Algorithmique et programmation. Fonctions et procédures (2)

Pierre Boudes

9 octobre 2012

Types char et double

- Représentation des réels en virgule flottante
- Types et entrées sorties
- Conversions automatiques entre types

Fonctions et procédures, compléments

- Rappel sur les fonctions en C
- 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

Longue démo (menu)

Représentation des réels en virgule flottante

- La représentation informatique usuelle des réels s'inspire de la notation scientifique :

$$\pi = 3,141592653589793 \quad \text{(pi)}$$

$$-700 \text{ milliards} = -7 \times 10^{11} \quad \text{(Paulson)}$$

$$h = 6,626068 \times 10^{-34} \quad \text{(Planck)}$$

$$\text{Univers} = 1 \times 10^{80} \quad \text{(Atomes)}$$

Représentation des réels en virgule flottante

- La représentation informatique usuelle des réels s'inspire de la notation scientifique :

$$\pi = 3,141592653589793 \quad \text{(pi)}$$

$$-700 \text{ milliards} = -7 \times 10^{11} \quad \text{(Paulson)}$$

$$h = 6,626068 \times 10^{-34} \quad \text{(Planck)}$$

$$\text{Univers} = 1 \times 10^{80} \quad \text{(Atomes)}$$

- Les bits sont séparés en :
  - bit de signe (1 bit)
  - mantisse (53 bits)
  - exposant (11 bits)

Représentation des réels en virgule flottante

- La représentation informatique usuelle des réels s'inspire de la notation scientifique :

$$\pi = 3,141592653589793 \quad \text{(pi)}$$

$$-700 \text{ milliards} = -7 \times 10^{11} \quad \text{(Paulson)}$$

$$h = 6,626068 \times 10^{-34} \quad \text{(Planck)}$$

$$\text{Univers} = 1 \times 10^{80} \quad \text{(Atomes)}$$

- Les bits sont séparés en :
  - bit de signe (1 bit)
  - mantisse (53 bits)
  - exposant (11 bits)
- En **double** précision (64 bits) :
  - exposant : entre 10^{-308} et 10^{308} (environ).
  - mantisse : 16 chiffres décimaux (environ).

Représentation des réels en virgule flottante

- La représentation informatique usuelle des réels s'inspire de la notation scientifique :

$$\pi = 3,141592653589793 \quad \text{(pi)}$$

$$-700 \text{ milliards} = -7 \times 10^{11} \quad \text{(Paulson)}$$

$$h = 6,626068 \times 10^{-34} \quad \text{(Planck)}$$

$$\text{Univers} = 1 \times 10^{80} \quad \text{(Atomes)}$$

- Les bits sont séparés en :
  - bit de signe (1 bit)
  - mantisse (53 bits)
  - exposant (11 bits)
- En **double** précision (64 bits) :
  - exposant : entre 10^{-308} et 10^{308} (environ).
  - mantisse : 16 chiffres décimaux (environ).
- Infini positif, infini négatif.
- NaN : not a number.

Type double en C et entrées/sorties associées

- Type des entiers relatifs `int` (rappel) :
  - Déclaration et initialisation : `int n = -23;`.
  - Représentation en complément à deux.
  - E/S : `%d`.
- Type des réels `double` :
  - Déclaration et initialisation : `double x = 3.14e-3;`.
  - Représentation en virgule flottante sur 64 bits.
  - E/S : `%lg` (mais plutôt %g avec `printf`).
  - Attention : toujours mettre le point (équivalent anglais de la virgule) pour les constantes réelles (`1.0`).

Entiers

```
int n;
...
printf("Entrer un nombre entier\n");
scanf("%d", &n);
```

Entiers

```
int n;
...
printf("Entrer un nombre entier\n");
scanf("%d", &n);
```

Réels

```
double x;
...
printf("Entrer un nombre reel\n");
scanf("%lg", &x);
```

Entiers

```
int n;
...
printf("Entrer un nombre entier\n");
scanf("%d", &n);
```

Réels

```
double x;
...
printf("Entrer un nombre reel\n");
scanf("%lg", &x);
printf("Vous avez saisi : %g\n", x);
```

Remarque : on tombe vite sur un problème (boucle infinie) avec scanf car cette fonction s'occupe à la fois de reconnaître ce que tape l'utilisateur et de *purger* cette entrée. Mais scanf ne purge pas ce qui n'est pas reconnu (démo) ! Il faudra séparer purge et reconnaissance.

Type char en C et entrées/sorties associées

Type des caractères `char` :

- Déclaration et initialisation : `char c = 'A';`.
- Représentation sur 8 bits, ASCII, ISO-8859-x, UTF-8.
- E/S : `%c`.

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Source : Wikimedia Commons, public domain.

Caractères

```
char c;
...
printf("Entrer un caractère\n");
scanf("%c", &c);
```

Caractères

```
char c;
...
printf("Entrer un caractère\n");
scanf("%c", &c);
```

Attention : mieux vaut utiliser `scanf("␣%c", &c);`

Caractères

```
char c;
...
printf("Entrer un caractère\n");
scanf("%c", &c);
```

Attention : mieux vaut utiliser `scanf("␣%c", &c);`

Chaînes de caractères (semaine prochaine)

```
char nom[64];
...
printf("Entrer votre nom\n");
scanf("%s", nom);
```

Caractères

```
char c;
...
printf("Entrer un caractère\n");
scanf("%c", &c);
```

Attention : mieux vaut utiliser `scanf("␣%c", &c);`

Chaînes de caractères (semaine prochaine)

```
char nom[64];
...
printf("Entrer votre nom\n");
scanf("%s", nom);
```

Conversions automatiques entre types

- Sans changement de représentation :
  - `char` vers `int`
  - `int` vers `char` (troncature)

Conversions automatiques entre types

- Sans changement de représentation :
  - `char` vers `int`
  - `int` vers `char` (troncature)

```
char c;
int n;

n = 'A' + 1; /* voir table ascii */
c = n + 24; /* quel caractere vaut c ? */
```

- Avec changement de représentation :
  - char ou entiers vers réels
  - réels vers entiers ou char

```
double x;
int n;

n = 3.1; /* que vaut n ? */
x = n;
```

Rappel sur les fonctions en C

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**.

Rappel sur les fonctions en C

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**.

Rappel sur les fonctions en C

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.

Rappel sur les fonctions en C

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.
avant cela, quelques compléments (`void`, bibliothèques de fonction).

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

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

Appeler

```
double x;

x = log(3.5);
```

Définir

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

Traces (rappel)

Pour étudier l'exécution de nos programmes, nous faisons la trace de chaque appel de chaque fonction que l'on a défini (les fonctions utilisateur, pas les fonctions externes, comme printf).

`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		

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.

main()

ligne	x	y	Affichage
initialisation	1	2	
6			

permute_valeurs(1, 2)

ligne	a	b	aux	Aff.
initialisation	1	2	?	

```
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.

main()

ligne	x	y	Affichage
initialisation	1	2	
6			
7			x =

permute_valeurs(1, 2)

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

```
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",
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			
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  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",
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 }
```

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;c
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			

```
1   doubl                                    e y);
2   doubl                                    );
3   int main() {
4       double a = -4.2;c
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   double                                    e y);
2   double                                    );
3   int main() {
4       double a = -4.2;c
5       double b = 1.3;
6       a = valeur_absolue(a) + va        _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

```
1   double                                  e y);
2   double                                  );
3   int main() {
4       double a = -4.2;c
5       double b = 1.3;
6       a = valeur_absolue(a) + va
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  }
```

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		

Un exemple avec plus d'appels de fonctions

```
1   double
2   double
3   int main() {
4       double a = -4.2;c
5       double b = 1.3;
6       a = valeur_absolue(a) + va
7       printf("%g\n", a);
8       return EXIT_SUCCESS;
9   }
10  double maximum(double x, 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	

maximum(-4.2, 4.2)

ligne	x	y
init.	-4.2	4.2
14	renvoie 4.2	

17

Un exemple avec plus d'appels de fonctions

```
1  double valeur_absolue(double x);
2  double maximum(double x, double y);
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 }
```

main()

ligne	a	b	Aff.
initialisation	-4.2	1.3	
6			
6	5.5		
7			5.5
8	SUCCÈS		

valeur_absolue(-4.2)

ligne	x
init.	-4.2
17	
17	renvoie 4.2

maximum(-4.2, 4.2)

ligne	x	y
init.	-4.2	4.2
14	renvoie 4.2	

valeur_absolue(1.3)

ligne	x
init.	1.3
17	
17	renvoie 1.3

maximum(1.3, -1.3)

ligne	x	y
init.	1.3	-1.3
12	renvoie 1.3	

Un exemple avec plus d'appels de fonctions

```
1   doubl                                   e y);
2   doubl                                   );
3   int r
4           c
5           c
6           a
7           b
8           r
9   }
10  doubl
11          r
12
13          }
14          r
15  }
16  doubl
17          r
18  }
```

main()

ligne	a	b	Aff.
initialisation	-4.2	1.3	
6			
6			
6	5.5		
7			5.5
8	SUCCÈS		

valeur_absolue(-4.2)

ligne	x
init.	-4.2
17	
17	renvoie 4.2

maximum(-4.2, 4.2)

ligne	x	y
init.	-4.2	4.2
14	renvoie 4.2	

valeur_absolue(1.3)

ligne	x
init.	1.3
17	
17	renvoie 1.3

maximum(1.3, -1.3)

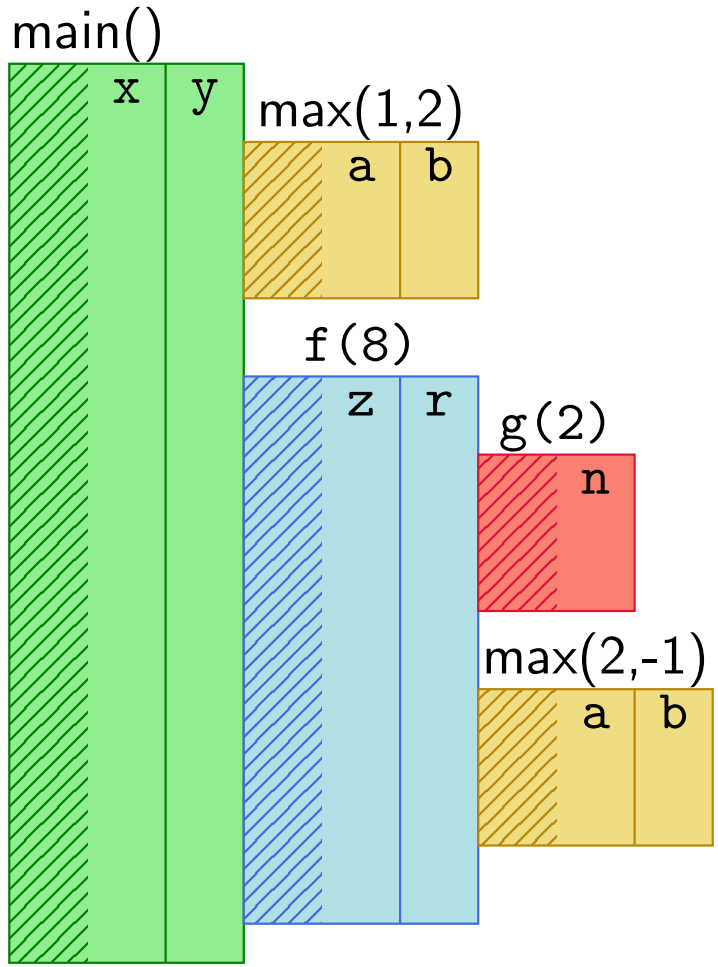
ligne	x	y
init.	1.3	-1.3
12	renvoie 1.3	

Traces : la mémoire et le temps

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

Traces : la mémoire et le temps

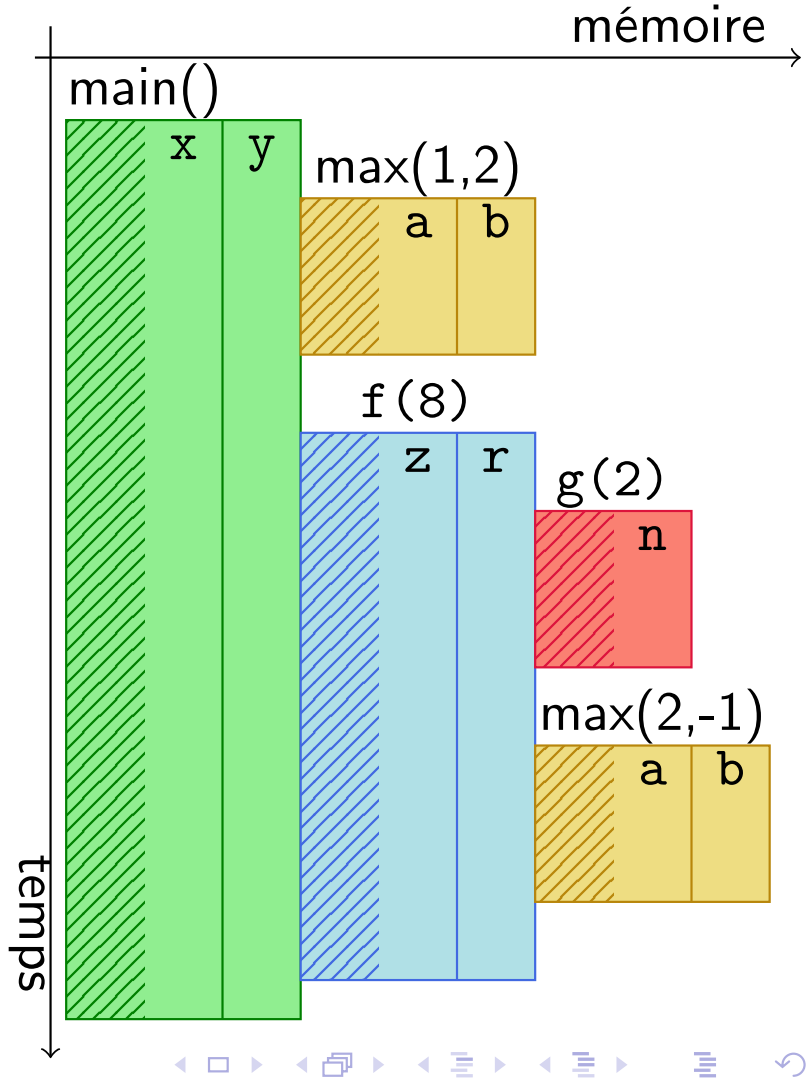
La trace d'un programme donne schématiquement ce type de dessin :



Traces : la mémoire et le temps

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

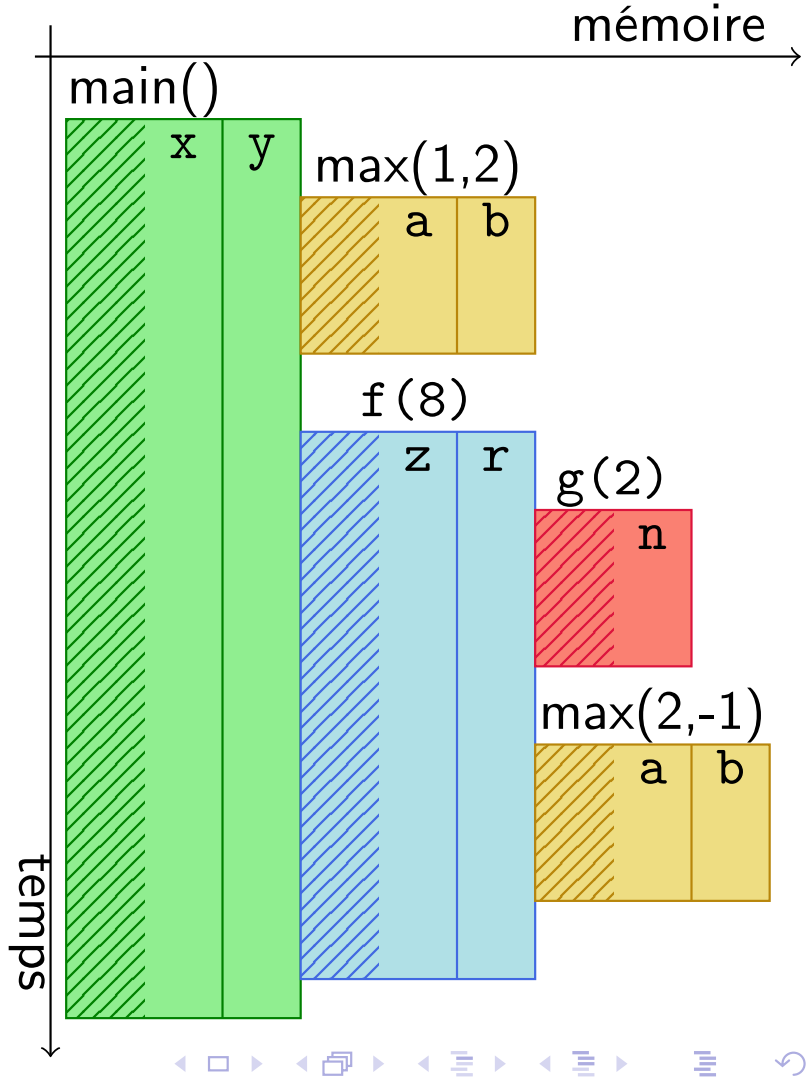
- verticalement, c'est le temps
- et horizontalement, l'occupation mémoire



Traces : la mémoire et le temps

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

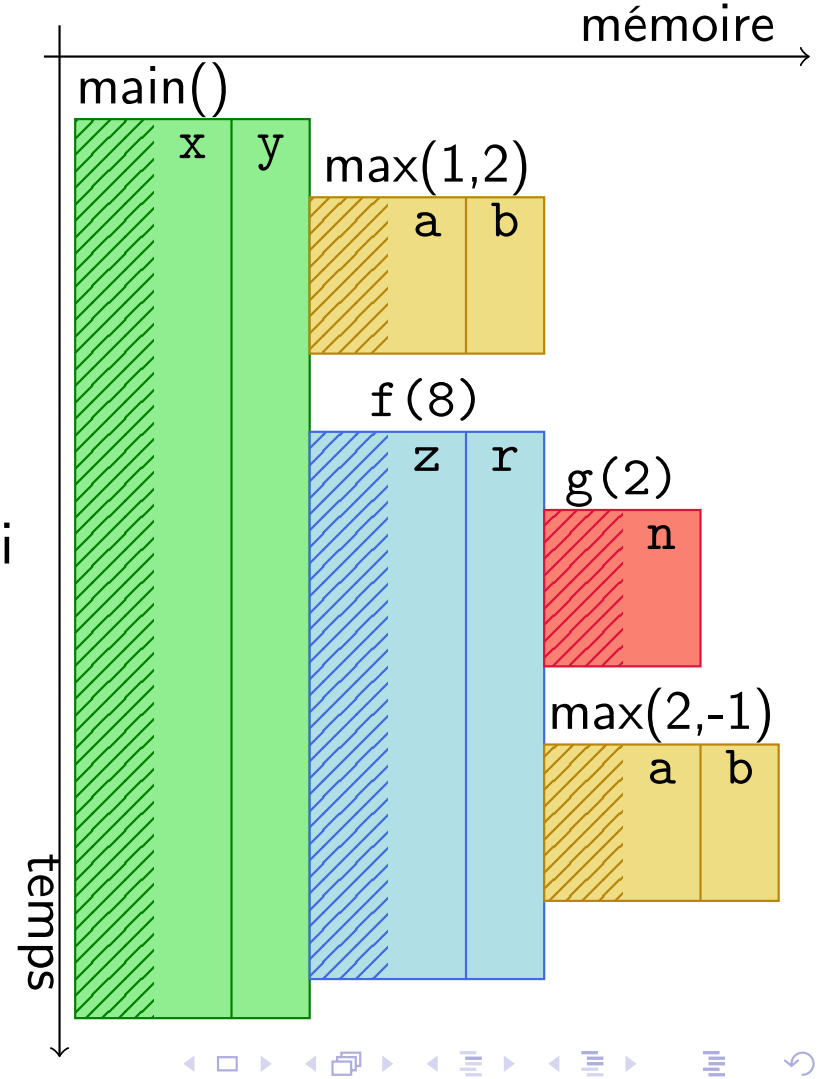
- verticalement, c'est le temps
- et horizontalement, l'occupation mémoire
- un appel de fonction occupe une portion de mémoire, puis la libère.



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.).

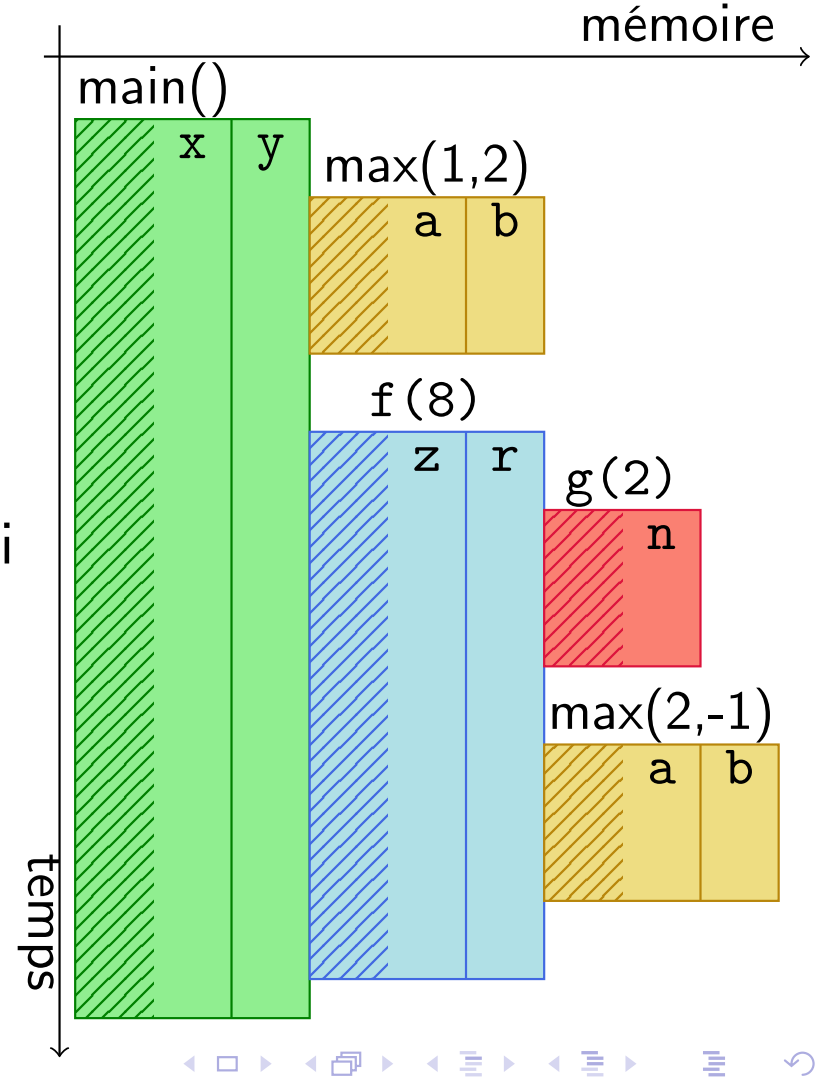


Traces : la mémoire et le temps

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

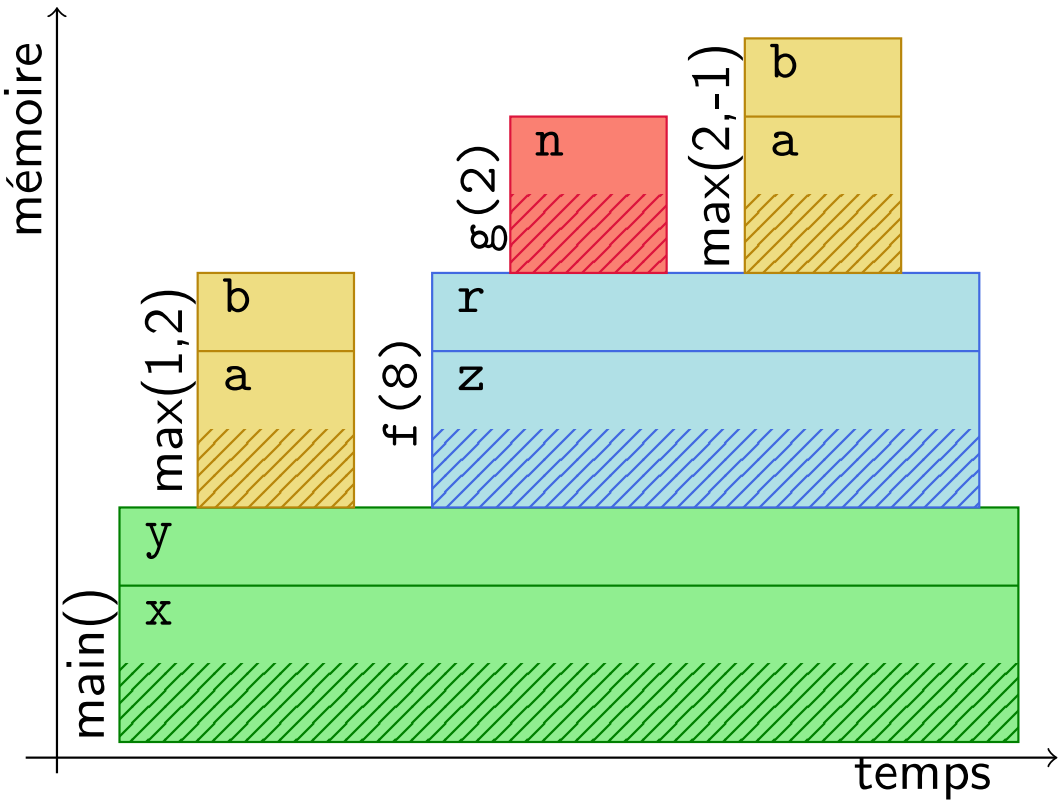
- verticalement, c'est le temps
- et horizontalement, l'occupation mémoire
- un appel de fonction occupe une portion de mémoire, puis la libère.
- la trace représente réellement ce qui arrive dans vos programmes (durée de vie et localisation en mémoire des variables, etc.).

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



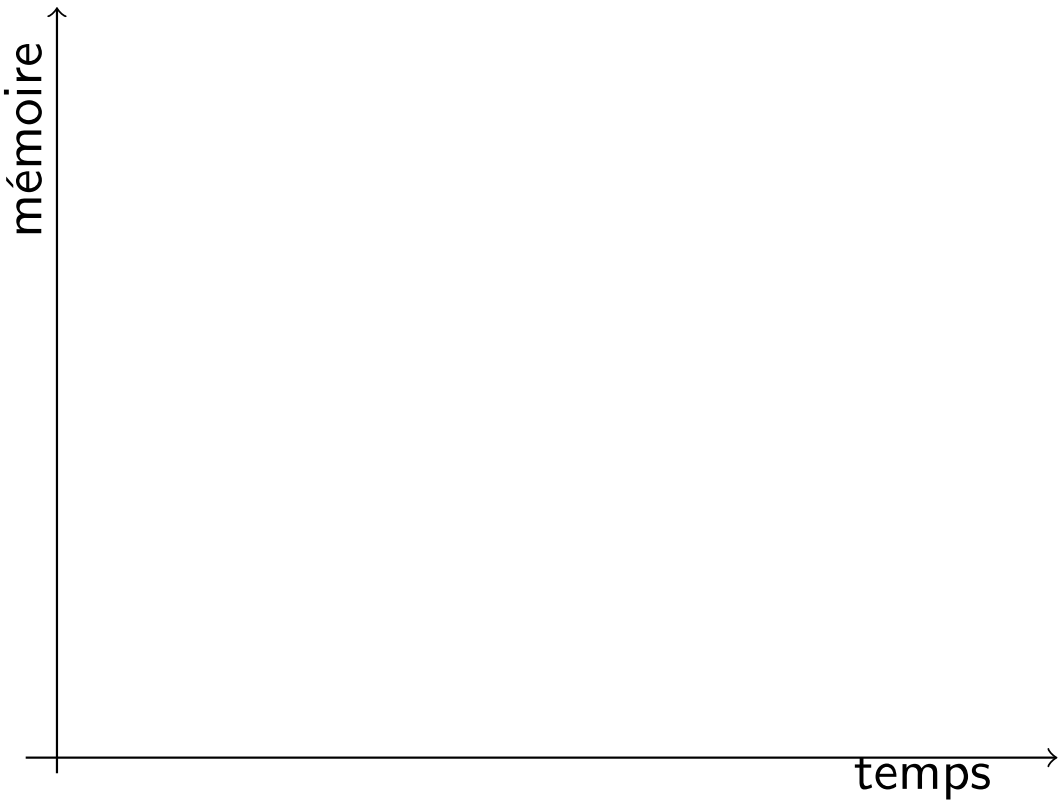
Pile d'appel

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



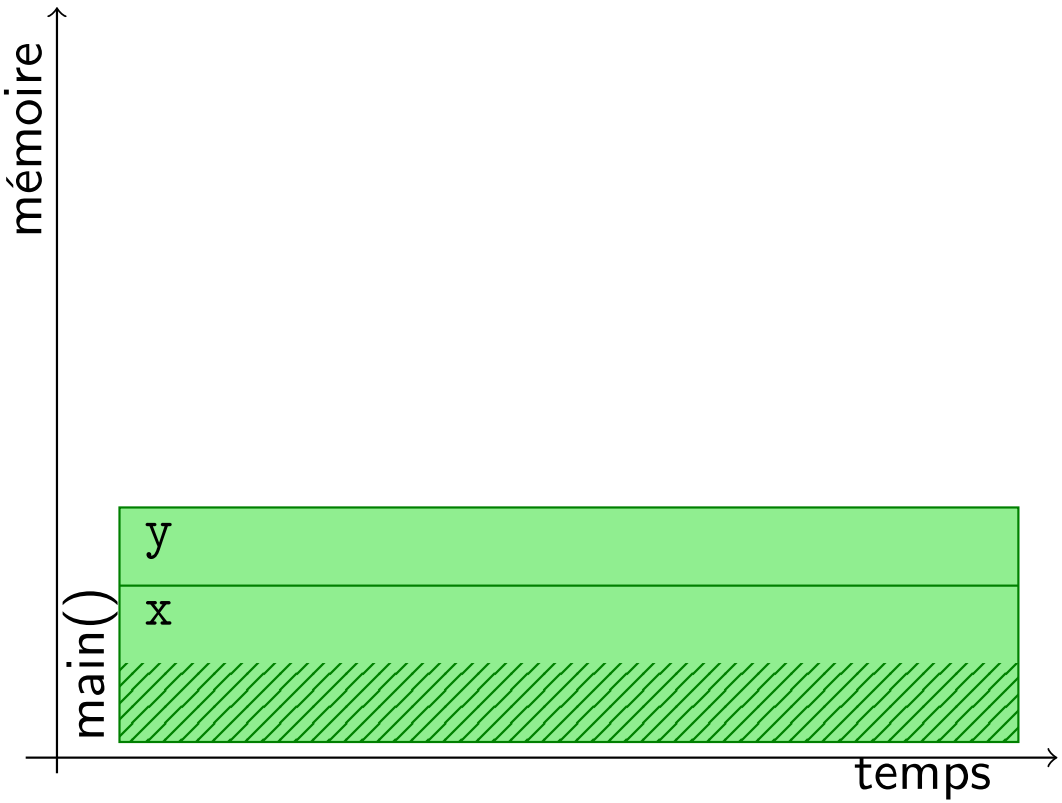
Pile d'appel

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



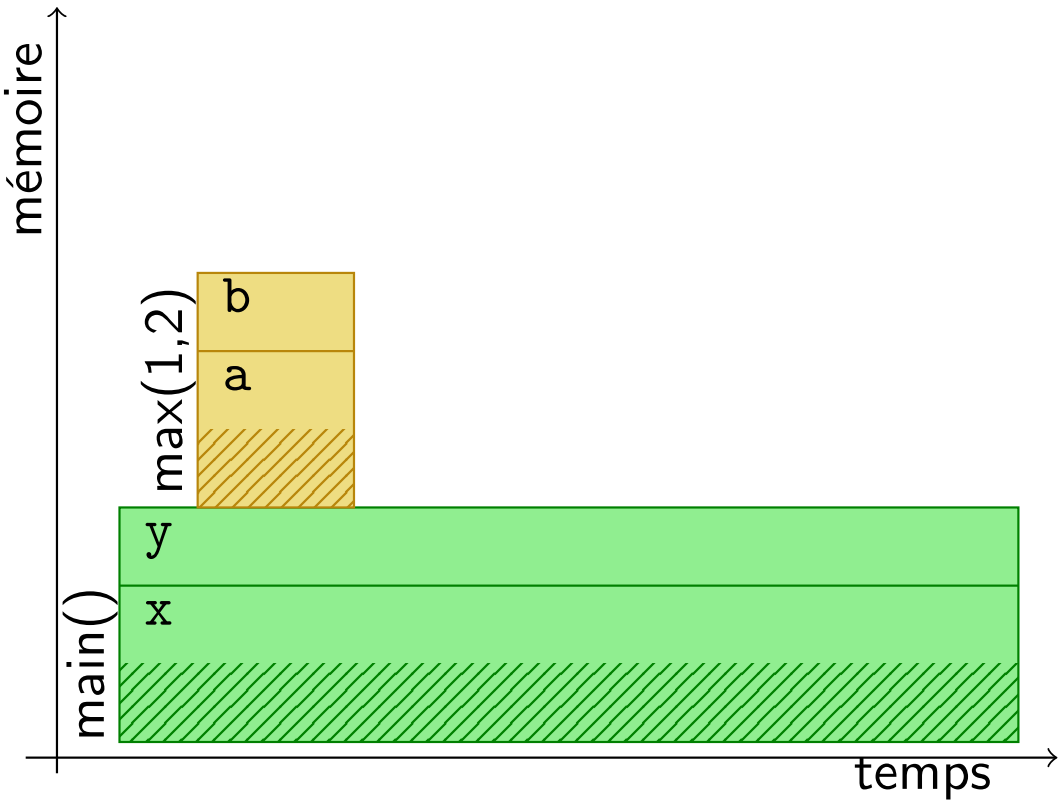
Pile d'appel

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



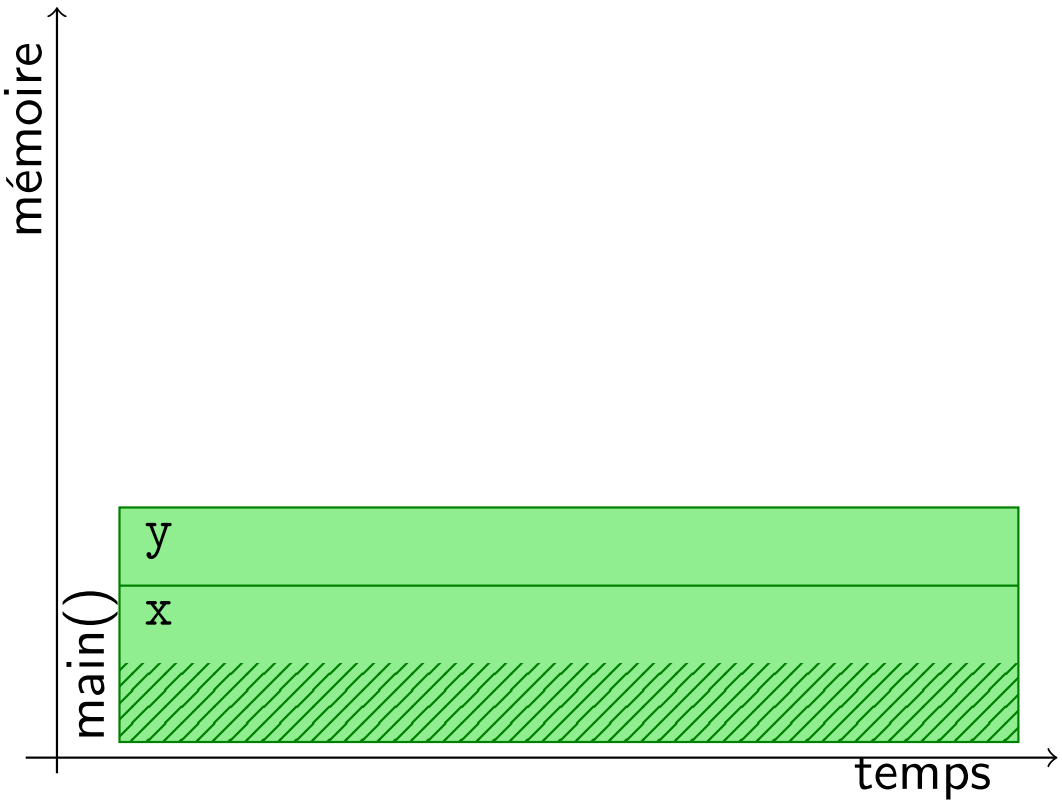
Pile d'appel

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



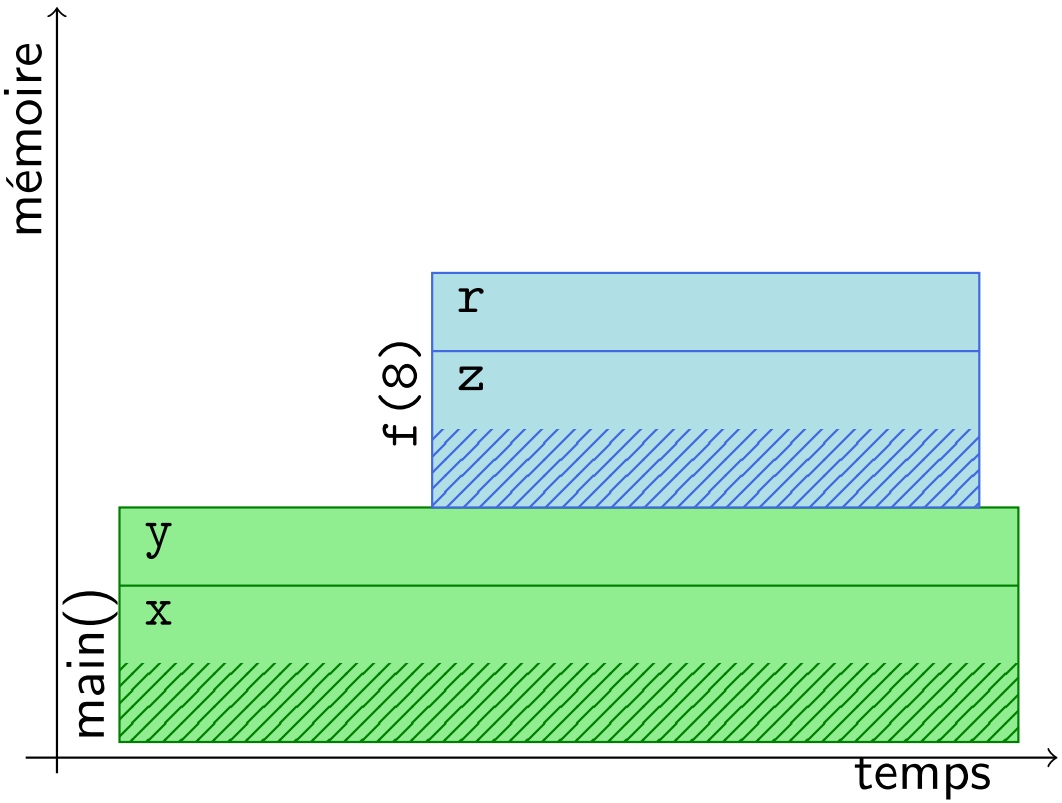
Pile d'appel

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



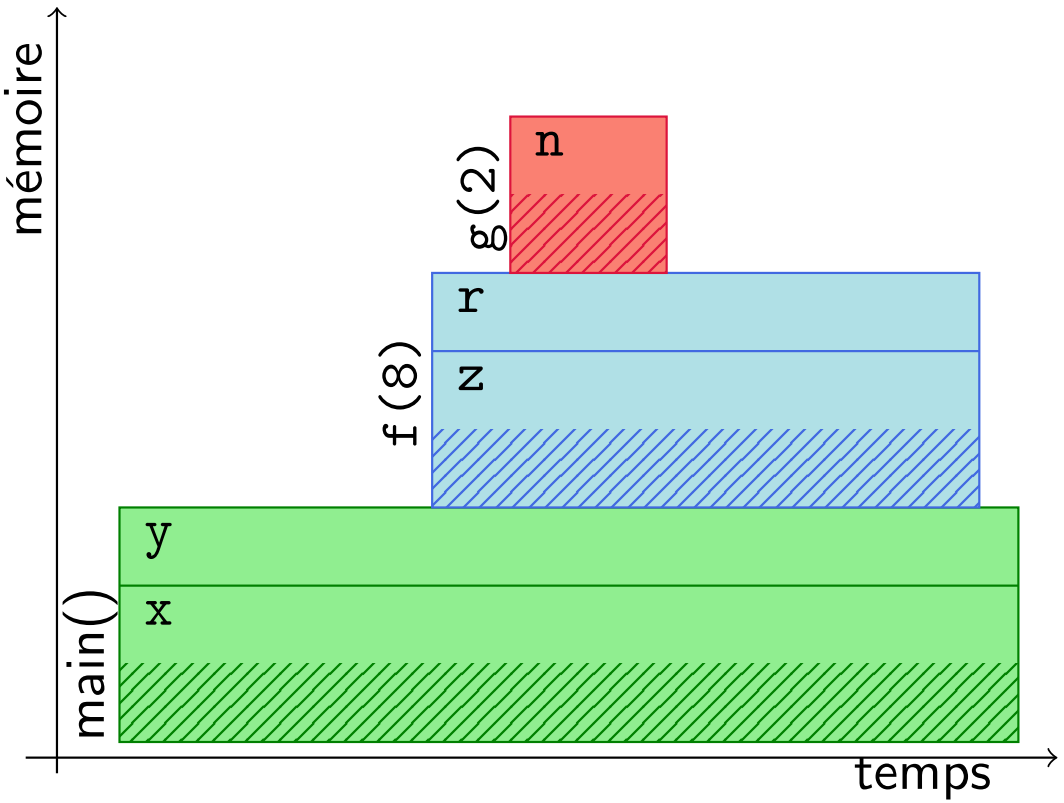
Pile d'appel

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



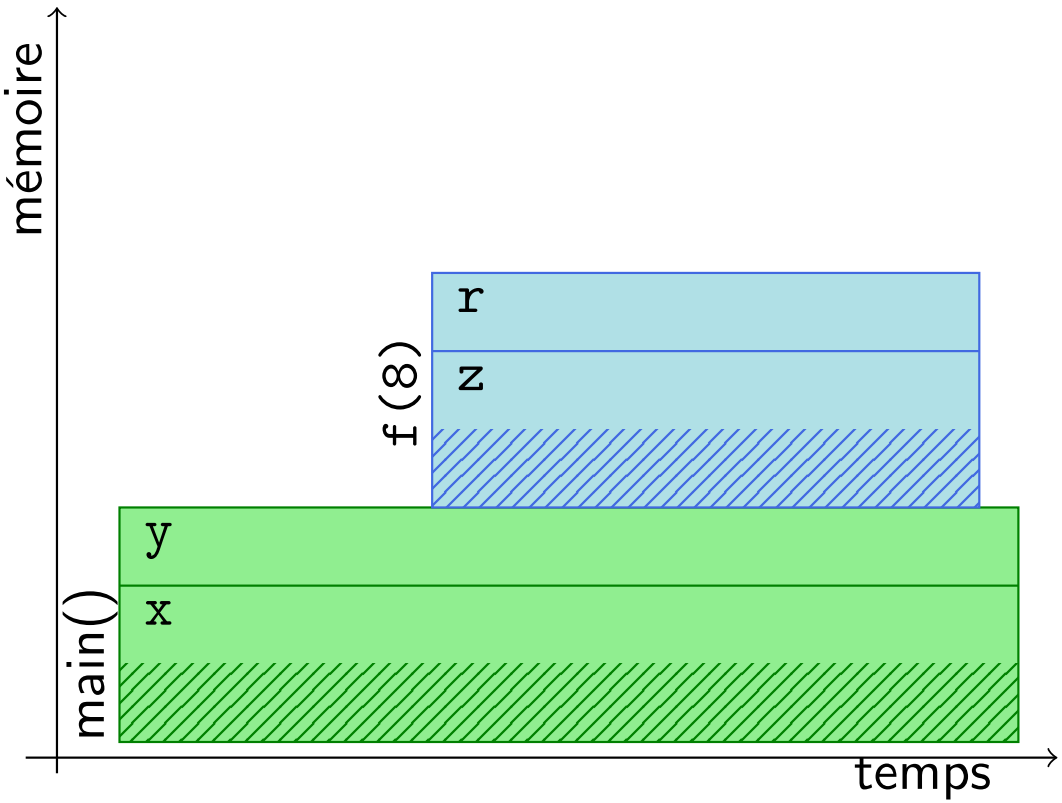
Pile d'appel

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



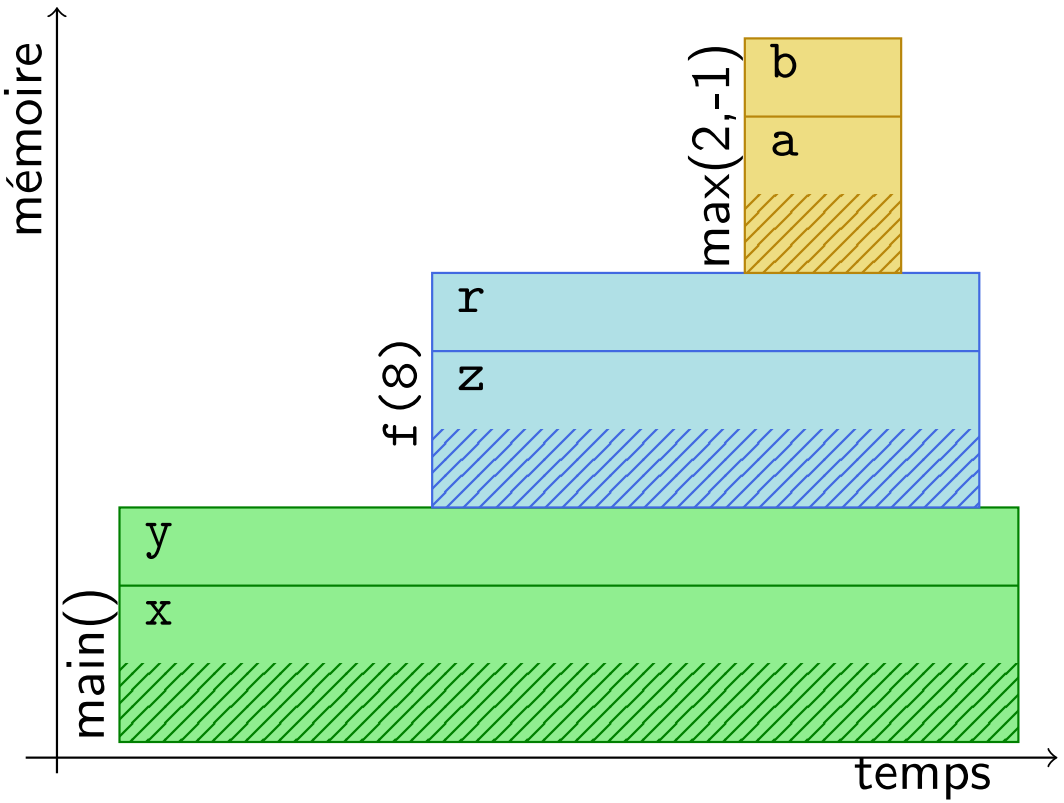
Pile d'appel

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



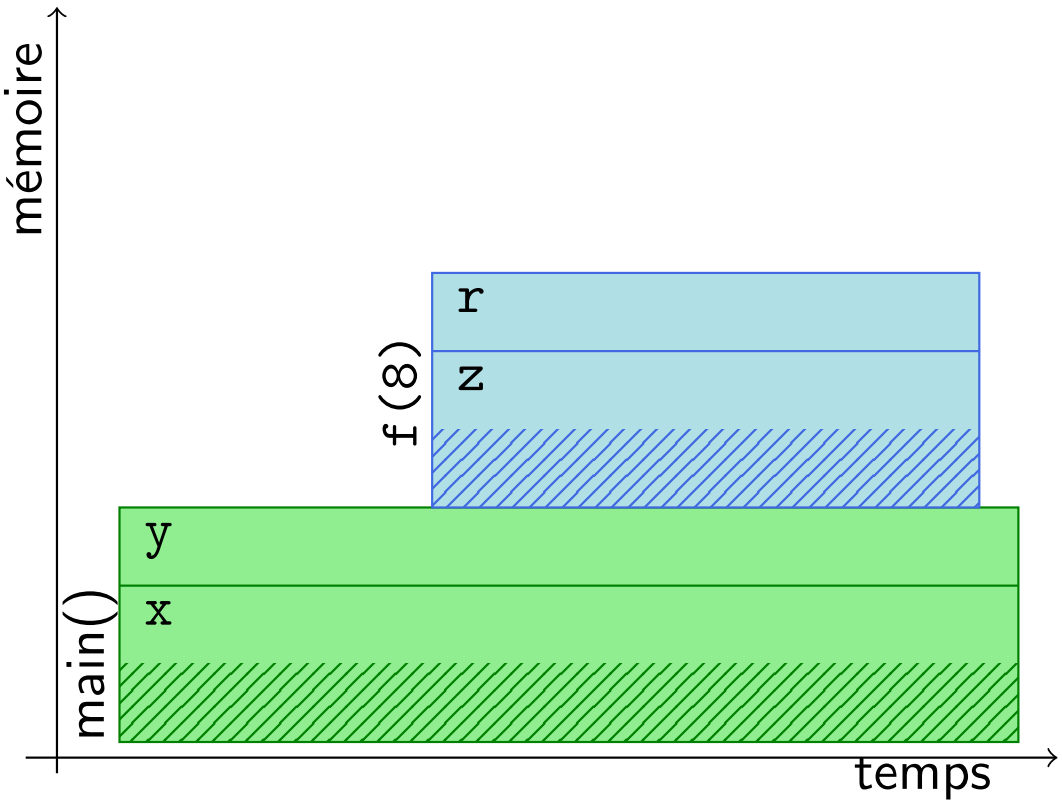
Pile d'appel

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



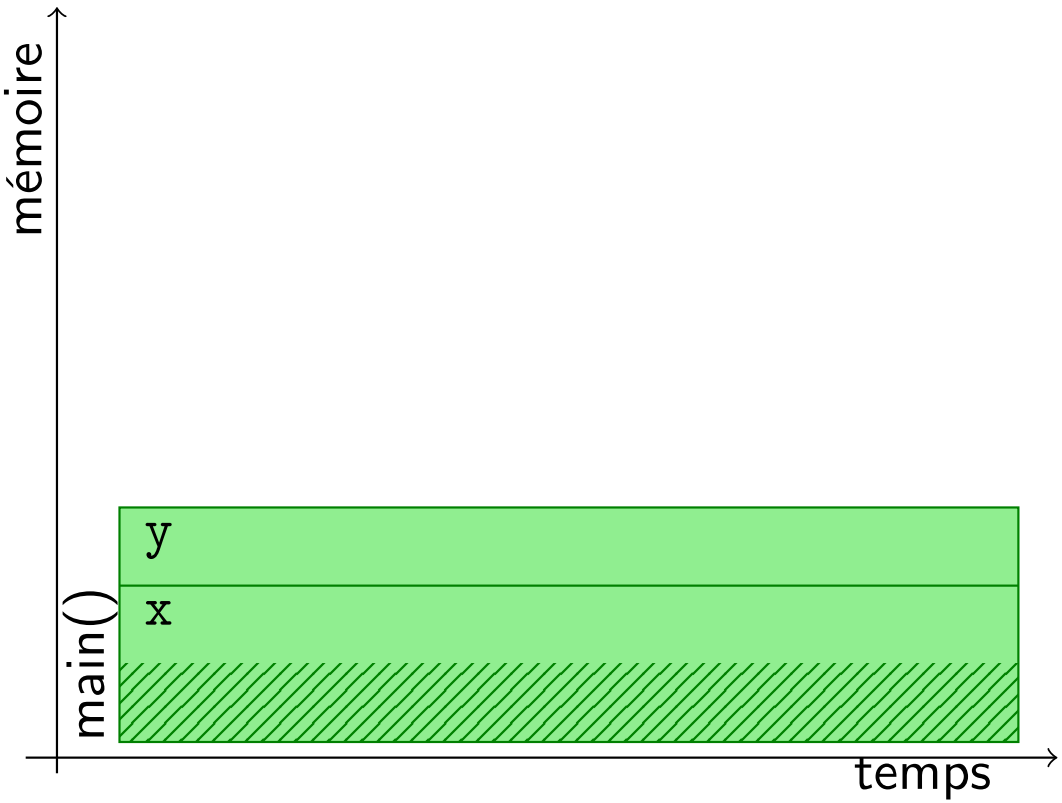
Pile d'appel

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



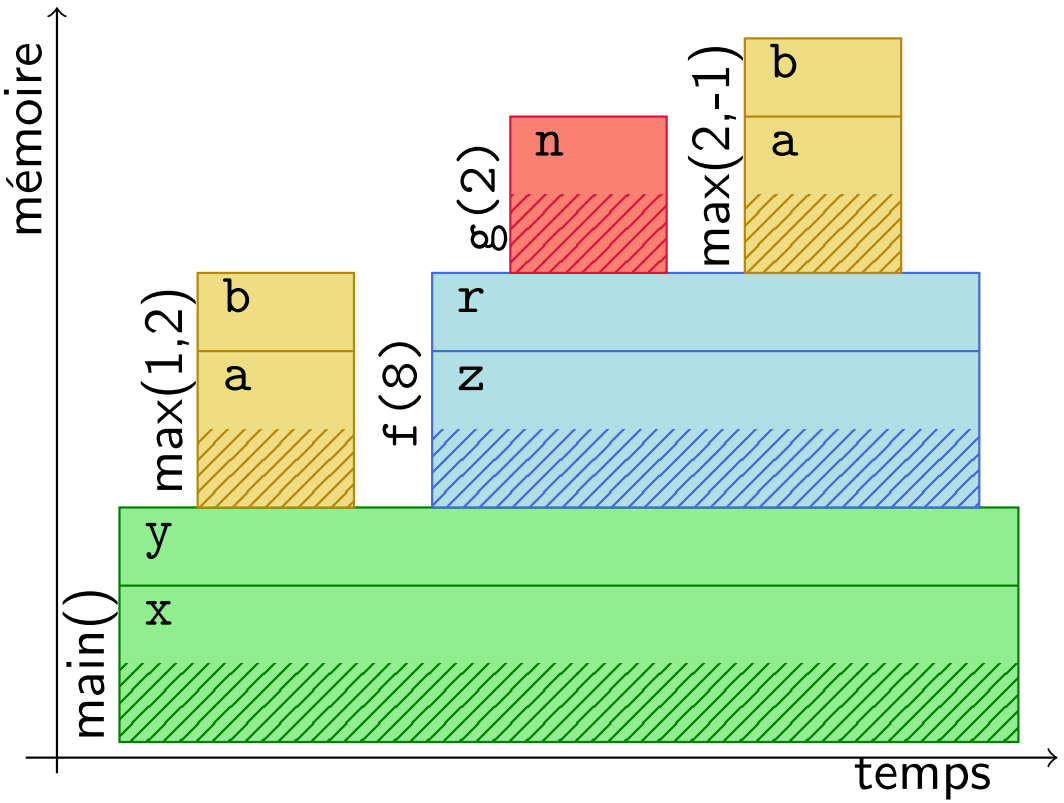
Pile d'appel

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



Pile d'appel

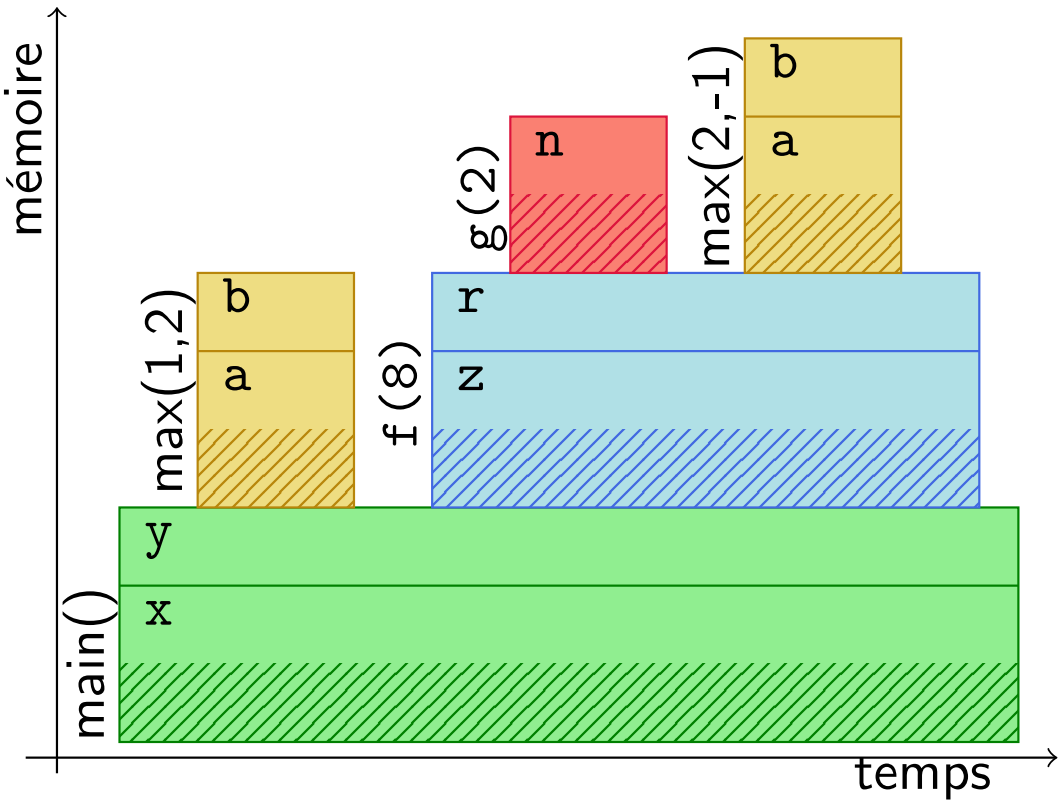
On parle de pile d'appel car les appels de fonctions s'empilent... *comme sur une pile d'assiettes.*



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 ? (La même fonction avec des contenus différents)



Factorielle récursive (teaser)

Définition

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

Factorielle récursive (teaser)

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 (teaser)

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);
}
```

Longue démo (menu)