Structures

Programmation séquentielle en C, 2022-2023

Orestis Malaspinas (A401) et un tout petit peu Michaël El Kharroubi

2022-10-18

Informatique et Systèmes de Communication, HEPIA

Types composés: `struct` (1/6)

- Appelées aussi **structures**.

Fractions

- Numérateur: `int num;`
- Dénominateur: `int denom.`

Addition

```
int num1 = 1, denom1 = 2;
int num2 = 1, denom2 = 3;
int num3 = num1 * denom2 + num2 * denom1;
int denom3 = denom1 * denom2;
```

Pas super pratique....

Types composés: `struct` (2/6)

On peut faire mieux

- Plusieurs variables qu'on aimerait regrouper dans un seul type: `struct`.

```
struct fraction { // déclaration du type
    int32_t num, denom;
};

struct fraction frac; // déclaration de frac
```

Types composés: `struct` (3/6)

Simplifications

- `typedef` permet de définir un nouveau type.

```
typedef unsinged int uint;
typedef struct fraction fraction_t;
typedef struct fraction {
    int32_t num, denom;
} fraction_t;
```

- L'initialisation peut aussi se faire avec

```
fraction_t frac = {1, -2};       // num = 1, denom = -2
fraction_t frac = {.denom = 1, .num = -2}; // idem
fraction_t frac = {.denom = 1}; // argl! .num non initialisé
fraction_t frac2 = frac;         // copie
```

Types composés: `struct` (4/6)

Pointeurs

- Comme pour tout type, on peut avoir des pointeurs vers un `struct`.
- Les champs sont accessible avec le sélecteur ->

```
fraction_t *frac; // on crée un pointeur
frac->num = 1;    // seg fault...
frac->denom = -1; // mémoire pas allouée.
```

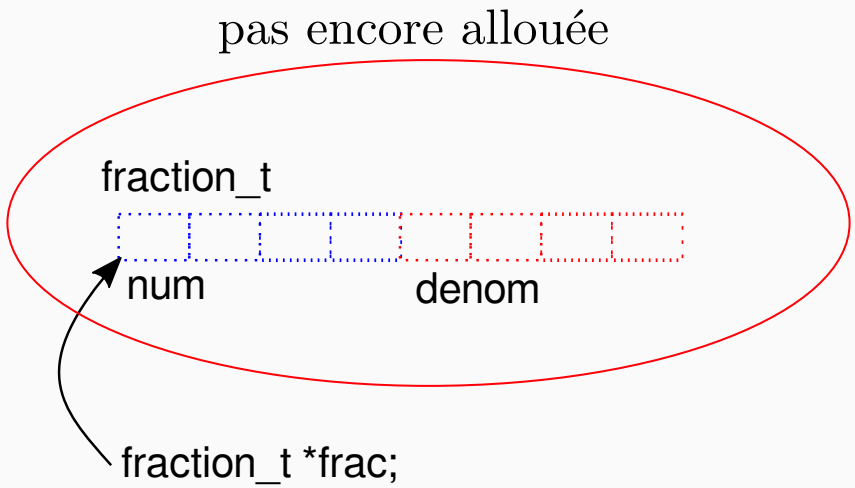

Figure 1: La représentation mémoire de `fraction_t`.

Types composés: `struct` (5/6)

Initialisation

- Avec le passage par **référence** on peut modifier un struct *en place*.
- Les champs sont accessible avec le sélecteur ->

```
void fraction_init(fraction_t *f,
                   int32_t num, int32_t denom)
{
    // f a déjà été allouée
    f->num   = num;
    f->denom = denom;
}
int main() {
    fraction_t frac; // on alloue une fraction
    fraction_init(&frac, 2, -1); // on l'initialise
}
```

Types composés: `struct` (6/6)

Initialisation version copie

- On peut allouer une fraction, l'initialiser et le retourner.
- La valeur retournée peut être copiée dans une nouvelle structure.

```
fraction_t fraction_create(int32_t num, int32_t denom) {
    fraction_t f;
    f.num = num; f.denom = denom;
    return f;
}
int main() {
    // on crée une fraction et on l'initialise
    // en copiant la fraction créé par fraction_create
    // deux allocation et une copie
    fraction_t frac = fraction_create(2, -1);
}
```

Modification en place vs retour

Quelle est la différence entre `fraction_init` et `fraction_create`?

```
void fraction_init(fraction_t *f, int32_t num, int32_t denom);
fraction_t fraction_create(int32_t num, int32_t denom);
```

Modification en place vs retour

Quelle est la différence entre `fraction_init` et `fraction_create`?

```
void fraction_init(fraction_t *f, int32_t num, int32_t denom);
fraction_t fraction_create(int32_t num, int32_t denom);
```

- `fraction_init` modifie une fraction **déjà** allouée quelque part.
- `fraction_create` alloue une nouvelle fraction et la retourne.

Modification en place vs retour

Quelle est la différence entre `fraction_init` et `fraction_create`?

```
void fraction_init(fraction_t *f, int32_t num, int32_t denom);
fraction_t fraction_create(int32_t num, int32_t denom);
```

- `fraction_init` modifie une fraction **déjà** allouée quelque part.
- `fraction_create` alloue une nouvelle fraction et la retourne.

Quelle impact sur les performances?

Modification en place vs retour

Quelle est la différence entre `fraction_init` et `fraction_create`?

```
void fraction_init(fraction_t *f, int32_t num, int32_t denom);
fraction_t fraction_create(int32_t num, int32_t denom);
```

- `fraction_init` modifie une fraction **déjà** allouée quelque part.
- `fraction_create` alloue une nouvelle fraction et la retourne.

Quelle impact sur les performances?

- `init`: une allocation et modification des données.
- `create`: deux allocation, une modification et une copie.
- Important pour des structures contenance *beaucoup* de données.