

Module algorithmie et programmation

Représentation des nombres et types composés

Pierre Künzli

Adapté des cours de Paul Albuquerque, Guido Bologna et Orestis Malaspinas

Représentation des nombres

- Le nombre 247.

Nombres décimaux : Les nombres en base 10

10^2	10^1	10^0
2	4	7

$$247 = 2 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0.$$

Représentation des nombres

- Le nombre 11110111.

Nombres binaires : Les nombres en base 2

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	0	1	1	1

$$1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

Représentation des nombres

- Le nombre 11110111.

Nombres binaires : Les nombres en base 2

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	0	1	1	1

$$1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

$$= 247.$$

Conversion de décimal à binaire

Convertir 11 en binaire ?

Conversion de décimal à binaire

Convertir 11 en binaire ?

- On décompose en puissances de 2 en partant de la plus grande possible

$$11 / 8 = 1, \quad 11 \% 8 = 3$$

$$3 / 4 = 0, \quad 3 \% 4 = 3$$

$$3 / 2 = 1, \quad 3 \% 2 = 1$$

$$1 / 1 = 1, \quad 1 \% 1 = 0$$

- On a donc

$$1011 \Rightarrow 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11.$$

Les additions en binaire

Que donne l'addition 1101 avec 0110 ?

- L'addition est la même que dans le système décimal

1101		8+4+0+1 = 13
+ 0110	+	0+4+2+0 = 6
-----		-----
10011		16+0+0+2+1 = 19

- Les entiers sur un ordinateur ont une précision **fixée** (ici 4 bits).
- Que se passe-t-il donc ici ?

Les additions en binaire

Que donne l'addition 1101 avec 0110 ?

- L'addition est la même que dans le système décimal

1101	8+4+0+1 = 13
+ 0110	+ 0+4+2+0 = 6
-----	-----
10011	16+0+0+2+1 = 19

- Les entiers sur un ordinateur ont une précision **fixée** (ici 4 bits).
- Que se passe-t-il donc ici ?

Dépassement de capacité : le nombre est “tronqué”

- 10011 (19) -> 0011 (3).
- On fait “le tour”.

Entier non-signés minimal/maximal

- Quel est l'entier non-signé maximal représentable avec 4 bit ?

Entier non-signés minimal/maximal

- Quel est l'entier non-signé maximal représentable avec 4 bit ?

$$(1111)_2 = 8 + 4 + 2 + 1 = 15$$

- Quel est l'entier non-signé minimal représentable avec 4 bit ?

Entier non-signés minimal/maximal

- Quel est l'entier non-signé maximal représentable avec 4 bit ?

$$(1111)_2 = 8 + 4 + 2 + 1 = 15$$

- Quel est l'entier non-signé minimal représentable avec 4 bit ?

$$(0000)_2 = 0 + 0 + 0 + 0 = 0$$

- Quel est l'entier non-signé min/max représentable avec N bit ?

Entier non-signés minimal/maximal

- Quel est l'entier non-signé maximal représentable avec 4 bit ?

$$(1111)_2 = 8 + 4 + 2 + 1 = 15$$

- Quel est l'entier non-signé minimal représentable avec 4 bit ?

$$(0000)_2 = 0 + 0 + 0 + 0 = 0$$

- Quel est l'entier non-signé min/max représentable avec N bit ?

$$0 \text{ et } 2^N - 1.$$

- Donc `uint32_t` maximal est ?

Entier non-signés minimal/maximal

- Quel est l'entier non-signé maximal représentable avec 4 bit ?

$$(1111)_2 = 8 + 4 + 2 + 1 = 15$$

- Quel est l'entier non-signé minimal représentable avec 4 bit ?

$$(0000)_2 = 0 + 0 + 0 + 0 = 0$$

- Quel est l'entier non-signé min/max représentable avec N bit ?

$$0 \text{ et } 2^N - 1.$$

- Donc `uint32_t` maximal est ?

$$4294967295$$

Les multiplications en binaire

Que donne la multiplication de 1101 avec 0110 ?

- La multiplication est la même que dans le système décimal

1101	13
* 0110	* 6
-----	-----
0000	78
11010	
110100	
+ 0000000	
-----	-----
1001110	64+0+0+8+4+2+0

Les multiplications en binaire

Que fait la multiplication par 2 ?

Les multiplications en binaire

Que fait la multiplication par 2 ?

- Décalage de un bit vers la gauche !

```
      0110
*     0010
-----
      0000
+     01100
-----
     01100
```


Les multiplications en binaire

Que fait la multiplication par 2 ?

- Décalage de un bit vers la gauche !

$$\begin{array}{r} 0110 \\ * 0010 \\ \hline 0000 \\ + 01100 \\ \hline 01100 \end{array}$$

Que fait la multiplication par 2^N ?

Les multiplications en binaire

Que fait la multiplication par 2 ?

- Décalage de un bit vers la gauche !

```
    0110
  * 0010
  -----
    0000
  + 01100
  -----
    01100
```

Que fait la multiplication par 2^N ?

- Décalage de N bits vers la gauche !

Entiers signés

Pas de nombres négatifs encore...

Comment faire ?

Entiers signés

Pas de nombres négatifs encore...

Comment faire ?

Solution naïve :

- On ajoute un bit de signe (le bit de poids fort) :

00000010: +2

10000010: -2

Problèmes ?

Entiers signés

Pas de nombres négatifs encore...

Comment faire ?

Solution naïve :

- On ajoute un bit de signe (le bit de poids fort) :

00000010: +2

10000010: -2

Problèmes ?

- Il y a deux zéros (pas trop grave) : 10000000 et 00000000
- Les additions différentes que pour les non-signés (très grave)

00000010	2
+ 10000100	+ -4
-----	----
10000110 = -6	!= -2

Entiers signés

Beaucoup mieux

- Complément à un :
 - on inverse tous les bits : $1001 \Rightarrow 0110$.
 - il y a toujours deux 0 (0000 et 1111)

Encore un peu mieux

- Complément à deux :
 - on inverse tous les bits,
 - on ajoute 1 (on ignore les dépassements).

Entiers signés

Beaucoup mieux

- Complément à un :
 - on inverse tous les bits : $1001 \Rightarrow 0110$.
 - il y a toujours deux 0 (0000 et 1111)

Encore un peu mieux

- Complément à deux :
 - on inverse tous les bits,
 - on ajoute 1 (on ignore les dépassements).
- Comment écrit-on -4 en 8 bits ?

Entiers signés

Beaucoup mieux

- Complément à un :
 - on inverse tous les bits : 1001 => 0110.
 - il y a toujours deux 0 (0000 et 1111)

Encore un peu mieux

- Complément à deux :
 - on inverse tous les bits,
 - on ajoute 1 (on ignore les dépassements).
- Comment écrit-on -4 en 8 bits ?

4 = 00000100

-4 => -----
 00000100

 11111011
+ 00000001

 11111100

Le complément à 2

Questions :

- Comment on écrit $+0$ et -0 ?
- Comment calcule-t-on $2 + (-4)$?
- Quel est le complément à 2 de 1000 0000 ?

Le complément à 2

Questions :

- Comment on écrit +0 et -0 ?
- Comment calcule-t-on $2 + (-4)$?
- Quel est le complément à 2 de 1000 0000 ?

Réponses

- Comment on écrit +0 et -0 ?

$$+0 = 00000000$$

$$-0 = 11111111 + 00000001 = 100000000 \Rightarrow 00000000$$

on retombe donc sur la même représentation

Le complément à 2

Réponses

- Comment calcule-t-on $2 + (-4)$?

00000010	2
+ 11111100	+ -4
-----	-----
11111110	-2

- En effet

$$11111110 \Rightarrow 00000001 + 00000001 = 00000010 = 2.$$

Le complément à 2

Réponses

- Quel est le complément à 2 de 10000000 ?

$$10000000 \Rightarrow 01111111 + 00000001 = 10000000$$

- Cas particulier, le complément à 2 de 10000000 est 10000000.

Le complément à 2

Quels sont les entiers représentables en 8 bits ?

Le complément à 2

Quels sont les entiers représentables en 8 bits ?

01111111 => 127

10000000 => -128 // par définition

Quels sont les entiers représentables sur N bits ?

Le complément à 2

Quels sont les entiers représentables en 8 bits ?

01111111 => 127

10000000 => -128 // par définition

Quels sont les entiers représentables sur N bits ?

$$-2^{N-1} \dots 2^{N-1} - 1.$$

Remarque : dépassement de capacité en C

- Comportement indéfini !

Nombres à virgule

Comment manipuler des nombres à virgule ?

$$0.1 + 0.2 = 0.3.$$

Facile non ?

Nombres à virgule

Comment manipuler des nombres à virgule ?

$$0.1 + 0.2 = 0.3.$$

Facile non ?

Et ça ?

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    float a = atof(argv[1]);
    float b = atof(argv[2]);
    printf("%.10f\n", (double)(a + b));
}
```

Nombres à virgule

Comment manipuler des nombres à virgule ?

$$0.1 + 0.2 = 0.3.$$

Facile non ?

Et ça ?

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    float a = atof(argv[1]);
    float b = atof(argv[2]);
    printf("%.10f\n", (double)(a + b));
}
```

Que se passe-t-il donc ?

Nombres à virgule

Nombres à virgule fixe

2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	1	0	.	0	1	0	1

Qu'est-ce ça donne en décimal ?

Nombres à virgule

Nombres à virgule fixe

2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	1	0	.	0	1	0	1

Qu'est-ce ça donne en décimal ?

$$2^3 + 2^1 + \frac{1}{2^2} + \frac{1}{2^4} = 8 + 2 + 0.5 + 0.0625 = 10.5625.$$

Limites de cette représentation ?

Nombres à virgule

Nombres à virgule fixe

2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	1	0	.	0	1	0	1

Qu'est-ce ça donne en décimal ?

$$2^3 + 2^1 + \frac{1}{2^2} + \frac{1}{2^4} = 8 + 2 + 0.5 + 0.0625 = 10.5625.$$

Limites de cette représentation ?

- Tous les nombres > 16 .
- Tous les nombres < 0.0625 .
- Tous les nombres dont la décimale est pas un multiple de 0.0625 .

Nombres à virgule

Nombres à virgule fixe

- Nombres de $0 = 0000.0000$ à $15.9375 = 1111.1111$.
- Beaucoup de “trous” (au moins 0.0625) entre deux nombres.

Solution partielle ?

Nombres à virgule

Nombres à virgule fixe

- Nombres de $0 = 0000.0000$ à $15.9375 = 1111.1111$.
- Beaucoup de “trous” (au moins 0.0625) entre deux nombres.

Solution partielle ?

- Rajouter des bits.
- Bouger la virgule.

Nombres à virgule flottante

Notation scientifique

- Les nombres sont représentés en terme :
 - Une mantisse
 - Une base
 - Un exposant

$$\underbrace{22.1214}_{\text{nombre}} = \underbrace{221214}_{\text{mantisse}} \cdot \underbrace{10}_{\text{base}} \overset{\text{exp.}}{\hat{-4}},$$

Nombres à virgule flottante

Notation scientifique

- Les nombres sont représentés en terme :
 - Une mantisse
 - Une base
 - Un exposant

$$\underbrace{22.1214}_{\text{nombre}} = \underbrace{221214}_{\text{mantisse}} \cdot \underbrace{10}_{\text{base}}^{\text{exp. } -4},$$

On peut donc séparer la représentation en 2 :

- La mantisse
- L'exposant

Nombres à virgule flottante

Quel est l'avantage ?

Nombres à virgule flottante

Quel est l'avantage ?

On peut représenter des nombres sur énormément d'ordres de grandeur avec un nombre de bits fixés.

Différence fondamentale avec la virgule fixe ?

Nombres à virgule flottante

Quel est l'avantage ?

On peut représenter des nombres sur énormément d'ordres de grandeur avec un nombre de bits fixés.

Différence fondamentale avec la virgule fixe ?

La précision des nombres est **variable** :

- On a uniquement un nombre de chiffres **significatifs**.

$$123456 \cdot 10^{23} + 123456 \cdot 10^{-23}.$$

Quel inconvénient y a-t-il ?

Nombres à virgule flottante

Quel est l'avantage ?

On peut représenter des nombres sur énormément d'ordres de grandeur avec un nombre de bits fixés.

Différence fondamentale avec la virgule fixe ?

La précision des nombres est **variable** :

- On a uniquement un nombre de chiffres **significatifs**.

$$123456 \cdot 10^{23} + 123456 \cdot 10^{-23}.$$

Quel inconvénient y a-t-il ?

Ce mélange d'échelles entraîne une **perte de précision**.

Nombres à virgule flottante simple précision

Aussi appelés *IEEE 754 single-precision binary floating point*.

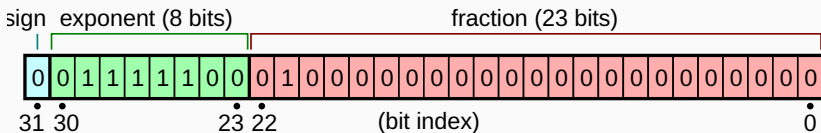


Fig. 1 : Nombres à virgule flottante 32 bits. Source : [Wikipedia](#)

Spécification

- 1 bit de signe,
- 8 bits d'exposant,
- 23 bits de mantisse.

Nombres à virgule flottante simple précision

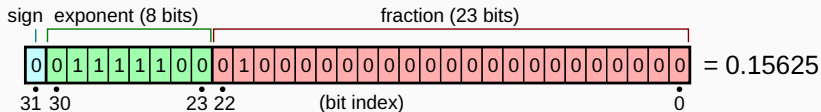


Fig. 2 : Un exercice de nombres à virgule flottante 32 bits. Source : [Wikipedia](#)

$$(-1)^{b_{31}} \cdot 2^{(b_{30}b_{29}\dots b_{23})_2 - 127} \cdot (1.b_{22}b_{21} \dots b_0)_2$$

ou

$$(-1)^{sign} \times 2^{exposant-127} \times 1.mantisse$$

Mantisse \rightarrow nombre à virgule fixe

Nombres à virgule flottante simple précision

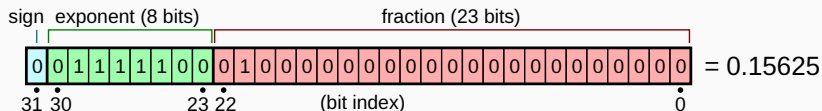


Fig. 3 : Un exercice de nombres à virgule flottante 32 bits. Source : [Wikipedia](#)

$$\text{exposant} = \sum_{i=0}^7 b_{23+i} 2^i = 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = 124 - 127, \quad (1)$$

$$\text{mantisse} = 1 + \sum_{i=1}^{23} b_{23-i} 2^{-i} = 1 + 2^{-2} = 1.25, \quad (2)$$

$$\Rightarrow (-1)^0 \cdot 2^{-3} \cdot 1.25 = 0.15625 \quad (3)$$

Nombres à virgule flottante simple précision

Quel nombre ne peut pas être vraiment représenté ?

Nombres à virgule flottante simple précision

Quel nombre ne peux pas être vraiment représenté ?

Zéro : exception pour l'exposant

Exponent	fraction = 0	fraction \neq 0	Equation
$00_H = 00000000_2$	$\pm zero$	subnormal number	$(-1)^{sign} \times 2^{-126} \times 0.fraction$
$01_H, \dots, FE_H = 00000001_2, \dots, 11111110_2$	normal value		$(-1)^{sign} \times 2^{exponent-127} \times 1.fraction$
$FF_H = 11111111_2$	$\pm infinity$	NaN (quiet, signalling)	

Fig. 4 : Cas particuliers de nombres à virgule flottante 32 bits. Source : [Wikipedia](#)

Nombres à virgule flottante simple précision

Les plus petits/grands nombres positifs représentables

$$0\ 0 \dots 0\ 0 \dots 01 = 2^{-126} \cdot 2^{-23} = 1.4\dots \cdot 10^{-45}, \quad (4)$$

$$0\ 1 \dots 10\ 1 \dots 1 = 2^{127} \cdot (2 - 2^{-23}) = 3.4\dots \cdot 10^{38}. \quad (5)$$

Nombre de chiffres significatifs en décimal ?

- 24 bits ($23 + 1$) sont utiles pour la mantisse, soit $2^{24} - 1$:
 - La mantisse fait $\sim 2^{24} \sim 10^7$, ou encore
 - Ou encore $\sim \log_{10}(2^{24}) \sim 7$,
- Environ **sept** chiffres significatifs.

Nombres à virgule flottante double précision (64bits)

Spécification

- 1 bit de signe,
- 11 bits d'exposant,
- 52 bits de mantisse.

Nombre de chiffres significatifs

- La mantisse fait $\sim 2^{53} \sim 10^{16}$,
- Ou encore $\sim \log_{10}(2^{53}) \sim 16$,
- Environ **seize** chiffres significatifs.

Plus petit/plus grand nombre représentable

- Plus petite mantisse et exposant : $\sim 2^{-52} \cdot 2^{-1022} \sim 4 \cdot 10^{-324}$,
- Plus grande mantisse et exposant : $\sim 2 \cdot 2^{1023} \sim 1.8 \cdot 10^{308}$.

Erreur de représentation

- Les nombres réels ont potentiellement un **nombre infini** de décimales
 - $1/3 = 0.\overline{3}$,
 - $\pi = 3.1415926535\dots$
- Les nombres à virgule flottante peuvent en représenter qu'un **nombre fini**.
 - $1/3 \cong 0.33333$, erreur $0.00000\overline{3}$.
 - $\pi \cong 3.14159$, erreur $0.0000026535\dots$

On rencontre donc des **erreurs de représentation** ou **erreurs d'arrondi**.

Précision finie

Erreur de représentation

- Les nombres réels ont potentiellement un **nombre infini** de décimales
 - $1/3 = 0.\overline{3}$,
 - $\pi = 3.1415926535\dots$
- Les nombres à virgule flottante peuvent en représenter qu'un **nombre fini**.
 - $1/3 \cong 0.33333$, erreur $0.00000\overline{3}$.
 - $\pi \cong 3.14159$, erreur $0.0000026535\dots$

On rencontre donc des **erreurs de représentation** ou **erreurs d'arrondi**.

Et quand on calcule ?

- Avec deux chiffres significatifs

$$8.9 + (0.02 + 0.04) = 8.96 = 9.0, \quad (6)$$

$$(8.9 + 0.02) + 0.04 = 8.9 + 0.04 = 8.9. \quad (7)$$

Même pas associatif !

Erreur de représentation virgule flottante

$$(1.2)_{10} = 1.\overline{0011} \cdot 2^0 \Rightarrow 0\ 01111111\ 00110011001100110011010.$$

Erreur d'arrondi dans les deux derniers bits et tout ceux qui viennent ensuite

$$\varepsilon_2 = (000000000000000000000011)_2.$$

Ou en décimal

$$\varepsilon_{10} = 4.76837158203125 \cdot 10^{-8}.$$

Comment définir l'égalité de 2 nombres à virgule flottante ?

Comment définir l'égalité de 2 nombres à virgule flottante ?

Ou en d'autres termes, pour quel $\varepsilon > 0$ (appelé epsilon-machine) on a

$$1 + \varepsilon = 1,$$

pour un nombre à virgule flottante ?

Comment définir l'égalité de 2 nombres à virgule flottante ?

Ou en d'autres termes, pour quel $\varepsilon > 0$ (appelé epsilon-machine) on a

$$1 + \varepsilon = 1,$$

pour un nombre à virgule flottante ?

Pour un float (32 bits) la différence est à

$$2^{-23} = 1.19 \cdot 10^{-7},$$

Soit la précision de la mantisse.

Erreurs d'arrondi

Et jusqu'ici on a encore pas fait d'arithmétique !

Multiplication avec deux chiffres significatifs, décimal

$$(1.1)_{10} \cdot (1.1)_{10} = (1.21)_{10} = (1.2)_{10}.$$

En continuant ce petit jeu :

$$\underbrace{1.1 \cdot 1.1 \dots 1.1}_{10 \text{ fois}} = 2.0.$$

Alors qu'en réalité

$$1.1^{10} = 2.5937...$$

Soit une erreur de près de $1/5e$!

Erreurs d'arrondi

Et jusqu'ici on a encore pas fait d'arithmétique !

Multiplication avec deux chiffres significatifs, décimal

$$(1.1)_{10} \cdot (1.1)_{10} = (1.21)_{10} = (1.2)_{10}.$$

En continuant ce petit jeu :

$$\underbrace{1.1 \cdot 1.1 \dots 1.1}_{10 \text{ fois}} = 2.0.$$

Alors qu'en réalité

$$1.1^{10} = 2.5937...$$

Soit une erreur de près de 1/5e !

Le même phénomène se produit (à plus petite échelle) avec les float ou double.

Types composés : struct

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

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

Simplifications

- `typedef` permet de définir un nouveau type.

```
typedef unsigned int uint;
typedef struct fraction fraction_t;
typedef struct fraction {
    int 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};
fraction_t frac = {.denom = 1}; // argl! .num non initialisé
fraction_t frac2 = frac; // copie
```


Types composés : struct

Accès aux champs

- On peut accéder aux champs d'un struct avec le sélecteur .

```
fraction t frac;  
frac.num = 2;  
int n = frac.num;
```

Types composés : struct

Pointeurs

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

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

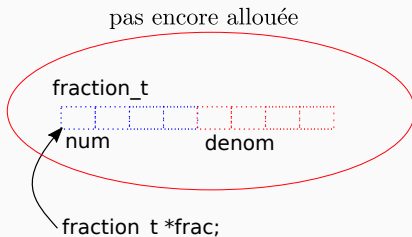


Fig. 5 : La représentation mémoire de `fraction_t`.

Types composés : struct

Initialisation

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

```
void fraction_init(fraction_t *frac,
                  int32_t re, int32_t im)
{
    // frac a déjà été allouée
    frac->num    = re;
    frac->denom  = im;
}

int main() {
    fraction_t frac; // on alloue une fraction
    fraction_init(&frac, 2, -1); // on l'initialise
}
```

Types composés : struct

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 re, int32_t im) {  
    fraction_t frac;  
    frac.num = re;  
    frac.denom = im;  
    return frac;  
}  
  
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);  
}
```