Institut
Mines-Télécom

Génération de code et conventions d'appels

ou comment transformer un IR en assembleur

Alexis Polti

Licence de droits d'usage

Contexte académique } sans modification

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après, et à l'exclusion de toute utilisation commerciale.

Le droit d'usage défini par la licence autorise un usage dans un cadre académique, par un utilisateur donnant des cours dans un établissement d'enseignement secondaire ou supérieur et à l'exclusion expresse des formations commerciales et notamment de formation continue. Ce droit comprend :

- le droit de reproduire tout ou partie du document sur support informatique ou papier,
- le droit de diffuser tout ou partie du document à destination des élèves ou étudiants.

Aucune modification du document dans son contenu, sa forme ou sa présentation n'est autorisé.

Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité.

Le droit d'usage défini par la licence est personnel, non exclusif et non transmissible.

Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur :

alexis.polti@telecom-paristech.fr

Avant tout

- **Si vous souhaitez installer votre chaîne de cross-compilation**
  - Linaro / GCC ARM Embedded :
    - https://launchpad.net/gcc-arm-embedded/
    - Ubuntu : https://launchpad.net/~terry.guo/+archive/ubuntu/gcc-arm-embedded
      - bien lire la description du package, surtout le dernier paragraphe
    - autres distributions :
      - récupérer `gcc-arm-none-eabi-4_9-2015q3-20150921-linux.tar.bz2`
      - et l'installer où vous voulez (`/opt` par exemple)
      - pensez à mettre à jour votre `$PATH` !

tl;dr

- **assembleur rapide**
- **génération de code**
- **gestion de la pile**
  - caller saved / callee saved
  - stack frame / AR
  - dynamic link, static link
- **conventions d'appel ARM**
  - passage d'arguments
  - valeur de retour
- **résumé ABI RAM**

Génération de code

- **But**
  - transformer l'AST en des suites d'instructions assembleur
  - qui seront ensuite transformées instructions machine par un assembleur

Génération de code

- **Code machine**
  - Le code machine
    - est simple : x^y n'existe pas
    - a peu d'arguments : `f(u, v, w, x, y, z)` est impossible
    - est peu structuré : `for(i=0; i<10; i++)` n'est pas représentable simplement
  - Il faut transformer l'IR en instructions élémentaires

Génération de code

- **Constructions ternaires**
  - classiquement, chaque opération est transformée en une suite de constructions généralement ternaires, car cela :
    - correspond aux possibilités usuelles d'un microprocesseur
    - permet d'optimiser indépendamment chaque instruction
    - permet d'unifier les sous-expressions communes
  - certains jeux d'instructions ARM disposent de quelques opérations plus que ternaires (MLA, STM, LDM).

Génération de code

- **AST :**
  - contient des variables
    - pas de notion d'adresse mémoire
    - seulement un flag d'escaping
  - des pseudo-instructions
  - des appels de fonction, mais sans spécifier comment
  - peut contenir des constructions évoluées (boucles) qui n'existent pas en assembleur
  - peut contenir des appels de fonctions dont les arguments sont des expressions

Génération de code

- **Assembleur :**
  - instructions basiques (souvent ternaires) spécifiques au processeur cible
  - pas de notions de variables :
    - registres (en nombre fini)
    - notions d'adresses mémoire, de pile
    - pas de notion d'appartenance de variable à une fonction
  - doit mettre en place des mécanismes d'appel de fonctions :
    - placer les arguments au bon endroit
    - mettre en place un environnement d'exécution de la fonction sain
    - savoir comment transmettre l'éventuelle valeur de retour

Génération de code

- **Il va donc falloir :**
  - mettre l'AST sous une forme canonique (« IR »)
    - évaluer les arguments de fonctions avant de procéder à l'appel
    - supprimer les structures de contrôle évoluées (boucles)
  - préparer les accès aux variables :
    - définir où elles seront placées (registres, pile, mémoire)
      - certaines variables ne tiennent pas dans des registres
      - certaines variables doivent être stockées en mémoire
      - le nombre de registres physiques est limité
    - généralement fait en deux temps
      - décision de placer une variable dans un registre ou non
      - puis utilisation d'un allocateur de registres

Génération de code

- **(suite)**
  - Générer les appels de fonctions :
    - pour permettre à chaque sous-programme présent dans l'arbre d'être appelé
    - moyens :
      - pour chaque instruction, générer du code effectuant les bonnes opérations
      - générer du code pour l'entrée (prologue) et la sortie (épilogue) du sous-programme

Génération de code

- **En pratique:**
  - on travaillera le plus possible sur l'IR
  - on ne passera à l'assembleur qu'au dernier moment possible

  - Ça implique qu'on se retrouvera, juste avant le passage à l'assembleur, à un IR spécialisé pour un processeur particulier.
    - facilite énormément la génération de l'assembleur

Où en est-on ?

- **On a vu :**
  - différences principales entre IR et assembleur
- **Ce qu'on va voir :**
  - un bref aperçu de l'assembleur ARM
  - comment traduire les constructions usuelles en assembleur
- **Plus loin :**
  - l'organisation de la pile
  - les conventions d'appel (ABI ARM)

Assembleur ARM : un spoiler

- **Cible**
  - on ciblera dans cette UE les ARM Cortex M
  - 16 registres 32 bits
    - R0 à R15
    - 3 réservés à des usages spéciaux
      - R15 : PC
      - R14 : LR (Link Register)
      - R13 : SP (Stack Pointer)
    - on réservera éventuellement d'autre registres à des fonctions spéciales à nous

Assembleur ARM : un spoiler

- **Généralités**
  - L'ARM est une architecture load / store : toutes les opérations arithmétiques et logiques sont exclusivement faites entre registres.

  - Instructions arithmétiques et logiques
    - format : `OPE Rdest, Rop1, Rop2`
    - exemples :
      - `AND R0, R1, R2` pour (R0 = R1 & R2)
      - `ADD PC, PC, R5` pour (PC = PC + r5)

Assembleur ARM : un spoiler

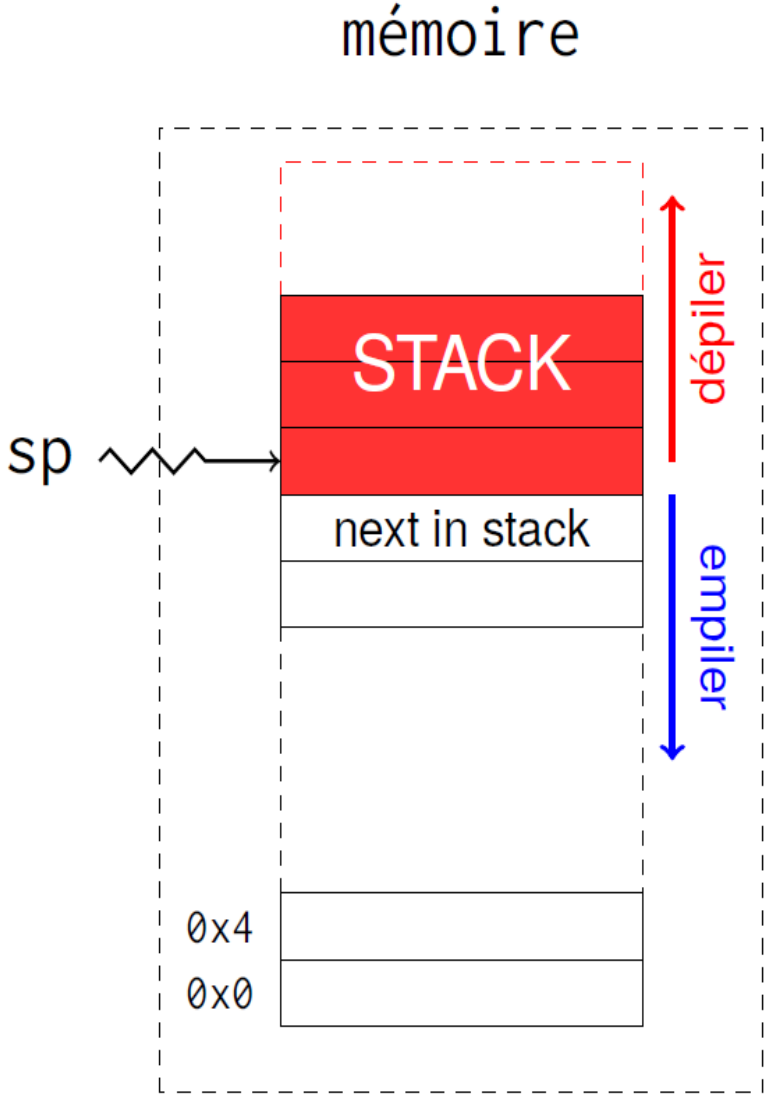
- **Accès aux données**
  - Déplacement de données :
    - `MOV R0, R1` : R0 = R1
    - `LDR R0, =valeur` : R0 = valeur
  - Accès à la mémoire :
    - `LDR R0, [R1]` : R0 = RAM[R1]
    - `STR R0, [R1]` : RAM[R1] = R0

Assembleur ARM : un spoiler

- **Branchements**
  - Branchements inconditionnels :
    - `B R0` : sauter à l'adresse contenue dans R0
    - `BL R1` : sauter à l'adresse contenue dans R1 et charger dans LR l'adresse de retour
  - Branchements conditionnels
    - `CMP R0, R1` : compare R0 à R1 et positionne des flags d'égalité / de « plus grand que » dans un registre interne de l'ARM
    - `BEQ R3` : effectue le branchement si R0 == R1
    - `BGT R3` : effectue le branchement si R0 > R1
    - `BGE R3` : effectue le branchement si R0 >= R1

Assembleur ARM : un spoiler

- **Pile**
  - Le registre R13 (SP) est le pointeur de pile (stack pointer)
  - Le pointeur de pile contient l'adresse de la dernière donnée empilée
  - Avant chaque empilement le pointeur de pile doit être décrémenté
  - La pile est dite "Full Descending"



Assembleur ARM : un spoiler

- **Pile**
  - Pour manipuler la pile :
    - Empiler :
      - `PUSH R0`
      - `PUSH {R1-R5}`
    - Dépiler :
      - `POP R0`
      - `POP {R1-R5}`

Assembleur ARM : un spoiler

- **En résumé :**
  - Ceci est juste un bref aperçu pour la suite de ce cours
  - Vous aurez bientôt un cours complet d'assembleur ARM

  - `ADD R1, R0, R1 :?`
  - `LDR R1, = valeur :?`
  - `LDR R1, [R1] :?`
  - `PUSH R0 :?`
  - `POP {R1-R5} :?`
  - `BL R0 :?`
  - `BEQ LR :?`

Où en est-on ?

- **On a vu**
  - différences principales entre IR et assembleur
  - un aperçu de l'assembleur ARM
- **Ce qu'on va voir:**
  - comment traduire les constructions usuelles en assembleur
- **Plus loin :**
  - l'organisation de la pile
  - les conventions d'appel (ABI ARM)

Parenthèse de C

- **En C, les goto existent !**
  - On peut définir des labels et sauter à l'un de ces labels

```
{
  …
start:
  …
  if (…)
    goto end;
  goto start;
  …
end:
  …
}
```

  - attention : à consommer avec modération !
  - http://cs.sjsu.edu/~mak/CS185C/KnuthStructuredProgrammingGoTo.pdf

Parenthèse de C

- **En C, les goto existent !**
  - cas d'utilisation légitimes :

```
void foo()
{
  if (!try_A())
    goto exit;
  if (!try_B())
    goto cleanupA;
  if (!try_C())
    goto cleanupB;

  // everything succeeded
  return;

 cleanupB:
  undoB();
 cleanupA:
  undoA();
 exit:
  return;
}
```

- gestion des erreurs

```
void foo()
{
  while (...) {
    while (...) {
      if (...)
        goto end;
      // loop action
    }
  }
 end:
  // end action
  ...
}
```

- sortie de boucles imbriquées

Constructions usuelles

- **Sauts simples et appels de fonctions :**

En C :

```
a:
   ...
   goto a;
```

devient :

```
a:
   ...
   b a
```

En C :

```
   ...
   f();
   ...
```

devient :

```
   ...
   bl f
   ...
```

Constructions usuelles

- **Boucles**
  - Les boucles sont généralement ré-écrites dans l'IR selon la structure suivante :
    - 1 : initialisation des paramètres de la boucle
    - 2 : saut par-dessus l'étape 3 si le test est en fin de boucle
    - 3 : test de sortie, saut après 6 si positif
    - 4 : corps de la boucle
    - 5 : exécution de la partie finale de la boucle
    - 6 : saut inconditionnel en 3
    - 7 : suite du programme

Constructions usuelles

- **Boucles**
  - Exemple : `for (i=0;i<10;i++) {...}`

```
1 : i = 0
2 :
3 : if NOT(i < 10) goto 7
4 : ...
5 : i = i + 1
6 : goto 3
7 :
```

Constructions usuelles

- **Boucles**
  - Exemple : `while(c) {...}`

```
1 :
2 :
3 : if NOT(c) goto 7
4 : ...
5 :
6 : goto 3
7 :
```

Constructions usuelles

- **Boucles**
  - Exemple : `do {...} while(c)`

```
1 :
2 : goto 4
3 : if NOT(c) goto 7
4 : ...
5 :
6 : goto 3
7 :
```

Constructions usuelles

- **Accès aux variables**
  - sur les architectures load / store :
    - une instruction ne peut manipuler que des registres
    - l'accès à la mémoire se fait par les instructions LDR et STR
  - pour accéder à une variable, il faut donc :
    - d'abord stocker son adresse dans un registre
    - puis accéder à la mémoire (LDR ou STR)

Constructions usuelles

- **Accès aux variables**
  - Un symbole représente
    - en C / Tiger / langage de haut niveau : *la valeur* d'une variable
    - en assembleur ou linker script : *l'adresse* de cette variable
  - ainsi, avec a entier 32 bits, `a = 3` pourrait devenir :

```
 0:    ldr r3, [pc, #8]    ; r3 ← &a
 4:    mov r2, #3          ; r2 ← 3
 8:    str r2, [r3]        ; 3 → mem[&a]
 c:    bx  lr              ; passage à la suite
10:    .word a             ; adresse de a
```

```
0:  ldr r3, =a          ; r3 ← &a
4:  mov r2, #3          ; r2 ← 3
8:  str r2, [r3]        ; 3 → mem[&a]
8:  bx lr               ; passage à la suite
```

Constructions usuelles

- **Accès aux variables**

```
int32_t a, b;
a = b;
```

pourrait devenir :

```
 0:    ldr r3, [pc, #c]    ; r3 ← &b
 4:    ldr r2, [r3]        ; r2 ← mem[&b]=b
 8:    ldr r3, [pc, #8]    ; r3 ← &a
 c:    str r2, [r3]        ; b → mem[&a]
10:    bx  lr              ; on passe à la suite
14:    .word b             ; adresse de b
18:    .word a             ; adresse de a
```

Constructions usuelles

- **Accès aux variables**

```
int32_t a, b, c;
a = a + b*c;
```

pourrait devenir :

```
 0:   ldr   r3, =a           ; r3 ← &a
 4:   ldr   r1, =b           ; r1 ← &b
 8:   ldr   r2, =c           ; r2 ← &c
 c:   ldr   r0, [r1]         ; r0 ← b
10:   ldr   r2, [r2]         ; r2 ← c
14:   ldr   r1, [r3]         ; r1 ← a
18:   mla   r2, r0, r2, r1   ; r2 ← b*c + a
1c:   str   r2, [r3]         ; r2 → mem[&a]
```

Constructions usuelles

- **Accès aux variables**

```
*p++ = 3;
```

pourrait devenir :

```
 0: ldr   r2, =p          ; r2 ← &p
 4: ldr   r3, [r2]        ; r3 ← p
 8: mov   r1, #3          ; r1 ← 3
 c: str   r1, [r3], #4    ; 3 → mem[p] puis r3 ← p+4
10: str   r3, [r2]        ; p+4 → mem[&p]
```

Chouette, des exercices !

- **À vous de travailler :**
  - exercice 1 : facile
  - exercice 2 : facile
  - exercice 3 : moyen

Exercices de compilation

- **Exercice 1**
  - Traduire en assembleur ARM le code Tiger suivant :

```
let
    var a := 3
    var b := 8
in
    if a>b then 10 else 20
end
```

Exercices de compilation

- **Exercice 2**
  - Traduire en assembleur ARM le code C suivant :

```
uint32_t a; // global variable
...
for (uint8_t i=0; i<=a; i++)
    g();
```

  - Même question si `i` est un `unsigned int`. Conclusion ?

  - Indice : pour voir ce que produit GCC pour ARM :

```
__attribute__((naked)) void f() {
    for (uint8_t i=0; i<=a; i++)
        g();
}
```

Puis : `arm-none-eabi-gcc -Os -S t.c -o -`

Exercices de compilation

- **Exercice 3**
  - Traduire en assembleur ARM le code C suivant :

```
uint32_t *a; // global variable
uint32_t *b; // global variable
uint32_t *c; // global variable
…
*a += *c;
*b += *c;
```

  - Comparez avec ce que produit GCC, ainsi :
    -
```
void f() {
    *a += *c ;
    *b += *c ;
}
```
    - Puis : `arm-none-eabi-gcc -O2 -S t.c -o -`

      Pourquoi GCC charge-t-il deux fois le contenu de `*c` au lieu d'une seule ?

Où en est-on ?

- **Ce qu'on a vu :**
  - différences principales entre IR et assembleur
  - un aperçu de l'assembleur ARM
  - comment traduire les constructions usuelles en assembleur
- **Ce qu'on va voir :**
  - la gestion des fonctions
    - à quoi sert une pile
    - ce qu'est un activation record
    - le passage d'argument
    - le passage de la valeur de retour
  - les conventions d'appel ARM

ABI

- **ABI : application binary interface**
  - Convention regroupant :
    - les types de données, leur tailles et alignements
    - les formats des exécutables, objets et bibliothèques
    - les conventions d'usage de registres
    - les conventions d'appel de fonctions
    - les appels système
    - le format des informations de débug
    - le format et traitement des exceptions
    - etc.
  - Pour ARM :
    - OABI : obsolète
    - EABI : récente et plus performante (basée sur l'AAPCS)
    - EABIHF : la même, le coprocesseur flottant en plus

Structure d'une fonction

- **Architecture d'une fonction (C,Tiger, ...)**
  - une fonction se compose :
    - d'un prologue : met en place un environnement d'exécution adéquat
    - du corps de la fonction
    - d'un épilogue :
      - fait le ménage,
      - place l'éventuelle valeur de retour au bon endroit,
      - et transfère le contrôle à la fonction appelante.
  - On peut demander en C la suppression des prologues / épilogues par l'attribut `naked`
    - exemple : `__attribute__ ((naked)) void f(void);`

Structure d'une fonction

- **Environnement d'exécution**
  - Une fonction a souvent besoin d'utiliser des registres
  - mais elle ne sait pas si ces registres sont déjà utilisés ou non par les fonctions appelantes !
  - → elle doit les remettre dans leur état original avant de retourner

  - Conclusion : *Merci de laisser cet endroit aussi propre en sortant que vous l'avez trouvé en entrant*

Sauvegarde des registres

- **Stratégies**
  - *caller saved registers* : la procédure appelante sait quels sont les registres qui doivent ne doivent pas être modifiés par l'appel, et les sauvegarde.
    - inconvénient : si tous les registres sont caller-saved, on risque d'en sauvegarder beaucoup trop
  - *callee saved registers* : la procédure appelée sait quels sont les registres qu'elle va modifier, et les sauvegarde.
    - inconvénient : si tous les registres sont callee-saved, on risque d'en sauvegarder beaucoup trop
  - Une stratégie optimale consiste en une approche intermédiaire.

Sauvegarde des registres

- **Stratégie intermédiaire**
  - Les registres sont partagés en deux groupes :
    - `n` registres caller saved
    - `m` registres callee saved
  - Quelle répartition ?
    - optimisation de la taille du code
    - optimisation de la rapidité d'exécution du code

Sauvegarde des registres

- Optimisation de la taille du code
  - Le code de sauvegarde / restauration est généré :
    - *caller saved register* : 1 fois par appel de procédure
    - *callee saved register* : 1 fois par procédure
- Optimisation de la vitesse d'exécution
  - Une procédure terminale a tout intérêt à utiliser des registres caller saved.
  - Une procédure non terminale a tout intérêt à utiliser :
    - pour les registres devant survivre à un appel de fonction, des registres callee saved
    - sinon, des registres caller saved.

Sauvegarde des registres

- EABI ARM (AAPCS) :
  - R0 à R3 : caller saved
  - R4 à R11 : callee saved

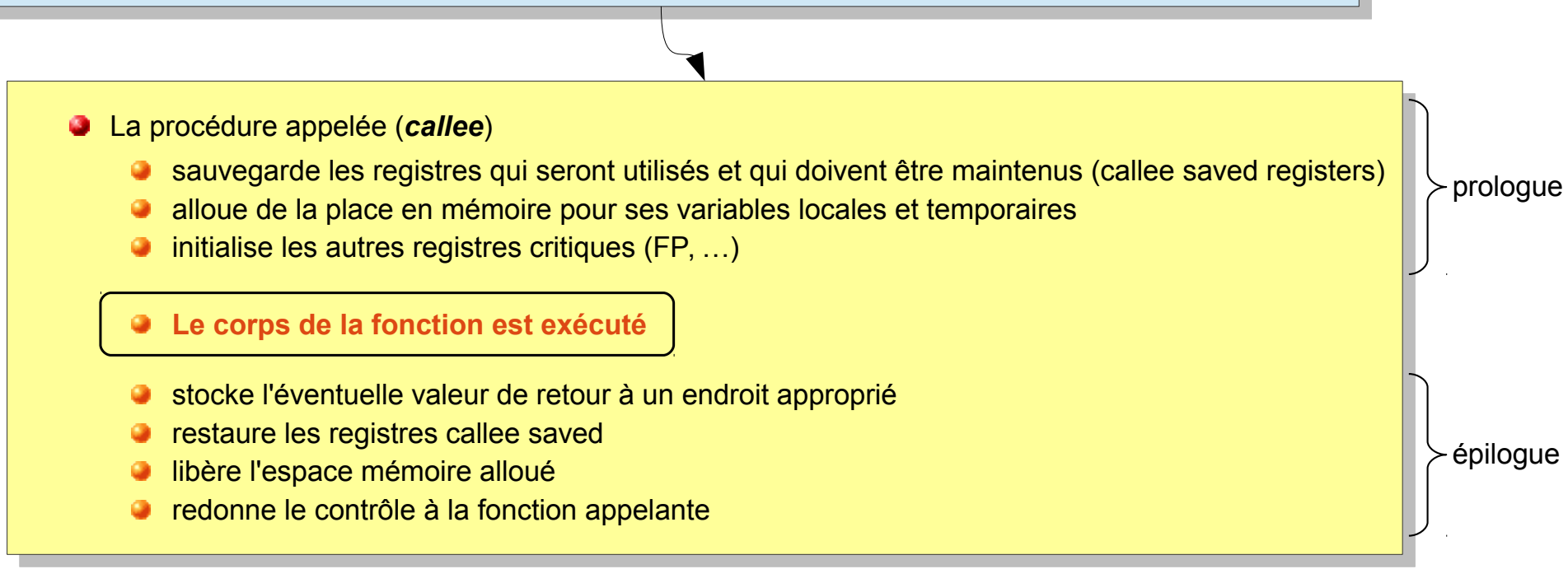
- Quizz :
  - qu'en est-il de :
    - R15 ?
    - R14 ?

Mécanisme d'appel de fonction

- La procédure appelante (***caller***) :
  - évalue les arguments et les place à des endroits appropriés
  - sauve l'adresse de retour
  - sauvegarde les registres qu'elle utilise (caller saved registers) et qui doivent être préservés
  - passe le contrôle à la procédure appelée

Mécanisme d'appel de fonction

- La procédure appelante (***caller***) :
  - évalue les arguments et les place à des endroits appropriés
  - sauve l'adresse de retour
  - sauvegarde les registres qu'elle utilise (caller saved registers) et qui doivent être préservés
  - passe le contrôle à la procédure appelée



Mécanisme d'appel de fonction

- La procédure appelante (***caller***) :
  - évalue les arguments et les place à des endroits appropriés
  - sauve l'adresse de retour
  - sauvegarde les registres qu'elle utilise (caller saved registers) et qui doivent être préservés
  - passe le contrôle à la procédure appelée

- La procédure appelée (***callee***)
  - sauvegarde les registres qui seront utilisés et qui doivent être maintenus (callee saved registers)
  - alloue de la place en mémoire pour ses variables locales et temporaires
  - initialise les autres registres critiques (FP, …)

prologue

  - **Le corps de la fonction est exécuté**
  - stocke l'éventuelle valeur de retour à un endroit approprié
  - restaure les registres callee saved
  - libère l'espace mémoire alloué
  - redonne le contrôle à la fonction appelante

épilogue

- La procédure appelante :
  - restaure ses registres sauvegardés,
  - désalloue les éventuels arguments
  - et continue son exécution.

Où en est-on ?

- **À venir :**
  - Où sauvegarder les choses ?
  - De quelle façon ?
  - Comment transmettre les arguments ?
  - Comment transmettre la valeur de retour ?

Pile / activation record

- **Pile et activation record:**
  - Dans la plupart des langages de haut niveau, les fonctions peuvent être récursives ou ré-entrantes. Chaque instance d'appel de fonction doit donc pouvoir disposer de son propre espace de stockage pour ses objets.
  - Les garder dans des endroits statiques est très difficile
    - quizz : pourquoi ?
  - Il faut donc pouvoir allouer dynamiquement de la mémoire pour stocker ces objets
  - La plupart du temps, cela est fait avec :
    - une ou plusieurs pile(s) : C, C++, Java, Ada, C#, Pascal, …
    - le tas : LISP, Scheme, …
  - L'espace contenant les objets propres à un appel de fonction est appelé « activation record » ou « activation frame »

Pile / activation record

- **Rôle de "la" pile :**
  - Originellement : stocker l'adresse de retour d'une fonction (call stack)
  - Implémentation :
    - matérielle
    - logicielle
    - mixte
  - On en profite généralement pour y stocker aussi :
    - certains paramètres lors d'un appel de fonction
    - les sauvegardes de registres / état du processeur
    - des informations diverses…
  - Selon les langages et architectures, une ou plusieurs piles (Forth, Stackless Python)

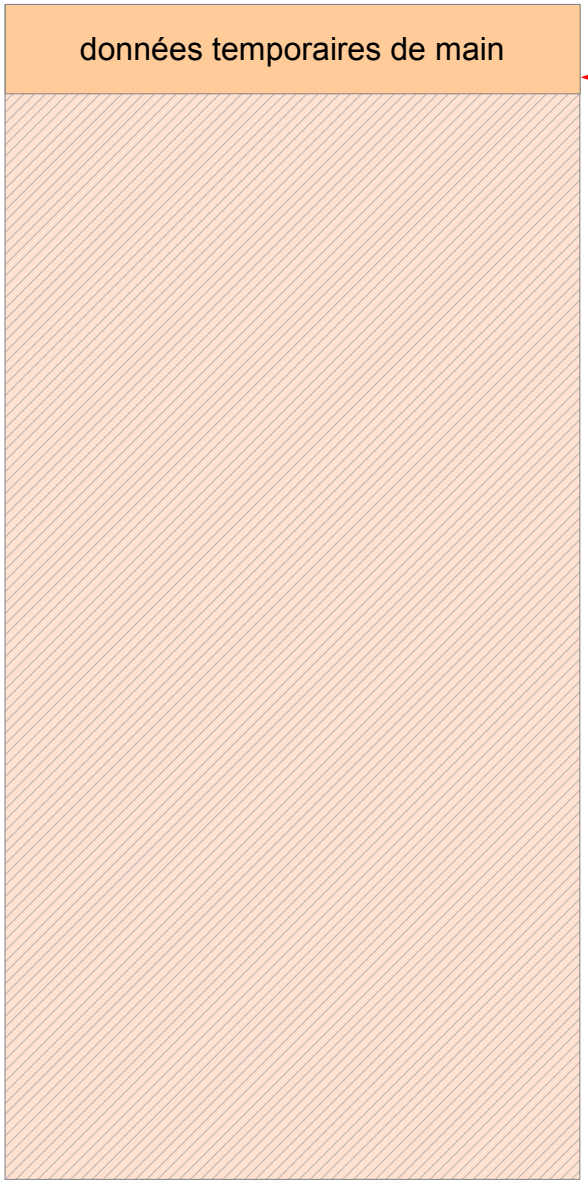
Pile / activation record

- **Contenu usuel d'un activation record (AR)**
  - adresse de retour (si nécessaire)
  - arguments de la fonction
  - sauvegardes de registres et de l'état du processeur
  - dynamic link : lien vers l'AR de la fonction appelante
  - static link : lien vers l'AR de la la fonction dans le code où est définie la fonction actuelle
  - données locales
    - variables locales
    - valeurs intermédiaires dans certaines expressions
    - données dynamiques
  - `this` (ou équivalent, en langages Objets)
  - pointeurs vers handlers d'exception
  - …

Pile / activation record en C

ATTENTION
Sera modifié plus tard !

```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```

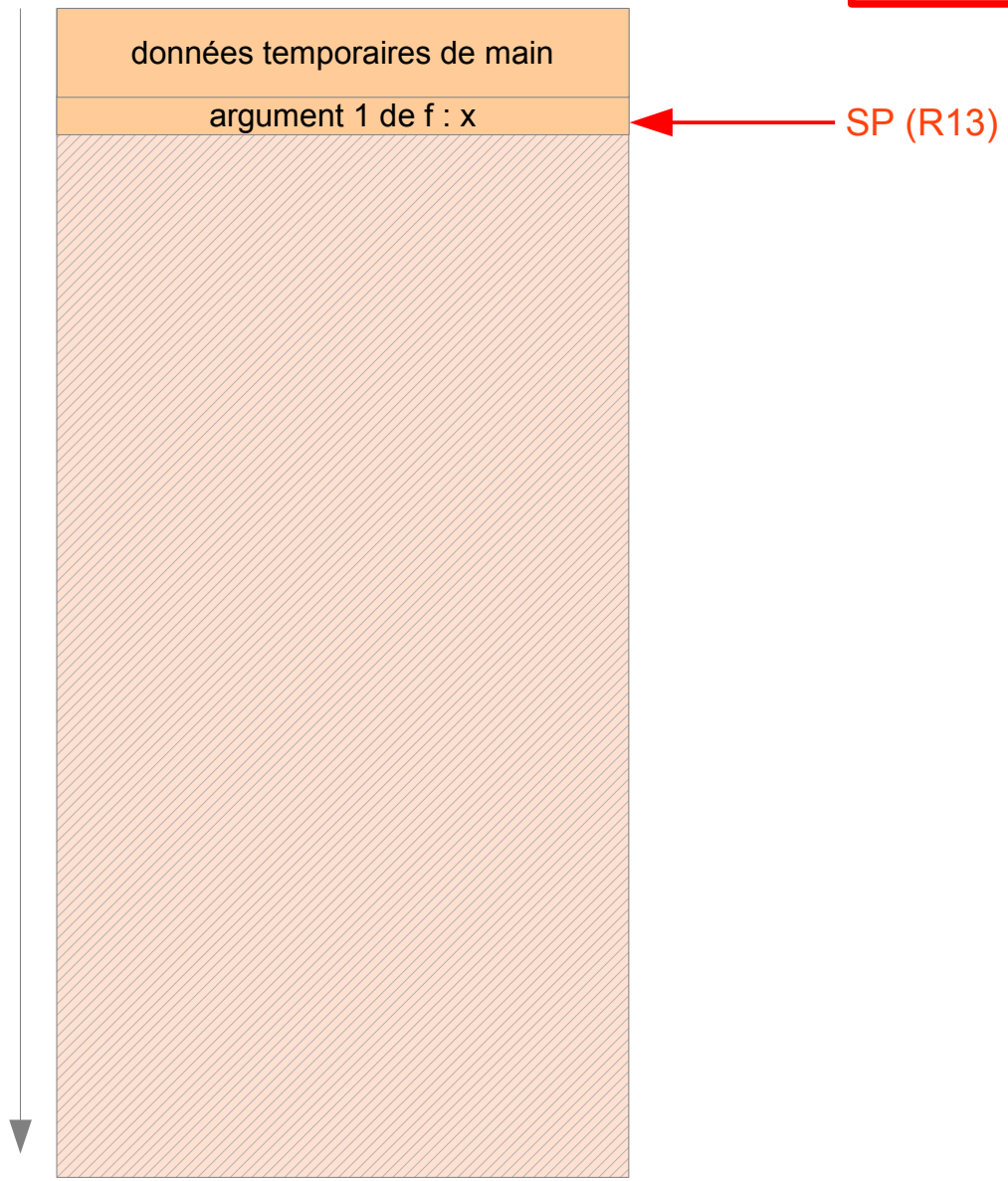


direction de la pile
(ici, full-descending)

Pile / activation record en C

ATTENTION
Sera modifié plus tard !

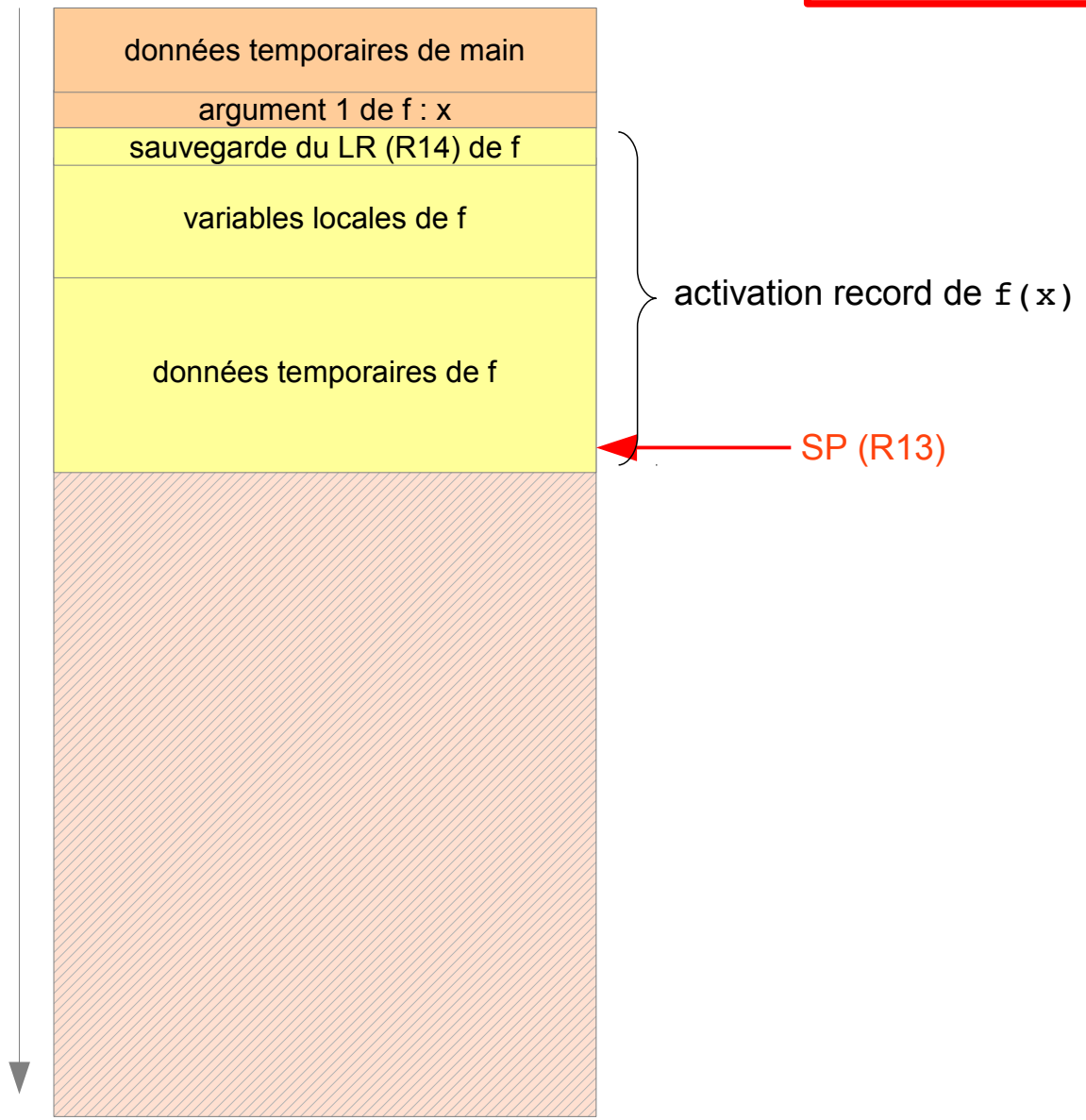
```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```



Pile / activation record en C

ATTENTION
Sera modifié plus tard !

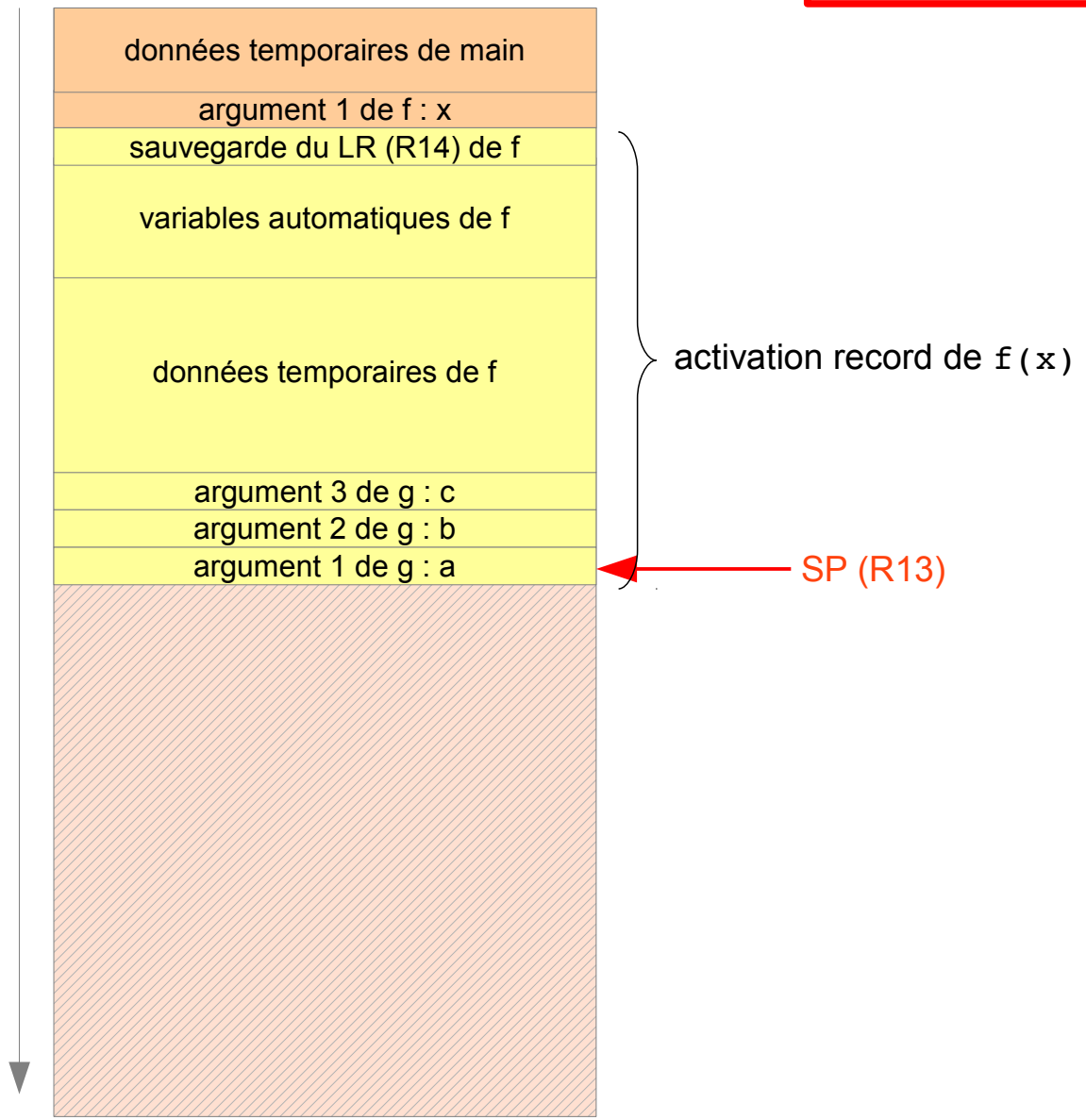
```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```



Pile / activation record en C

ATTENTION
Sera modifié plus tard !

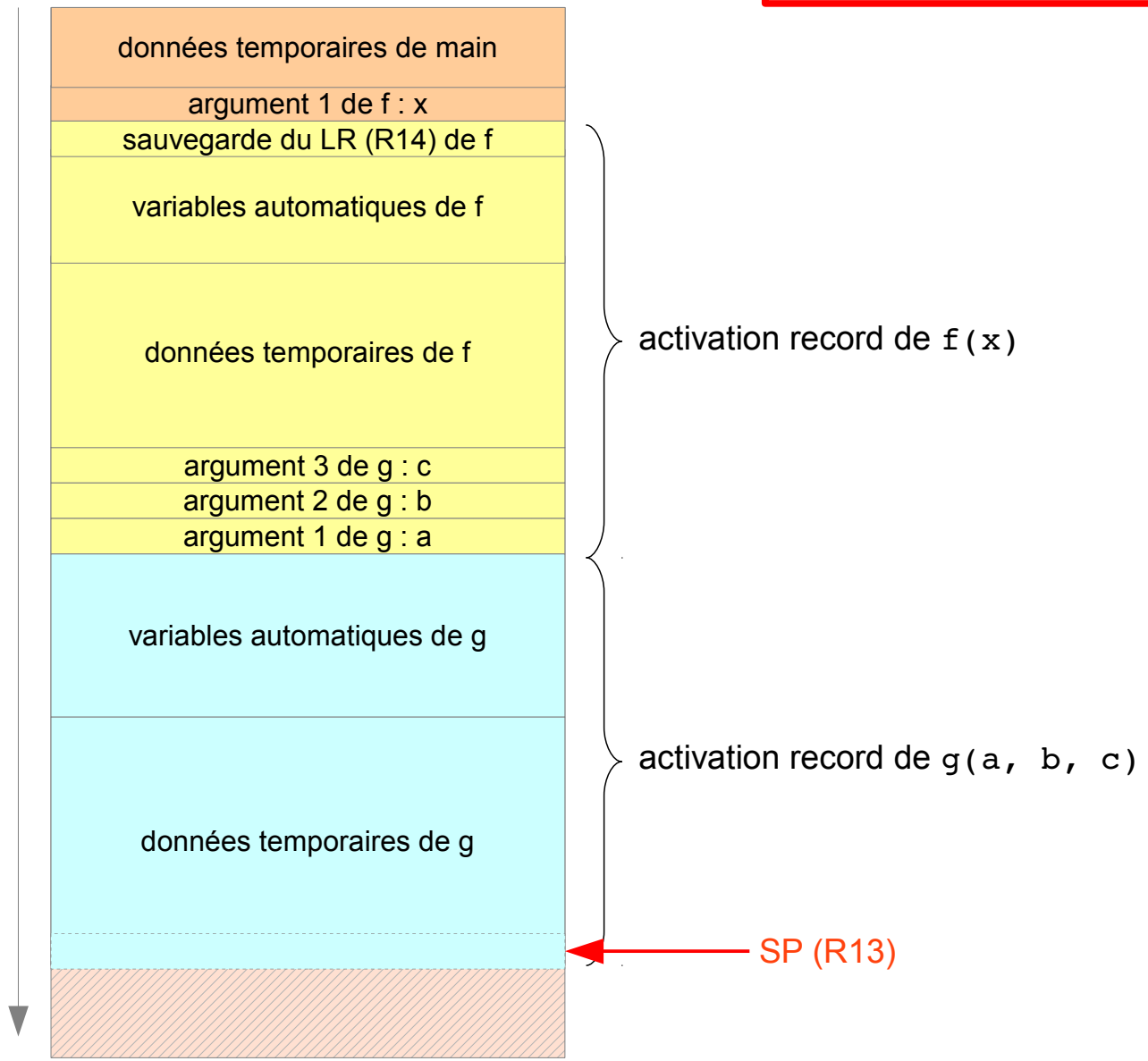
```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```



Pile / activation record en C

ATTENTION
Sera modifié plus tard !

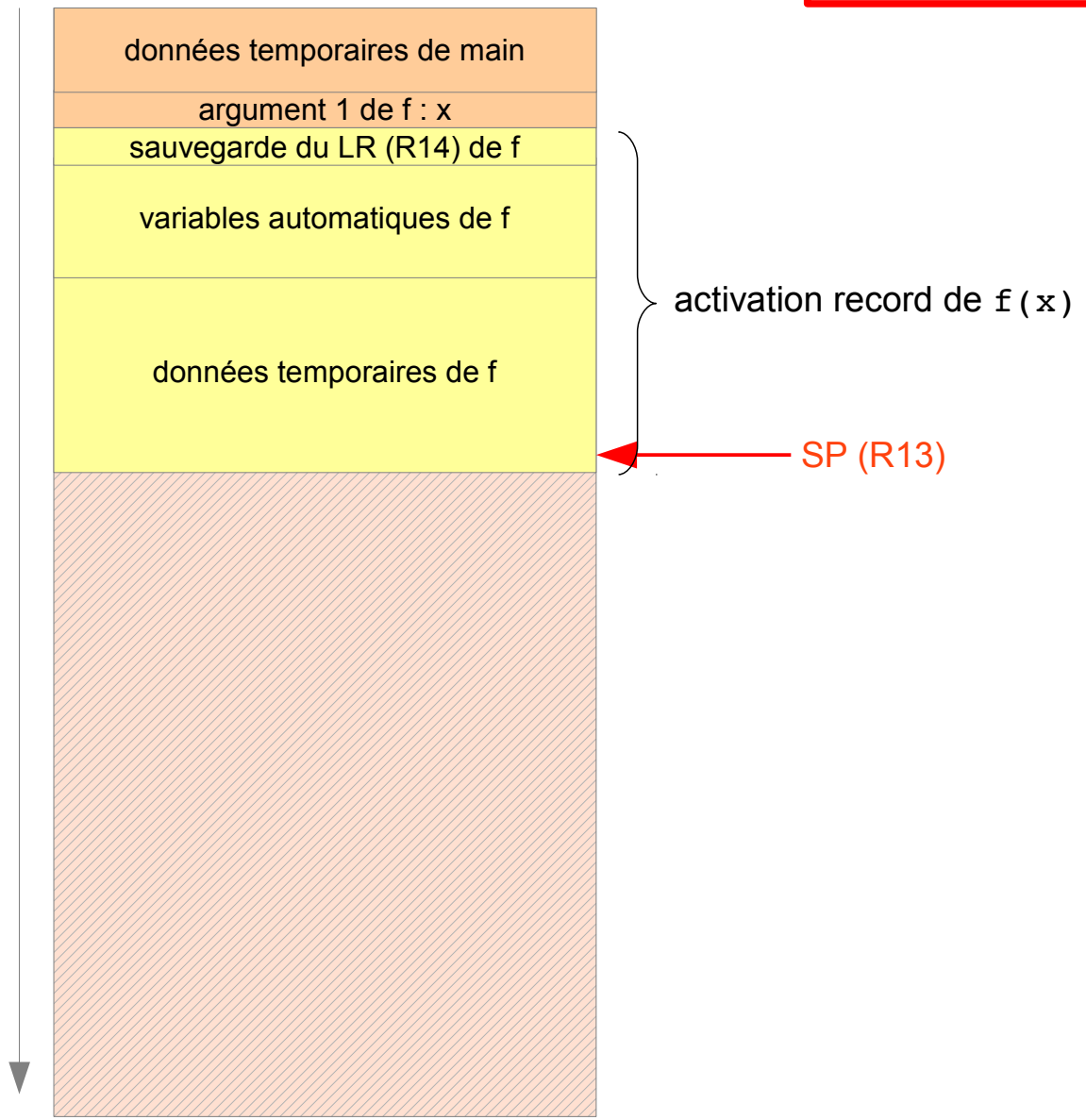
```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```



Pile / activation record en C

ATTENTION
Sera modifié plus tard !

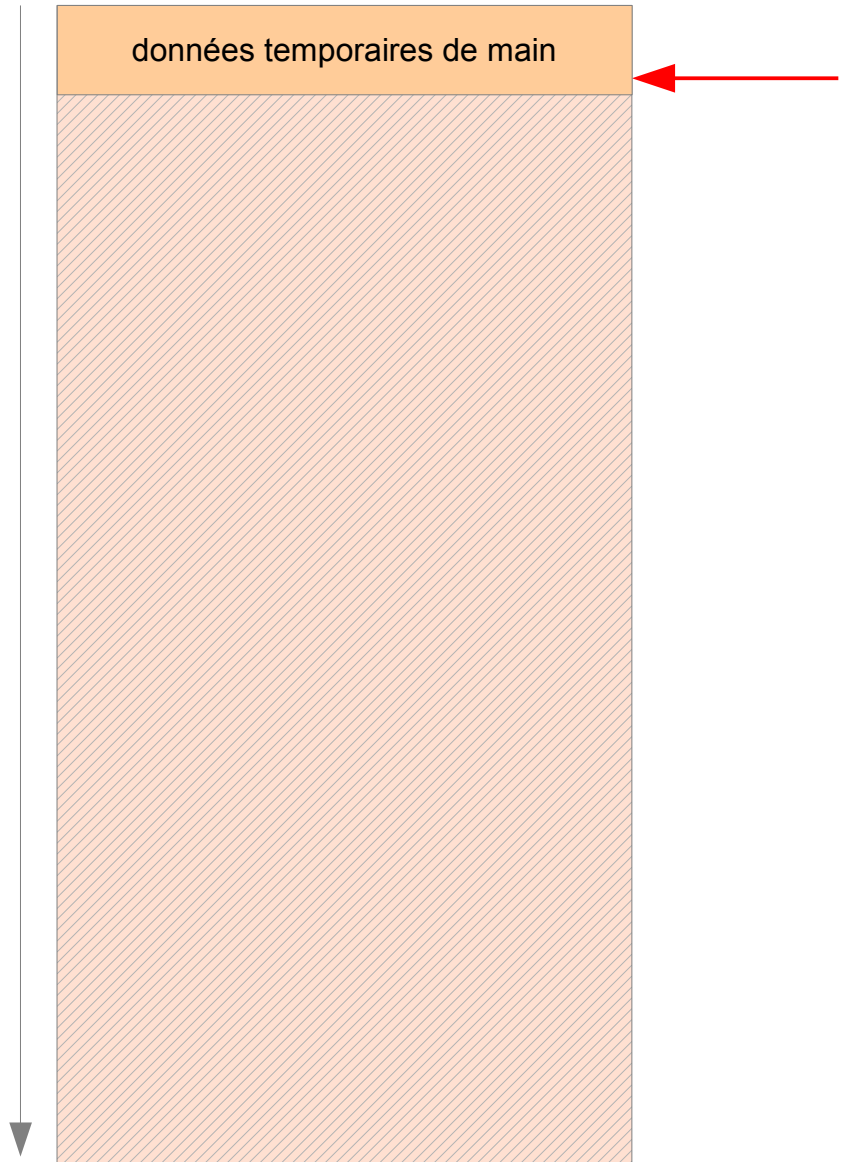
```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```



Pile / activation record en C

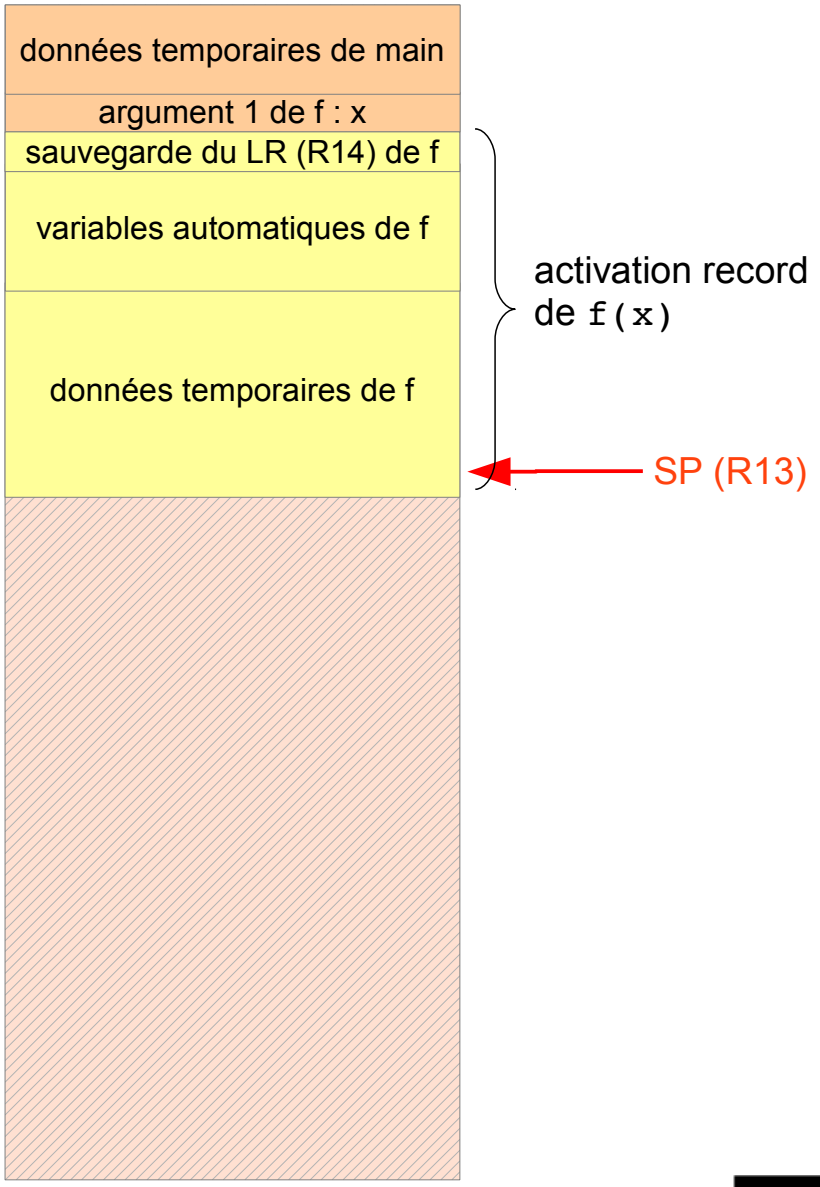
ATTENTION
Sera modifié plus tard !

```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```



Activation frame / Frame pointer

- **Adressage des données**
  - Si nous sommes actuellement dans f(x), et si toutes ses données sont de taille connue à la compilation :
    - comment fait-on référence aux arguments de f ?
    - comment fait-on référence aux variables de f ?
    - comment fait-on référence aux données temporaires de f ?



Réponse : ?

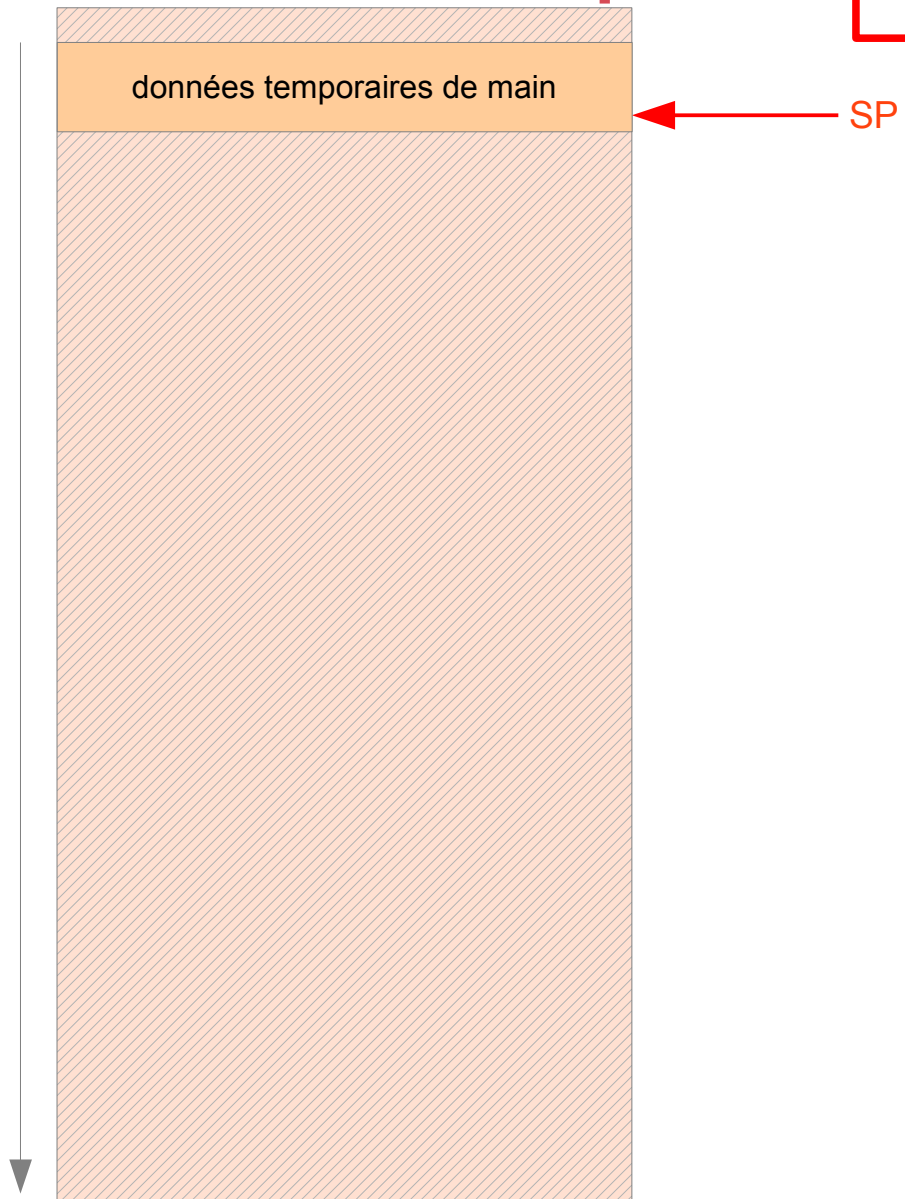
Activation frame / Frame pointer

- **Problème**
  - En C, comme dans beaucoup de langages, les données locales peuvent être de taille non connue à la compilation :
    - exemples ?
  - Conséquence :
    - la taille des activation record n'est donc pas connue à la compilation
    - la position du SP n'est donc pas connue à la compilation
    - il est impossible de calculer les adresses des variables locales / arguments en se basant sur SP
  - Il faut un pointeur stable : Frame Pointer (FP, R11 sur ARM)

Activation frame / Frame pointer

ATTENTION :
Sera modifié plus tard !

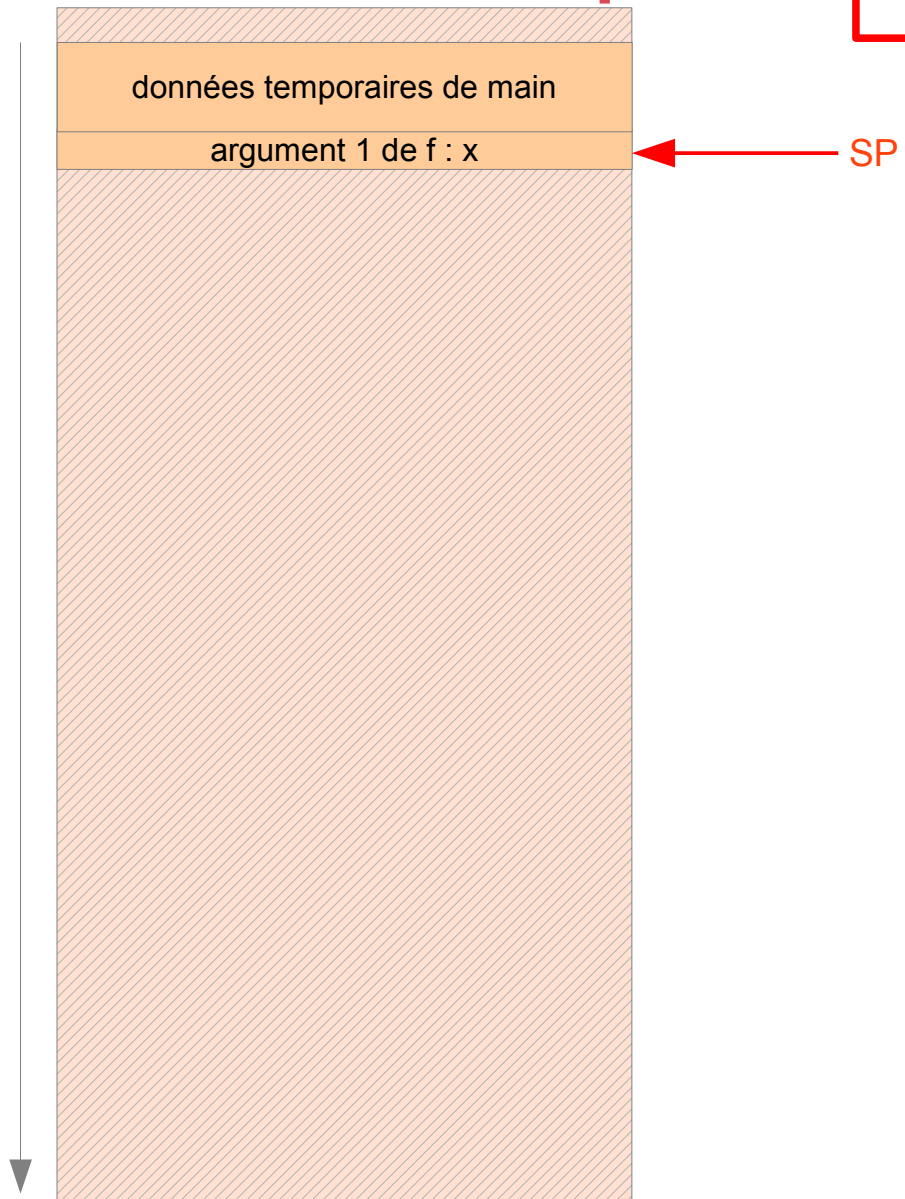
```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```



Activation frame / Frame pointer

ATTENTION :
Sera modifié plus tard !

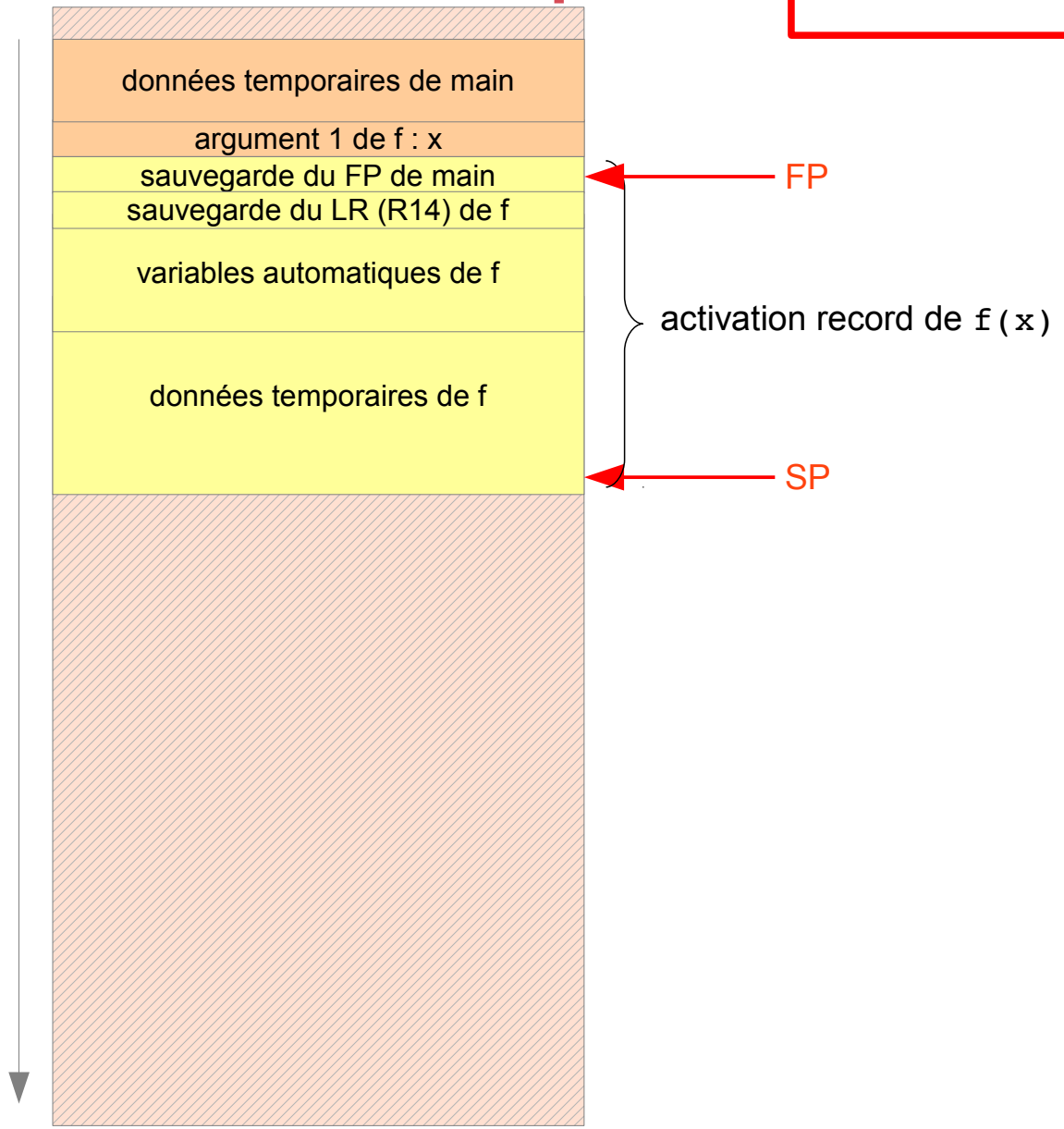
```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```



Activation frame / Frame pointer

ATTENTION :
Sera modifié plus tard !

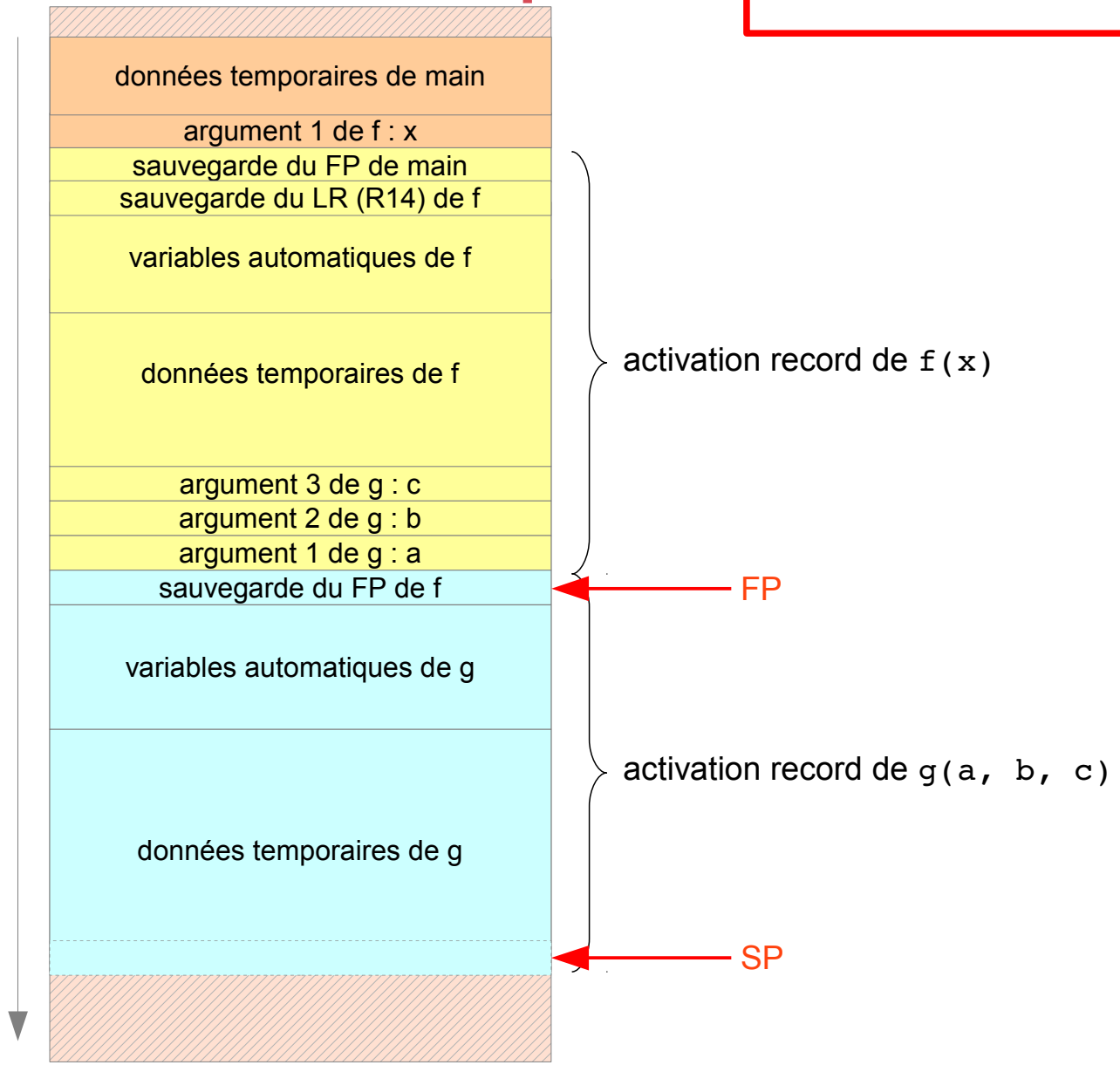
```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```



Activation frame / Frame pointer

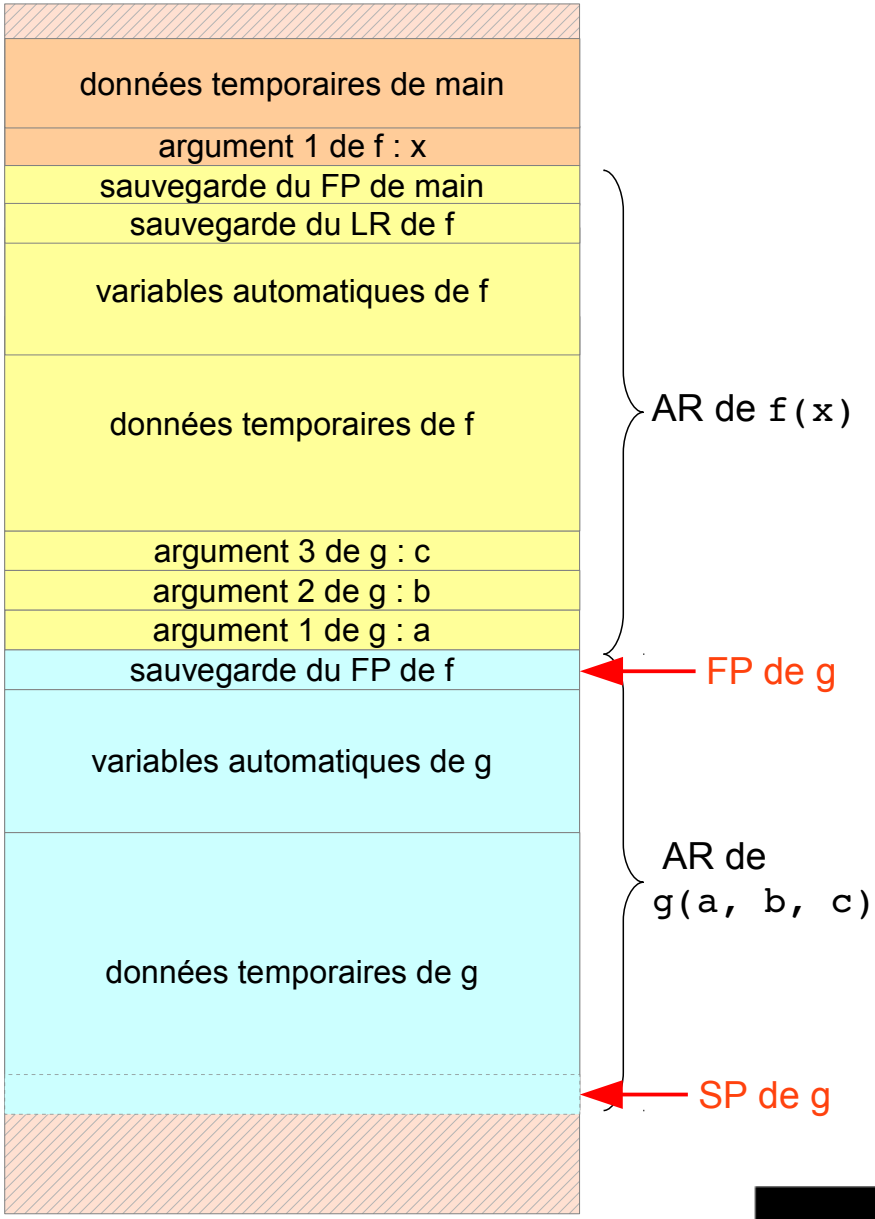
ATTENTION :
Sera modifié plus tard !

```
void g(int a,
       int b,
       int c)
{
  ...
}

int f(int x)
{
  ...
  g(a, b, c);
  ...
}

int main()
{
  ...
  res = f(x);
  ...
}
```



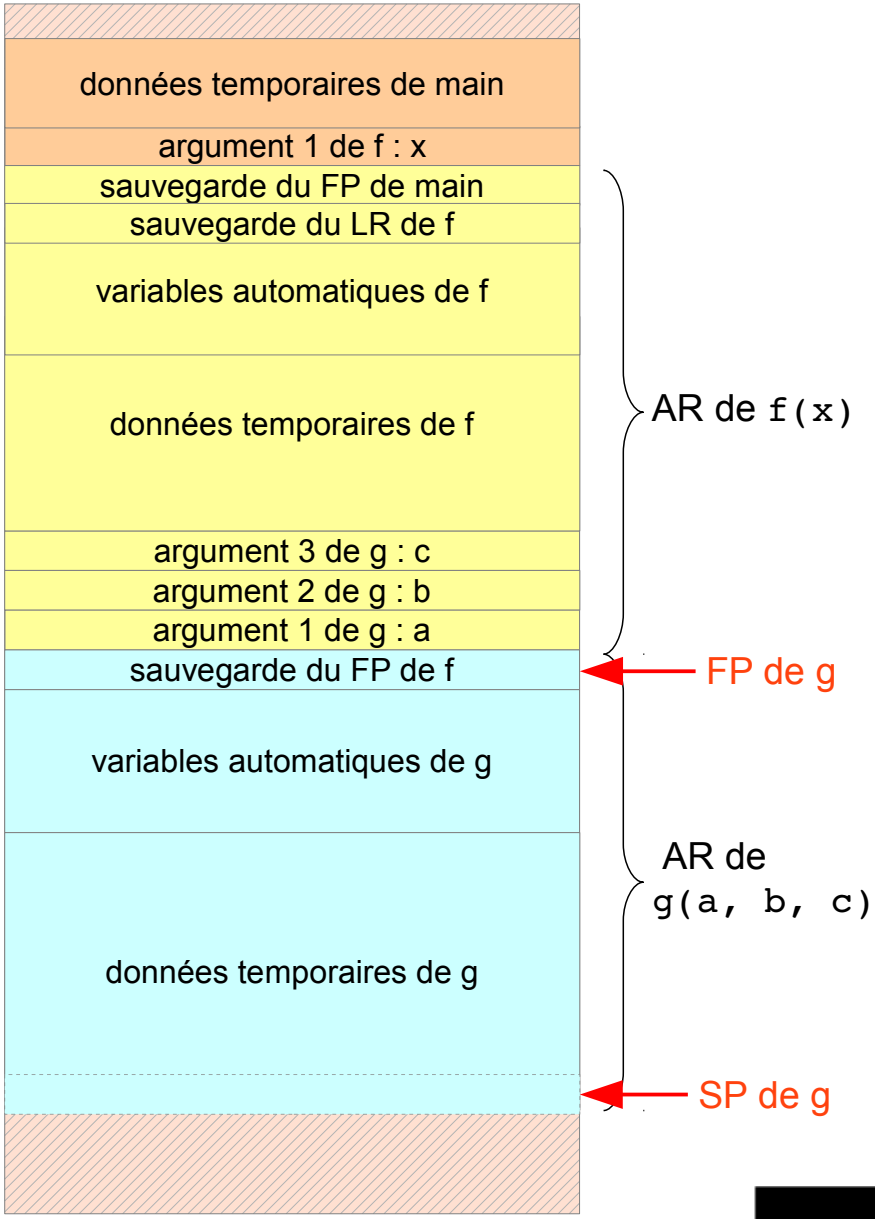
Activation frame / Frame pointer

- **Frame Pointer**
  - permet de calculer les adresses des données d'une fonction, même en cas de SP variable
    - les arguments sont en FP + 4, FP + 8, FP + C, …
    - les variables locales et temporaires sont en FP – 4n
  - Sur ARM, traditionnellement le Frame Pointer est R11
  - Si on sait qu'il n'y aura pas d'allocation dynamique sur la pile, on peut supprimer le frame pointer grâce à l'option de compilation `-fomit-frame-pointer`.



Activation frame / Frame pointer

- **Retour de fonction**
  - SP <= FP - 4
  - FP <= Ancien FP
  - L'appelant désalloue ensuite les arguments



- **Quizz**
  - Où se trouve le dynamic link ?

Static link

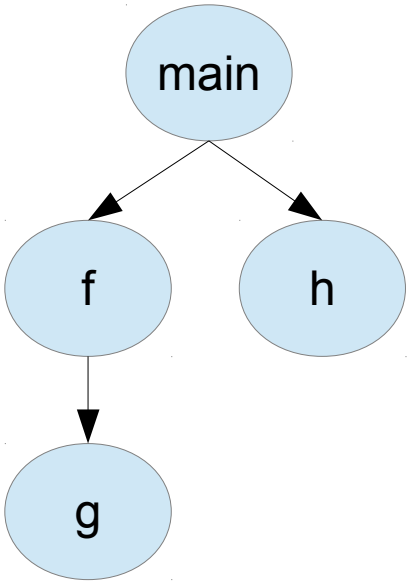
- **Problème**
  - Certains langages (Tiger, Pascal, Python, …) autorisent à déclarer des fonctions à l'intérieur d'autres fonctions.

  - Exemple :

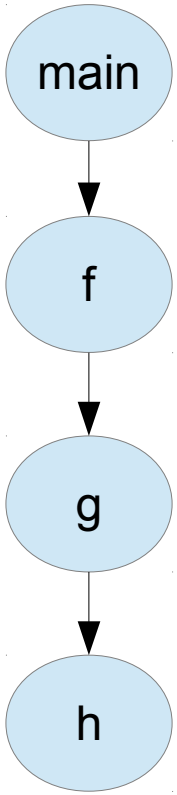
```
let
   var a := 5
   var b := 10

   function h(z:int):int =
    z+a+b

   function f(x:int):int =
      let
         function g(y:int):int =
            h(y)+x+a
      in
         g(x)+b
      end
in
   f(a)
end
```

Static link

- **Graphes**
  - graphe de déclaration

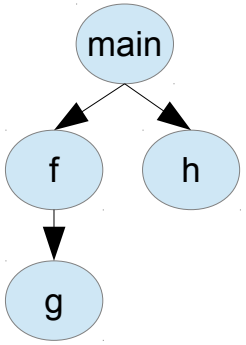


  - graphe d'appel



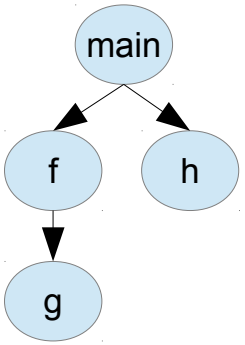
  - mais
    - f fait référence à des variables de main : `b`
    - g fait référence à des variables de main :`a`, et f : `x`
    - h fait référence à des variables de main : `a`

Static link

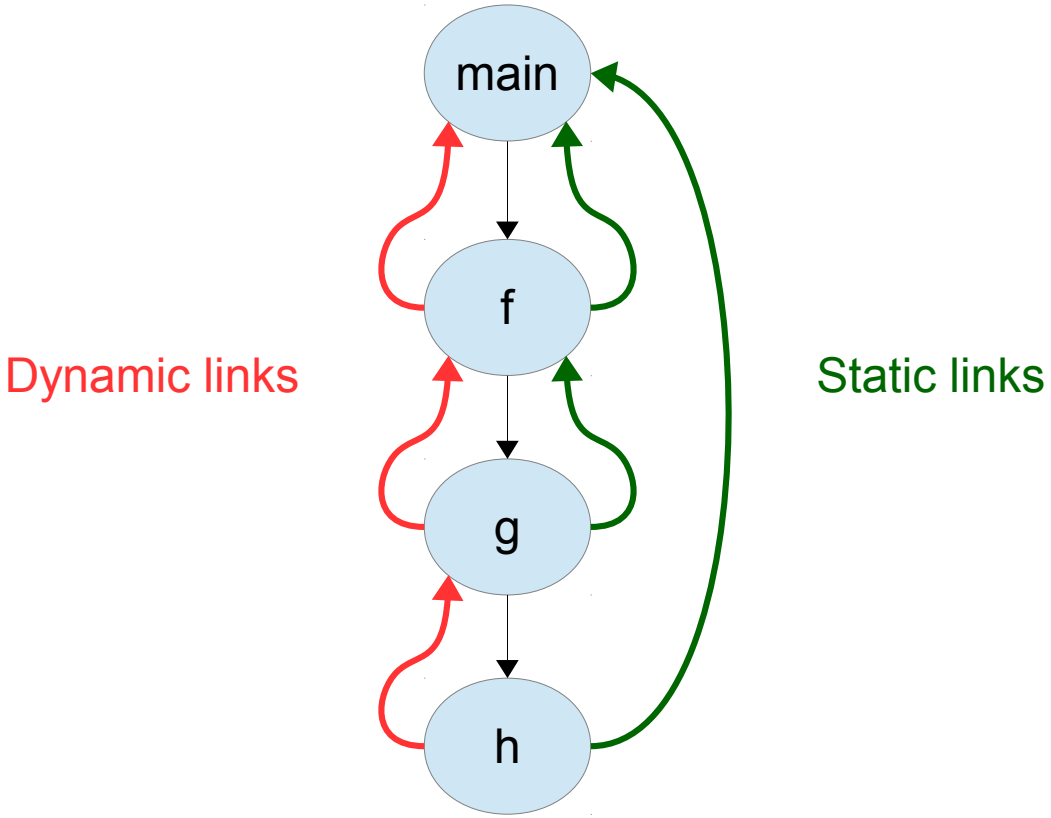


- **Static link**
  - Pour qu'une fonction puisse faire référence aux variables de la fonction où elle a été définie, on utilise un autre pointeur : le static link (ou access link)
  - À chaque appel de fonction, on passera le static link vers la fonction englobante en premier paramètre de la fonction.

Static link

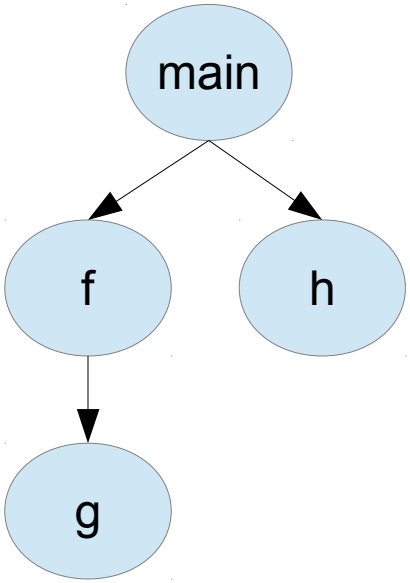


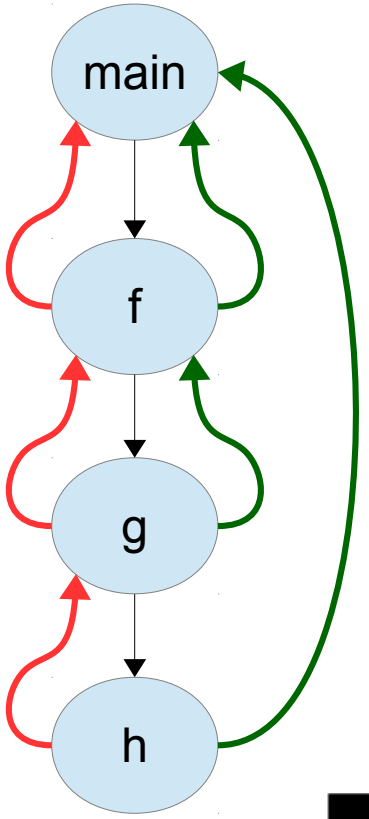
- **Static et dynamic links**



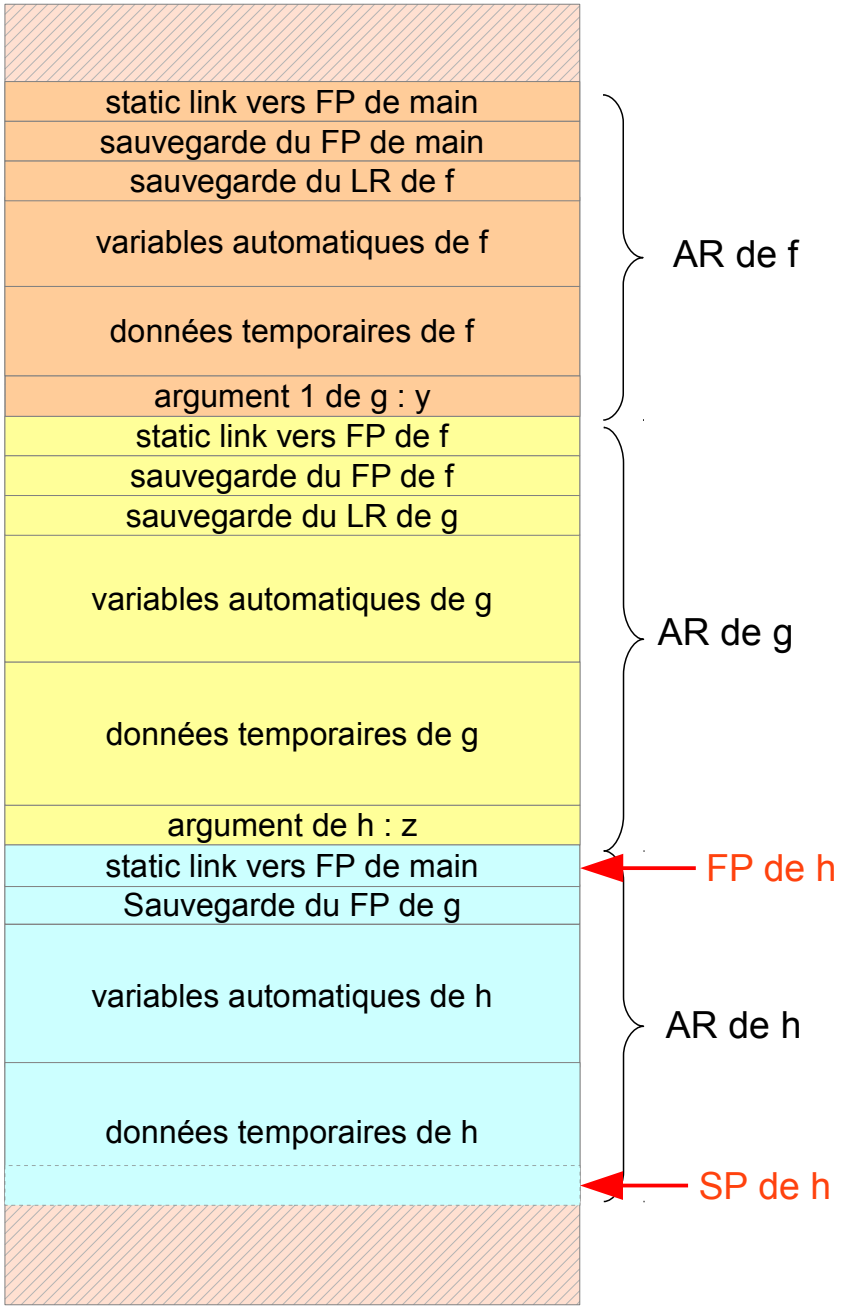
Static link

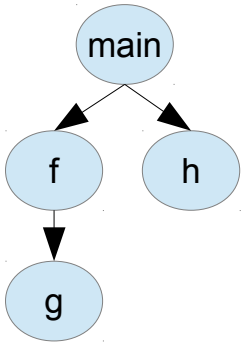
- Il sera stocké sur la pile, à l'endroit où pointe le FP.
  - pour le retrouver, il suffira d'un `LDR Rx, [FP]`
  - pour remonter de plusieurs niveaux, on recommence l'opération (« `[[FP]]` », etc.)
    - exemple : g, qui a besoin d'accéder aux variables de main

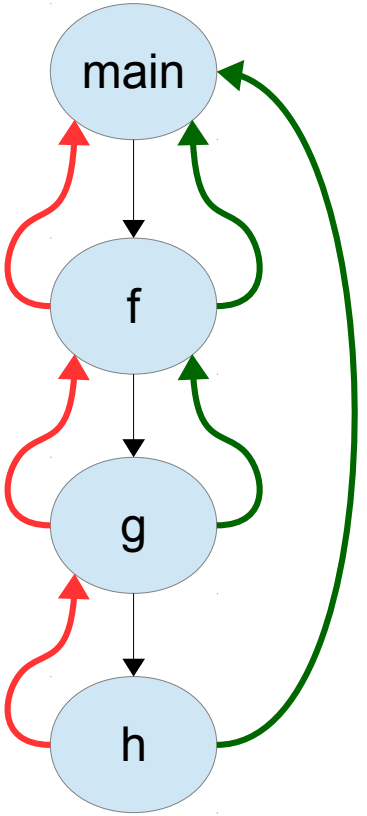




Bilan







Où en est-on ?

- **On a vu :**
  - rôle de la pile
  - structure des activation frames / records
  - rôle du SP / FP
  - dynamic link / static link

- **À venir :**
  - Comment transmettre les arguments ?
  - Comment transmettre la valeur de retour ?

Passage d'arguments

- **Plusieurs scénarios possibles**
  - L'appelant peut créer une structure, y stocker les arguments et passer l'adresse de cette structure (par un registre, qui deviendra important et qu'il faudra sauver en cas de sous-appel)
  - L'appelant connaît l'emplacement du stack frame de l'appelé, et peut stocker les argument juste à côté de cette frame, de façon à ce que l'appelé les trouve (cf. schémas précédents)
  - L'appelant peut stocker des arguments dans des registres, et d'autres sur la pile (scénario maintenant le plus fréquent)

Passage d'arguments

- **Sur ARM**
  - Les 4 premiers arguments sont passés dans R0 à R3
  - Les arguments suivants sont passés sur la pile
  - Les types de 64 bits sont passés sur :
    - R0 et R1
    - R2 et R3
  - Les grosses structures sont passées
    - soit sur la pile,
    - soit un bout par les registres et le reste sur la pile
  - pour les tableaux, on passe l'adresse du premier élément
  - le static link, s'il existe, sera passé dans R0

Passage de la valeur de retour

- **En général**
  - Dans un registre, pour les petites données
  - Pour les structures, il est tentant de les allouer sur la pile et de retourner un pointeur dessus. Pourquoi est-ce une très mauvaise idée ?
  - La solution correcte est que l'appelant alloue lui-même l'espace nécessaire à la valeur de retour, et transmette à l'appelé un pointeur dessus (ARM : dans R0).

Résumé convention ARM

- **En général sur ARM**
  - PC = R15, LR = 14, SP = R13, IP = R12, FP = R11 (pour gcc)
  - R4 à R11 : utilisés pour les variables locales, callee saved
  - R0 à R3 : utilisés pour passer les arguments, caller saved
  - Évidemment :
    - FP et SP sont callee saved
    - LR et IP sont caller saved
  - La valeur de retour est transmise dans R0 (ou R0 + R1)
  - Les procédures ne sauvegardent leur LR que si elles appellent d'autres routines
  - Le static link est passé dans R0
  - Le pointeur de pile doit être aligné sur 8 octets aux frontières d'unités de compilation

Licence de droits d'usage

Contexte académique } sans modification

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après, et à l'exclusion de toute utilisation commerciale.

Le droit d'usage défini par la licence autorise un usage dans un cadre académique, par un utilisateur donnant des cours dans un établissement d'enseignement secondaire ou supérieur et à l'exclusion expresse des formations commerciales et notamment de formation continue. Ce droit comprend :

- le droit de reproduire tout ou partie du document sur support informatique ou papier,
- le droit de diffuser tout ou partie du document à destination des élèves ou étudiants.

Aucune modification du document dans son contenu, sa forme ou sa présentation n'est autorisé.

Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité.

Le droit d'usage défini par la licence est personnel, non exclusif et non transmissible.

Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur :

alexis.polti@telecom-paristech.fr