Compilation : théorie, techniques et outils

Thème 6 : Gestion de la mémoire

Habib ABDULRAB (Institut National des Sciences Appliquées de Rouen)
Claude MOULIN (Université de Technologie de Compiègne)
Sid TOUATI (Université de Versailles Saint-Quentin en Yvelines)

Table des matières

Objectifs

processus d'allocation, processus de substitution. Comment fait un compilateur pour transformer des données haut niveau en données bas niveau (mémoire linéaire)

Lectures conseillées :

- Aho, Sethi et Ullman : chapitre 7
- Compilateurs : chapitres 5 et 6.2

Présentation générale

1

A. Présentation générale

Données et langages

- Les langages haut niveau manipulent des données structurées (tableaux, variables, objets, etc.).
- Le processeur lui ne connaît que des adresses mémoire.
- Le compilateur doit attribuer une place (une adresse mémoire) à toute donnée présente dans le programme source.

Allocation et substitution

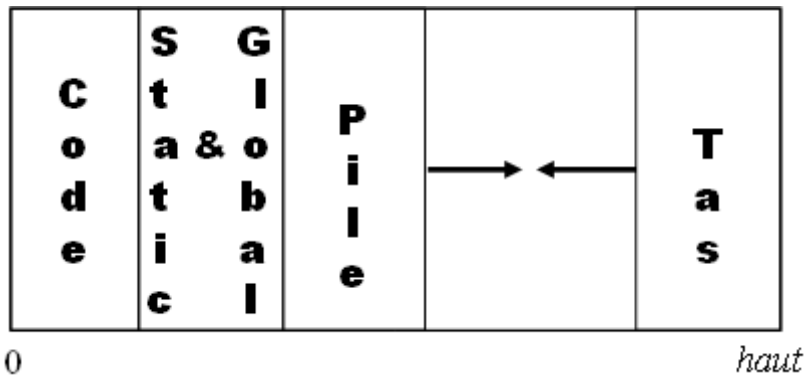
Le processus qui consiste à attribuer une zone mémoire pour chaque donnée du programme source s'appelle le processus d'**allocation** mémoire.

Le compilateur doit également être capable de substituer chaque référence à une variable par son adresse. Cela s'appelle le processus de substitution.

Ex : référence à un élément de tableau, champs de structure, fonction virtuelle, etc.

Schéma classique d'allocation mémoire

Espace logique d'adressage unique du programme (mémoire virtuelle) alloué par l'OS



- Les zones contenant le code, les données statiques et globales ont des tailles connues (par le compilateur par ex).

  - Le programme source les accède implicitement avec des étiquettes (noms).
- Le tas et la pile s'accroissent et rétrécissent durant le temps d'exécution du programme.
- meilleure utilisation de la mémoire si la pile et le tas s'accroissent chacun vers l'autre (Knuth).
- Codes & données séparés ou entrelacés
- on utilise l'espace d'adressage virtuel géré par le proc: on n'alloue pas dynamiquement cet espace.

Complément

Tas : *heap*

Pile : *stack*

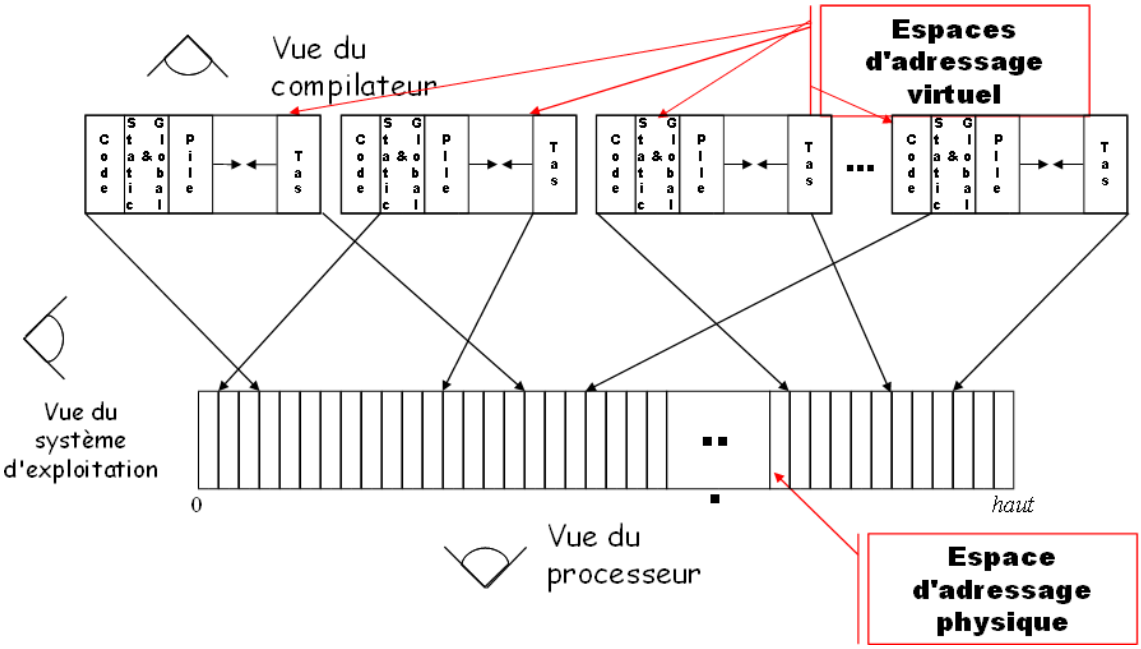
zone « Code » : contient le programme binaire (zone où l'on n'écrit pas par défaut).

zone « statique » : contient les données globales du programmes, i.e., celles qui sont en vie durant tout le temps d'exécution.

zone « tas » : utilisée pour stocker les données allouées dynamiquement.

zone « pile » : passage de paramètres et de contextes lors d'appels de fonctions, variables locales, gestion de la récursivité, etc.

Vue globale de la mémoire



Complément

Un compilateur voit l'unique espace d'adressage virtuel du programme qu'il est en train de compiler.

Le système d'exploitation voit tous les espaces d'adressages des programmes et leur affectation à l'espace physique.

Le processeur lui ne voit que des adresses mémoire : un long tableau uni-dimensionnel.

Représentation de l'information

II

A.Scalaires

En général, un scalaire est traduit par le compilateur en un mot mémoire.

Ex : les données de type *int*, les pointeurs, etc. ont la taille du mot mémoire.

En fonction du langage et de l'architecture, on peut utiliser des demi-mots et doubles (*char, long int*).

B.Ensembles

- Généralement, ils sont de petits intervalles d'entiers.
- Facilement implémentable en un ensemble de bits. Le bit i est à 1 ssi la valeur appartient à l'ensemble.
  - Ex: si on a n éléments, on utilise un mot de n bits.
  - Les opérations sur les ensembles deviennent naturellement des opérations sur les bits.
- Autre méthode : assigner un entier pour chaque élément (langage C).

Complément

L'union des ensembles devient un *ou logique* entre deux mots

L'intersection devient un *et logique*

Le test d'appartenance d'un élément : *et logique* avec *un masque*

C.Tableaux

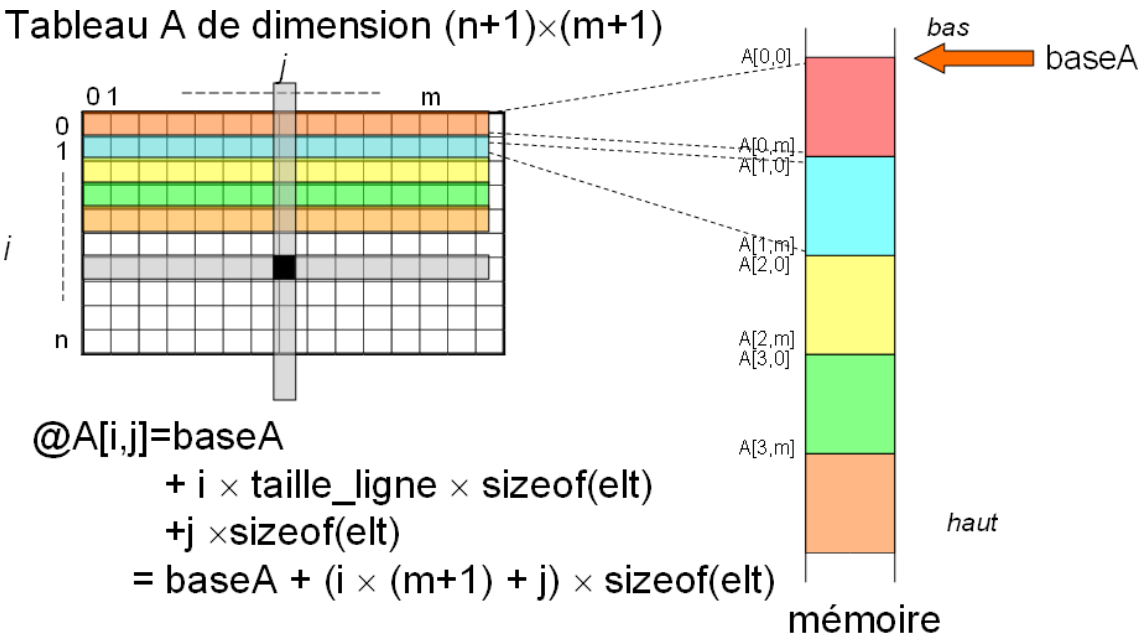
- La mémoire est un tableau à une dimension.

- Les langages autorisent l'emploi de tableaux multidimensionnels.
- Problème : comment linéariser un tableau ? i.e., comment mapper un tableau multidimensionnel en un tableau mono-dimensionnel (la mémoire).

Tableaux bidimensionnels

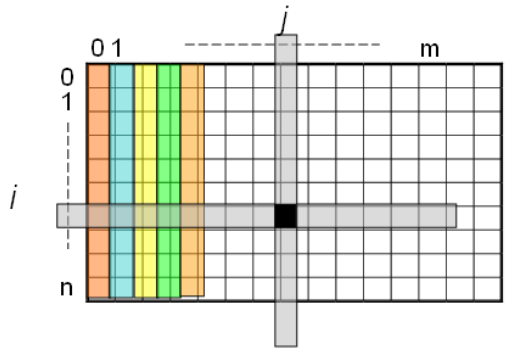
- Par analogie aux matrices (tableaux bidimensionnels), on peut stocker les tableaux à deux dimensions soit:
  - par ligne (langage C) : les éléments d'une ligne de la matrice sont consécutifs en mémoire
  - par colonne (fortran) : les éléments d'une colonne de la matrice sont consécutifs en mémoire

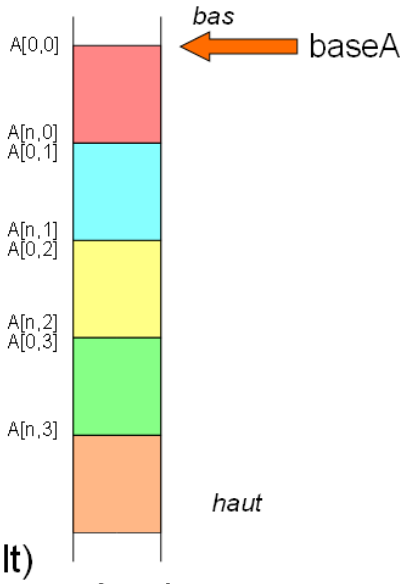
Tableaux bidimensionnels



Tableaux bidimensionnels

Tableau A de dimension (n+1)×(m+1)



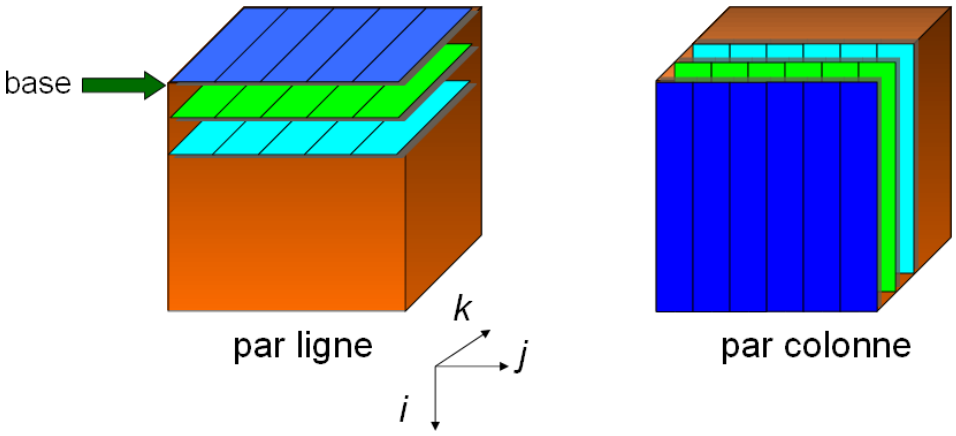


@A[i,j]=baseA
+ j × taille_colonne × sizeof(elt)
+i ×sizeof(elt)
= baseA + (j × (n+1) + i) × sizeof(elt)

Tableaux multidimensionnels

- La dénomination est restée la même : stockage par ligne (*line major*) ou par colonne (*column major*). Soit $A[d_0, ..., d_n]$
- Par ligne : les éléments consécutifs dans la mémoire sont ceux des dimensions les plus à droite :
  - $A[0, ..., 0, 0], A[0, ..., 0, 1], A[0, ..., 0, 2], ...$
- Par colonne : les éléments consécutifs dans la mémoire sont ceux des dimensions les plus à gauche:
  - $A[0, ..., 0, 0], A[1, ..., 0, 0], A[2, ..., 0, 0], ...$

Exemple : un cube



Adressage des éléments

- Soit à calculer l'adresse de $A[i_1, i_2, ..., i_n]$ d'un tableau à n dimensions.
- On suppose L_k le nombre d'éléments de la dimension k.
- Représentation par ligne

$$@A[i_1, i_2, ..., i_n] = \text{baseA} + (i_1 \times L_2 \times ... \times L_n + i_2 \times L_3 \times ... \times L_n + ... + i_n) \times sizeof(elt)$$

- Représentation par colonne

$$@A[i_1, i_2, ..., i_n] = \text{baseA} + (in \times L_1 \times ... \times L_{n-1} + i_{n-1} \times L_1 \times ... \times L_{n-2} + ... + i_1) \times sizeof(elt)$$

Complément

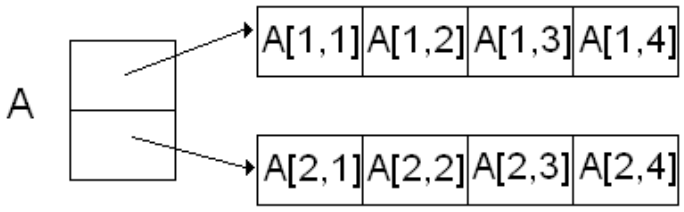
La formule peut être retrouvée récursivement à partir de l'adressage dans un tableau à une dimension, puis à deux dimensions, puis à trois dimensions, etc.

- Le compilateur stocke dans la table des symboles les différentes informations sur les tableaux (taille, dimension, intervalles des indices, etc.).
- Lors d'un accès à un élément d'un tableau dans un programme, c'est le compilateur qui génère le code de calcul d'adresse afin d'accéder à la bonne case mémoire.

Tableaux indirects

- Dans les représentations précédentes, tous les éléments du tableau étaient contigus en mémoire.
- Dans le schéma d'Iliffe, les tableaux sont représentés par des pointeurs indirects
  - Les éléments d'une ligne (ou colonne) sont contigus
  - le tableau multidimensionnel est un vecteur de pointeurs vers pointeurs ...vers pointeurs
  - Représentation non adéquate pour l'analyse de code, consomme plus

d'espace, mais simplifie l'adressage des éléments



Complément

Avec cette représentation, il n'y a pas de calcul d'adresse compliqué. Tout élément est accédé par une suite d'indirections.

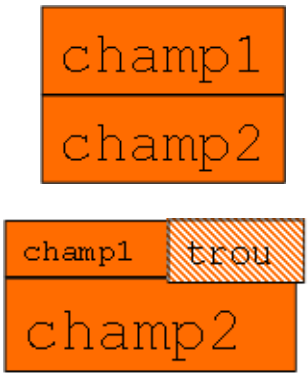
Java utilise les tableaux indirects.

Tableaux dynamiques

- Si les tailles des dimensions ne sont pas connues par le compilateur, ce dernier ne saurait pas la taille de la zone à allouer, ni comment adresser les éléments.
- Le tableau devient alloué à l'exécution. Le compilateur stocke néanmoins un **descripteur** de tableau (une structure), qui a une taille fixe.

D.Structures

```
structure exemple{
    int champ1;
    double champ2;
}
```



L'adressage d'un champ est seulement un déplacement (*offset*) par rapport à l'adresse de début de la structure.

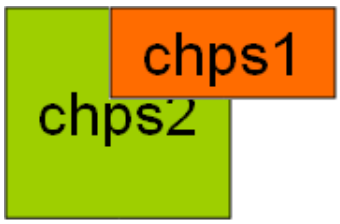
Complément

Souvent, les architectures des processeurs ont des contraintes d'alignement : par ex, les entiers (4octets) doivent être alignés sur des adresses multiples de 4, les doubles à des adresses multiples de 8, etc. Dans notre exemple, si champ1 est aligné sur une frontière de 4 octets, il faut s'assurer que champ2 soit aligné sur 8

octets en insérant éventuellement un trou.

Structure avec variante

```
union{
    int champ1;
    double champ2;
}
```



- L'adresse d'un champ est seulement l'adresse de base de l'union.
- La taille allouée est la taille du plus grand champ.
- L'alignement se fait sur le *ppcm* des alignements des différents champs.

Complément

PPCM : Plus Petit Commun Multiple

E.Objets

- La programmation orientée objet est une bénédiction pour le programmeur, mais une « malédiction » pour les concepteurs des compilateurs.

- Un objet est une entité contenant des méthodes (codes) et des données (attributs).
- On peut considérer les objets comme des structures, mais il faut savoir que certains attributs de l'objet ne sont accessibles que par des méthodes internes à l'objet. Aussi, d'autres méthodes ou attributs ne sont pas accessibles.
- Les données des objets sont représentées en mémoire comme des structures.
- Les méthodes quant à elles sont stockées à part
  - Elles sont partagées par tous les objets de la même classe.
  - Implicitement, le compilateur donne l'adresse de l'objet comme paramètre à la méthode afin qu'elle puisse accéder aux différents champs.
- Le compilateur doit générer des mécanismes pour empêcher l'accès à des attributs ou méthodes interdites.
- Le système d'exploitation et le processeur n'ont aucune *visibilité* à ce sujet. Le compilateur doit se débrouiller seul !
- Ex : bien que le compilateur puisse générer une erreur si un programmeur écrit `objet.x`, comment faire pour empêcher un hacker d'accéder à l'attribut `x` en se servant de l'adresse de `y` comme base et en utilisant un offset.

Allocation statique

III

A. Introduction

Exemple : Où allouer ces données ?

```
int bleu;
float taux;
int A[20,30];
void transfert(int x){
  int *tableau;
  char toto;
  …
  tableau=malloc(20);
  …
}
int main(){
 int bleu;
 float taux;
}
```

sur la zone statique globale

statiquement sur la pile

dynamiquement sur le tas

statiquement sur la pile

Complément

"tableau" est un pointeur de taille fixe.
la zone qu'il pointe (allouée par `malloc`) est dynamique.

B.Allocation statique

L'allocation statique s'occupe des données à taille fixe dans le programme.

Le compilateur analyse les déclarations dans une fonction.

Il peut donc calculer la taille globale de la mémoire requise.

Il alloue le montant nécessaire sur la pile pour les déclarations locales, et sur la zone globale statique dans le cas des déclarations globales.

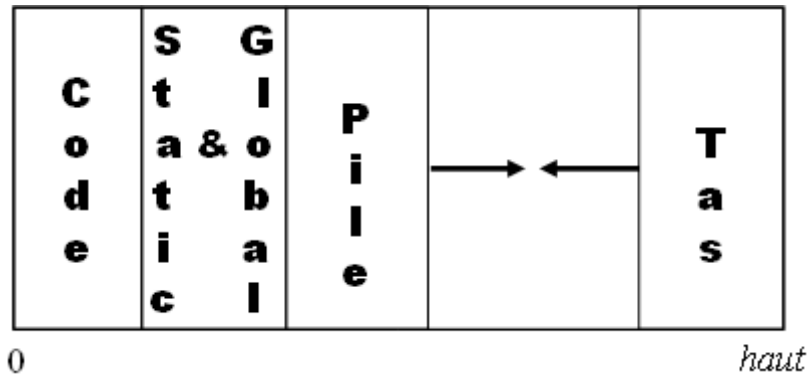
Complément

Rien n'empêche d'allouer les données locales sur le tas. Mais l'utilisation de la pile permet une meilleure gestion de la récursivité.

Rien n'empêche non plus d'utiliser la zone globale statique pour stocker des données locales. Mais cela serait une perte d'espace énorme, car les données locales sont périssables (en vie seulement pendant l'exécution de la fonction).

Schéma classique d'allocation mémoire

Espace logique d'adressage unique du programme (mémoire virtuelle)



Complément

Tas : *heap*

Pile : *stack*

zone « Code » : contient le programme binaire (zone où l'on n'écrit pas par défaut).

zone « statique » : contient les données globales du programmes, i.e., celles qui sont en vie durant tout le temps d'exécution.

zone « tas » : utilisée pour stocker les données allouées dynamiquement.

zone « pile » : passage de paramètres et de contextes lors d'appels de fonctions, variables locales, gestion de la récursivité, etc.

C.Adressage des données statiques

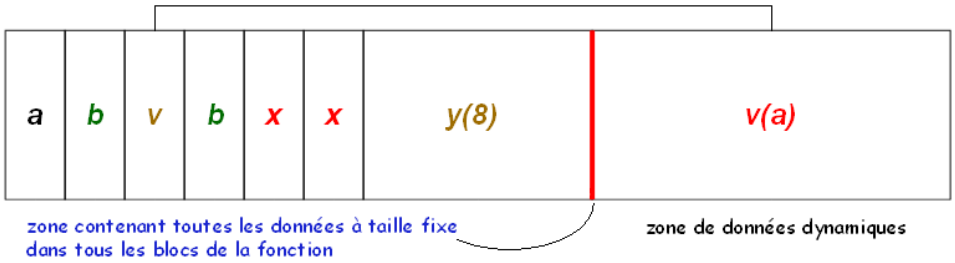
- Si les données sont allouées sur la pile, l'adresse de la variable devient de la forme `base+offset`, où la base est celle de la pile.
- L'offset est assigné par le compilateur (stocké dans la table des symboles).
- Si les données sont allouées sur la zone globale, alors la donnée devient une adresse absolue dans la mémoire virtuelle (une étiquette dans le programme

assembleur).

Exemple : Variables locales

```
B0: {
      int a, b
      a=...
B1:   {
            int v(a), b, x
B2:         {
                  int x, y(8)
                  ....
            }
```

Tableaux
→ Si taille fixe, stocker dans une zone délimitée (pile).
→ Si taille variable, stocker un descripteur ayant une taille fixe, et qui pointe sur une zone allouée dynamiquement.
→ la zone des données dynamiques peut être soit le tas (gérée par le système), soit une zone sur la pile située après les données statiques.

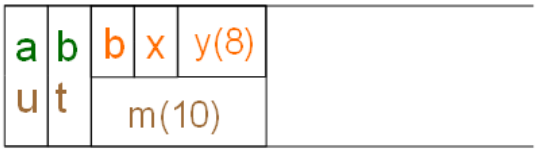


Optimisation de l'allocation statique

- Certaines variables ou données locales ne sont jamais en vie simultanément : elles peuvent partager le même espace.
- C'est le cas quand elles sont déclarées dans deux blocs non imbriqués.
- Le compilateur peut donc leur allouer la même zone mémoire.

```
B0: {
      int a, b
B1:   {
            int  b, x
            int y(8)
                  ....
      }
   }
B2: {
            int u, t,
            int m(10)
   }
```

allocation mémoire « bête »



allocation mémoire optimisée

- Après la sortie du bloc B0, les données en vert n'ont pas besoin de demeurer sur la pile
- Idem pour les données en orange après la sortie du bloc B1
- Idem pour les données en marron après la sortie du bloc B2

Initialisation des variables statiques

Le compilateur génère explicitement le code qui initialise les variables.

Si le compilateur génère du code assembleur, certains assembleurs permettent d'allouer des zones statiques initialisées avec des constantes.

Allocation dynamique

IV

A.Allocation dynamique

- Le système d'exploitation offre des mécanismes d'allocation dynamique.
  - la fonction `malloc` recherche un segment libre (dans le tas) de la taille désirée et le marque comme réservé.
- Du côté du langage : `new` et `free`.
  - Parfois ils sont implicites (entrée et sortie d'un bloc)
- Le compilateur insère des appels de fonctions systèmes pour obtenir la place nécessaire.

B.Gestion de l'espace dynamique

- La libération des espaces dynamiques pose le problème de fragmentation de la mémoire
  - Utilisation d'un ramasse miettes (*garbage collector*) pour compacter les espaces libres.
- Problèmes :
  - à quel moment lancer le ramasse miette ? compromis performance-espace.
  - Est-ce le programme, le compilateur ou l'OS qui doit s'occuper du problème de fragmentation ?
  - comment détecter les zones "zombies" ? les zones allouées mais qui ne sont référencées nulle part ?

Complément

Si on exécute souvent le ramasse miette, le programme de l'utilisateur devient lent.

Si on l'exécute très peu, on risque de fragmenter la mémoire : certains `malloc` peuvent ne pas être satisfaits si des zones mémoire assez larges ne sont pas disponibles.

Fonctions et passage de paramètres

V

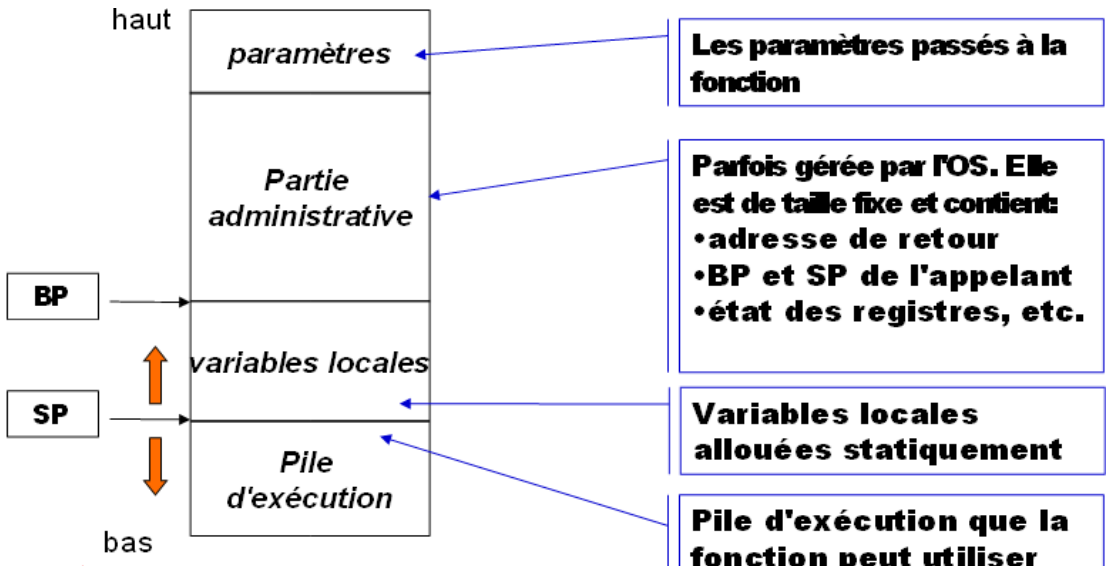
A.Appel de fonction

- Les compilateurs doivent gérer efficacement les appels aux fonctions et sous programmes.
- Un appel à une fonction est une combinaison de quatre points :
  - fournir un environnement de calcul avec un peu de mémoire temporaire et un peu de variables locales
  - passer les paramètres
  - transférer le flot de contrôle, avec (en principe) un retour « garanti »
  - renvoyer une valeur de retour

B.Zone d'activation d'une fonction

- C'est une structure de données stockée sur la pile qui contient le contexte d'exécution
  - valeurs des registres, paramètres, variables locales, adresse de retour,...
- Dans beaucoup de langages, une pile est la meilleure technique d'allocation des zones d'activation.

Exemple



Avantage
@variable locale=BP+offset
@paramètre=BP+taille(partie admin) + offset

Complément

D'autres structures pour la zone d'activation sont possibles, tout dépend du langage et de la machine cible.

BP : *base pointer*

SP : *stack pointer*

Où stocker la zone d'activation ?

- Si l'intervalle de vie d'une zone d'activation est le même que celui de l'appel de fonction, ET si le code exécute un "return" normal
  - Alors : stocker la zone d'activation sur la pile (cas le plus commun)
- Si la fonction est en vie au delà de son appelant, OU si elle envoie un objet qui référence (pointe) sur son état d'exécution
  - Alors: la zone d'activation doit être stockée sur le tas.
- Si une fonction ne fait aucun appel à une autre fonction
  - Alors : la zone d'activation peut être allouée en statique
- L'efficacité du code généré préfère l'allocation statique, ensuite sur pile, en enfin sur le tas.

Passage de paramètres

- Les langages offrent la possibilité de passer des paramètres.
  - Une expression utilisée par l'appelant devient une variable dans le contexte de l'appelé.
- Il existe deux mécanismes communs de passage de paramètres
- *passage par référence* : passer un pointeur sur le paramètre
  - seul un pointeur est empilé dans la zone d'activation
  - effet de bord : plusieurs noms avec la même adresse.
- *passage par valeur* : passer une copie de sa valeur au moment de l'appel
  - une donnée avec une taille potentiellement élevée peut être empilée dans la zone d'activation

- Chaque nom a un seul emplacement, mais plusieurs noms peuvent avoir la même valeur.
- Les tableaux sont généralement passés par référence, non par valeur.

Complément

De toute façon, il y a toujours la possibilité d'utiliser des variables globales pour faire passer des paramètres à une fonction. Mais n'oubliez pas qu'un compilateur doit être efficace. La taille mémoire est un des critères d'efficacité.

C.Protocole d'appel

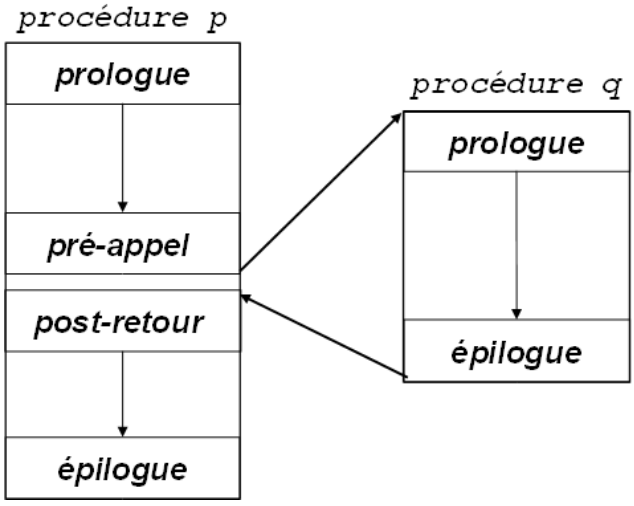
- Lors de la compilation, nous ne pouvons pas toujours inspecter la fonction appelée ou la fonction appelante.
  - plusieurs appels sont possibles dans plusieurs unités de compilations (fichier et/ou codes objets distincts).
  - Le compilateur ne peut pas faire la différence entre un code système et un code d'un utilisateur.
  - Tous les appels de fonctions doivent avoir le même protocole.
- Le compilateur doit utiliser une séquence standardisée d'opérations
  - Partage les responsabilités entre l'appelant et l'appelé.
- Habituellement, c'est en accord avec le système (**interopérabilité**)

Complément

registres *caller-save* : registres que doit sauvegarder l'appelant.

registres *callee-save* : registres que doit sauvegarder l'appelé.

Protocole standard



La procédure a

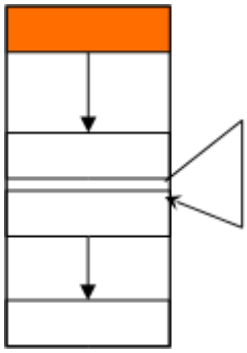
- **un prologue standard**
- **un épilogue standard**

Chaque appel implique

- **une séquence de pré-appel**
- **une séquence de post-retour**

Séquence prologue de l'appelé

- L'appelé q finit de mettre en place son environnement
- L'appelé q met à l'abri les éléments de l'environnement de p susceptibles d'être modifiés.



sequence_prologue

En détails

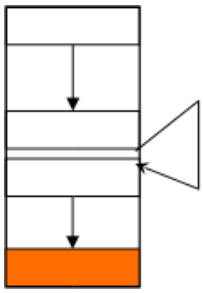
- Sauvegarder les registres *callee-save*
- Allouer de l'espace pour les données locales
  - Le scénario le plus simple est de modifier le SP.
- Effectuer les initialisations de variables locales, si nécessaire

Remarque

Si la ZA est allouée sur le tas, vous pouvez allouer une zone séparée pour les variables locales

Séquence épilogue de l'appelé

- L'appelé q conclut ses tâches
- Commencer à restaurer l'environnement de l'appelant p.



sequence_epilogue

En détails

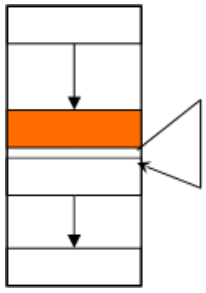
- Sauvegarder la valeur de retour
- Restaurer les registres *callee-save*
- Libérer l'espace alloué pour les données locales, si nécessaire (sur le tas).
- Charger l'adresse de retour de la ZA de l'appelant
- Restaurer les pointeurs de pile de l'appelant p
- Effectuer un saut vers l'adresse de retour

Remarque

Si la ZA est stockée sur la pile, ceci peut ne pas être nécessaire. L'appelé peut remettre le pointeur de sommet de pile à sa valeur de pré-appel

Séquence de pré-appel de l'appelant

- L'appelant p initialise les bases de la zone d'activation de l'appelé q.
- L'appelant p essaye de protéger son propre environnement.



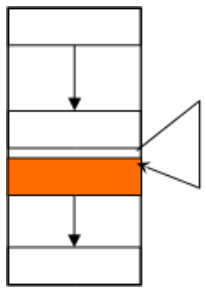
sequence_pre_appel

En détails

- Allouer de l'espace pour l'appelé q
  - **sauf pour les variables locales (seul q les connaît)**
- Calculer chaque paramètre et stocker sa valeur ou son adresse dans la ZA de q.
- L'appelant p sauvegarde l'adresse de retour et les pointeurs de sa pile dans la ZA de q
- Sauvegarder les registres *caller-save* dans la ZA de q.
- Faire un saut vers l'adresse de début de la séquence prologue de l'appelé.

Séquence post-retour de l'appelant

- L'appelant finit de restaurer son environnement
- Affecter chaque valeur à sa place.



sequence_post_retour

En détails

- L'appelant p copie, si nécessaire, la valeur de retour de la ZA de l'appelé q
- Restaurer les registres *caller-save*
- Libérer l'espace alloué à la ZA de l'appelé q.
- Continuer l'exécution juste après l'appel de q.

Conclusion

- Plus les objets sont abstraits, plus la gestion de la mémoire est compliquée.
- Le compilateur doit substituer chaque donnée du programme par un calcul d'adresse mémoire qui soit le plus ressemblant à celui effectué par la machine cible.
- La gestion de la mémoire dans un compilateur doit faire attention à
  - optimiser la taille globale allouée
  - permettre un accès efficace aux données
  - protection et sécurité dans certains langages