Université Savoie Mont Blanc,
UFR SceM
INFO505, L3 INFO

September 3, 2023
J.-O. Lachaud

Fiche 4 — Mécanismes d'allocation mémoire

(pile, tas, allocation dynamique)

Cette fiche explique les mécanismes d'allocation mémoire en C : la *pile*, que vous utilisez déjà, et le *tas* qui permet d'avoir des données persistantes à la durée de vie d'une fonction ou d'un bloc d'instruction.

1 Pile d'exécution

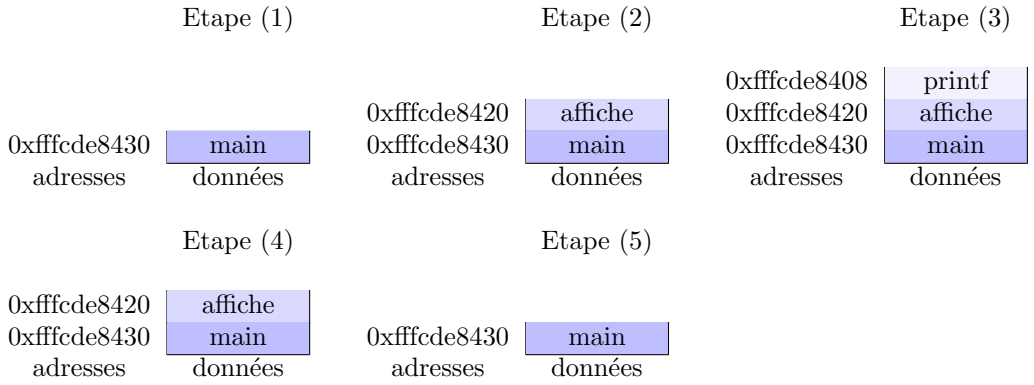
La *pile d'exécution* ou simplement *pile* (*call stack* en anglais) est une structure de données propre à chaque fil d'exécution d'un processus, dont le rôle principal est de garder trace des appels successifs des fonctions.

1.1 Fonctionnement général

Son fonctionnement est celui d'une pile, d'où son nom. Son sommet correspond aux informations nécessaires à l'exécution de la fonction courante: paramètres et variables locales, valeur de retour, adresse de retour dans le code de la fonction appelante. En général, la pile descend en mémoire à chaque appel de fonction, mais ce n'est pas obligatoire.

Appeler une fonction va rajouter les données nécessaires sur le sommet de la pile, de façon à ce que le fil d'exécution puisse exécuter cette fonction. *Quitter* une fonction va supprimer les données sur le sommet de la pile, de façon à revenir à la fonction qui l'avait appelée avant.

```c
void affiche( char* t )
{
  // Etape (2)
  printf( ">>>>>>> %s <<<<<<<\n", t ); // Etape (3)
  // Etape (4)
}
int main( int argc, char* argv[] )
{ // Etape (1)
  affiche( "Bonjour" );
  // Etape (5)
  return 0;
}
```

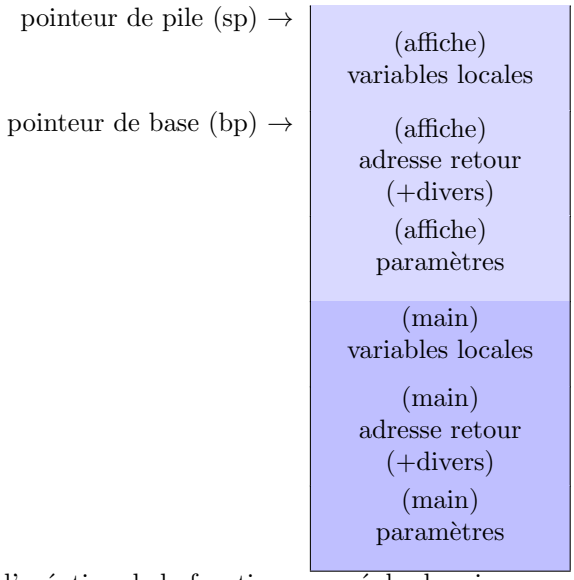


Chaque bloc fonction ou *stack frame*, contient les informations nécessaires à son exécution. Plus précisément, on retrouve:

- les paramètres de la fonction (ce sont des variables)
- l'adresse de retour dans le code assembleur de la fonction appelante (un pointeur)

- les variables locales à la fonction
- et possiblement la valeur de retour, des sauvegardes de contexte, de privilèges, etc.

Leur ordre respectif dépend de l'architecture et de l'OS, mais le diagramme ci-contre montre une possibilité de structuration à l'étape (2) de l'exemple précédent.
On voit que le *pointeur de pile (sp)* pointe le sommet de pile en mémoire (en général l'adresse la plus basse). Dès qu'on a besoin de plus de variables locales, il remonte en mémoire pour mettre les nouvelles variables.
Le *pointeur de base* (*base pointer (bp)* ou *frame pointer (fp)*) pointe la zone mémoire stockant l'adresse de retour. Il est utilisé pour réinitialiser le pointeur de pile, restaurer le contexte et ramener le fil d'exécution à la fonction appelante.



Le pointeur de pile pourrait donc changer pendant l'exéution de la fonction, au gré des besoins en nouvelles variables. Au contraire, le pointeur de base ne change pas pendant l'exécution de la fonction. Une variable locale pourra donc toujours avoir une adresse relative fixe par rapport au pointeur de base.

On note cependant que, dans du code optimisé, le pointeur de base est souvent omis car le compilateur est à même de tout faire avec le pointeur de pile dans la plupart des cas, la taille des variables étant connue à la compilation.

1.2 Suite d'appels imbriqués des fonctions

La pile sert principalement à mémoriser l'état actuel de la suite imbriquée des appels de fonctions du fil d'exécution. Elle le fait en mémorisant simplement l'adresse dans le code assembleur où le fil d'exécution doit revenir. Dans du code assembleur `x86_64`, cela se fait via l'instruction `call` pour appeler une fonction, et l'instruction `ret` qui quitte une fonction. Si on regarde le code assembleur généré par `gcc` on voit les instructions suivantes:

```
_affiche:                               ## @affiche
        pushq   %rbp                    ## sauvegarde le base pointer
        movq    %rsp, %rbp              ## bp = sp
        subq    $16, %rsp               ## déplace la pile de 16 octets
        movq    %rdi, -8(%rbp)          ## pour sauver du contexte
        movq    -8(%rbp), %rsi
        leaq    L_.str(%rip), %rdi      ## 1er paramètre mis dans un registre
        movb    $0, %al
        callq   _printf                 ## met l'adresse de retour _adr dans la pile
                                        ## et saute à printf
_adr:   movl    %eax, -12(%rbp)         ## valeur de retour de printf
        addq    $16, %rsp               ## remonte la pile de 16 octets
        popq    %rbp                    ## récupère le base pointer
        retq                            ## lit l'adresse de retour sur la pile
                                        ## et saute à cette adresse

        .section        __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
        .asciz  ">>>>>> %s <<<<<<\n"
```

Tout cela vous est caché par le compilateur, mais c'est ce mécanisme qui permet d'appeler des fonctions à partir de n'importe quelle fonction ou qui permet de faire des appels récursifs. En effet, rien n'empêche la pile d'exécution de stocker plusieurs fois la même adresse de retour dans le code.

1.3 Allocation statique

Une deuxième fonction essentielle de la pile d'exécution est de s'occuper de l'allocation et la désallocation des variables locales (et donc des paramètres aussi, sauf optimisations particulières du compilateur). On parle d'*allocation statique* ou *automatique*.

En fait chaque variable locale occupe une taille mémoire (que l'on peut connaître via `sizeof`). La zone mémoire de chaque variable locale est donc attribuée sur la pile, par un simple déplacement correspondant du pointeur de pile. Cela explique pourquoi les valeurs initiales des variables sont indéterminées, car les valeurs viennent des données antérieures qui étaient sur la pile.

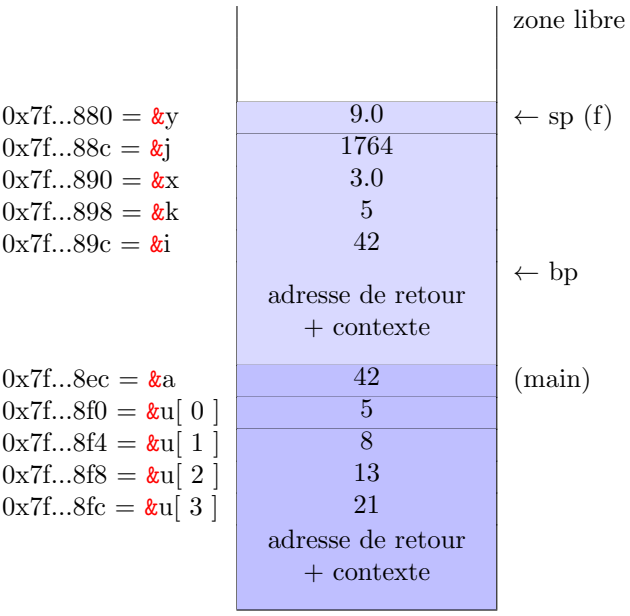
L'exemple suivant a été compilé avec clang sous macos (Darwin kernel 19.6.0).

```
void f( int i, int k, double x )
{
  int    j = i*i;
  double y = x*x;
  printf( "&i    : %p %d\n", &i   , i   );
  printf( "&k    : %p %d\n", &k   , k   );
  printf( "&x    : %p %f\n", &x   , x   );
  printf( "&j    : %p %d\n", &j   , j   );
  printf( "&y    : %p %f\n", &y   , y   );
}

int main( int argc, char* argv[] )
{
  int a = 42;
  int u[ 4 ] = { 5, 8, 13, 21 };
  printf( "&a    : %p %d\n", &a   , a    );
  printf( "&u[0] : %p %d\n", &u[0], u[0] );
  printf( "&u[1] : %p %d\n", &u[1], u[1] );
  printf( "&u[2] : %p %d\n", &u[2], u[2] );
  printf( "&u[3] : %p %d\n", &u[3], u[3] );
  printf( "--- appel  f( a, 5, 3.0 ) -----\n" );
  f( a, 5, 3.0 );
  printf( "--- retour f( a, 5, 3.0 ) -----\n" );
  printf( "&u[0] : %p %d\n", &u[0], u[0] );
  return 0;
}
```

qui s'exécute ainsi:

```
&a    : 0x7ffee0e0e8ec 42
&u[0] : 0x7ffee0e0e8f0 5
&u[1] : 0x7ffee0e0e8f4 8
&u[2] : 0x7ffee0e0e8f8 13
&u[3] : 0x7ffee0e0e8fc 21
--- appel  f( a, 5, 3.0 ) -----
&i    : 0x7ffee0e0e89c 42
&k    : 0x7ffee0e0e898 5
&x    : 0x7ffee0e0e890 3.000000
&j    : 0x7ffee0e0e88c 1764
&y    : 0x7ffee0e0e880 9.000000
--- retour f( a, 5, 3.0 ) -----
&u[0] : 0x7ffee0e0e8f0 5
```



Dans cette architecture, à la fois les variables locales et les paramètres sont placés au-dessus de l'adresse de retour et du contexte. Le principe est le même que décrit précédemment.

Cette manière de mettre les variables dans la pile a les conséquences suivantes:

- allouer l'espace mémoire nécessaire à toutes les variables locales ne prend quasi pas de temps, il s'agit juste de décaler le pointeur de pile de la bonne taille mémoire.

  Par exemple, si vous allouez 3 `int` et 2 `double`, le pointeur de pile sera décalé de $3*4+2*8=28$ octets vers les plus petites adresses.

- désallouer la mémoire attribuée aux variables locales est aussi ultra-rapide: il s'agit juste de décaler le pointeur de pile dans l'autre sens.

  Dans l'exemple ci-dessus, le pointeur de pile sera décalé de $3*4+2*8=28$ octets vers les plus grandes adresses.

- lors d'un appel de fonction, les valeurs qu'attendent les paramètres sont empilés sur la pile d'exécution, exactement à l'endroit mémoire où la fonction attend les paramètres.

- lors d'un appel de fonction interne à la fonction courante, les variables locales ne sont plus accessibles mais les variables sont toujours alloués en mémoire (on a empilé de nouvelles informations sur la pile).

- en revanche, lorsqu'on quitte une fonction, toutes les variables locales et paramètres sont désalloués.

- la seule manière de retourner des données est soit d'utiliser la valeur de retour (mise dans un registre `eax` sous `x86_64`) ou placée sur la pile pour de grosses données, soit d'utiliser le passage par adresse vu précédemment.

- ce type d'allocation est limité par la taille maximale allouée à une pile par le système d'exploitation : on ne peut mettre de grands tableaux comme variables locales, ou avoir des profondeurs de récursivité trop importantes.

2 Allocation dynamique et tas d'exécution

Le mécanisme d'allocation statique présente de nombreux avantages mais, par sa définition même, manque de souplesse à l'exécution. En effet, tout doit être connu dès la compilation. Or la plupart des programmes informatique traitent des quantités de données inconnues à l'étape de compilation. Il faut donc disposer d'un autre mécanisme plus souple lorsque l'allocation statique ne suffit plus. On parle d'*allocation dynamique*.

Note historique: jusqu'en 1990, Fortran ne manipulait que des tableaux de taille fixe. Son domaine de prédilection est le calcul scientifique et la simulation numérique.

2.1 Principes

Le principe choisi en C, qui reste très "bas niveau", est de laisser le programmeur demander explicitement de la mémoire lorsqu'il en a besoin, et de le laisser libérer explicitement cette mémoire lorsqu'il n'en a plus besoin. La mémoire dynamique est prise sur une zone mémoire attribuée au processus, appelé *tas d'exécution* ou simplement *tas* (*heap* en anglais). Par défaut, seul ce processus y a accès.

Pour ce faire, le programmeur dispose des fonctions suivantes définies dans `libc`:

```
/* stdlib.h */
void* malloc ( size_t size);
void* calloc ( size_t nmemb, size_t size );
void  free   ( void* ptr );
void* realloc( void* ptr, size_t size );
```

Le paramètre `size` désigne un nombre d'octets. On voit que les pointeurs sont de type `void*` : c'est donc au programmeur de les caster/transtyper dans le type de pointeur voulu.

- `malloc` : alloue un bloc mémoire sur le tas et retourne son adresse.

- `free` : désalloue le bloc mémoire à l'adresse spécifiée (sur le tas).
- `calloc` : alloue un tableau de `nmemb` éléments de taille spécifiée et l'initialise à 0.
- `realloc` : réalloue un bloc mémoire (en général souhaité plus grand).

Le bout du code suivant fabrique une nouvelle chaîne de caractères qui est la concaténation des deux chaînes données.

```
// return "ccccc....c" (where c repeated n times)
char* make_n_letters( int n, char c )
{
  char* s = (char*) malloc( (n+1) * sizeof( char ) );
  char* q = s + n;
  for ( char* p = s; p != q; p++ )
    *p = c;
  *q = 0;                 // termine la chaîne.
  return s;
}

int main()
{
  char* stars = make_n_letters( 50, '*' );
  printf( "%s\n", stars ); // l'affiche
  free( stars );           // libère la mémoire
}
```

Une règle simple dans un code C : il doit y avoir autant de `malloc+calloc+realloc` que de `free`, sauf si évidemment certaines de ces fonctions sont dans un bloc conditionnel.

2.2 Utilisation de l'allocation dynamique

Les zones mémoires allouées par allocation dynamique persistent même après la fin du bloc où elles ont été allouées. Cela permet de retourner une adresse mémoire d'une zone allouée dynamiquement ou de faire des structures de données avec des données à la fois de taille non prévisible et persistantes.

A titre illustratif, nous allons construire une structure de données pour manipuler les chaînes de caractères un peu à la façon C++ ou JAVA. On stockera donc la longueur du texte, mais aussi l'espace réellement alloué. On veillera à toujours avoir un octet alloué de plus que nécessaire dans notre structure, afin de faciliter l'export vers les chaînes C usuelles.

```
#ifndef __STRING_H__
#define __STRING_H__

struct SString {
  char* data;   // pointer to dyn. alloc. data
  int   length; // effective length of text
  int   alloc;  // allocated space for text (>= length)
};
typedef struct SString String;

// Initialise une chaîne vide
void String_init( String* s );
// Initialise une chaîne à partir d'un texte C
void String_init_with_c_str( String* s, char* t );
// Termine la chaîne donnée
void String_end ( String* s );
// Affecte à la chaîne s la chaîne t (ie "s = t")
void String_assign( String* s, String* other );
// Ajoute à la chaîne s la chaîne t (ie "s += t")
void String_append( String* s, String* other );
// Ajoute à la chaîne s le texte C t (ie "s += t")
void String_append_c_str( String* s, char* t );
// Retourne la longueur effective de la chaine s.
int   String_length( String* s );
// Retourne le texte C  qui correspond à la chaîne s
char* String_c_str( String* s );
// Retourne le caractère en position k dans s, ou 0 si en dehors.
char  String_at( String* s, int k );
// Fonction interne qui garantit que la chaîne s a au moins un buffer
```

```
// de taille size alloué.
void __String_minimal_alloc( String* s, int size );
#endif
```

Comme on ne connaît pas la taille de nos futures chaînes de caractères, on voit que le buffer de données n'est pas un tableau de taille fixe, mais un pointeur vers une zone de données. On pourra ainsi le modifier au gré de nos besoins.

Par exemple, on pourra utiliser nos `String` ainsi:

```
int main( int argc, char* argv[] )
{
  String s1, s2;
  String_init_with_c_str( &s1, "Bonjour" );
  String_init_with_c_str( &s2, "Titi" );
  String_append_c_str( &s1, " Toto et " );
  String_append( &s1, &s2 );
  String_append_c_str( &s1, " !!!" );
  printf( "%s\n", String_c_str( &s1 ) );
  for ( int k = 0; k < 1000; k += 2 ) // avance de 2 en 2
    printf( "%c", String_at( &s1, k ) );
  printf("\n");
  String_end( &s1 );
  String_end( &s2 );
  return 0;
}
```

affichera:

```
chaine 'Bonjour Toto et Titi !!!' de longueur 24
BnorTt tTt !
```

Observons d'abord le code des fonctions d'initialisation.

```
void String_init( String* s )
{
  s->length = 0;
  s->alloc  = 16;
  s->data   = (char*) malloc( s->alloc * sizeof( char ) );
}
void String_init_with_c_str( String* s, char* t )
{
  char* p = t;
  for ( ; *p != 0; ++p ) ; // p moves till the end of C-string t
  s->length = p - t;
  s->alloc  = s->length + 1;
  s->data   = (char*) malloc( s->alloc * sizeof( char ) );
  // copy literal C-string into our structure
  for ( p = s->data; *t != 0; ++p, ++t )
    *p = *t;
}
```

Dans les deux cas, on alloue dynamiquement de la mémoire. En effet, il faut que notre structure "possède" la zone de données où on va stocker les caractères. Autrement on prendrait le risque de ne pouvoir la modifier ou insérer des données. C'est pourquoi on alloue une zone mémoire assez grande (au moins 16 octets, ou au moins la taille de la chaine C) et on sauvegarde cette adresse dans la structure.

Question 1. Dessinez les différents états de la pile et du tas sur le code suivant (avant et pendant et après l'appel de `String_init...`).

```
int main( int argc, char* argv[] )
{
  String s1;
  String_init_with_literal( &s1, "Bonjour" );
  printf( "La longueur de s1 est %d\n", s1->length );
  String_termine( &s1 );
}
```

La concaténation de deux String (ie `s += t`) se ferait via un code similaire au suivant:

```
void String_append( String* s, String* other )
{
  __String_minimal_alloc( s, s->length + other->length + 1 );
  char* p = other->data;
  char* e = other->data + other->length;
  char* q = s->data + s->length;
  while ( p != e ) *q++ = *p++;
  s->length += other->length;
}
void __String_minimal_alloc( String* s, int size )
{
  if ( s->alloc < size )
    {
      // NB: not always optimal
      s->data  = (char*) realloc( s->data, size * sizeof( char ) );
      s->alloc = size;
    }
}
```

Notez l'utilisation d'une fonction auxiliaire qui sert à agrandir le buffer courant à la taille nécessaire et qui est utilisé dans plusieurs autres fonctions. Les fonctions d'accès aux données ou de conversion vers une chaîne sont faciles à écrire. Cela donne :

```
int String_length( String* s )
{
  return s->length;
}
char* String_c_str( String* s )
{
  s->data[ s->length ] = 0;
  return s->data;
}
char  String_at( String* s, int k )
{
  return ( 0 <= k && k < s->length ) ? s->data[ k ] : 0;
}
```

Question 2. Ecrire le code des fonctions restantes.

```
// Affecte à la chaîne s la chaîne t (ie "s = t")
void String_assign( String* s, String* other );
// Ajoute à la chaîne s la chaîne t (ie "s += t")
void String_append_c_str( String* s, char* t );
```

Question 3. Proposez une fonction d'insertion d'une chaîne dans une autre à une position quelconque.

```
// Insert string t at position k in s.
void String_insert( String* s, String* t, int k );
```

- La pile d'exécution est un mécanisme fondamental commun à la plupart des langages compilés qui s'occupe à la fois de la gestion des appels de fonction et de l'allocation mémoire statique des variables et des paramètres, tout cela de façon extrêmement efficace.
- Les valeurs des arguments et les valeurs de retour sont transmises sur la pile (ou comme registres après optimisation du code).
- La pile d'exécution est un mécanisme trop rigide pour traiter des données dont on ne connait pas a priori la masse.
- On utilise alors l'allocation dynamique, qui alloue des zones mémoire quelconque à un autre endroit de la mémoire, le tas du processus, via notamment les fonctions `malloc` et `free`.
- L'utilisateur doit gérer lui-même quand il veut allouer cette mémoire, où il va stocker les informations associées, et quand il a fini et veut désallouer cette mémoire.