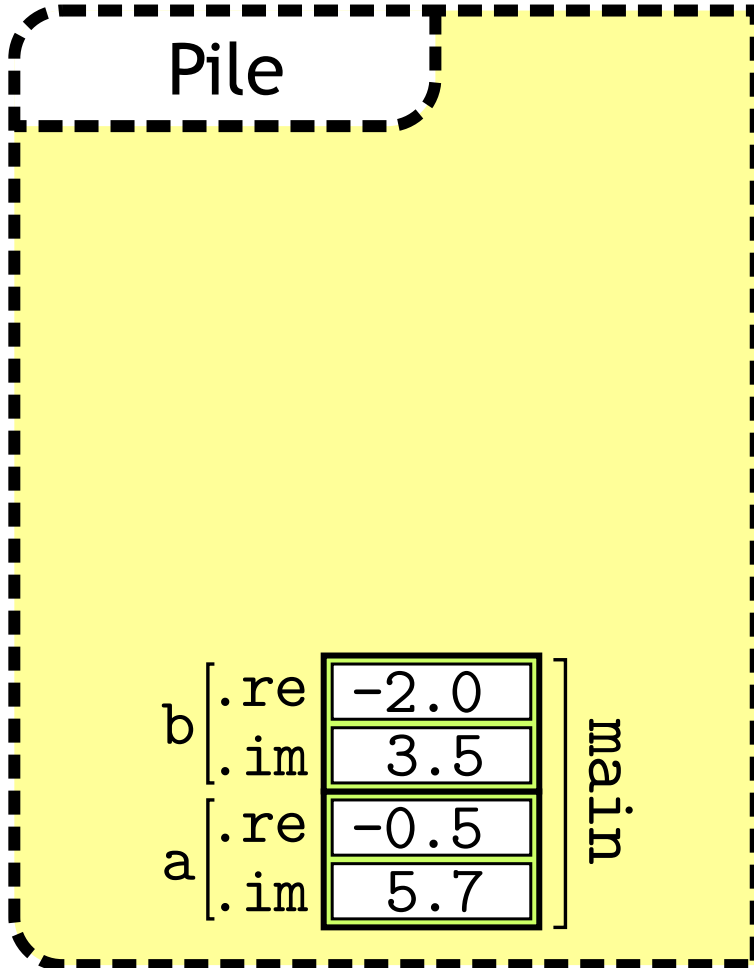
```



```

1 typedef struct {
2     double re, im;
3 } complex;
4
5 complex cplx_add(complex a,
6                 complex b) {
7     complex res;
8     res.re = a.re + b.re;
9     res.im = a.im + b.im;
10    ► return res;
11 }
12 int main() {
13     complex a = {1.5, 2.2};
14     complex b = {-2, 3.5};
15     a = cplx_add(a,b);
16     return 0;
17 }

```



```

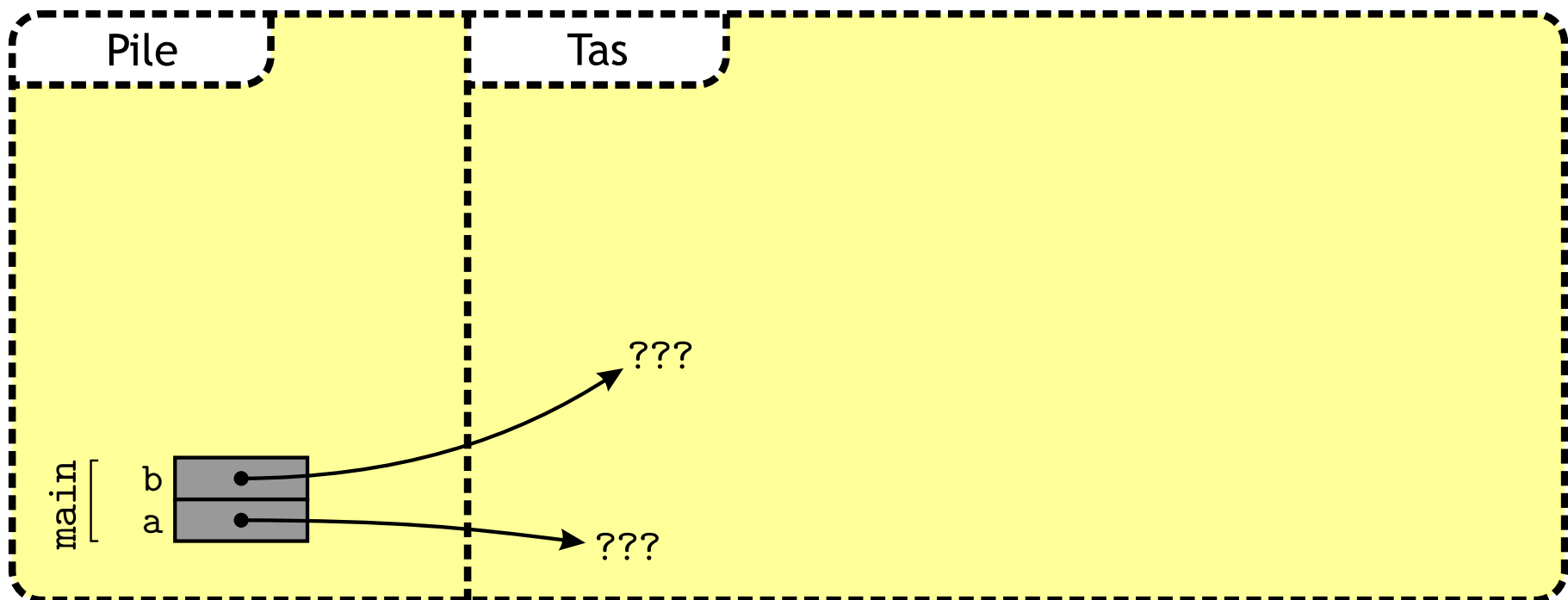
1 typedef struct {
2     double re, im;
3 } complex;
4
5 complex cplx_add(complex a,
6                 complex b) {
7     complex res;
8     res.re = a.re + b.re;
9     res.im = a.im + b.im;
10    return res;
11 }
12 int main() {
13     complex a = {1.5, 2.2};
14     complex b = {-2, 3.5};
15     a = cplx_add(a,b);
16     ► return 0;
17 }

```

- ✘ Pour éviter la copie de structures on utilise le passage par adresse
 - ✘ on ne copie alors qu'une adresse mémoire
 - la taille du pointeur ne dépend pas de la taille de la structure.
- ✘ En général on préfère aussi stocker les structures dans le tas :
 - ✘ on déclare uniquement les pointeurs en variables locales,
 - ✘ on alloue les structures avec `malloc`,
 - ✘ il faut ensuite les libérer avec `free`.
- ✘ Avec des pointeurs uniquement, le code est un peu plus long à écrire, mais on ne copie qu'une adresse
 - on peut gagner beaucoup en efficacité.
- ✘ Pour accéder à une composante il faut d'abord suivre un pointeur
 - ✘ `(*a).re` est un peu lourd,
 - ✘ la notation `a->re` est équivalente (et plus lisible).

```
1 complex* cplx_add(complex* a,  
2                 complex* b) {  
3     complex* res;  
4     res = malloc(sizeof(complex));  
5     res->re = a->re + b->re;  
6     res->im = a->im + b->im;  
7     return res;  
8 }
```

```
9 int main() {  
10  ► complex *a, *b;  
11     a = malloc(sizeof(complex));  
12     b = malloc(sizeof(complex));  
13     a->re=1.5; a->im=2.2;  
14     b->re=-2; b->im=3.5;  
15     a = cplx_add(a,b);  
16     free(a); free(b); }
```




```

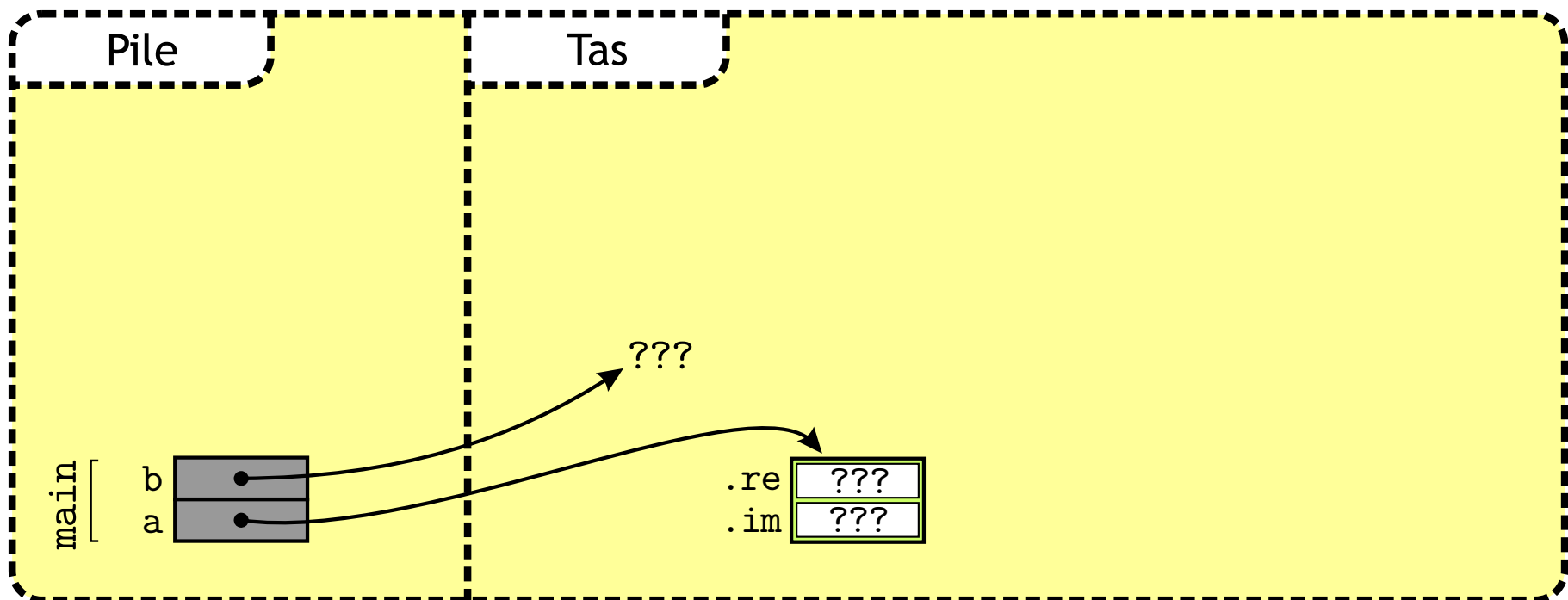
1 complex* cplx_add(complex* a,
2                 complex* b) {
3     complex* res;
4     res = malloc(sizeof(complex));
5     res->re = a->re + b->re;
6     res->im = a->im + b->im;
7     return res;
8 }

```

```

9 int main() {
10     complex *a, *b;
11     ► a = malloc(sizeof(complex));
12     b = malloc(sizeof(complex));
13     a->re=1.5; a->im=2.2;
14     b->re=-2; b->im=3.5;
15     a = cplx_add(a,b);
16     free(a); free(b); }

```



Structures et fonctions

Exemple avec passage par adresse

```

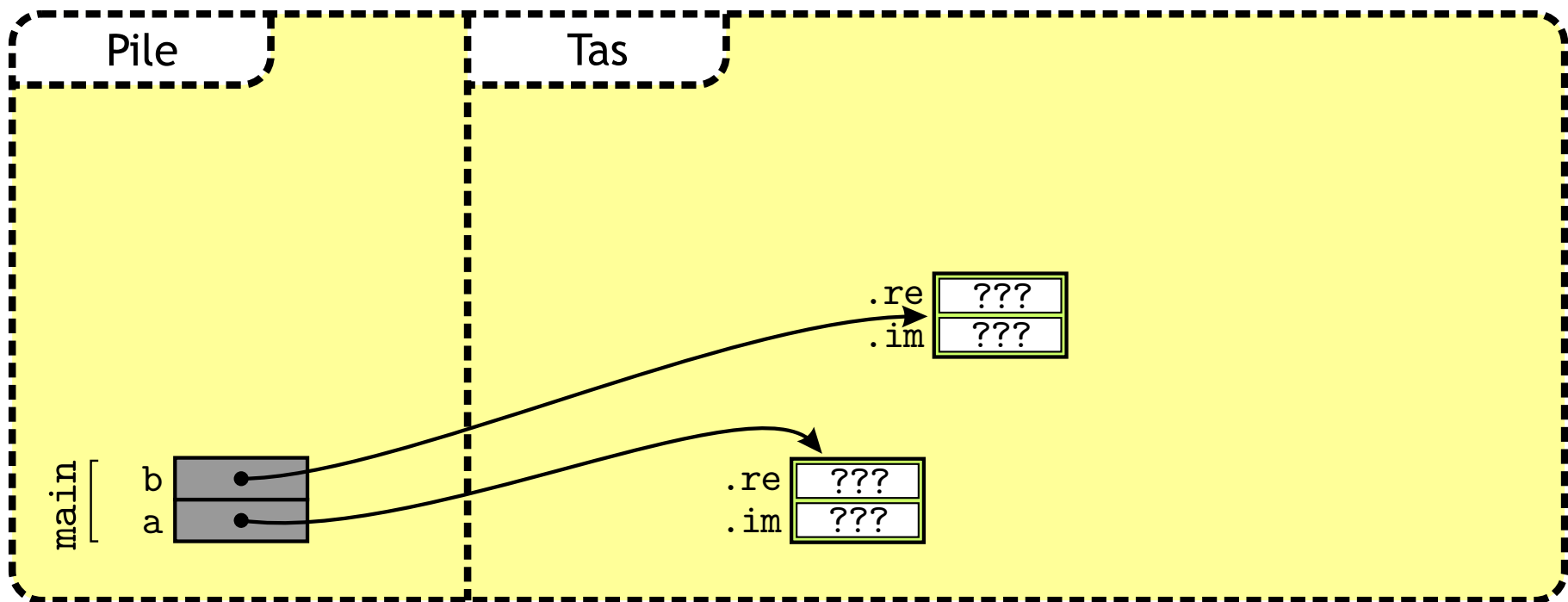
1 complex* cplx_add(complex* a,
2                 complex* b) {
3     complex* res;
4     res = malloc(sizeof(complex));
5     res->re = a->re + b->re;
6     res->im = a->im + b->im;
7     return res;
8 }

```

```

9 int main() {
10     complex *a, *b;
11     a = malloc(sizeof(complex));
12     ► b = malloc(sizeof(complex));
13     a->re=1.5; a->im=2.2;
14     b->re=-2; b->im=3.5;
15     a = cplx_add(a,b);
16     free(a); free(b); }

```



```

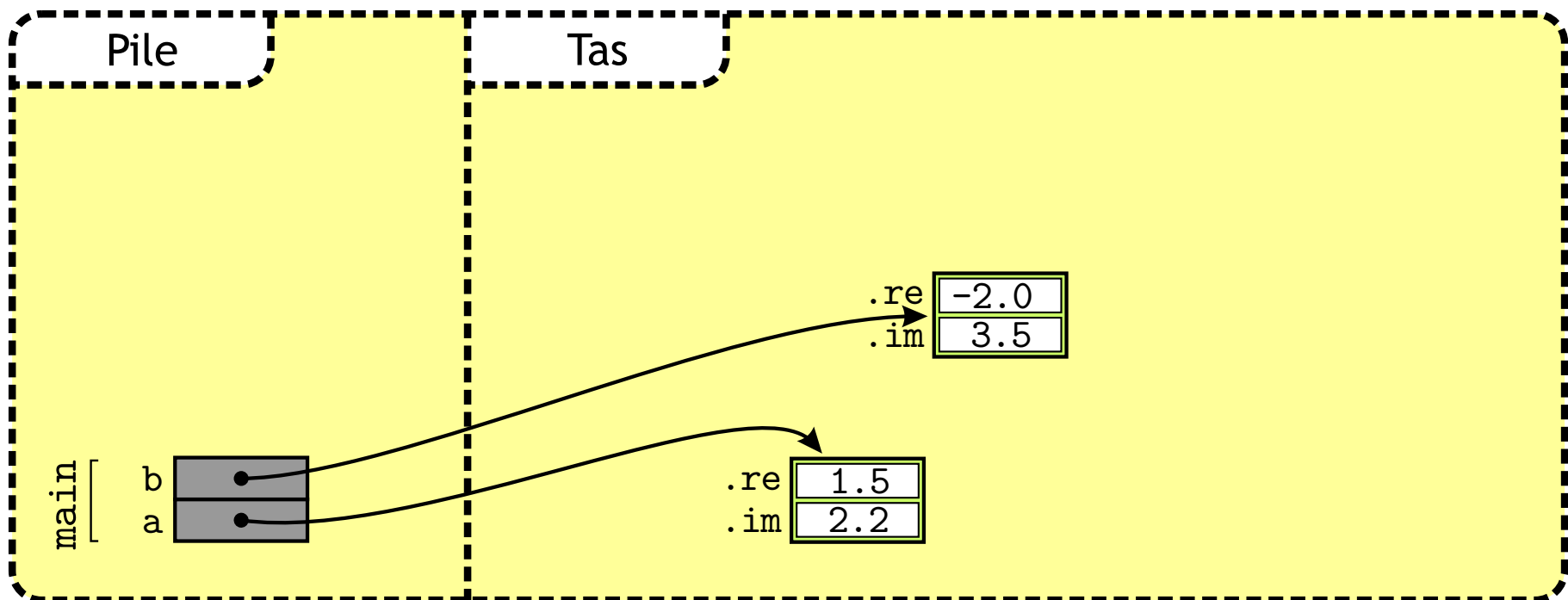
1 complex* cplx_add(complex* a,
2                 complex* b) {
3     complex* res;
4     res = malloc(sizeof(complex));
5     res->re = a->re + b->re;
6     res->im = a->im + b->im;
7     return res;
8 }

```

```

9 int main() {
10     complex *a, *b;
11     a = malloc(sizeof(complex));
12     b = malloc(sizeof(complex));
13     ► a->re=1.5; a->im=2.2;
14     b->re=-2; b->im=3.5;
15     a = cplx_add(a,b);
16     free(a); free(b); }

```



Structures et fonctions

Exemple avec passage par adresse

```

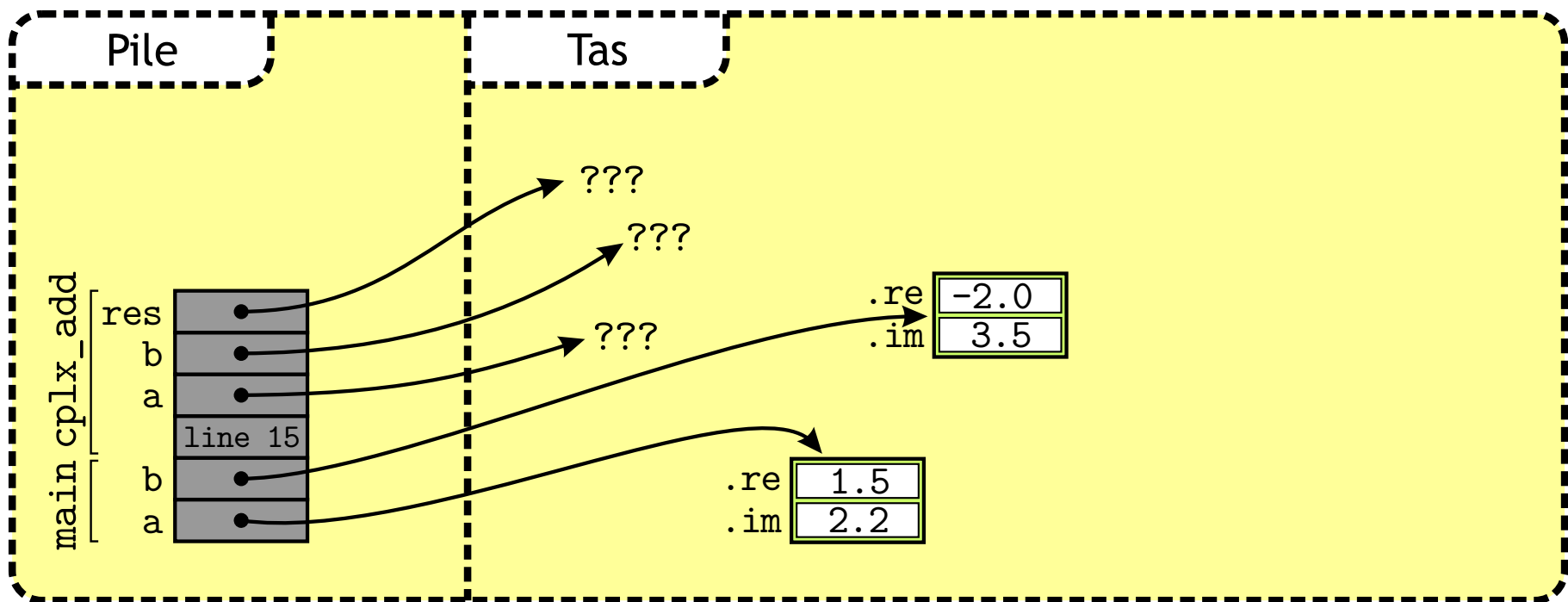
1 complex* cplx_add(complex* a,
2                 complex* b) {
3     complex* res;
4     res = malloc(sizeof(complex));
5     res->re = a->re + b->re;
6     res->im = a->im + b->im;
7     return res;
8 }

```

```

9 int main() {
10     complex *a, *b;
11     a = malloc(sizeof(complex));
12     b = malloc(sizeof(complex));
13     a->re=1.5; a->im=2.2;
14     b->re=-2; b->im=3.5;
15     ► a = cplx_add(a,b);
16     free(a); free(b); }

```



Structures et fonctions

Exemple avec passage par adresse

```

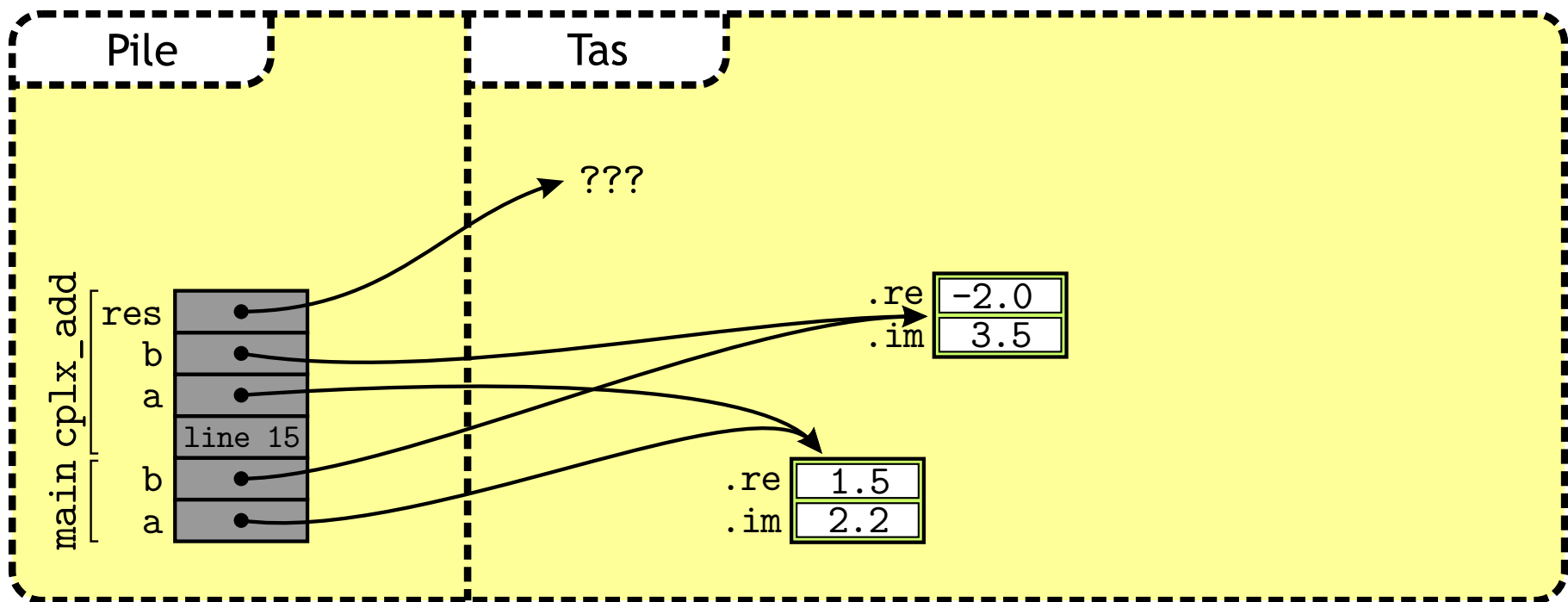
1 complex* cplx_add(complex* a,
2                 complex* b) {
3     complex* res;
4     res = malloc(sizeof(complex));
5     res->re = a->re + b->re;
6     res->im = a->im + b->im;
7     return res;
8 }

```

```

9 int main() {
10     complex *a, *b;
11     a = malloc(sizeof(complex));
12     b = malloc(sizeof(complex));
13     a->re=1.5; a->im=2.2;
14     b->re=-2; b->im=3.5;
15     ► a = cplx_add(a,b);
16     free(a); free(b); }

```



Structures et fonctions

Exemple avec passage par adresse

```

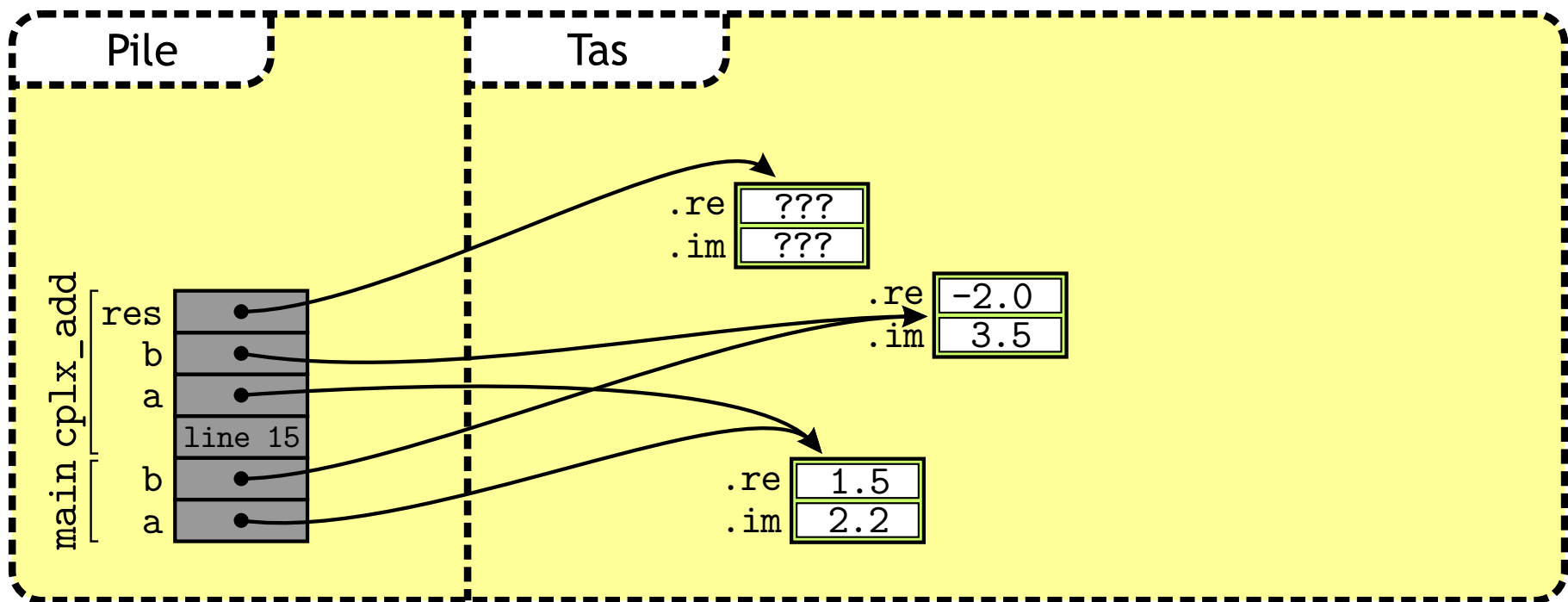
1 complex* cplx_add(complex* a,
2                 complex* b) {
3     complex* res;
4     ► res = malloc(sizeof(complex));
5     res->re = a->re + b->re;
6     res->im = a->im + b->im;
7     return res;
8 }

```

```

9 int main() {
10     complex *a, *b;
11     a = malloc(sizeof(complex));
12     b = malloc(sizeof(complex));
13     a->re=1.5; a->im=2.2;
14     b->re=-2; b->im=3.5;
15     a = cplx_add(a,b);
16     free(a); free(b); }

```



Structures et fonctions

Exemple avec passage par adresse

```

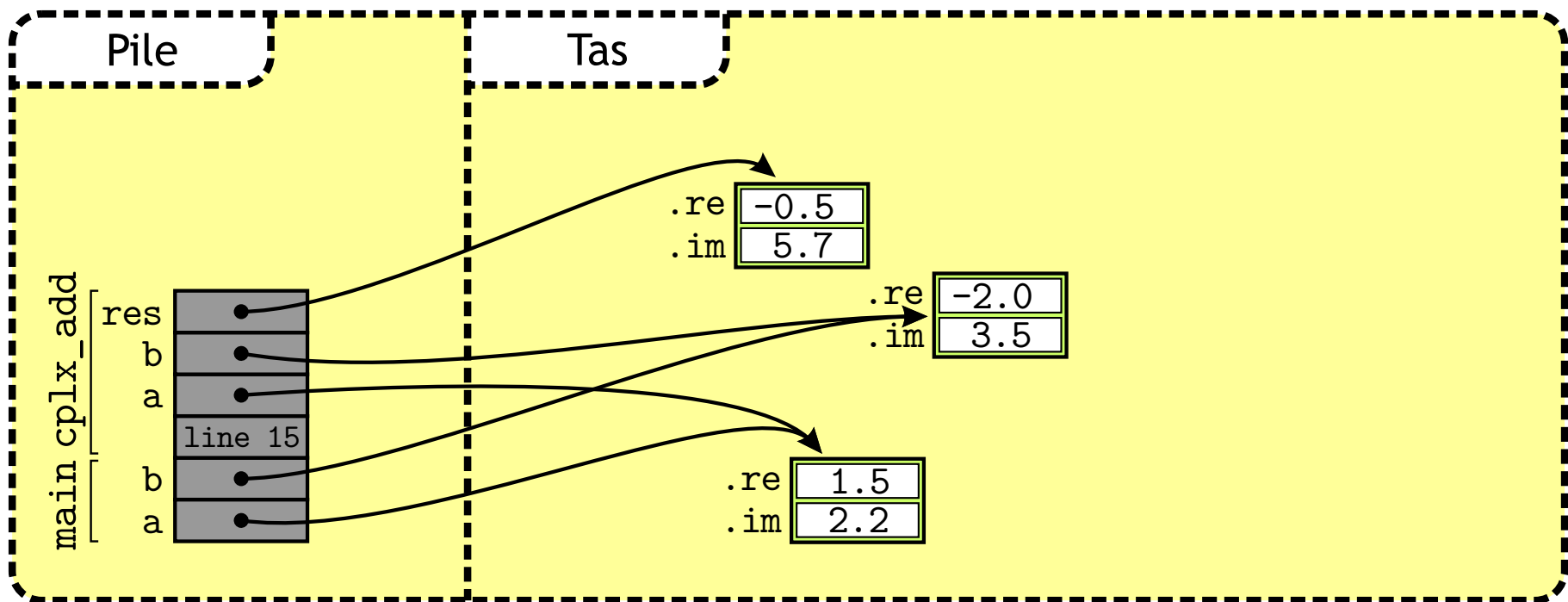
1 complex* cplx_add(complex* a,
2                 complex* b) {
3     complex* res;
4     res = malloc(sizeof(complex));
5     ► res->re = a->re + b->re;
6     res->im = a->im + b->im;
7     return res;
8 }

```

```

9 int main() {
10     complex *a, *b;
11     a = malloc(sizeof(complex));
12     b = malloc(sizeof(complex));
13     a->re=1.5; a->im=2.2;
14     b->re=-2; b->im=3.5;
15     a = cplx_add(a,b);
16     free(a); free(b); }

```



```

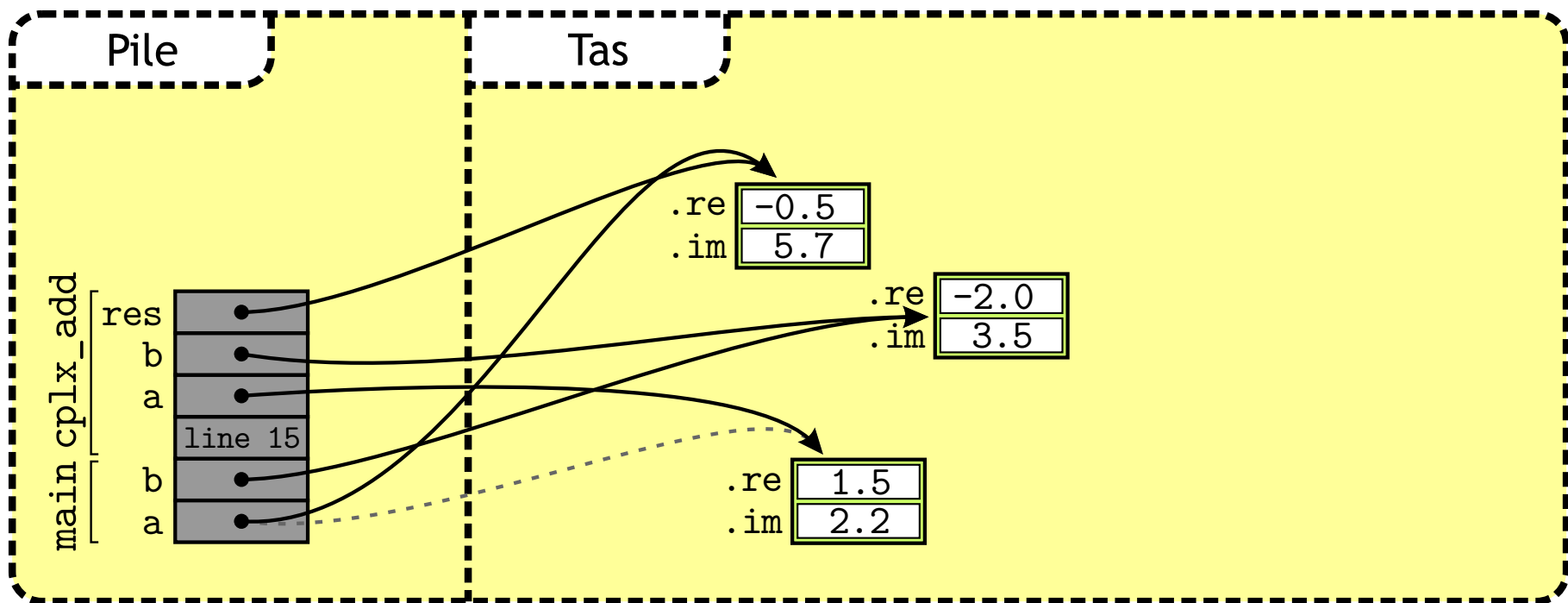
1 complex* cplx_add(complex* a,
2                 complex* b) {
3     complex* res;
4     res = malloc(sizeof(complex));
5     res->re = a->re + b->re;
6     res->im = a->im + b->im;
7     ► return res;
8 }

```

```

9 int main() {
10     complex *a, *b;
11     a = malloc(sizeof(complex));
12     b = malloc(sizeof(complex));
13     a->re=1.5; a->im=2.2;
14     b->re=-2; b->im=3.5;
15     a = cplx_add(a,b);
16     free(a); free(b); }

```



Structures et fonctions

Exemple avec passage par adresse

```

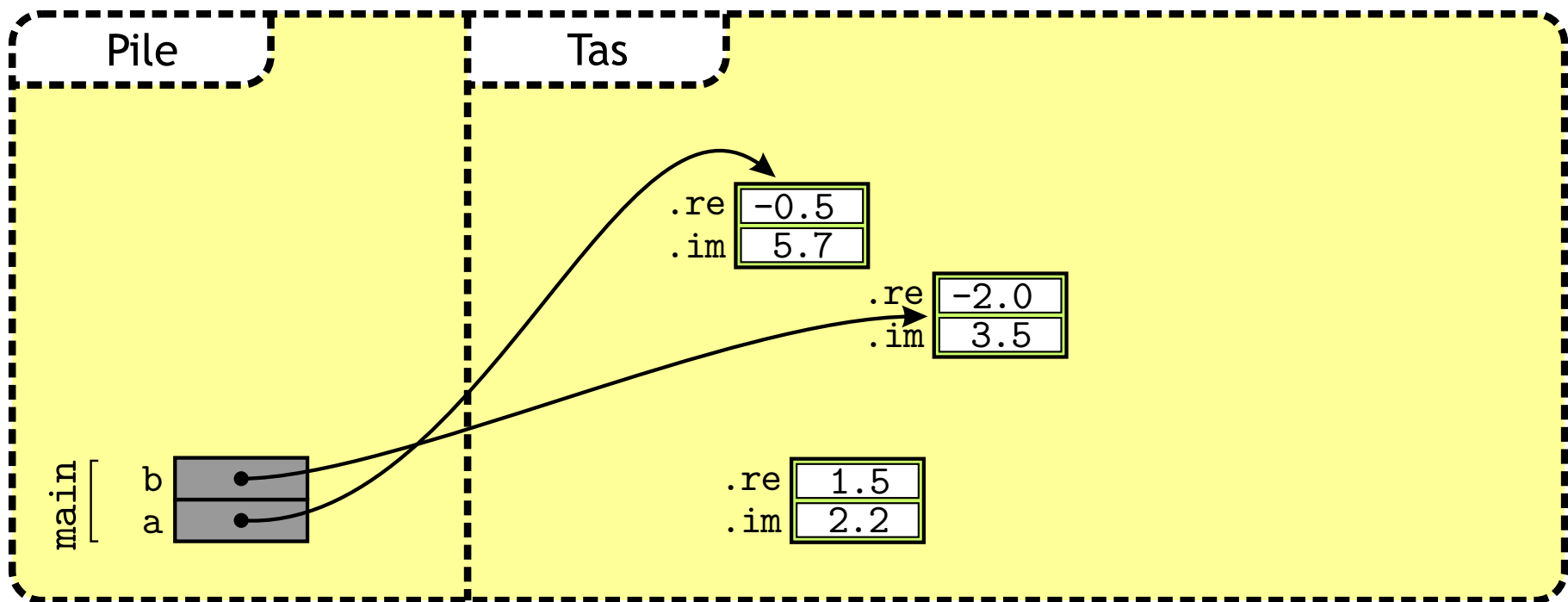
1 complex* cplx_add(complex* a,
2                 complex* b) {
3     complex* res;
4     res = malloc(sizeof(complex));
5     res->re = a->re + b->re;
6     res->im = a->im + b->im;
7     return res;
8 }

```

```

9 int main() {
10     complex *a, *b;
11     a = malloc(sizeof(complex));
12     b = malloc(sizeof(complex));
13     a->re=1.5; a->im=2.2;
14     b->re=-2; b->im=3.5;
15     a = cplx_add(a,b);
16     ► free(a); free(b); }

```



Structures et fonctions

Exemple avec passage par adresse

```

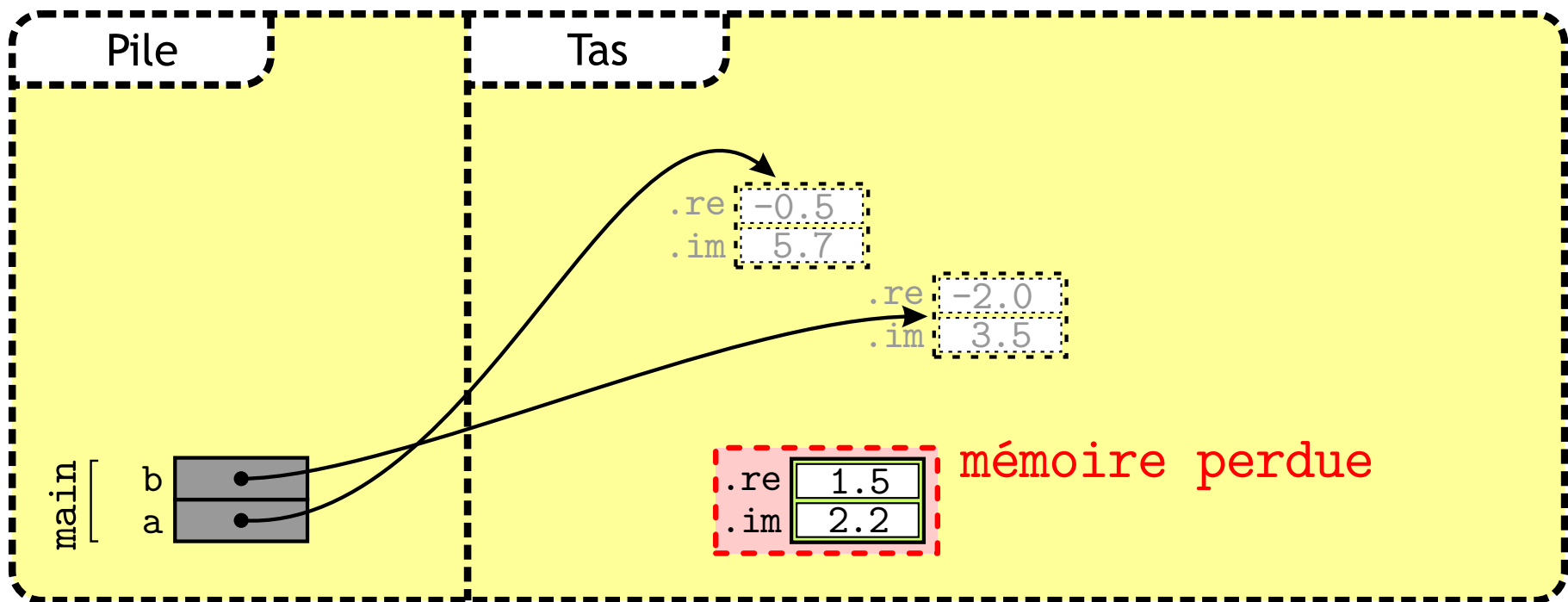
1 complex* cplx_add(complex* a,
2                 complex* b) {
3     complex* res;
4     res = malloc(sizeof(complex));
5     res->re = a->re + b->re;
6     res->im = a->im + b->im;
7     return res;
8 }

```

```

9 int main() {
10     complex *a, *b;
11     a = malloc(sizeof(complex));
12     b = malloc(sizeof(complex));
13     a->re=1.5; a->im=2.2;
14     b->re=-2; b->im=3.5;
15     a = cplx_add(a,b);
16     ► free(a); free(b); }

```



✘ Il est aussi possible de faire des tableaux de structures

✘ soit des tableaux simples :

```
1 complex* tab;  
2 tab = (complex*) malloc(10*sizeof(complex));  
3 tab[5].re = 2;
```

✘ soit des tableaux de pointeurs :

```
1 complex** tab;  
2 tab = (complex**) malloc(10*sizeof(complex*));  
3 for (i=0; i<10; i++) {  
4     tab[i] = (complex*) malloc(sizeof(complex));  
5 }  
6 tab[5]->re = 2;
```

✘ Il faut bien regarder si `tab[5]` est une structure ou un pointeur et utilise le bon symbole `.` ou `->`

- ✘ Il est aussi possible de définir des structure récurrentes
 - ✘ pas des structures qui contiennent un élément de leur type
 - problème pour connaître la taille effective de la structure...
 - ✘ mais contenant un pointeur vers une structure du même type.

```
1 typedef struct cell_st {  
2     int val;  
3     struct cell_st* next;  
4 } cell;
```

- ✘ On ne peut pas encore utiliser le nom du typedef à ce moment
 - obligatoire d'avoir un nom de structure.
- ✘ Nous reverrons cela pour les listes chaînées, arbres...

(cf. prochains cours)

Structures et bibliothèques

IN101 - 2011-2012

- ✘ Une structure sert à représenter un type de données
 - ✘ soit très spécifique → ne sert qu'à un programme,
 - ✘ soit plus général → peut resservir dans d'autres programmes.

- ✘ Quand on définit une structure, on veut en général aussi définir :
 - ✘ une fonction d'initialisation,
 - ✘ une fonction d'affichage,
 - ✘ une fonction de "destruction" (libération de la mémoire)...→ on ne veut pas tout récrire à chaque fois.

- ✘ Il sera plus facile de réutiliser cela dans une "bibliothèque"
 - ✘ plutôt que de tout écrire dans un seul fichier, faire un fichier qui contient toutes les fonctions propres à la bibliothèque, et un fichier à côté avec le `main` et autres fonctions spécifiques.

- ✘ Une bibliothèque peut simplement être un fichier `.c` à inclure en début de programme
 - ✘ oblige l'utilisateur à tout lire pour connaître les fonctions,
 - ✘ le code de la bibliothèque est recompilé à chaque compilation.

- ✘ Pour plus de lisibilité on fait en général deux fichiers :
 - ✘ un fichier `.c` qui contient le code des fonctions,
 - ✘ un fichier `.h` qui ne contient que les en-têtes de fonctions
 - seul ce fichier est inclus et suffit pour utiliser la bibliothèque.

- ✘ Le fichier `.c` doit quand même être inclus à la compilation :
 - > `gcc -Wall prog.c lib.c -o prog`
 - et `prog.c` doit contenir `#include "lib.h"`

`cplx.h`

```
1 struct cplx_st {
2     double re,im;
3 };
4 typedef struct cplx_st* cplx;
5
6 cplx cplx_init(double re, double im);
7
8 cplx cplx_add(cplx a, cplx b);
9 cplx cplx_mul(cplx a, cplx b);
10 cplx cplx_inv(cplx a);
11 double cplx_norm(cplx a);
12
13 void cplx_print(cplx a);
14 void cplx_free(cplx a);
```

`cplx.c`

```
1 #include "cplx.h"
2 cplx cplx_init(double re, double im){
3     cplx res = malloc(sizeof(struct cplx_st));
4     if (res != NULL) {
5         res->re = re;
6         res->im = im;
7     }
8     return res;
9 }
10 ...
11 void cplx_print(cplx a) {
12     printf("%f + i%f", a->re, a->im);
13 }
14 void cplx_free(cplx a) {
15     free(a);
16 }
```

`prog.c`

```
1 #include "cplx.h"
2
3 int main() {
4     cplx a,b,c;
5     a = cplx_init(1.5, 3.2);
6     b = cplx_init(-2, 2.5);
7     c = cplx_add(a, b);
8     cplx_print(c);
9     printf("\n");
10    cplx_free(a);
11    cplx_free(b);
12    cplx_free(c);
13    return 0;
14 }
15
```

✘ Compiler avec : `gcc -Wall -O3 cplx.c prog.c -o prog`

- ✘ Les structures sont un outil centrale en C
 - ✘ permettent de regrouper des variables,
 - ✘ simplifient la manipulation de données complexes.
- ✘ Les structures sont souvent très efficaces
 - ✘ attention à ne pas faire trop de recopies inutiles,
 - ✘ en général on les utilise toujours avec des pointeurs
 - la notation -> existe pour une raison !
- ✘ Les structures ne remplacent pas des vrais objets comme en C++, java ou OCaml : pas de méthodes, d'héritage...
- ✘ Elles servent surtout pour les structures de données :
 - ✘ matrices,
 - ✘ listes chaînées,
 - ✘ arbres,
 - ✘ graphes...