Bases de programmation – Cours 4. Fonctions (1/2)

Pierre Boudes

20 novembre 2014

This work is licensed under the *Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License*.

Cours en plusieurs parties sur les fonctions

1. Principe de base, intérêt et analogie mathématique
2. *Déclarer, appeler, définir* ou *appeler, déclarer, définir*
3. Fonctions sans entrée ou sans sortie, effets de bord
4. Pile d'appel, fonctions récursives
5. Cours suivant : autres types que les entiers pour les paramètres et la sortie

Plan de la séance

Définition et intérêt des fonctions en informatique
- Analogie mathématique
- La notion mathématique de fonction

Les fonctions en langage C
- Appeler des fonctions
- Créer des fonctions : déclarer, définir
- Fonctions sans valeurs de retour (void)
- Utiliser les fonctions d'une bibliothèque

Traces et pile d'appel
- Traces : flot de contrôle et données
- Traces : la mémoire et le temps
- Pile d'appel

Définition et intérêt des fonctions en informatique

En informatique, une fonction est une portion de code représentant un sous programme, qui effectue une tâche ou un calcul relativement indépendant du reste du programme. (wikipédia)

Définition et intérêt des fonctions en informatique

En informatique, une fonction est une portion de code représentant un sous programme, qui effectue une tâche ou un calcul relativement indépendant du reste du programme. (wikipédia)

Les fonctions c'est de la sous-traitance en interne.

Définition et intérêt des fonctions en informatique

En informatique, une fonction est une portion de code représentant un sous programme, qui effectue une tâche ou un calcul relativement indépendant du reste du programme. (wikipédia)

Les fonctions c'est de la sous-traitance en interne.

Intérêt des fonctions :

- *factorisation* : éviter de recopier du code.

Définition et intérêt des fonctions en informatique

En informatique, une fonction est une portion de code représentant un sous programme, qui effectue une tâche ou un calcul relativement indépendant du reste du programme. (wikipédia)

Les fonctions c'est de la sous-traitance en interne.

Intérêt des fonctions :

- *factorisation* : éviter de recopier du code.
- *réutilisation* : copier-coller dans d'autres programmes, utiliser des bibliothèques (`rand()`, etc.) ;

Définition et intérêt des fonctions en informatique

En informatique, une fonction est une portion de code représentant un sous programme, qui effectue une tâche ou un calcul relativement indépendant du reste du programme. (wikipédia)

Les fonctions c'est de la sous-traitance en interne.

Intérêt des fonctions :

- *factorisation* : éviter de recopier du code.
- *réutilisation* : copier-coller dans d'autres programmes, utiliser des bibliothèques (`rand()`, etc.) ;
- *lisibilité* : faciliter la relecture du code et cacher les détails ;

Définition et intérêt des fonctions en informatique

En informatique, une fonction est une portion de code représentant un sous programme, qui effectue une tâche ou un calcul relativement indépendant du reste du programme. (wikipédia)

Les fonctions c'est de la sous-traitance en interne.

Intérêt des fonctions :

- *factorisation* : éviter de recopier du code.
- *réutilisation* : copier-coller dans d'autres programmes, utiliser des bibliothèques (`rand()`, etc.) ;
- *lisibilité* : faciliter la relecture du code et cacher les détails ;
- *structuration* : découper en sous-problèmes de tailles pensables ; distribuer la programmation à différente personnes, à différentes étapes de la réalisation d'un projet.

Définition et intérêt des fonctions en informatique

En informatique, une fonction est une portion de code représentant un sous programme, qui effectue une tâche ou un calcul relativement indépendant du reste du programme. (wikipédia)

Les fonctions c'est de la sous-traitance en interne.

Intérêt des fonctions :

- *factorisation* : éviter de recopier du code.
- *réutilisation* : copier-coller dans d'autres programmes, utiliser des bibliothèques (`rand()`, etc.) ;
- *lisibilité* : faciliter la relecture du code et cacher les détails ;
- *structuration* : découper en sous-problèmes de tailles pensables ; distribuer la programmation à différente personnes, à différentes étapes de la réalisation d'un projet.

Analogie mathématique

- De même que le terme *variable* a un sens différent en informatique et en mathématiques, le terme *fonction* recouvre des réalités différentes.
- Pour les fonctions, les deux notions sont historiquement très proches. C'est bien pour écrire des fonctions mathématiques que les fonctions informatiques ont été introduites.
- Mais qu'est-ce qu'une fonction mathématique ?

La notion mathématique de fonction

Une fonction mathématique est :

La notion mathématique de fonction

Une fonction mathématique est :

- Une courbe ?
- Une expression avec inconnues ($x^2 + 1$), une formule ?
- Un terme dans une algèbre de fonctions ($f \circ g$, f') ?

La notion mathématique de fonction

Une fonction mathématique est :

- Une courbe ?
- Une expression avec inconnues ($x^2 + 1$), une formule ?
- Un terme dans une algèbre de fonctions ($f \circ g$, f') ?
- Une relation entre deux ensembles, entrées et sorties, telle que chaque entrée est en relation avec au plus une sortie (toujours la même). Le graphe de la fonction.

La notion mathématique de fonction

Une fonction mathématique est :

- Une courbe ?
- Une expression avec inconnues ($x^2 + 1$), une formule ?
- Un terme dans une algèbre de fonctions ($f \circ g$, f') ?
- Une relation entre deux ensembles, entrées et sorties, telle que chaque entrée est en relation avec au plus une sortie (toujours la même). Le graphe de la fonction.

À retenir

En mathématiques, la manière dont la fonction est calculée ne fait pas partie de l'objet défini. Une fonction est une boîte noire.

Déclarer, appeler, définir.

Déclarer, appeler, définir.

- *Déclarer* : comme les variables, les fonctions doivent être déclarées avant usage pour fixer le nombre et le type des arguments et de la sortie.

Déclarer, appeler, définir.

- *Déclarer* : comme les variables, les fonctions doivent être déclarées avant usage pour fixer le nombre et le type des arguments et de la sortie.
- *Appeler* : utiliser une fonction, faire appel à son résultat en fixant les valeurs des arguments (paramètres effectifs).

Déclarer, appeler, définir.

- *Déclarer* : comme les variables, les fonctions doivent être déclarées avant usage pour fixer le nombre et le type des arguments et de la sortie.
- *Appeler* : utiliser une fonction, faire appel à son résultat en fixant les valeurs des arguments (paramètres effectifs).
- *Définir* : décrire le corps de la fonction, c'est à dire la suite d'instructions qui constitue son calcul (sur des paramètres formels).

Déclarer, appeler, définir.

- *Déclarer* : comme les variables, les fonctions doivent être déclarées avant usage pour fixer le nombre et le type des arguments et de la sortie.
- *Appeler* : utiliser une fonction, faire appel à son résultat en fixant les valeurs des arguments (paramètres effectifs).
- *Définir* : décrire le corps de la fonction, c'est à dire la suite d'instructions qui constitue son calcul (sur des paramètres formels).

Ordre de la lecture (compilation) : déclaration puis appels et définition

Ordre de l'écriture (création) : appels puis déclaration et définition

Utiliser des fonctions : appeler

Supposons que l'on dispose d'une fonction `maximum`, qui lorsqu'on lui donne deux entiers nous renvoie le plus grand des deux.

Utiliser des fonctions : appeler

Supposons que l'on dispose d'une fonction `maximum`, qui lorsqu'on lui donne deux entiers nous renvoie le plus grand des deux. Nous pouvons l'utiliser de différentes façons :

```
int main ()
{
  int x = 3;
  int y = 4;
  int z;

  z = maximum(x, y);
```

Utiliser des fonctions : appeler

Supposons que l'on dispose d'une fonction `maximum`, qui lorsqu'on lui donne deux entiers nous renvoie le plus grand des deux. Nous pouvons l'utiliser de différentes façons :

```
int main ()
{
  int x = 3;
  int y = 4;
  int z;

  z = maximum(x, y);
  z = 2 + maximum(3 * 2, z);
```

Utiliser des fonctions : appeler

Supposons que l'on dispose d'une fonction `maximum`, qui lorsqu'on lui donne deux entiers nous renvoie le plus grand des deux. Nous pouvons l'utiliser de différentes façons :

```
int main ()
{
  int x = 3;
  int y = 4;
  int z;

  z = maximum(x, y);
  z = 2 + maximum(3 * 2, z);
  x = maximum(maximum(3, y) + 1, z - 1);
```

Utiliser des fonctions : appeler

Supposons que l'on dispose d'une fonction `maximum`, qui lorsqu'on lui donne deux entiers nous renvoie le plus grand des deux. Nous pouvons l'utiliser de différentes façons :

```
int main ()
{
  int x = 3;
  int y = 4;
  int z;

  z = maximum(x, y);
  z = 2 + maximum(3 * 2, z);
  x = maximum(maximum(3, y) + 1, z - 1);
```

Chaque entrée prend une valeur en fonction de l'expression passée en argument. La fonction prend alors sa valeur de sortie.

Créer des fonctions (1) déclarer

Déclarer une fonction c'est donner son prototype (ou signature), c'est à dire son nom, le type de ses arguments, le type de sa sortie.

Créer des fonctions (1) déclarer

Déclarer une fonction c'est donner son prototype (ou signature), c'est à dire son nom, le type de ses arguments, le type de sa sortie.

```
type_sortie nom_fonction(type1 param1, ..., typen paramn);
```

Créer des fonctions (1) déclarer

Déclarer une fonction c'est donner son prototype (ou signature), c'est à dire son nom, le type de ses arguments, le type de sa sortie.

```
type_sortie nom_fonction(type1 param1, ..., typen paramn);
```

Remarque

Seule la déclaration des types est importante pour l'analyse sémantique. Cependant, le choix de noms de paramètres explicites fournit une documentation minimale.

Créer des fonctions (1) déclarer

Déclarer une fonction c'est donner son prototype (ou signature), c'est à dire son nom, le type de ses arguments, le type de sa sortie.

```
type_sortie nom_fonction(type1 param1, ..., typen paramn);
```

Remarque

Seule la déclaration des types est importante pour l'analyse sémantique. Cependant, le choix de noms de paramètres explicites fournit une documentation minimale.

Exemple

```
int puissance(int, int);
int puissance(int base, int exposant);
```

Créer des fonctions (1) déclarer

Déclarer une fonction c'est donner son prototype (ou signature), c'est à dire son nom, le type de ses arguments, le type de sa sortie.

```
type_sortie nom_fonction(type1 param1, ..., typen paramn);
```

Remarque

Seule la déclaration des types est importante pour l'analyse sémantique. Cependant, le choix de noms de paramètres explicites fournit une documentation minimale.

Exemple

```
int puissance(int, int);
int puissance(int base, int exposant);
```

La déclaration doit se trouver avant tout appel.

Créer des fonctions (2) définir

Créer des fonctions (2) définir

Définir une fonction c'est donner l'ensemble des instructions qui permettent, à partir des paramètres d'entrée, de calculer la valeur de la fonction, dite valeur de retour, valeur renvoyée.

Créer des fonctions (2) définir

Définir une fonction c'est donner l'ensemble des instructions qui permettent, à partir des paramètres d'entrée, de calculer la valeur de la fonction, dite valeur de retour, valeur renvoyée.

```
type_sortie nom_fonction(type1 param1, ..., typen paramn)
{
  /* declaration et initialisation variables */

  /* instructions */
}
```

Créer des fonctions (2) définir

Définir une fonction c'est donner l'ensemble des instructions qui permettent, à partir des paramètres d'entrée, de calculer la valeur de la fonction, dite valeur de retour, valeur renvoyée.

```
type_sortie nom_fonction(type1 param1, ..., typen paramn)
{
  /* declaration et initialisation variables */

  /* instructions */
}
```

Les noms des paramètres formels sont `param1`, ..., `paramn`, ils sont utilisés dans le corps du calcul.

Créer des fonctions (2) définir

Définir une fonction c'est donner l'ensemble des instructions qui permettent, à partir des paramètres d'entrée, de calculer la valeur de la fonction, dite valeur de retour, valeur renvoyée.

```
type_sortie nom_fonction(type1 param1, ..., typen paramn)
{
  /* declaration et initialisation variables */

  /* instructions */
}
```

Les noms des paramètres formels sont `param1, ..., paramn`, ils sont utilisés dans le corps du calcul.
Dès que la valeur de la fonction est calculée, on utilise l'instruction `return` qui marque la fin du calcul et donne sa valeur à la fonction.

Créer des fonctions (2) définir

Définir une fonction c'est donner l'ensemble des instructions qui permettent, à partir des paramètres d'entrée, de calculer la valeur de la fonction, dite valeur de retour, valeur renvoyée.

```
type_sortie nom_fonction(type1 param1, ..., typen paramn)
{
  /* declaration et initialisation variables */

  /* instructions */
}
```

Les noms des paramètres formels sont `param1, ..., paramn`, ils sont utilisés dans le corps du calcul.
Dès que la valeur de la fonction est calculée, on utilise l'instruction `return` qui marque la fin du calcul et donne sa valeur à la fonction.

```
return expression_resultat;
```

La définition est obligatoire pour que l'édition de liens réussisse et que l'exécutable soit créé.

Variables locales, portée, durée

Le corps de la fonction peut déclarer des variables additionnelles, qui sont locales à la fonction (portée) et se voient allouer un espace mémoire pour chaque appel de la fonction (durée).

Remarque

`main` est une fonction (appelée au lancement du programme).

Résumé

Utilisation des fonctions :

- *déclaration* (types des paramètres et de la valeur de retour)
- *définition* (code, paramètres formels)
- *appel* (paramètres effectifs, espace mémoire)

Résumé

Utilisation des fonctions :

- *déclaration* (types des paramètres et de la valeur de retour)
- *définition* (code, paramètres formels)
- *appel* (paramètres effectifs, espace mémoire)

Convention de nommage

Il est pratique de faire commencer le nom de chaque fonction par un verbe à l'infinitif `convertir_...`, `tester_...`, etc. ou éventuellement par un verbe conjugué : par exemple `est_` pour des fonctions à valeur de test (`est_majeur`, `est_premier`, `......`)
Exceptions : la fonction est connue sous un autre nom (`factorielle(n)`, `racine(x)`, `moyenne(...)`)

Fonctions sans valeurs de retour (void)

On parle plutôt de procédure ou de routine car l'analogie avec les fonctions mathématiques est perdue.

Déclarer

```
void afficher_valeurs(int x, int y);
```

Appeler

```
afficher_valeurs(5, 3);
```

Définir

Comme d'habitude mais pas de `return` (ou `return sans argument`).

Fonctions sans arguments

Déclarer

```
int nombre_aleatoire();
int saisie_utilisateur();
```

Appeler

```
int n;
int secret;

secret = nombre_aleatoire();

n = saisie_utilisateur();
```

Définir

Comme d'habitude.

Utiliser les fonctions d'une bibliothèque (math.h)

Utilisation de la bibliothèque math.h

```
$ man math
```

Déclarer

```c
#include <math.h>
```

Appeler

```c
double x; /* nombre a virgule (cours 5) */
x = log(3.5); /* la virgule se note avec un point */
```

Définir

```
$ gcc -lm -Wall prog.c -o prog.exe
```

Traces et pile d'appel

Utilisation des fonctions :

- *déclaration* **types** (`int`, `char`, `double`, `void`) des paramètres et de la valeur de retour ;
- *définition* code, paramètres formels (chacun est une variable locale) ;
- *appel* paramètres effectifs (chaque expression donnant sa valeur au paramètre formel correspondant), **espace mémoire**.

Nous allons voir de façon plus précise cette question d'espace mémoire, avec la *pile d'appel*...

Traces

Pour étudier l'exécution de nos programmes, nous simulons leur exécution à la main. Comment tenir compte de chaque appel de chaque fonction que l'on a défini ?

`main()`

ligne	n	Affichage
initialisation	9	
16		
22		9 n'est pas premier
26	renvoie EXIT_SUCCESS	

`est_premier(9)`

ligne	n	i	Affichage
initialisation	9	?	
34		2	
40		3	
38	renvoie false		

On ne *trace* pas les fonctions externes, comme `printf`.

Focus sur le passage de valeurs

Notez que les fonctions communiquent **des valeurs**, pas des noms de variables.

```
1   void permute_valeurs(int a,int b);
2
3   int main() {
4       int x = 1;
5       int y = 2;
6       permute_valeurs(x,y);
7       printf("x␣=␣%d␣et␣y␣=␣%d\n",x,y);
8       return EXIT_SUCCESS;
9   }
10
11  void permute_valeurs(int a,int b) {
12      int aux;
13      aux = a;
14      a = b;
15      b = aux;
16  }
```

Focus sur le passage de valeurs

Notez que les fonctions communiquent **des valeurs**, pas des noms de variables.

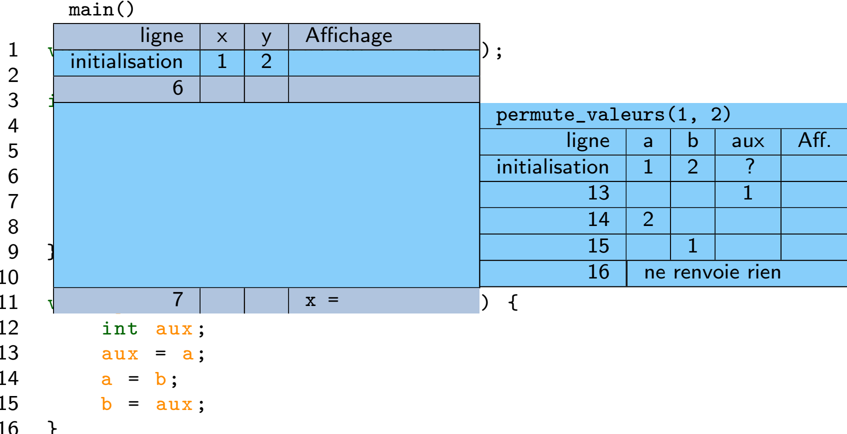
main()

ligne	x	y	Affichage
initialisation	1	2	
6			

```
1   v                                          );
2
3   int main() {
4       int x = 1;
5       int y = 2;
6       permute_valeurs(x,y);
7       printf("x␣=␣%d␣et␣y␣=␣%d\n",x,y);
8       return EXIT_SUCCESS;
9   }
10
11  void permute_valeurs(int a,int b) {
12      int aux;
13      aux = a;
14      a = b;
15      b = aux;
16  }
```

Focus sur le passage de valeurs

Notez que les fonctions communiquent **des valeurs**, pas des noms de variables.



Focus sur le passage de valeurs

Notez que les fonctions communiquent **des valeurs**, pas des noms de variables.

main()

ligne	x	y	Affichage
initialisation	1	2	
6			
7			x = 1 et y = 2
8	SORTIE AVEC SUCCÈS		

permute_valeurs(1, 2)

ligne	a	b	aux	Aff.
initialisation	1	2	?	
13			1	
14	2			
15		1		
16	ne renvoie rien			

```
1   v                                   );
2
3   i
4
5
6
7
8
9   }
10
11  v                                   ) {
12
13      aux = a;
14      a = b;
15      b = aux;
16  }
```

Un exemple avec plus d'appels de fonctions

```
1   double maximum(double x, double y);
2   double valeur_absolue(double x);
3   int main() {
4       double a = -4.2;
5       double b = 1.3;
6       a = valeur_absolue(a) + valeur_absolue(b);
7       printf("%g\n", a);
8       return EXIT_SUCCESS;
9   }
10  double maximum(double x, double y) {
11      if (x > y) {
12          return x;
13      }
14      return y;
15  }
16  double valeur_absolue(double x) {
17      return maximum(x, -x);
18  }
```

Un exemple avec plus d'appels de fonctions

main()

ligne	a	b	Aff.
initialisation	-4.2	1.3	
6			

```c
1  double maximum(double x, double y);
2  double valeur_absolue(double x);
3  int main() {
4      double a = -4.2;
5      double b = 1.3;
6      a = valeur_absolue(a) + valeur_absolue(b);
7      printf("%g\n", a);
8      return EXIT_SUCCESS;
9  }
10 double maximum(double x, double y) {
11     if (x > y) {
12         return x;
13     }
14     return y;
15 }
16 double valeur_absolue(double x) {
17     return maximum(x, -x);
18 }
```

Un exemple avec plus d'appels de fonctions

main()

ligne	a	b	Aff.
initialisation	-4.2	1.3	
6			

valeur_absolue(-4.2)

ligne	x
init.	-4.2
17	

```
1   doubl                              e y);
2   doubl                              );
3   int r
4       c
5       c
6       a                              _absolue(b);
7       p
8       r
9   }
10  doubl                              e y) {
11      if (x > y) {
12          return x;
13      }
14      return y;
15  }
16  double valeur_absolue(double x) {
17      return maximum(x, -x);
18  }
```

Un exemple avec plus d'appels de fonctions

main()

ligne	a	b	Aff.
initialisation	-4.2	1.3	
6			

valeur_absolue(-4.2)

ligne	x
init.	-4.2
17	

maximum(-4.2, 4.2)

ligne	x	y
init.	-4.2	4.2
14		

```
1  doubl                                  e y);
2  doubl                                  );
3  int r
4       c
5       c
6       a
7       p
8       r
9  }
10 doubl                                  e y) {
11      if (x > y) {
12          return x;
13      }
14      return y;
15 }
16 double valeur_absolue(double x) {
17      return maximum(x, -x);
18 }
```

Un exemple avec plus d'appels de fonctions

```
1   doubl                       e y);
2   doubl                       );
3   int m
4       d
5       d
6       a
7       p
8       r
9   }
10  doubl                            {
11      if (x > y) {
12          return x;
13      }
14      return y;
15  }
16  double valeur_absolue(double x) {
17      return maximum(x, -x);
18  }
```

main()

ligne	a	b	Aff.
initialisation	-4.2	1.3	
6			

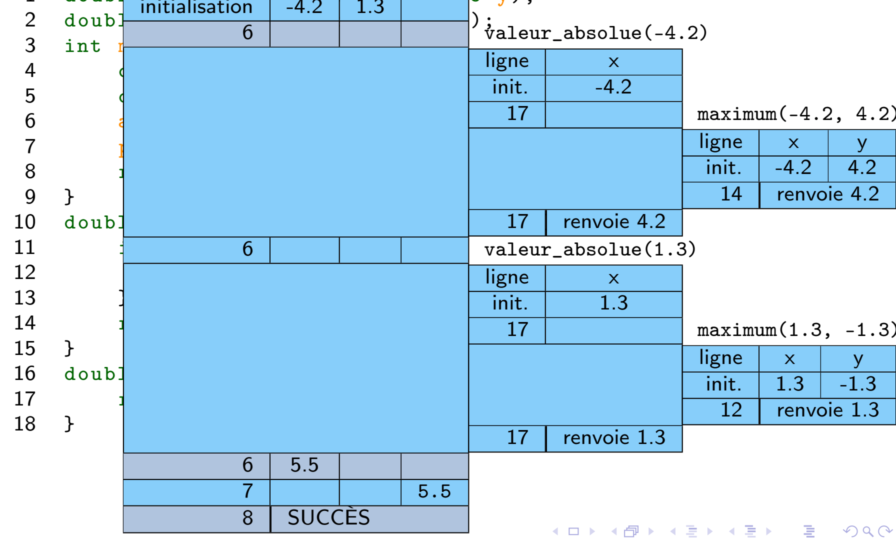
valeur_absolue(-4.2)

ligne	x
init.	-4.2
17	
17	

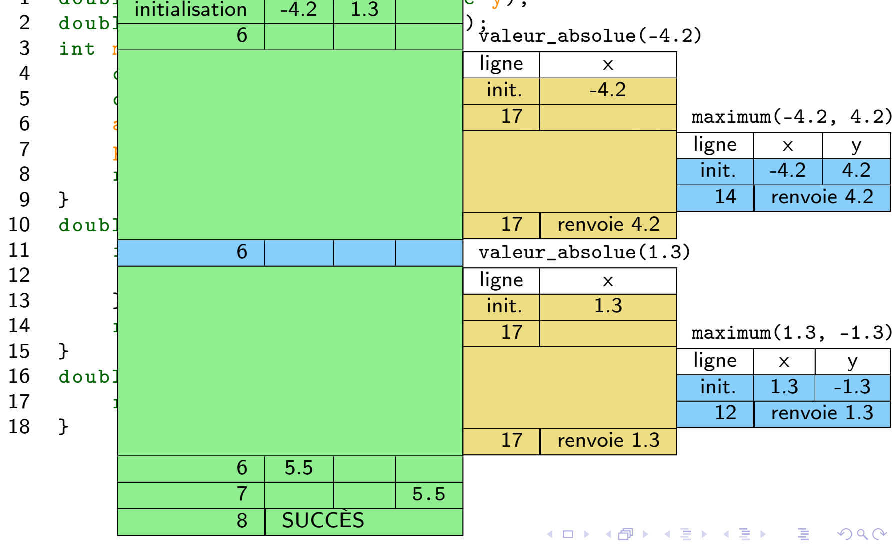
maximum(-4.2, 4.2)

ligne	x	y
init.	-4.2	4.2
14	renvoie 4.2	

Un exemple avec plus d'appels de fonctions



Un exemple avec plus d'appels de fonctions

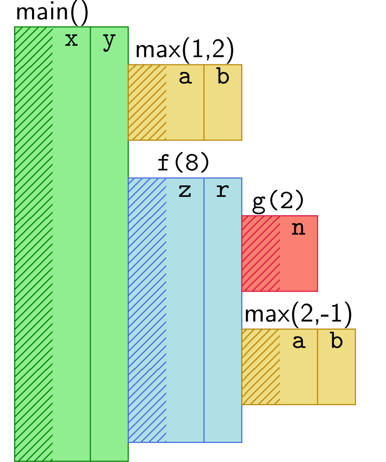


Traces : la mémoire et le temps

La trace d'un programme donne schématiquement ce type de dessin :

Traces : la mémoire et le temps

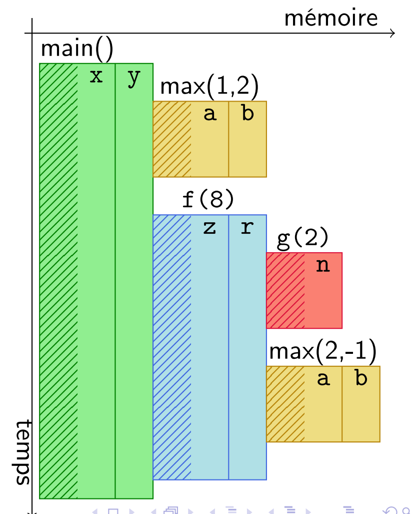
La trace d'un programme donne schématiquement ce type de dessin :



Traces : la mémoire et le temps

La trace d'un programme donne schématiquement ce type de dessin :

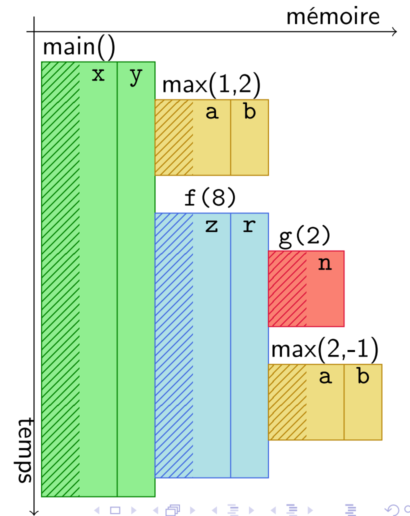
- verticalement, c'est le temps
- et horizontalement, l'occupation mémoire



Traces : la mémoire et le temps

La trace d'un programme donne schématiquement ce type de dessin :

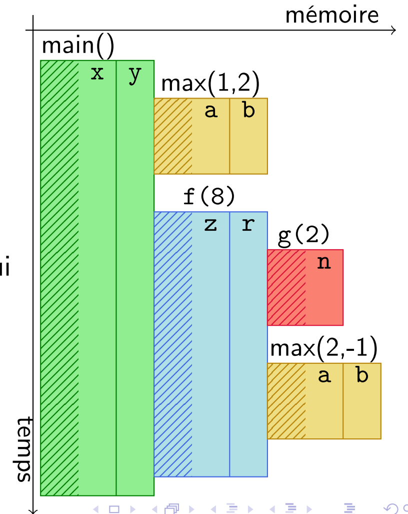
- verticalement, c'est le temps
- et horizontalement, l'occupation mémoire
- un appel de fonction occupe une portion de mémoire, puis la libère.



Traces : la mémoire et le temps

La trace d'un programme donne schématiquement ce type de dessin :

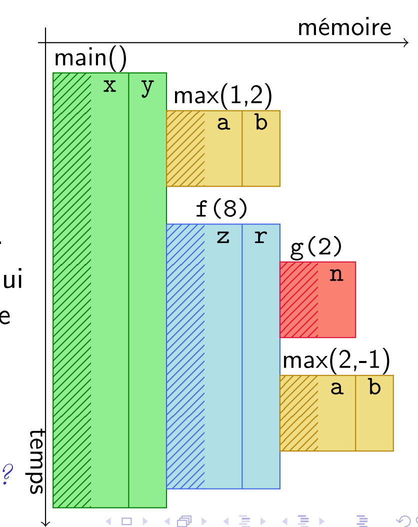
- verticalement, c'est le temps
- et horizontalement, l'occupation mémoire
- un appel de fonction occupe une portion de mémoire, puis la libère.
- la trace représente réellement ce qui arrive dans vos programmes (durée de vie et localisation en mémoire des variables, etc.).



Traces : la mémoire et le temps

La trace d'un programme donne schématiquement ce type de dessin :

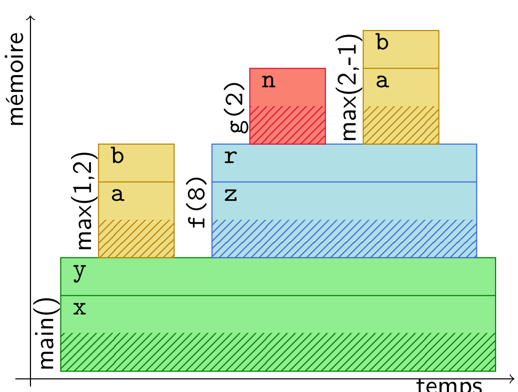
- verticalement, c'est le temps
- et horizontalement, l'occupation mémoire
- un appel de fonction occupe une portion de mémoire, puis la libère.
- la trace représente réellement ce qui arrive dans vos programmes (durée de vie et localisation en mémoire des variables, etc.).



Un appel de fonction peut-il modifier la mémoire d'une fonction appelante ?

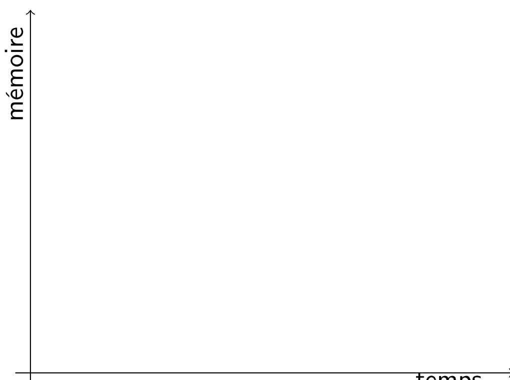
Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent... *comme sur une pile d'assiettes.*



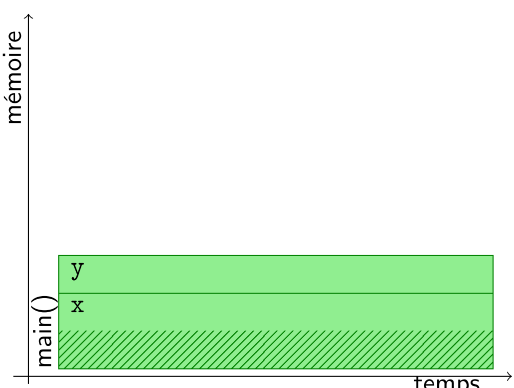
Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent... *comme sur une pile d'assiettes.*



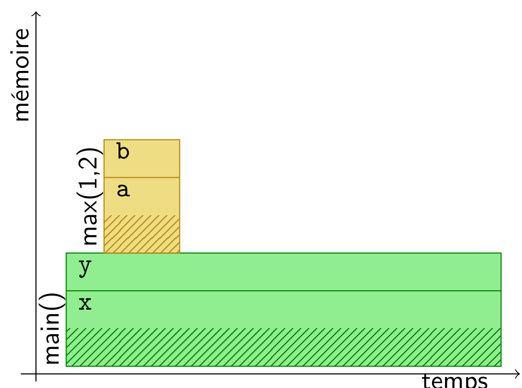
Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent… *comme sur une pile d'assiettes.*



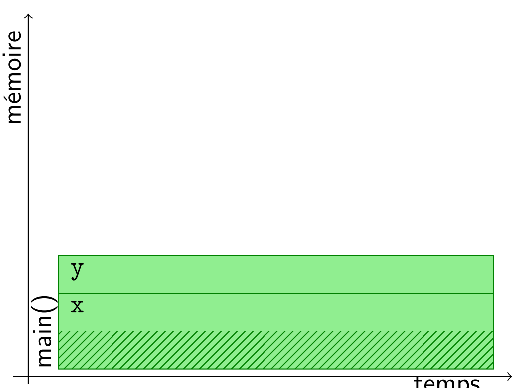
Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent... *comme sur une pile d'assiettes.*



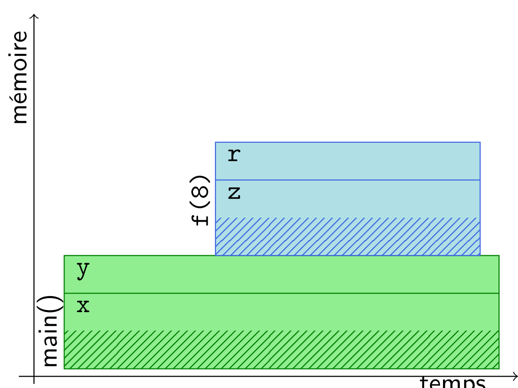
Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent... *comme sur une pile d'assiettes.*



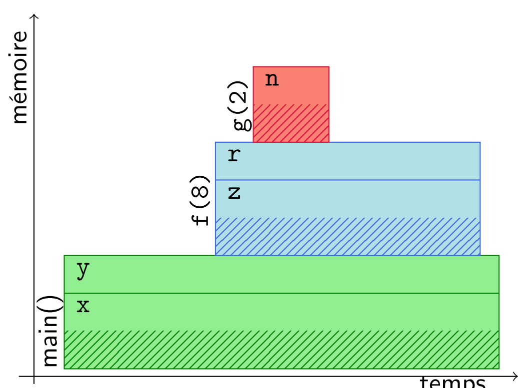
Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent…
comme sur une pile d'assiettes.



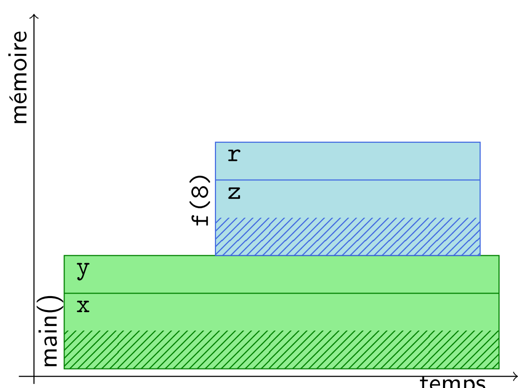
Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent...
comme sur une pile d'assiettes.



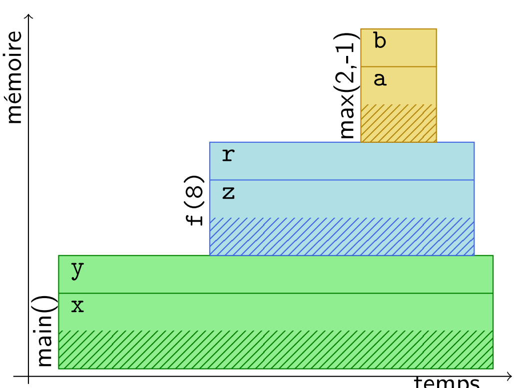
Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent... *comme sur une pile d'assiettes.*



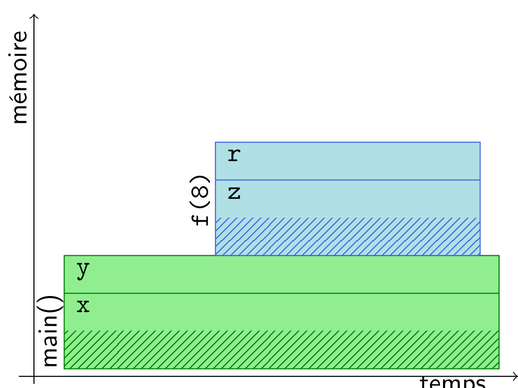
Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent...
comme sur une pile d'assiettes.



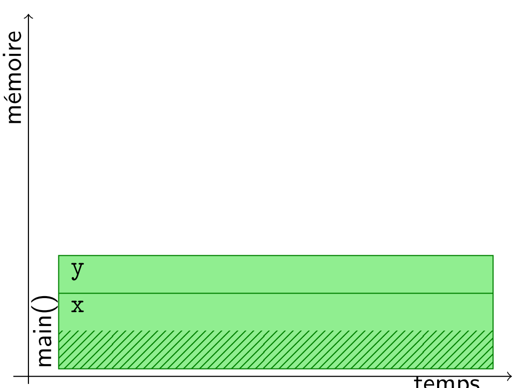
Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent... *comme sur une pile d'assiettes.*



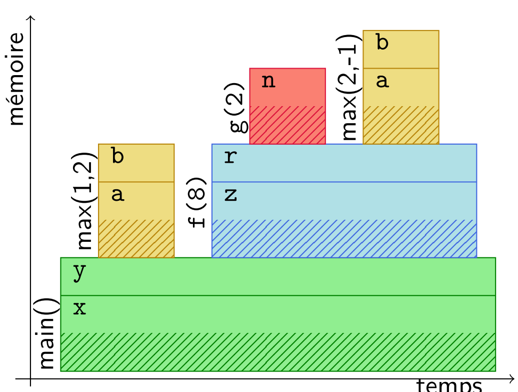
Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent... *comme sur une pile d'assiettes.*



Pile d'appel

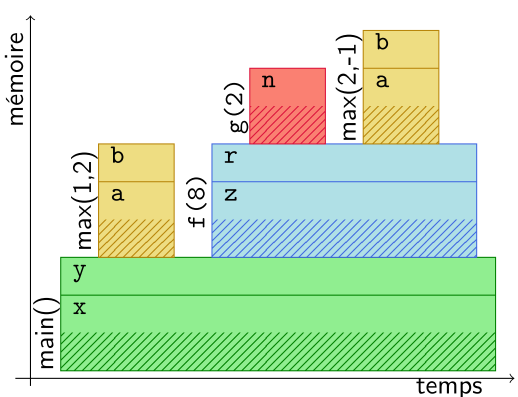
On parle de pile d'appel car les appels de fonctions s'empilent… *comme sur une pile d'assiettes.*



Pile d'appel

On parle de pile d'appel car les appels de fonctions s'empilent... *comme sur une pile d'assiettes.*

Peut-on avoir deux assiettes identiques dans la pile au même instant ? (La même fonction appelée deux fois simultanément)



Factorielle récursive

Définition

Une fonction récursive est une fonction dont la définition fait appel à la fonction *elle-même*.

Factorielle récursive

Définition

Une fonction récursive est une fonction dont la définition fait appel à la fonction *elle-même*.

Il y a une forte analogie avec les maths : $(n+1)! = (n+1) \times n!$

Factorielle récursive

Définition

Une fonction récursive est une fonction dont la définition fait appel à la fonction *elle-même*.

Il y a une forte analogie avec les maths : $(n+1)! = (n+1) \times n!$

```
int factorielle(int n)
{
    if (n < 2) /* cas de base */
    {
        return 1;
    }
    return n * factorielle(n - 1);
}
```