Université Claude Bernard Lyon 1

LIF5 - Algorithmique et programmation procédurale

Carole Knibbe

Samir Akkouche

LIRIS

Chapitre 3
Vie et mort des variables en mémoire

Sommaire

1. Motivation

2. Comment les données d'un programme sont-elles organisées en mémoire ?

3. Qu'est-ce qu'un pointeur ?

4. Que se passe-t-il en mémoire lorsqu'on appelle une fonction ou une procédure ?

5. Qu'est-ce que l'allocation dynamique de mémoire ?

Niveau langage (évolué)	tables, arbres, graphes, fichiers, listes chaînées, files, files de priorité, tableaux dynamiques, piles, tableaux, structures booléens, caractères, entiers, réels, **pointeurs**
Niveau machine	mots ⇧ bytes (octets) ⇧ bits 

?

1. Motivation

2. Comment les données d’un programme sont-elles organisées en mémoire ?

Un aperçu du langage machine

1 instruction en langage C :

```
res = x + y;
```

Plusieurs instructions en langage machine :

« Copier, dans le registre AX du processeur, les bits situés à l'emplacement numéro 0x0110 des barrettes mémoire »

Langage machine (binaire)	Langage machine (hexadécimal)	Assembleur
1010 0001 0000 0001 0001 0000	0xA10110	MOV AX,[0110]
0000 0011 0000 0001 0001 0010	0x030112	ADD AX,[0112]
1010 0011 0000 0001 0001 0100	0xA30114	MOV [0114],AX

Dans une expression, écrire un nom de variable revient à écrire "contenu de l'emplacement mémoire n° ..."

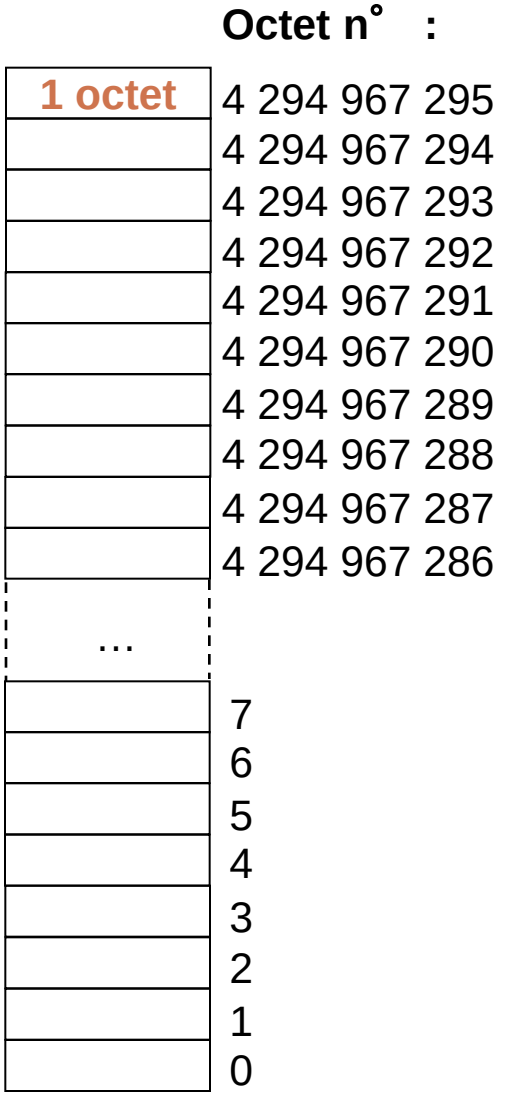
Notion de variable

- Chaque variable possède :
  - un nom
  - un type, et donc une taille en octets,
  - une adresse en mémoire, qui ne change pas au cours de l'exécution,
  - une valeur, qui, elle, peut changer au cours de l'exécution du programme

- Quand une nouvelle valeur est placée dans une variable (par une affectation, par **cin**, par **scanf…**), elle remplace et détruit la valeur précédente

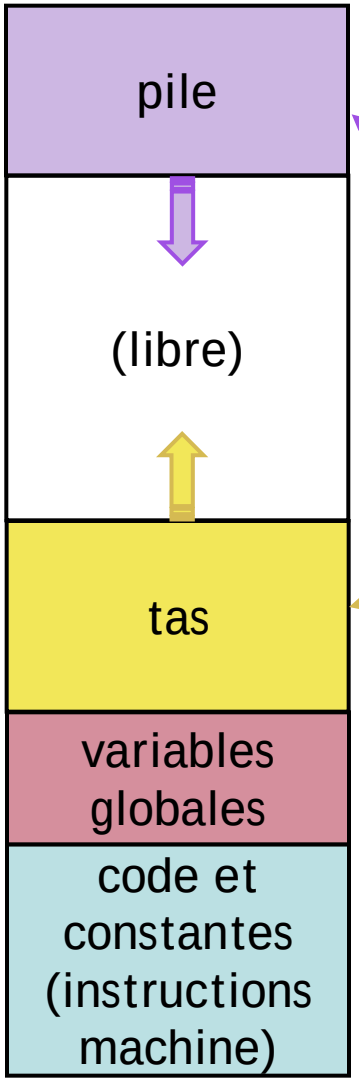
Notion d'adresse mémoire

- La mémoire est organisée en 1 dimension :
  1 seule « coordonnée » suffit pour repérer un emplacement
- Les adresses mémoire sont généralement codées sur 32 bits ⇨ $2^{32}-1 \approx 4$ milliards d'adresses possibles
- Plus petit élément adressable = l'octet ⇨ une adresse mémoire désigne un octet spécifique parmi les 4 milliards d'octets adressables
- Adresse mémoire d'une variable = adresse du 1er octet qu'elle occupe



L’espace d’adressage d’un programme sous Linux

L’espace d’adressage est segmenté :



adresses élevées
(ex. : 3 221 217 280)

Pour les fonctions et procédures en cours d’exécution :
- paramètres d’appel
- variables locales
- valeur de retour éventuelle

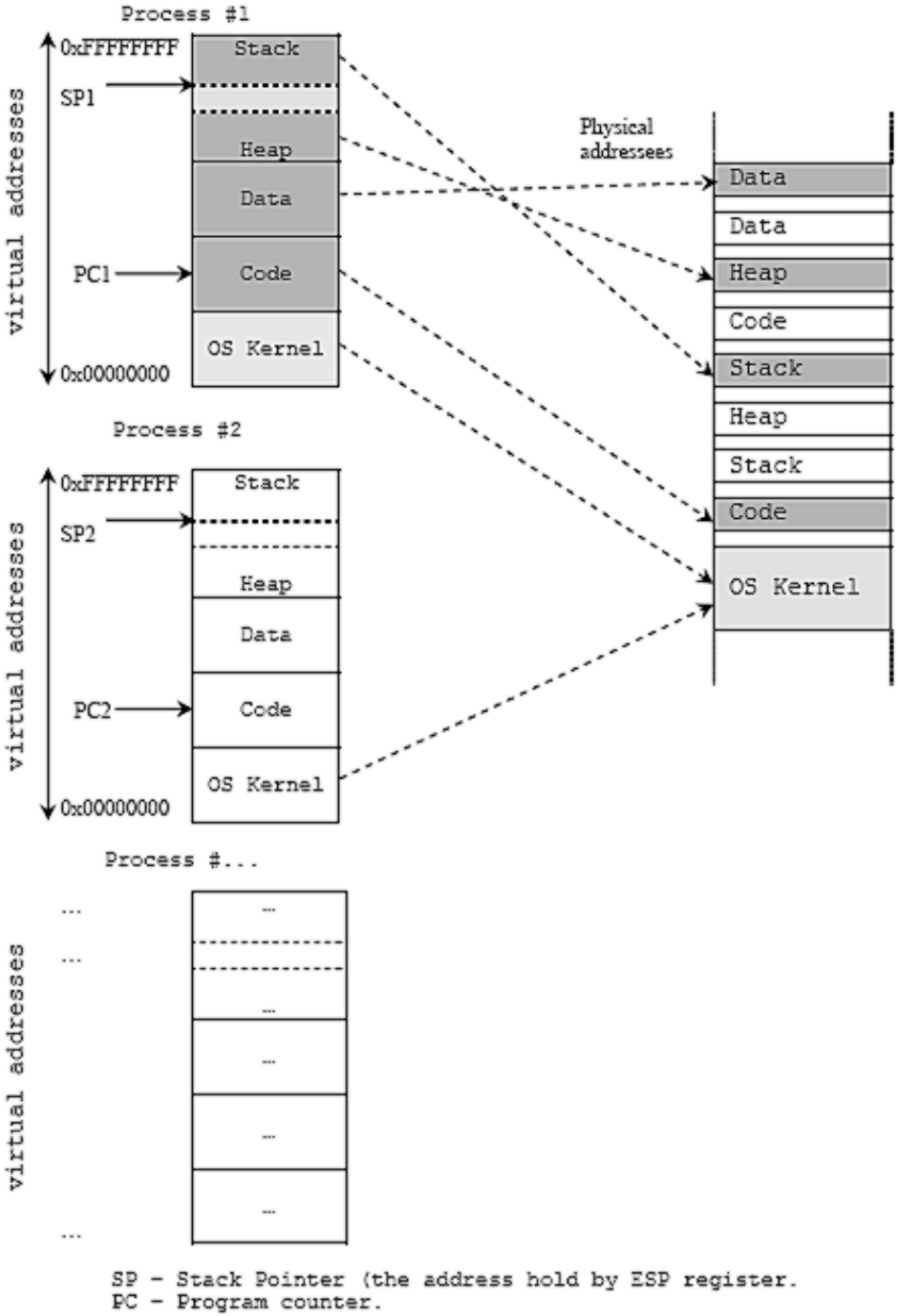
Données allouées dynamiquement, avec new ou malloc (nous y reviendrons…)

adresses basses
(ex. : 2 196)

L’espace d’adressage d’un programme

- Alloué par l’OS au lancement du processus
- 4 Go maximum, en général
- Permet de stocker :
  - les instructions machine du programme (code compilé)
  - les données manipulées par ces instructions (variables)

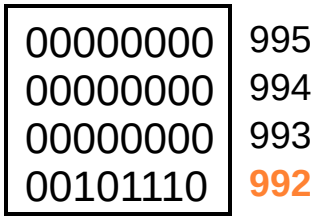
- Remarque : L’espace d’adressage est dit virtuel car il peut excéder la quantité de mémoire physique disponible (barrettes de mémoire RAM)
  - utilisation du disque dur comme espace mémoire annexe
  - chargement en RAM à la demande
  - un composant dédié (la MMU) traduit les adresses virtuelles en adresses physiques, et réclame les chargements en RAM si nécessaire
  - le processeur ne manipule, lui, que les adresses virtuelles


Process #1
0xFFFFFFFF
SP1
PC1
0x00000000
virtual addresses
Stack
Heap
Data
Code
OS Kernel
Physical addressees
Data
Data
Heap
Code
Stack
Heap
Stack
Code
OS Kernel
Process #2
0xFFFFFFFF
SP2
PC2
0x00000000
virtual addresses
Stack
Heap
Data
Code
OS Kernel
Process #...
virtual addresses
SP - Stack Pointer (the address hold by ESP register.
PC - Program counter.

Représentation visuelle d'une variable

Représentation détaillée :

monInt



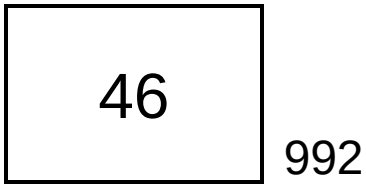
- La variable « monInt » occupe les emplacements mémoire (octets) numéros 992, 993, 994, 995. Son adresse est donc 992.
- Elle contient actuellement la valeur 46, car :

00000000 00000000 00000000 00101110 $_2$ = 46 $_{10}$

octet n° 995 | octet n° 994 | octet n° 993 | octet n° 992

Représentation abrégée :

monInt

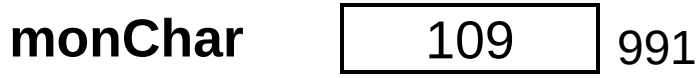


Représentation visuelle d'une variable

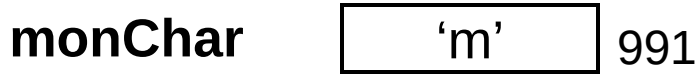
Représentation détaillée :

- La variable « monChar » occupe l'emplacement mémoire (octet) numéro 991. Son adresse est donc 991.
- Elle contient actuellement :
  - la valeur 109, car $01101101_2 = 109_{10}$
  - le caractère 'm', car 109 est le code ASCII du caractère 'm'

Représentation abrégée :



ou :



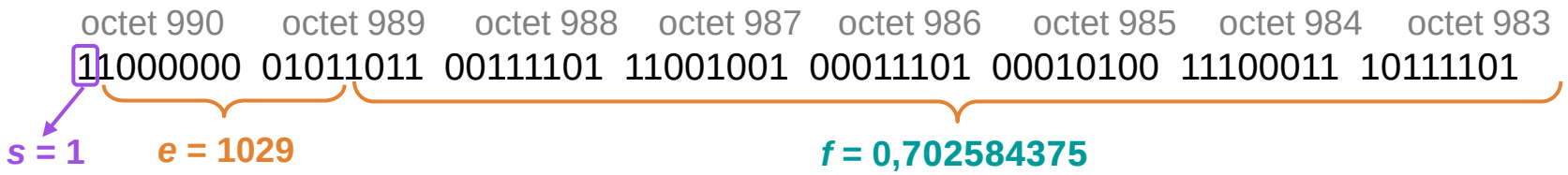
Représentation visuelle d'une variable

Représentation détaillée :

monDouble

11000000	990
01011011	989
00111101	988
11001001	987
00011101	986
00010100	985
11100011	984
10111101	**983**

- La variable « monDouble » occupe les emplacements mémoire (octets) 983 à 990. Son adresse est donc 983.
- Elle contient actuellement la valeur -108.9654 :



$(-1)^s . (1 + f) . 2^{(e-1023)} = - 108.9654$

Représentation abrégée :

monDouble

-108.9654

983

Visibilité (portée) des variables en C

- Variable globale :
  - déclarée **en dehors** de tout bloc d'instructions { }
  - utilisable dans n'importe quel bloc d'instructions
  - y compris dans un autre fichier source du même programme (en la redéclarant avec le mot-clé « extern »), sauf si elle avait été déclarée « static »
  - stockée dans le segment « Variables globales » de l'espace d'adressage
  - attention : risque de modifications non désirées ou non prévues

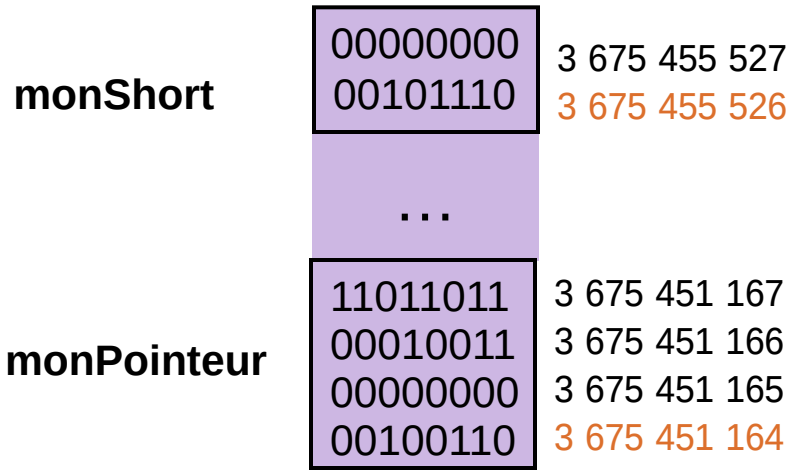
- Variable locale :
  - déclarée **à l'intérieur** d'un bloc d'instructions dans une fonction ou une boucle par exemple)
  - inutilisable dans un autre bloc
  - masque, le cas échéant, la variable globale du même nom
  - stockée dans le segment « Pile » de l'espace d'adressage

3. Qu’est-ce qu’un pointeur ?

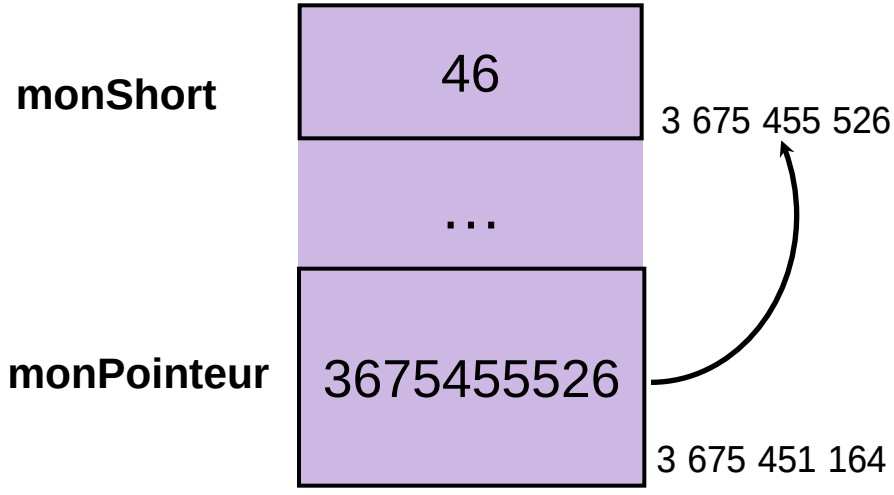
Les variables de type pointeur : définition

- **Variables destinées à contenir des adresses mémoire**
- Rappel : Les adresses mémoire sont généralement codées sur 4 octets, soit 32 bits ⇨ $2^{32} - 1 \approx 4$ milliards d'adresses possibles
- Exemple :

Représentation détaillée



Représentation abrégée



Les variables de type pointeur : norme algorithmique

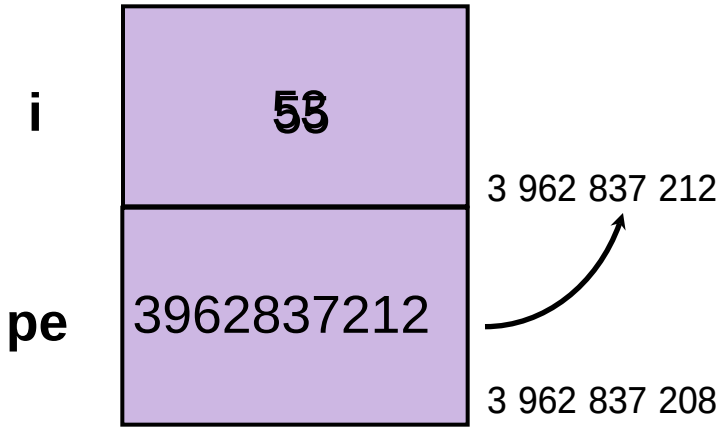
- Déclaration d'une variable pointeur :

  p : pointeur sur *type*

- Opérateur **&** : accès à l'adresse d'une variable

- Opérateur ↑ (déréférencement) : accès à la valeur qui se trouve à l'adresse pointée
  - précondition : adresse valide
  - le type du pointeur permet de savoir combien d'octets lire et comment les interpréter

```
Variables
        i : entier
        pe : pointeur sur entier
Début
        i ← 53
        pe ← &i
        pe↑ ← pe↑ + 2
        afficher(i)
Fin
```



Les variables de type pointeur : en C

- Déclaration : ***type* *p ;**
- Accès à l'adresse d'une variable : opérateur **&**
  - ne pas confondre avec le & du passage de paramètres par référence en C++
- Accès à la valeur se trouvant à l'adresse pointée :
  - opérateur *
  - opérateur [ ]
- Deux lectures possibles pour la déclaration
  - int *p; « *p (valeur pointée) est un int »
  - int* p; « p est un int* (pointeur sur int) »

```
int main()
{
   int i;
   int *pe;

   i = 53;
   pe = &i;
   *pe = *pe + 2;
   printf("%d \n", i);
   return 0;
}
```

Cas particulier des pointeurs non typés (void *)

void *p;

- p est une variable de type « adresse générique »
- Peut pointer sur n'importe quel type de donnée
- Utile par ex. pour traiter une suite d'octets à partir d'une adresse donnée
- Déréférencement interdit (*p interdit)
- Opérations arithmétiques interdites (cf chapitre suivant)
- Possibilité de cast en pointeur typé :
  - en C : float * pf = **(float *)** p;
  - en C++ : float * pf = **reinterpret_cast<double *>(**p**)**;

Question

unsigned short *x;

- Quelle est la valeur maximale que peut prendre x ?
- Quelle est la valeur maximale que peut pendre *x ?

4. Que se passe-t-il en mémoire lorsqu'on appelle une fonction ou une procédure ?

Appel d’un sous-programme : rappel

- Seule une instruction est exécutée à la fois

- Le sous-programme appelant doit attendre la fin de l’exécution du sous-programme appelé avant de continuer

Le cas d'une fonction

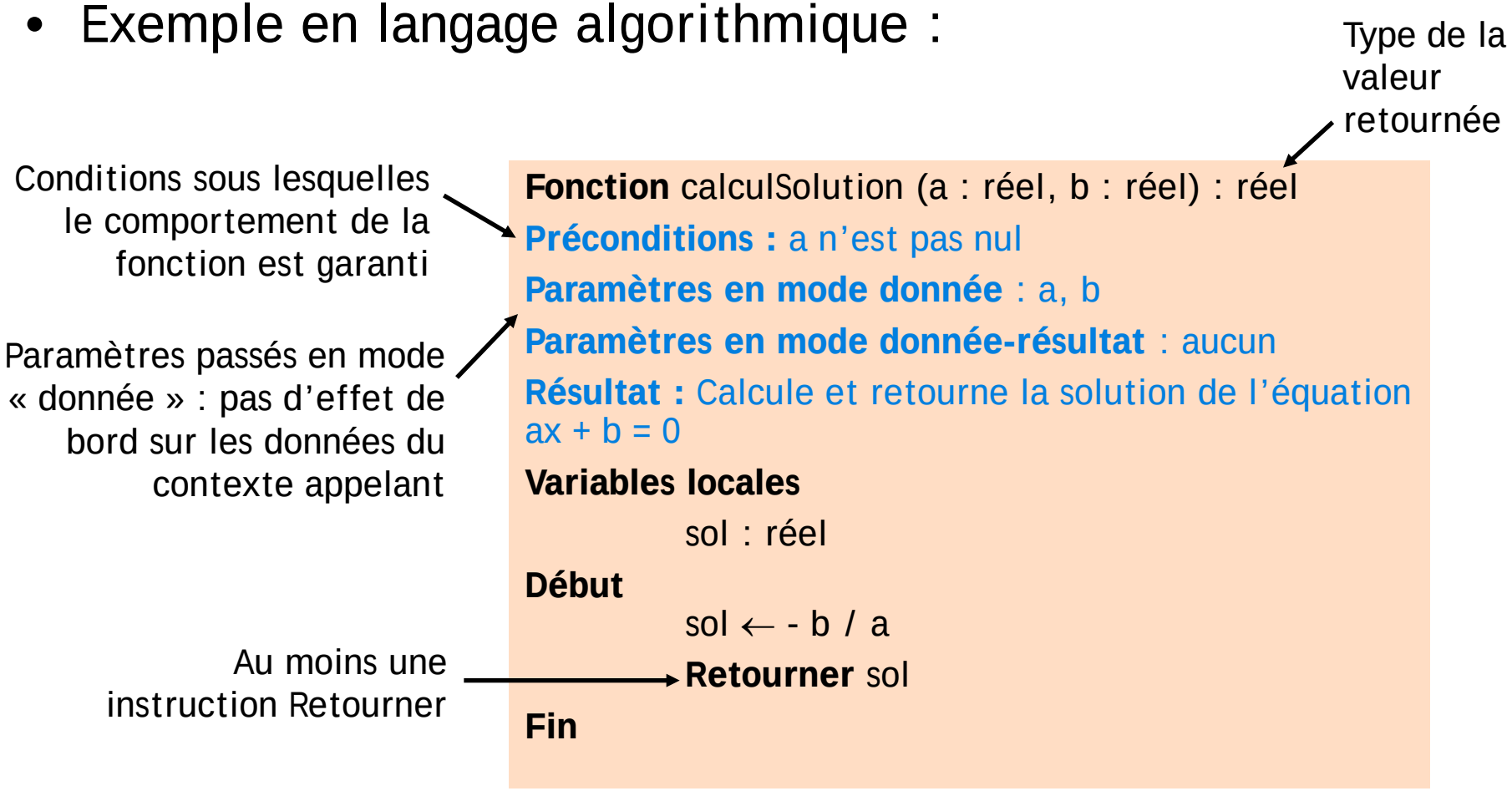
- Une fonction est une séquence d'actions qui produit une **valeur de retour** à partir de valeurs passées en paramètres

- Une fonction au sens algorithmique ne doit **pas avoir d'effet de bord** sur les données du sous-programme appelant

  ⇨ Les paramètres doivent être passés en mode « donnée »

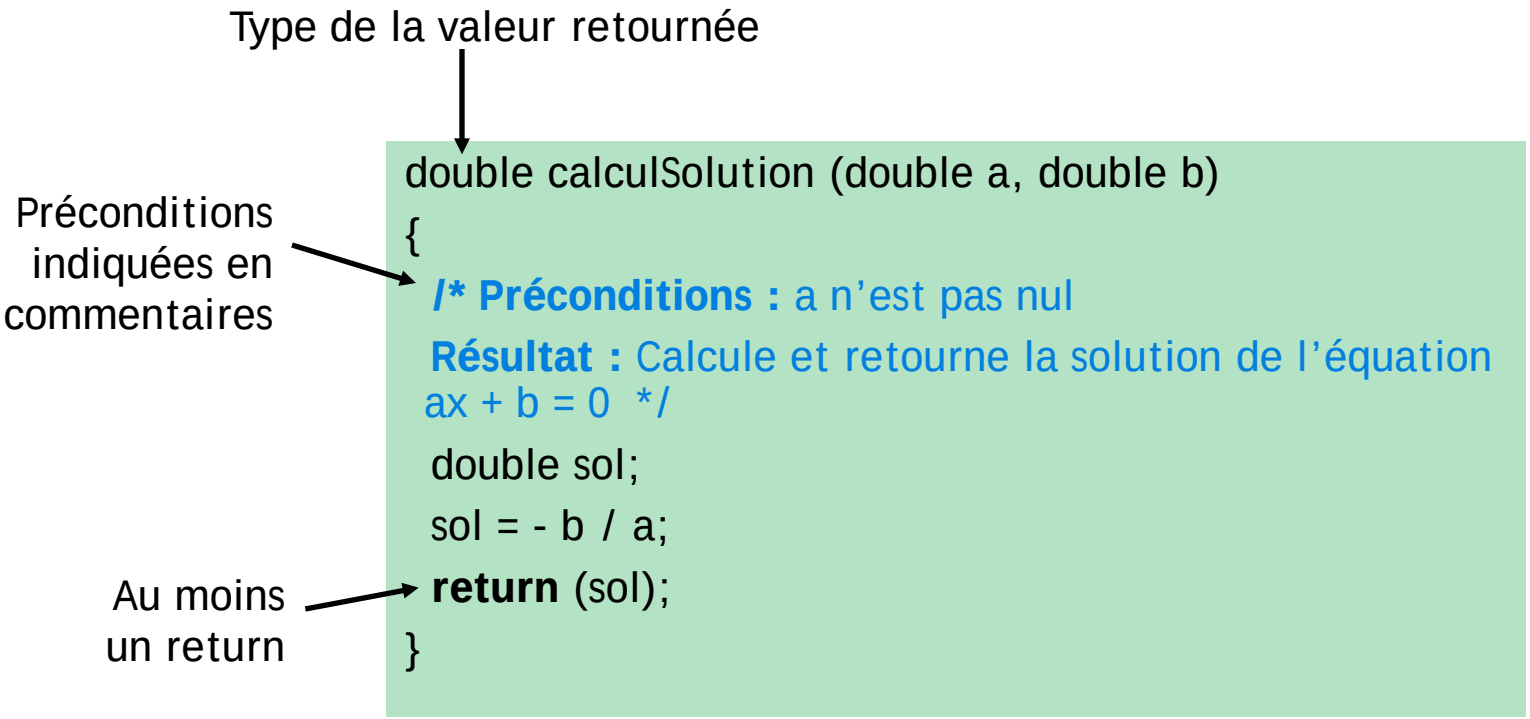
Le cas d'une fonction

- Exemple en langage algorithmique :



Le cas d'une fonction

- Le même exemple en langage C :



Le cas d'une fonction

```
#include <stdio.h>   /* pour printf */

double calculSolution (double a, double b)
{
  /* Préconditions : a n'est pas nul
  Résultat : Calcule et retourne la solution de
  l'équation ax + b = 0  */
  double sol;
  sol = - b / a;
  return (sol);
}

int main()
{
  double coef1 = 0.5;
  double coef2 = 128.2;
  double lasolution;
  lasolution = calculSolution(coef1, coef2);
  printf("La solution est %f \n", lasolution);
  return 0;
}
```

a et b sont les paramètres formels de calculSolution

Le programme principal est une fonction particulière qui renvoie généralement 0 quand tout s'est bien passé

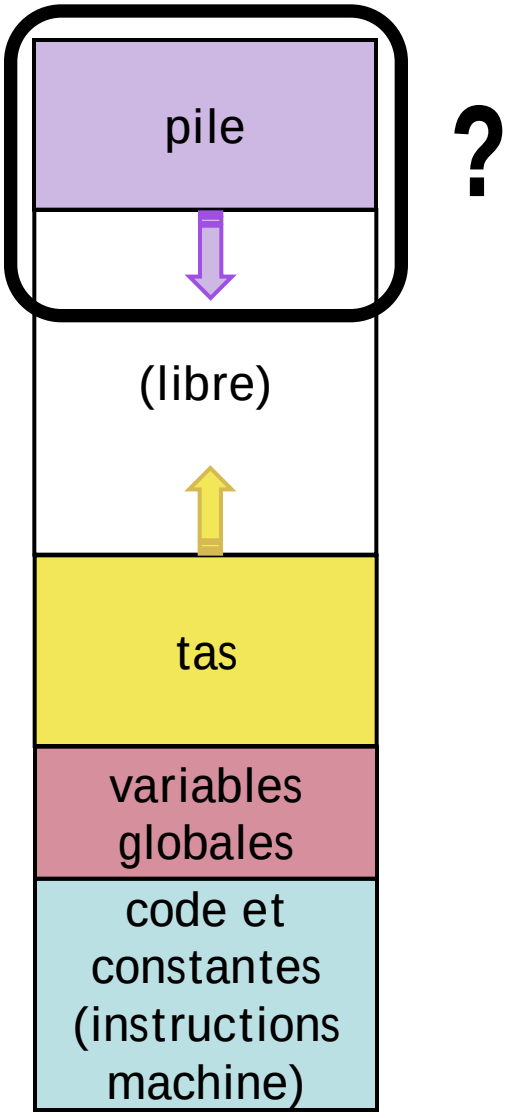
coef1 et coef2 sont les paramètres effectifs de calculSolution

Le fonctionnement de la pile : exemple avec une fonction

```
#include <stdio.h>   /* pour printf */

double calculSolution (double a, double b)
{
  /* Préconditions : a n'est pas nul
  Résultat : Calcule et retourne la solution de
  l'équation ax + b = 0  */
  double sol;
  sol = - b / a;
  return (sol);
}

int main()
{
  double coef1 = 0.5;
  double coef2 = 128.2;
  double lasolution;
  lasolution = calculSolution(coef1, coef2);
  printf("La solution est %f \n", lasolution);
  return 0;
}
```



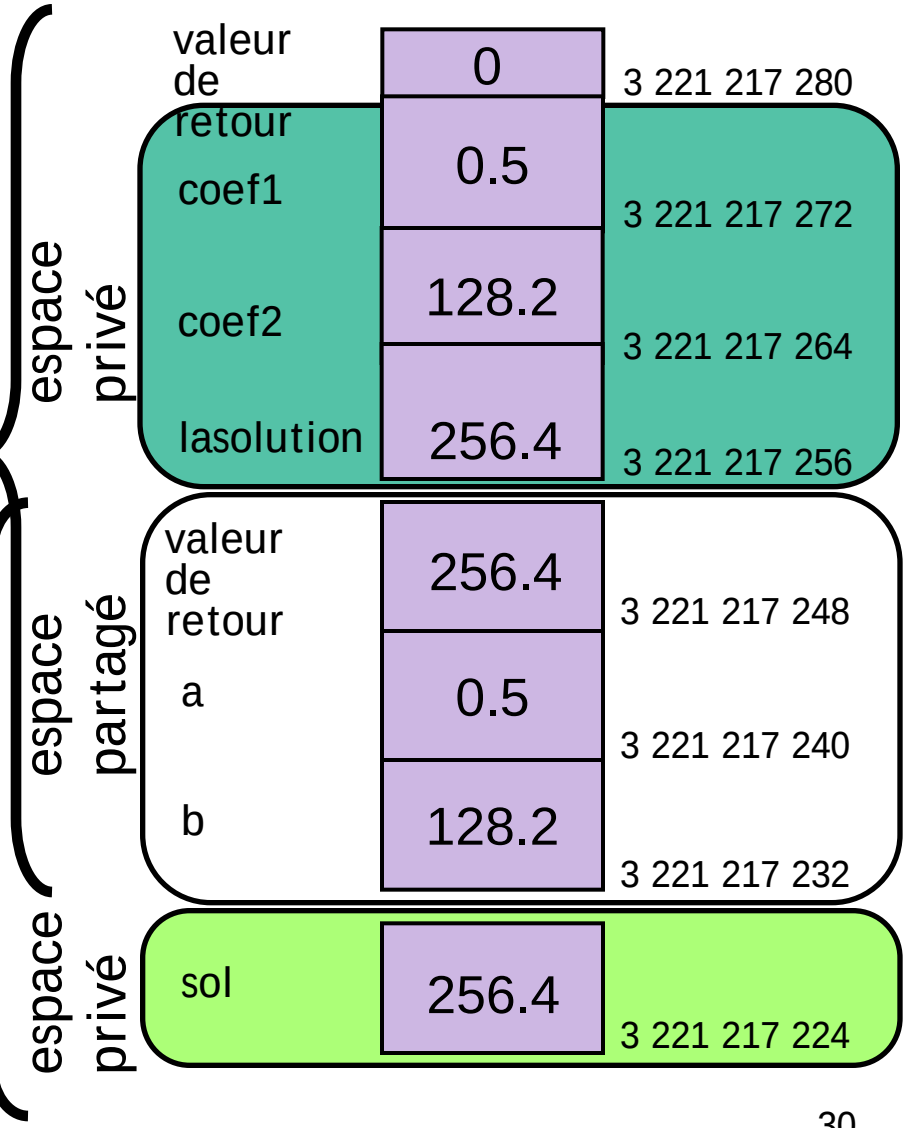
Le fonctionnement de la pile : exemple avec une fonction

```
double calculSolution (double a, double b)
{
  /* Préconditions : a n'est pas nul
  Résultat : Calcule et retourne la solution
  de l'équation ax + b = 0  */
double sol;
sol = - b / a;
return (sol);
}
```

Frame de main

```
int main()
{
double coef1 = 0.5;
double coef2 = 128.2;
double lasolution;
lasolution = calculSolution(coef1, coef2);
printf("La solution est %f \n", lasolution);
return 0;
}
```

Frame de calculSolution



Bilan : vie et mort des variables lors de l'appel d'une fonction

- Les valeurs de retour, les paramètres et les variables locales des fonctions sont stockés dans le segment « Pile » de l'espace d'adressage du programme

- Les données sont contiguës dans la pile (pas de trous)

- La taille de la pile augmente et diminue au fur et à mesure de l'exécution du programme

- La pile est organisée en « frames »

Bilan : vie et mort des variables lors de l'appel d'une fonction (modèle simplifié)

- Appel de fonction = création d'une nouvelle frame

- Au moment de l'appel, des emplacements mémoire sont créés pour :
  - la valeur de retour qui va être calculée
  - les paramètres passés par valeur (sans &) : recopie des données
  - les éventuelles variables locales

- Les instructions d'une fonction ne peuvent manipuler que les données de sa frame

Bilan : vie et mort des variables lors de l'appel d'une fonction (modèle simplifié)

- Lors du « return » :
  - l'expression qui suit le mot-clé return est évaluée et placée dans l'emplacement « valeur de retour » du contexte courant
  - les variables locales et les paramètres du contexte courant sont supprimés de la pile
- Lors de l'affectation (instruction du contexte appelant) :
  - la valeur de retour est affectée à la variable indiquée dans le code
  - la valeur de retour est supprimée de la pile
  - on revient au contexte appelant

Suppression du contexte de la fonction appelée, en 2 étapes

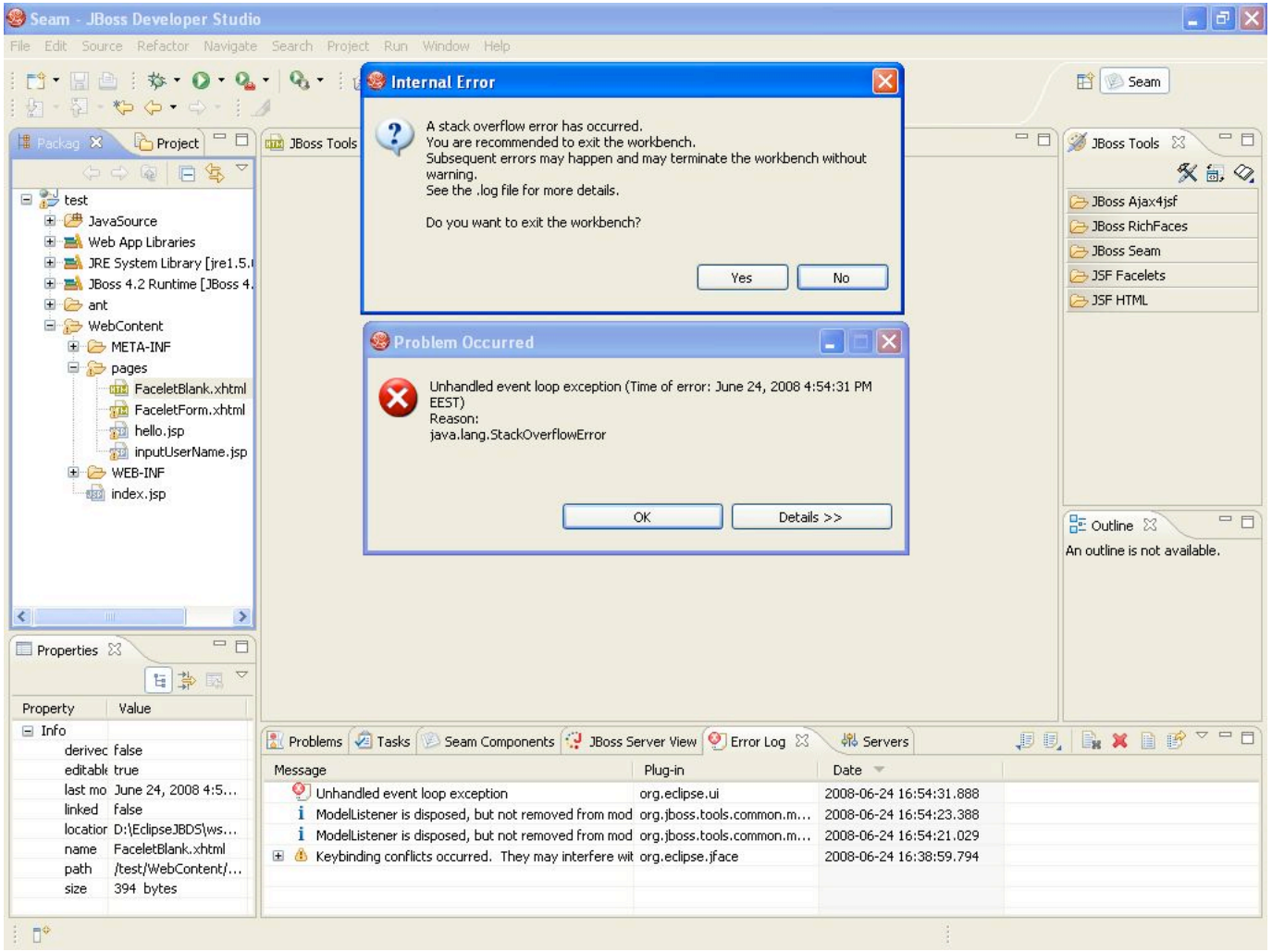
Question

Si le main

- appelle f1 qui appelle f2 qui s'appelle récursivement 3 fois,
- puis appelle g,

quelle est l'évolution de la pile ?


Microsoft Internet Explorer
Stack overflow at line: 0
OK


Seam - JBoss Developer Studio
File Edit Source Refactor Navigate Search Project Run Window Help
Seam
Internal Error
A stack overflow error has occurred.
You are recommended to exit the workbench.
Subsequent errors may happen and may terminate the workbench without warning.
See the .log file for more details.
Do you want to exit the workbench?
Yes
No
Problem Occurred
Unhandled event loop exception (Time of error: June 24, 2008 4:54:31 PM EEST)
Reason:
java.lang.StackOverflowError
OK
Details >>
Packag
Project
test
JavaSource
Web App Libraries
JRE System Library [jre1.5.
JBoss 4.2 Runtime [JBoss 4.
ant
WebContent
META-INF
pages
FaceletBlank.xhtml
FaceletForm.xhtml
hello.jsp
inputUserName.jsp
WEB-INF
index.jsp
JBoss Tools
JBoss Ajax4jsf
JBoss RichFaces
JBoss Seam
JSF Facelets
JSF HTML
Outline
An outline is not available.
Properties
Property
Value
Info
derivec false
editable true
last mo June 24, 2008 4:5...
linked false
locatior D:\EclipseJBDS\ws...
name FaceletBlank.xhtml
path /test/WebContent/...
size 394 bytes
Problems
Tasks
Seam Components
JBoss Server View
Error Log
Servers
Message
Plug-in
Date
Unhandled event loop exception
org.eclipse.ui
2008-06-24 16:54:31.888
ModelListener is disposed, but not removed from mod
org.jboss.tools.common.m...
2008-06-24 16:54:23.388
ModelListener is disposed, but not removed from mod
org.jboss.tools.common.m...
2008-06-24 16:54:21.029
Keybinding conflicts occurred. They may interfere wit
org.eclipse.jface
2008-06-24 16:38:59.794

Le cas d'une procédure

- Rappel : une procédure est un sous-programme qui ne renvoie pas de résultat, mais qui peut avoir des effets de bord sur les données du sous-programme appelant

- Trois modes de passage possibles pour les paramètres :
  - donnée
  - donnée-résultat
  - résultat

- Effets de bord possibles sur les paramètres passés en mode donnée-résultat ou résultat

Le cas d'une procédure

- Exemple en langage algorithmique :

Conditions sous lesquelles le comportement de la procédure est garanti

Tous les modes de passage sont possibles

Effets de la procédure sur les données

PAS d'instruction « retourner »

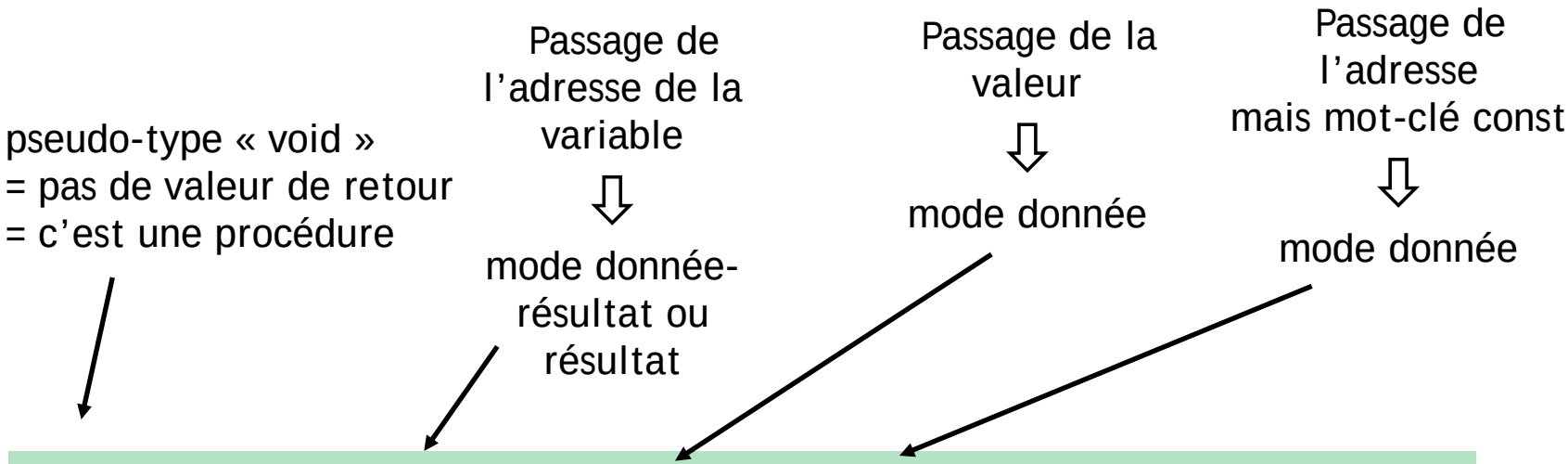
Procédure exemple (a : réel, x : réel, z : réel)
Préconditions : z n'est pas nul
Paramètres en mode donnée : x, z
Paramètres en mode donnée-résultat : a
Post-conditions : a^+ vaut a^-*x/z
Variables locales
aucune
Début
a ← a * x / z
Fin

Question

En C, comment faire pour qu'une procédure puisse avoir un effet de bord sur des données d'une autre « frame » que la sienne ?

Le cas d'une procédure

- La même en langage C :

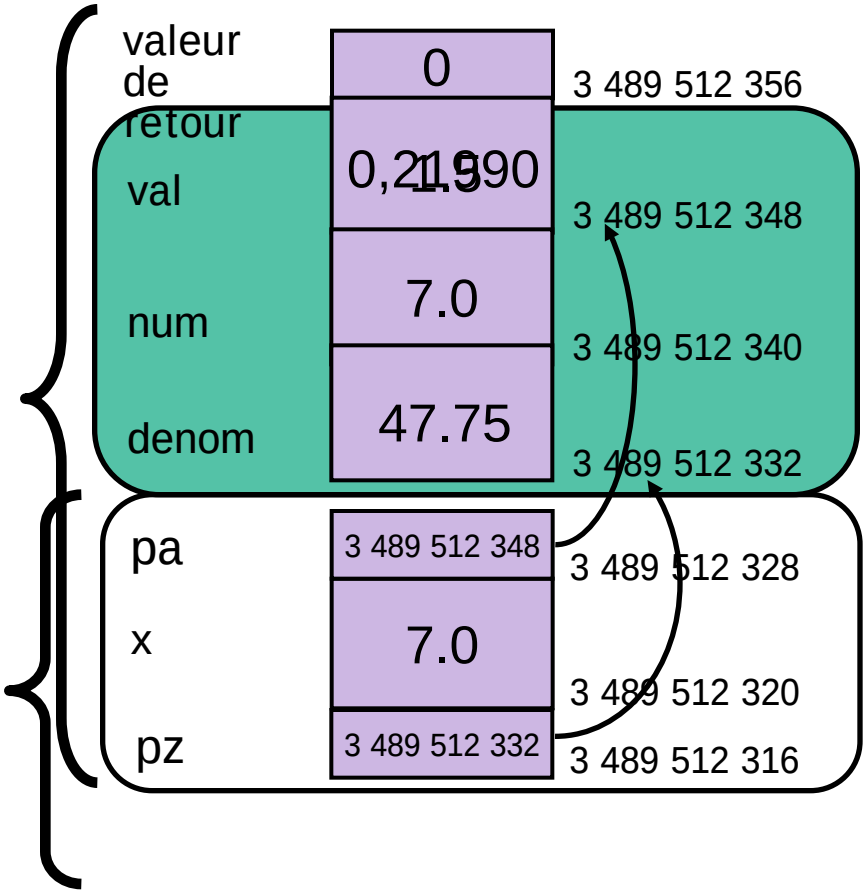



```
void exemple (double * pa, double x, const double * pz)
{
  /* Préconditions : la valeur stockée à l'adresse pz n'est pas nulle
     Post-conditions : à l'adresse pa, la nouvelle valeur stockée après exécution de la procédure vaut : ancienne valeur * x / valeur stockée à l'adresse pz   */
  *pa = (*pa) * x / (*pz) ;
}
```

Le cas d'une procédure

```
void exemple (double * pa, double x, \
const double * pz)
{
  /* Préconditions : z n'est pas nul
     Post-conditions : a+ vaut a-*x/z  */
  *pa = (*pa) * x / (*pz) ;
}
```

```
int main()
{
  double val = 1.5;
  double num = 7.0;
  double denom = num + 40.75;
  exemple (&val, num, &denom);
  printf("Nouvelle valeur de val = %f \n", val);
  return 0;
}
```

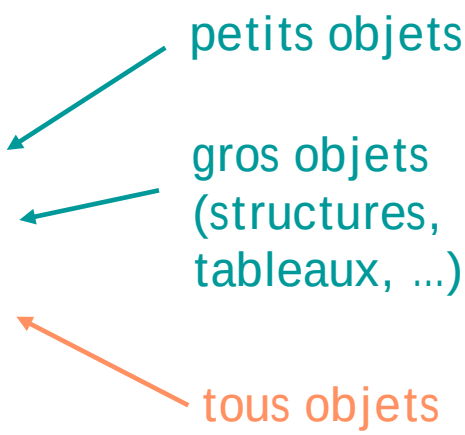


Passage de paramètres par adresse : à retenir

- Permet à une procédure d'accéder aux données (normalement privées) de la frame appelante
- Entorse au principe de séparation des données
- Permet à la procédure d'avoir des effets de bord
- Utilisation des pointeurs

Passage de paramètres : résumé

Norme algorithmique	Mise en œuvre C
Mode donnée	*type* a
	const *type* *a
Mode donnée-résultat ou résultat	*type* *a

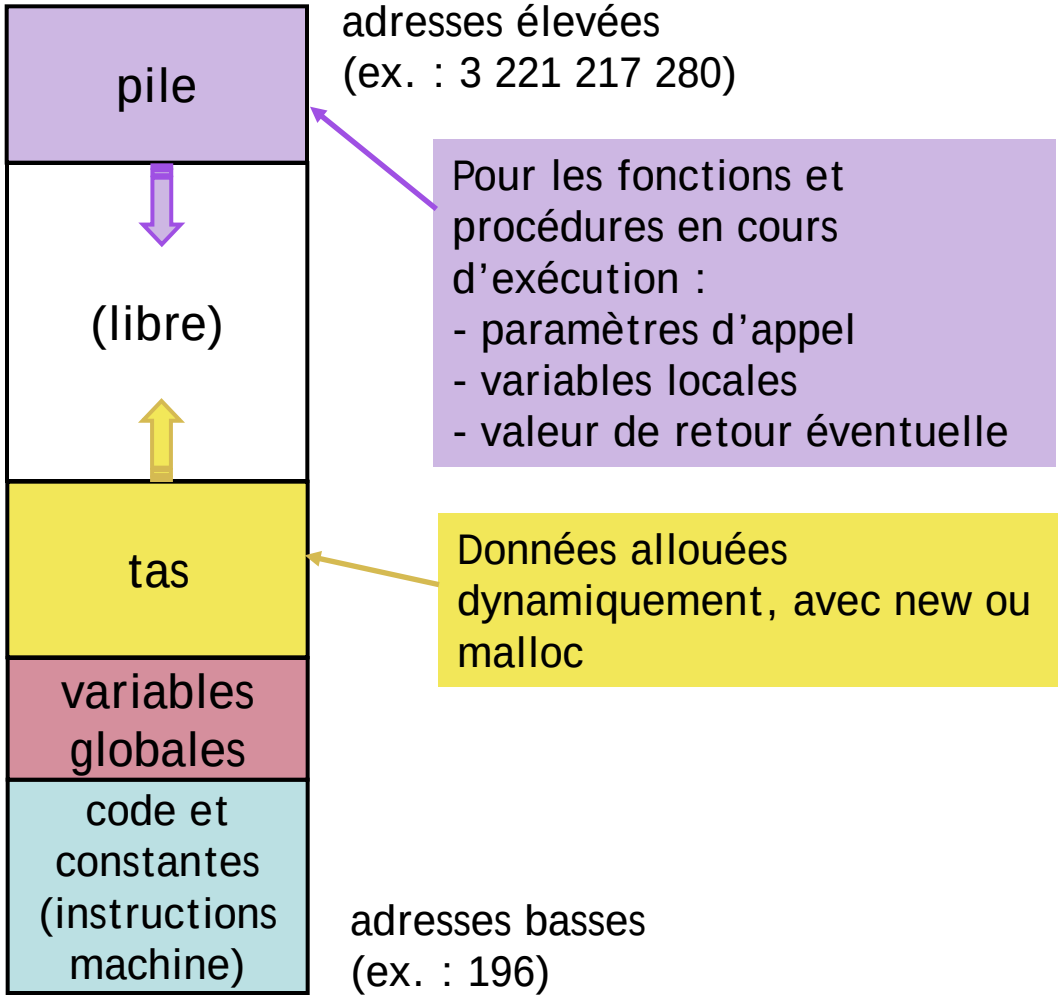


- Mot-clé « const » = la fonction ou la procédure ne modifie pas l'objet original, même si elle dispose de son adresse
  - dans la fonction ou la procédure, les tentatives de modification de l'objet sont en principe rejetées lors de la compilation

5. Autre utilité des pointeurs : l’allocation dynamique de mémoire

L’allocation dynamique de mémoire

- Réservation d’emplacements mémoire dans le tas
- La taille des données n’a pas besoin d’être connue à la compilation
- Les données mises dans le tas ne sont pas détruites en sortie de fonction ou de procédure



Allocation dynamique de mémoire : norme algorithmique

- Réservation : p ← **réserve** *type*
  ou : p ← **réserve** tableau [1..*n*] de *type*

- **réserve** fait deux choses :
  - alloue un emplacement mémoire de la taille nécessaire dans le tas
  - renvoie l'adresse mémoire de l'élément dans le tas (si tableau, renvoie l'adresse du premier élément du tableau)

- Libération : **libère** p

- **libère** rend l'emplacement mémoire disponible pour d'autres utilisations éventuelles (les données sont perdues)

- Allouer et oublier de libérer = « fuite de mémoire » (très mal !) (pas de ramasse-miettes en C)

Allocation dynamique de mémoire : mise en œuvre en C

Norme algorithmique	Langage C
p : pointeur sur réel	double *p;
p ← réserve réel ... libère p	p = (double *) **malloc**(sizeof(double)); ... **free**(p);

p ← réserve tableau[1..4] de réels
...
libère p
	p = (double *) malloc(**4***sizeof(double));
	...
	free(p);

malloc et free sont fournis par la librairie stdlib :

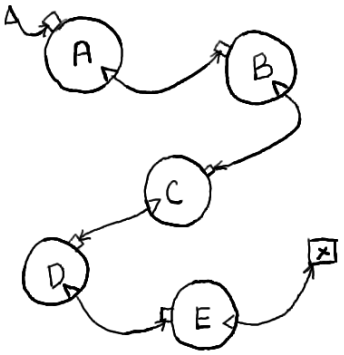
#include <stdlib.h> nécessaire

Utilité des pointeurs : bilan

- Passage de paramètres en mode donnée-résultat ou résultat
  - C++ : références = pointeurs masqués
  - C : pointeurs explicites

- Allocation dynamique dans le tas
  - gestion de structures dynamiques (taille variable)
  - possibilité d'allouer de la mémoire qui survivra au bloc dans lequel est réalisé son allocation

- Gestion des tableaux en C : cf. chapitre suivant



Linked List