Déclarations

Durant les chapitres précédents, nous avons principalement utilisé des variables locales. Celles-ci sont déclarées à l'intérieur des fonctions où elles sont utilisées. La façon dont les variables sont déclarées est importante dans un programme écrit en langage C. Dans cette section nous nous concentrerons sur des programmes C qui sont écrits sous la forme d'un seul fichier source. Nous verrons plus tard comment découper un programme en plusieurs modules qui sont répartis dans des fichiers différents et comment les variables peuvent y être déclarées.

La première notion importante concernant la déclaration des variables est leur **portée**. La portée d'une variable peut être définie comme étant la partie du programme où la variable est accessible et où sa valeur peut être modifiée. Le langage C définit deux types de portée à l'intérieur d'un fichier C. La première est la **portée globale**. Une variable qui est définie en dehors de toute définition de fonction a une portée globale. Une telle variable est accessible dans toutes les fonctions présentes dans le fichier. La variable `g` dans l'exemple ci-dessous a une portée globale.

```
float g;   // variable globale

int f(int i) {
int n;   // variable locale
// ...
for(int j=0;j<n;j++) {  // variable locale
  // ...
  }
//...
for(int j=0;j<n;j++) {  // variable locale
  // ...
  }
}
```

Dans un fichier donné, il ne peut évidemment pas y avoir deux variables globales qui ont le même identifiant. Lorsqu'une variable est définie dans un *bloc*, la portée de cette variable est locale à ce bloc. On parle dans ce cas de **portée locale**. La variable locale n'existe pas avant le début du bloc et n'existe plus à la fin du bloc. Contrairement aux identifiants de variables globales qui doivent être uniques à l'intérieur d'un fichier, il est possible d'avoir plusieurs variables locales qui ont le même identifiant à l'intérieur d'un fichier. C'est fréquent notamment pour les définitions d'arguments de fonction et les variables de boucles. Dans l'exemple ci-dessus, les variables `n` et `j` ont une portée locale. La variable `j` est définie dans deux blocs différents à l'intérieur de la fonction `f`.

Le programme **`/C/S3-src/portee.c`** illustre la façon dont le compilateur C gère la portée de différentes variables.

```


int g1;
int g2=1;

void f(int i) {
  int loc;    //def1a
  int loc2=2; //def2a
  int g2=-i*i;
  g1++;

  printf("[f-%da] \t\t %d \t %d \t %d \t %d\n",i,g1,g2,loc,loc2);
  loc=i*i;
  g1++;
  g2++;
  printf("[f-%db] \t\t %d \t %d \t %d \t %d\n",i,g1,g2,loc,loc2);

}


int main(int argc, char *argv[]) {
  int loc; //def1b
  int loc2=1; //def2b

  printf("Valeurs de : \t g1 \t g2\t loc\t loc2\n");
  printf("=================================================\n");

  printf("[main1] \t %d \t %d \t %d \t %d\n",g1,g2,loc,loc2);

  loc=1252;
  loc2=1234;
  g1=g1+1;
  g1=g1+2;

  printf("[main2] \t %d \t %d \t %d \t %d\n",g1,g2,loc,loc2);

  for(int i=1;i<3;i++) {
    int loc=i; //def1c
    int g2=-i;
    loc++;
    g1=g1*2;
    f(i);
    printf("[main-for-%d] \t %d \t %d \t %d \t %d\n",i,g1,g2,loc,loc2);
```

```
  }
  f(7);
  g1=g1*3;
  g2=g2+2;
  printf("[main3] \t %d \t %d \t %d \t %d\n",g1,g2,loc,loc2);

  return(EXIT_SUCCESS);
}
```

Ce programme contient deux variables qui ont une portée globale : `g1` et `g2`. Ces deux variables sont définies en dehors de tout bloc. En pratique, elles sont généralement déclarées au début du fichier, même si le compilateur C accepte un définition en dehors de tout bloc et donc par exemple en fin de fichier. La variable globale `g1` n'est définie qu'une seule fois. Par contre, la variable `g2` est définie avec une portée globale et est redéfinie à l'intérieur de la fonction `f` ainsi que dans la boucle `for` de la fonction `main`. Redéfinir une variable globale de cette façon n'est pas du tout une bonne pratique, mais cela peut arriver lorsque par mégarde on importe un fichier header qui contient une définition de variable globale. Dans ce cas, le compilateur C utilise la variable qui est définie dans le bloc le plus proche. Pour la variable `g2`, c'est donc la variable locale `g2` qui est utilisée à l'intérieur de la boucle `for` ou de la fonction `f`.

Lorsqu'un identifiant de variable locale est utilisé à plusieurs endroits dans un fichier, c'est la définition la plus proche qui est utilisée. L'exécution du programme ci-dessus illustre cette utilisation des variables globales et locales.

```
Valeurs de :     g1      g2      loc     loc2
=================================================
[main1]          0       1       0       1
[main2]          3       1       1252    1234
[f-1a]           7       -1      0       2
[f-1b]           8       0       1       2
[main-for-1]     8       -1      2       1234
[f-2a]           17      -4      0       2
[f-2b]           18      -3      4       2
[main-for-2]     18      -2      3       1234
[f-7a]           19      -49     0       2
[f-7b]           20      -48     49      2
[main3]          60      3       1252    1234
```

Note

Utilisation des variables

En pratique, les variables globales doivent être utilisées de façon parcimonieuse et il faut limiter leur utilisation aux données qui doivent être partagées par plusieurs fonctions à l'intérieur d'un programme. Lorsqu'une variable globale a été définie, il est préférable de ne pas réutiliser son identifiant pour une variable locale. Au niveau des variables locales, les premières versions du langage C imposaient leur définition au début des blocs. Les standards récents **[C99]** autorisent la déclaration de variables juste avant leur première utilisation un peu comme en Java.

Les versions récentes de C **[C99]** permettent également de définir des variables dont la valeur sera constante durant toute l'exécution du programme. Ces déclarations de ces constants sont préfixées par le mot-clé `const` qui joue le même rôle que le mot clé `final` en Java.

```
// extrait de <math.h>
#define M_PI   3.14159265358979323846264338327950288;

const double pi=3.14159265358979323846264338327950288;

const struct fraction {
  int num;
  int denom;
}  demi={1,2};
```

Il y a deux façons de définir des constantes dans les versions récentes de C **[C99]**. La première est via la macro `#define` du préprocesseur. Cette macro permet de remplacer une chaîne de caractères (par exemple `M_PI` qui provient de **math.h**) par un nombre ou une autre chaîne de caractères. Ce remplacement s'effectue avant la compilation. Dans le cas de `M_PI` ci-dessus, le préprocesseur remplace toute les occurrences de cette chaîne de caractères par la valeur numérique de π. Lorsqu'une variable `const` est utilisée, la situation est un peu différente. Le préprocesseur n'intervient pas. Par contre, le compilateur réserve une zone mémoire pour la variable qui a été définie comme constante. Cela a deux avantages par rapport à l'utilisation de `#define`. Premièrement, il est possible de définir comme constante n'importe quel type de données en C, y compris des structures ou des pointeurs alors qu'avec un `#define` on ne peut définir que des nombres ou des chaînes de caractères. Ensuite, comme une `const` est stockée en mémoire, il est possible d'obtenir son adresse et de l'examiner via un **debugger**.

Unions et énumérations

Les structures que nous avons présentées précédemment permettent de combiner plusieurs données de types primitifs différents entre elles. Outre ces structures (`struct`), le langage C supporte également les `enum` et les `union`. Le mot-clé `enum` est utilisé pour définir un type énuméré, c'est-à-dire un type de donnée qui permet de stocker un nombre fixe de valeurs. Quelques exemples classiques sont repris dans le fragment de programme ci-dessous :

```
// les jours de la semaine
typedef enum  {
      monday, tuesday, wednesday, thursday, friday, saturday, sunday
```

```
  } day;

// jeu de carte
typedef enum {
  coeur, carreau, trefle, pique
} carte;

// bits
typedef enum {
        BITRESET = 0,
        BITSET = 1
} bit_t;
```

Le premier `enum` permet de définir le type de données `day` qui contient une valeur énumérée pour chaque jour de la semaine. L'utilisation d'un type énuméré rend le code plus lisible que simplement l'utilisation de constantes définies via le préprocesseur.

```
  bit_t bit=BITRESET;
  day jour=monday;
  if(jour==saturday||jour==sunday)
    printf("Congé\n");
```

En pratique, lors de la définition d'un type énuméré, le compilateur C associe une valeur entière à chacune des valeurs énumérées. Une variable permettant de stocker la valeur d'un type énuméré occupe la même zone mémoire qu'un entier.

Outre les structures, le langage C supporte également les unions. Alors qu'une structure permet de stocker plusieurs données dans une même zone mémoire, une `union` permet de réserver une zone mémoire pour stocker une données parmi plusieurs types possibles. Une `union` est parfois utilisée pour minimiser la quantité de mémoire utilisée pour une structure de données qui peut contenir des données de plusieurs types. Pour bien comprendre la différence entre une `union` et une `struct`, considérons l'exemple ci-dessous.

donnée

```
struct s_t {
  int i;
  char c;
} s;

union u_t {
  int i;
  char c;
} u;
```

Une union, `u` et une structure, `s` sont déclarées dans ce fragment de programme.

```
  // initialisation
  s.i=1252;
  s.c='A';
  u.i=1252;
  // u contient un int
  u.c='Z';
  // u contient maintenant un char (et u.i est perdu)
```

La structure `s` peut contenir à la fois un entier et un caractère. Par contre, l'`union` `u`, peut elle contenir un entier (`u.i`) *ou* un caractère (`u.c`), mais jamais les deux en même temps. Le compilateur C alloue la taille pour l'`union` de façon à ce qu'elle puisse contenir le type de donnée se trouvant dans l'`union` nécessitant le plus de mémoire. Si les unions sont utiles dans certains cas très particulier, il faut faire très attention à leur utilisation. Lorsqu'une `union` est utilisée, le compilateur C fait encore moins de vérifications sur les types de données et le code ci-dessous est considéré comme valide par le compilateur :

```
  u.i=1252;
  printf("char : %c\n", u.c);
```

Lors de son exécution, la zone mémoire correspondant à l'union `u` sera simplement interprétée comme contenant un `char`, même si on vient d'y stocker un entier. En pratique, lorsqu'une `union` est vraiment nécessaire pour des raisons d'économie de mémoire, on préférera la placer dans une `struct` en utilisant un type énuméré qui permet de spécifier le type de données qui est présent dans l'`union`.

```
typedef enum { INTEGER, CHAR } Type;

typedef struct
{
  Type type;
  union {
  int i;
  char c;
```

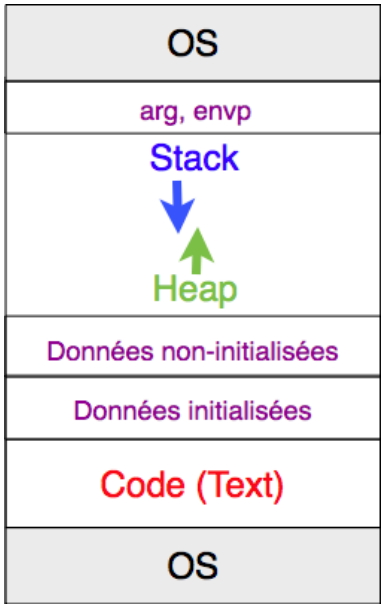
```
  } x;
} Value;
```

Le programmeur pourra alors utiliser cette structure en indiquant explicitement le type de données qui y est actuellement stocké comme suit.

```
Value v;
v.type=INTEGER;
v.x.i=1252;
```

Organisation de la mémoire

Lors de l'exécution d'un programme en mémoire, le système d'exploitation charge depuis le système de fichier le programme en langage machine et le place à un endroit convenu en mémoire. Lorsqu'un programme s'exécute sur un système Unix, la mémoire peut être vue comme étant divisée en six zones principales. Ces zones sont représentées schématiquement dans la figure ci-dessous.



Organisation d'un programme Linux en mémoire

La figure ci-dessus présente une vision schématique de la façon dont un processus Linux est organisé en mémoire centrale. Il y a d'abord une partie de la mémoire qui est réservée au système d'exploitation (OS dans la figure). Cette zone est représentée en grisé dans la figure.

Le segment text

La première zone est appelée par convention le **segment text**. Cette zone se situe dans la partie basse de la mémoire [3]. C'est dans cette zone que sont stockées toutes les instructions qui sont exécutées par le micro-processeur. Elle est généralement considérée par le système d'exploitation comme étant uniquement accessible en lecture. Si un programme tente de modifier son **segment text**, il sera immédiatement interrompu par le système d'exploitation. C'est dans le segment text que l'on retrouvera les instructions de langage machine correspondant aux fonctions de calcul et d'affichage du programme. Nous en reparlerons lorsque nous présenterons le fonctionnement du langage d'assemblage.

Le segment des données initialisées

La deuxième zone, baptisée **segment des données initialisées**, contient l'ensemble des données et chaînes de caractères qui sont utilisées dans le programme. Ce segment contient deux types de données. Tout d'abord, il comprend l'ensemble des variables globales explicitement initialisées par le programme (dans le cas contraire, elles sont initialisées à zéro par le compilateur et appartiennent alors au **segment des données non-initialisées**). Ensuite, les constantes et les chaînes de caractères utilisées par le programme.

```
#define MSG_LEN 10
int g;   // initialisé par le compilateur
int g_init=1252;
const int un=1;
int tab[3]={1,2,3};
int array[10000];
char cours[]="SINF1252";
char msg[MSG_LEN]; // initialisé par le compilateur

int main(int argc, char *argv[]) {
```

```
  int i;
  printf("g est à l'adresse %p et initialisée à %d\n",&g,g);
  printf("msg est à l'adresse %p contient les caractères :",msg);
  for(i=0;i<MSG_LEN;i++)
    printf(" %x",msg[i]);
  printf("\n");
  printf("Cours est à l'adresse %p et contient : %s\n",&cours,cours);
  return(EXIT_SUCCESS);
}
```

Dans le programme ci-dessus, la variable `g_init`, la constante `un` et les tableaux `tab` et `cours` sont dans la zone réservée aux variables initialisées. En pratique, leur valeur d'initialisation sera chargée depuis le fichier exécutable lors de son chargement en mémoire. Il en va de même pour toutes les chaînes de caractères qui sont utilisées comme arguments aux appels à `printf(3)`.

L'exécution de ce programme produit la sortie standard suivante.

```
g est à l'adresse 0x60aeac et initialisée à 0
msg est à l'adresse 0x60aea0 contient les caractères : 0 0 0 0 0 0 0 0 0 0
Cours est à l'adresse 0x601220 et contient : SINF1252
```

Cette sortie illustre bien les adresses où les variables globales sont stockées. La variable globale `msg` fait notamment partie du **segment des données non-initialisées**.

Le segment des données non-initialisées

La troisième zone est le **segment des données non-initialisées**, réservée aux variables non-initialisées. Cette zone mémoire est initialisée à zéro par le système d'exploitation lors du démarrage du programme. Dans l'exemple ci-dessus, c'est dans cette zone que l'on stockera les valeurs de la variable `g` et des tableaux `array` et `msg`.

Note

Initialisation des variables

Un point important auquel tout programmeur C doit faire attention est l'initialisation correcte de l'ensemble des variables utilisées dans un programme. Le compilateur C est nettement plus permissif qu'un compilateur Java et il autorisera l'utilisation de variables avant qu'elles n'aient été explicitement initialisées, ce qui peut donner lieu à des erreurs parfois très difficiles à corriger.

En C, par défaut les variables globales qui ne sont pas explicitement initialisées dans un programme sont initialisées à la valeur zéro par le compilateur. Plus précisément, la zone mémoire qui correspond à chaque variable globale non-explicitement initialisée contiendra des bits valant 0. Pour les variables locales, le langage C n'impose aucune initialisation par défaut au compilateur. Par souci de performance et sachant qu'un programmeur ne devrait jamais utiliser de variable locale non explicitement initialisée, le compilateur C n'initialise pas par défaut la valeur de ces variables. Cela peut avoir des conséquences ennuyeuses comme le montre l'exemple ci-dessous.

```
#define ARRAY_SIZE 1000

// initialise un tableau local
void init(void) {
  long a[ARRAY_SIZE];
  for(int i=0;i<ARRAY_SIZE;i++) {
    a[i]=i;
  }
}

// retourne la somme des éléments
// d'un tableau local
long read(void) {
  long b[ARRAY_SIZE];
  long sum=0;
  for(int i=0;i<ARRAY_SIZE;i++) {
    sum+=b[i];
  }
  return sum;
}
```

Cet extrait de programme contient deux fonctions erronées. La seconde, baptisée `read(void)` déclare un tableau local et retourne la somme des éléments de ce tableau sans l'initialiser. En Java, une telle utilisation d'un tableau non-initialisé serait détectée par le compilateur. En C, elle est malheureusement valide (mais fortement découragée évidemment). La première fonction, `init(void)` se contente d'initialiser un tableau local mais ne retourne aucun résultat. Cette fonction ne sert a priori à rien puisqu'elle n'a aucun effet sur les variables globales et ne retourne aucun résultat. L'exécution de ces fonctions via le fragment de code ci-dessous donne cependant un résultat interpellant.

```
  printf("Résultat de read() avant init(): %ld\n",read());
  init();
  printf("Résultat de read() après init() : %ld\n",read());
```

```
Résultat de read() avant init(): 7392321044987950589
Résultat de read() après init() : 499500
```

Le tas (ou *heap*)

La quatrième zone de la mémoire est le **tas** (ou **heap** en anglais). Vu l'importance pratique de la terminologie anglaise, c'est celle-ci que nous utiliserons dans le cadre de ce document. C'est une des deux zones dans laquelle un programme peut obtenir de la mémoire supplémentaire pour stocker de l'information. Un programme peut y réserver une zone permettant de stocker des données et y associer un pointeur.

Le système d'exploitation mémorise, pour chaque processus en cours d'exécution, la limite supérieure de son **heap**. Le système d'exploitation permet à un processus de modifier la taille de son heap via les appels systèmes **brk(2)** et **sbrk(2)**. Malheureusement, ces deux appels systèmes se contentent de modifier la limite supérieure du **heap** sans fournir une API permettant au processus d'y allouer efficacement des blocs de mémoire. Rares sont les processus qui utilisent directement **brk(2)** si ce n'est sous la forme d'un appel à `sbrk(0)` de façon à connaître la limite supérieure actuelle du **heap**.

En C, la plupart des processus allouent et libèrent de la mémoire en utilisant les fonctions **malloc(3)** et **free(3)** qui font partie de la librairie standard.

La fonction **malloc(3)** prend comme argument la taille (en bytes) de la zone mémoire à allouer. La signature de la fonction **malloc(3)** demande que cette taille soit de type `size_t`, c'est-à-dire le type retourné par l'expression `sizeof`. Il est important de toujours utiliser `sizeof` lors du calcul de la taille d'une zone mémoire à allouer. **malloc(3)** retourne normalement un pointeur de type `(void *)`. Ce type de pointeur est le type par défaut pour représenter dans un programme C une zone mémoire qui ne pointe pas vers un type de données particulier. En pratique, un programme va généralement utiliser **malloc(3)** pour allouer de la mémoire pour stocker différents types de données et le pointeur retourné par **malloc(3)** sera *casté* dans un pointeur du bon type.

Note

`typecast` en langage C

Comme le langage Java, le langage C supporte des conversions implicites et explicites entre les différents types de données. Ces conversions sont possibles entre les types primitifs et les pointeurs. Nous les rencontrerons régulièrement, par exemple lorsqu'il faut récupérer un pointeur alloué par **malloc(3)** ou le résultat de `sizeof`. Contrairement au compilateur Java, le compilateur C n'émet pas toujours de message de **warning** lors de l'utilisation de typecast qui risque d'engendrer une perte de précision. Ce problème est illustré par l'exemple suivant avec les nombres.

```
int i=1;
float f=1e20;
double d=1e100;

printf("i [int]: %d, [float]:%f, [double]:%f\n",i,(float)i,(double)i);
printf("f [int]: %d, [float]:%f, [double]:%f\n",(int)f,f,(double)f);
printf("d [int]: %d, [float]:%f, [double]:%f\n",(int)d,(float)d,d);
printf("sizeof -> int:%d float:%d double:%d\n",(int)sizeof(int), (int)sizeof(float), (int)sizeof(double));
```

La fonction de la librairie **free(3)** est le pendant de **malloc(3)**. Elle permet de libérer la mémoire qui a été allouée par **malloc(3)**. Elle prend comme argument un pointeur dont la valeur a été initialisée par **malloc(3)** et libère la zone mémoire qui avait été allouée par **malloc(3)** pour ce pointeur. La valeur du pointeur n'est pas modifiée, mais après libération de la mémoire il n'est évidemment plus possible **[1]** d'accéder aux données qui étaient stockées dans cette zone.

Le programme ci-dessous illustre l'utilisation de **malloc(3)** et **free(3)**.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>


typedef struct fraction {
  int num;
  int den;
} Fraction;

void error(char *msg) {
  fprintf(stderr,"Erreur :%s\n",msg);
  exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
  int size=1;
  if(argc==2)
```

```
    size=atoi(argv[1]);

  char * string;
  printf("Valeur du pointeur string avant malloc : %p\n",string);
  string=(char *) malloc((size+1)*sizeof(char));
  if(string==NULL)
    error("malloc(string)");

  printf("Valeur du pointeur string après malloc : %p\n",string);
  int *vector;
  vector=(int *)malloc(size*sizeof(int));
  if(vector==NULL)
    error("malloc(vector)");

  Fraction * fract_vect;
  fract_vect=(Fraction *) malloc(size*sizeof(Fraction));
  if(fract_vect==NULL)
    error("malloc(fract_vect)");

  free(string);
  printf("Valeur du pointeur string après free   : %p\n",string);
  string=NULL;
  free(vector);
  vector=NULL;
  free(fract_vect);
  fract_vect=NULL;

  return(EXIT_SUCCESS);
}
```

Ce programme alloue trois zones mémoires. Le pointeur vers la première est sauvé dans le pointeur `string`. Elle est destinée à contenir une chaîne de `size` caractères (avec un caractère supplémentaire pour stocker le caractère `\0` de fin de chaîne). Il y a deux points à remarquer concernant cette allocation. Tout d'abord, le pointeur retourné par **malloc(3)** est "casté" en un `char *`. Cela indique au compilateur que `string` va bien contenir un pointeur vers une chaîne de caractères. Cette conversion explicite rend le programme plus clair. Ensuite, la valeur de retour de **malloc(3)** est systématiquement testée. **malloc(3)** peut en effet retourner `NULL` lorsque la mémoire est remplie. Cela a peu de chance d'arriver dans un programme de test tel que celui-ci, mais tester les valeurs de retour des fonctions de la librairie est une bonne habitude à prendre lorsque l'on programme sous Unix. Le second pointeur, `vector` pointe vers une zone destiné à contenir un tableau d'entiers. Le dernier pointeur, `fract_vect` pointe vers une zone qui pourra stocker un tableau de `Fraction`. Lors de son exécution, le programme affiche la sortie suivante.

```
Valeur du pointeur string avant malloc : 0x7fff5fbfe1d8
Valeur du pointeur string après malloc : 0x100100080
Valeur du pointeur string après free   : 0x100100080
```

Dans cette sortie, on remarque que l'appel à fonction **free(3)** libère la zone mémoire, mais ne modifie pas la valeur du pointeur correspondant. Le programmeur doit explicitement remettre le pointeur d'une zone mémoire libérée à `NULL`.

Un autre exemple d'utilisation de **malloc(3)** est la fonction `duplicate` ci-dessous qui permet de retourner une copie d'une chaîne de caractères. Il est important de noter qu'en C la fonction **strlen(3)** retourne la longueur de la chaîne de caractères passée en argument sans prendre en compte le caractère `\0` qui marque sa fin. C'est la raison pour laquelle **malloc(3)** doit réserver un bloc de mémoire en plus. Même si généralement les `char` occupent un octet en mémoire, il est préférable d'utiliser explicitement `sizeof(char)` lors du calcul de l'espace mémoire nécessaire pour un type de données.

```
#include <string.h>

char *duplicate(char * str) {
  int i;
  size_t len=strlen(str);
  char *ptr=(char *)malloc(sizeof(char)*(len+1));
  if(ptr!=NULL) {
    for(i=0;i<len+1;i++) {
        *(ptr+i)=*(str+i);
      }
  }
  return ptr;
}
```

malloc(3) et **free(3)** sont fréquemment utilisés dans des programmes qui manipulent des structures de données dont la taille varie dans le temps. C'est le cas pour les différents sortes de listes chaînées, les piles, les queues, les arbres, ... L'exemple ci-dessous (**`/C/S3-src/stack.c`**) illustre une implémentation d'une pile simple en C. Le pointeur vers le sommet de la pile est défini comme une variable globale. Chaque élément de la pile est représenté comme un pointeur vers une structure qui contient un pointeur vers la donnée stockée (dans cet exemple des fractions) et l'élément suivant sur la pile. Les fonctions `push` et `pop` permettent respectivement d'ajouter un élément et de retirer un élément au sommet de la pile. La fonction `push` alloue la mémoire nécessaire avec **malloc(3)** tandis que la fonction `pop` utilise **free(3)** pour libérer la mémoire dès qu'un élément est retiré.

```
typedef struct node_t
{
  struct fraction_t *data;
```

```
  struct node_t *next;
} node;

struct node_t *stack; // sommet de la pile

// ajoute un élément au sommet de la pile
void push(struct fraction_t *f)
{
  struct node_t *n;
  n=(struct node_t *)malloc(sizeof(struct node_t));
  if(n==NULL)
    exit(EXIT_FAILURE);
  n->data = f;
  n->next = stack;
  stack = n;
}
// retire l'élément au sommet de la pile
struct fraction_t * pop()
{
  if(stack==NULL)
    return NULL;
  // else
  struct fraction_t *r;
  struct node_t *removed=stack;
  r=stack->data;
  stack=stack->next;
  free(removed);
  return (r);
}
```

Ces fonctions peuvent être utilisées pour empiler et dépiler des fractions sur une pile comme dans l'exemple ci-dessous. La fonction `display` permet d'afficher sur **stdout** le contenu de la pile.

```
// affiche le contenu de la pile
void display()
{
  struct node_t *t;
  t = stack;
  while(t!=NULL) {
    if(t->data!=NULL) {
      printf("Item at addr %p  :  Fraction %d/%d   Next %p\n",t,t->data->num,t->data->den,t->next);
    }
    else {
      printf("Bas du stack %p\n",t);
    }
    t=t->next;
  }
}

// exemple
int main(int argc, char *argv[]) {

  struct fraction_t demi={1,2};
  struct fraction_t tiers={1,3};
  struct fraction_t quart={1,4};
  struct fraction_t zero={0,1};

  // initialisation
  stack = (struct node_t *)malloc(sizeof(struct node_t));
  stack->next=NULL;
  stack->data=NULL;

  display();
  push(&zero);
  display();
  push(&demi);
  push(&tiers);
  push(&quart);
  display();

  struct fraction_t *f=pop();
  if(f!=NULL)
    printf("Popped : %d/%d\n",f->num,f->den);

  return(EXIT_SUCCESS);
}
```

Lors de son exécution le programme **`/C/S3-src/stack.c`** présenté ci-dessus affiche les lignes suivantes sur sa sortie standard.

```
Bas du stack 0x100100080
Item at addr 0x100100090  :  Fraction 0/1   Next 0x100100080
Bas du stack 0x100100080
Item at addr 0x1001000c0  :  Fraction 1/4   Next 0x1001000b0
Item at addr 0x1001000b0  :  Fraction 1/3   Next 0x1001000a0
Item at addr 0x1001000a0  :  Fraction 1/2   Next 0x100100090
Item at addr 0x100100090  :  Fraction 0/1   Next 0x100100080
Bas du stack 0x100100080
Popped : 1/4
```

Le tas (ou **heap**) joue un rôle très important dans les programmes C. Les données qui sont stockées dans cette zone de la mémoire sont accessibles depuis toute fonction qui possède un pointeur vers la zone correspondante

Note

Ne comptez jamais sur les **free(3)** implicites

Un programmeur débutant qui expérimente avec **malloc(3)** pourrait écrire le code ci-dessous et conclure que comme celui-ci s'exécute correctement, il n'est pas nécessaire d'utiliser **free(3)**. Lors de l'exécution d'un programme, le système d'exploitation réserve de la mémoire pour les différents segments du programme et ajuste si nécessaire cette allocation durant l'exécution du programme. Lorsque le programme se termine, via `return` dans la fonction `main` ou par un appel explicite à **exit(2)**, le système d'exploitation libère tous les segments utilisés par le programme, le text, les données, le tas et la pile. Cela implique que le système d'exploitation effectue un appel implicite à **free(3)** à la terminaison d'un programme.

```c
#define LEN 1024
int main(int argc, char *argv[]) {

  char *str=(char *) malloc(sizeof(char)*LEN);
  for(int i=0;i<LEN-1;i++) {
    *(str+i)='A';
  }
  *(str+LEN)='\0';  // fin de chaîne
  return(EXIT_SUCCESS);
}
```

Un programmeur ne doit cependant *jamais* compter sur cet appel implicite à **free(3)**. Ne pas libérer la mémoire lorsqu'elle n'est plus utilisée est un problème courant qui est généralement baptisé **memory leak**. Ce problème est particulièrement gênant pour les processus tels que les serveurs Internet qui ne se terminent pas ou des processus qui s'exécutent longtemps. Une petite erreur de programmation peut causer un **memory leak** qui peut après quelque temps consommer une grande partie de l'espace mémoire inutilement. Il est important d'être bien attentif à l'utilisation correcte de **malloc(3)** et de **free(3)** pour toutes les opérations d'allocation et de libération de la mémoire.

malloc(3) est la fonction d'allocation de mémoire la plus fréquemment utilisée **[5]**. La librairie standard contient cependant d'autres fonctions permettant d'allouer de la mémoire mais aussi de modifier des allocations antérieures. **calloc(3)** est nettement moins utilisée que **malloc(3)**. Elle a pourtant un avantage majeur par rapport à **malloc(3)** puisqu'elle initialise à zéro la zone de mémoire allouée. **malloc(3)** se contente d'allouer la zone de mémoire mais n'effectue aucune initialisation. Cela permet à **malloc(3)** d'être plus rapide, mais le programmeur ne doit jamais oublier qu'il ne peut pas utiliser **malloc(3)** sans initialiser la zone mémoire allouée. Cela peut s'observer en pratique avec le programme ci-dessous. Il alloue une zone mémoire pour `v1`, l'initialise puis la libère. Ensuite, le programme alloue une nouvelle zone mémoire pour `v2` et y retrouve les valeurs qu'il avait stocké pour `v1` précédemment. En pratique, n'importe quelle valeur pourrait se trouver dans la zone retournée par *malloc(3)*.

```c
#define LEN 1024

int main(int argc, char *argv[]) {

  int *v1;
  int *v2;
  int sum=0;
  v1=(int *)malloc(sizeof(int)*LEN);
  for(int i=0;i<LEN;i++) {
    sum+=*(v1+i);
    *(v1+i)=1252;
  }
  printf("Somme des éléments de v1 : %d\n", sum);
  sum=0;
  free(v1);
  v2=(int *)malloc(sizeof(int)*LEN);
  for(int i=0;i<LEN;i++) {
    sum+=*(v2+i);
  }

  printf("Somme des éléments de v2 : %d\n", sum);
  free(v2);
```

```
  return(EXIT_SUCCESS);
}
```

L'exécution du programme ci-dessus affiche le résultat suivant sur la sortie standard. Ceci illustre bien que la fonction **malloc(3)** n'initialise pas les zones de mémoire qu'elle alloue.

```
Somme des éléments de v1 : 0
Somme des éléments de v2 : 1282048
```

Lors de l'exécution du programme, on remarque que la première zone mémoire retournée par **malloc(3)** a été initialisée à zéro. C'est souvent le cas en pratique pour des raisons de sécurité, mais ce serait une erreur de faire cette hypothèse dans un programme. Si la zone de mémoire doit être initialisée, la mémoire doit être allouée par **calloc(3)** ou via une initialisation explicite ou en utilisant des fonctions telles que **bzero(3)** ou **memset(3)**.

Les arguments et variables d'environnement

Lorsque le système d'exploitation charge un programme Unix en mémoire, il initialise dans le haut de la mémoire une zone qui contient deux types de variables. Cette zone contient tout d'abord les arguments qui ont été passés via la ligne de commande. Le système d'exploitation met dans `argc` le nombre d'arguments et place dans `char *argv[]` tous les arguments passés avec dans `argv[0]` le nom du programme qui est exécuté.

Cette zone contient également les variables d'environnement. Ces variables sont généralement relatives à la configuration du système. Leurs valeurs sont définies par l'administrateur système ou l'utilisateur. De nombreuses variables d'environnement sont utilisées dans les systèmes Unix. Elles servent à modifier le comportement de certains programmes. Une liste exhaustive de toutes les variables d'environnement est impossible, mais en voici quelques unes qui sont utiles en pratique **[6]**:

- `HOSTNAME` : le nom de la machine sur laquelle le programme s'exécute. Ce nom est fixé par l'administrateur système via la commande **hostname(1)**
- `SHELL` : l'interpréteur de commande utilisé par défaut pour l'utilisateur courant. Cet interpréteur est lancé par le système au démarrage d'une session de l'utilisateur. Il est stocké dans le fichier des mots de passe et peut être modifié par l'utilisateur via la commande **passwd(1)**
- `USER` : le nom de l'utilisateur courant. Sous Unix, chaque utilisateur est identifié par un numéro d'utilisateur et un nom uniques. Ces identifiants sont fixés par l'administrateur système via la commande **passwd(1)**
- `HOME`: le répertoire d'accueil de l'utilisateur courant. Ce répertoire d'accueil appartient à l'utilisateur. C'est dans ce répertoire qu'il peut stocker tous ses fichiers.
- `PRINTER` : le nom de l'imprimante par défaut qui est utilisée par la commande **lp(1posix)**
- `PATH`: cette variable d'environnement contient la liste ordonnée des répertoires que le système parcourt pour trouver un programme à exécuter. Cette liste contient généralement les répertoires dans lesquels le système stocke les exécutables standards, comme `/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:` ainsi que des répertoires relatifs à des programmes spécialisés comme `/usr/lib/mozart/bin:/opt/python3/bin`. L'utilisateur peut ajouter des répertoires à son `PATH` avec **bash(1)** en incluant par exemple la commande `PATH=$PATH:$HOME/local/bin:.` dans son fichier `.profile`. Cette commande ajoute au `PATH` par défaut le répertoire `$HOME/local/bin` et le répertoire courant. Par convention, Unix utilise le caractère `.` pour représenter ce répertoire courant.

La librairie standard contient plusieurs fonctions qui permettent de manipuler les variables d'environnement d'un processus. La fonction **getenv(3)** permet de récupérer la valeur associée à une variable d'environnement. La fonction **unsetenv(3)** permet de supprimer une variable de l'environnement du programme courant. La fonction **setenv(3)** permet elle de modifier la valeur d'une variable d'environnement. Cette fonction alloue de la mémoire pour stocker la nouvelle variable d'environnement et peut échouer si il n'y a pas assez de mémoire disponible pour stocker de nouvelles variables d'environnement. Ces fonctions sont utilisées notamment par l'interpréteur de commande mais parfois par des programmes dont le comportement dépend de la valeur de certaines variables d'environnement. Par exemple, la commande **man(1)** utilise différentes variables d'environnement pour déterminer par exemple où les pages de manuel sont stockées et la langue (variable `LANG`) dans laquelle il faut afficher les pages de manuel.

Le programme ci-dessous illustre brièvement l'utilisation de **getenv(3)**, **unsetenv(3)** et **setenv(3)**. Outre ces fonctions, il existe également **clearenv(3)** qui permet d'effacer complètement toutes les variables d'environnement du programme courant et **putenv(3)** qui était utilisé avant **setenv(3)**.

```c
#include <stdio.h>
#include <stdlib.h>

// affiche la valeur de la variable d'environnement var
void print_var(char *var) {
  char *val=getenv(var);
  if(val!=NULL)
    printf("La variable %s a la valeur : %s\n",var,val);
  else
    printf("La variable %s n'a pas été assignée\n",var);
}

int main(int argc, char *argv[]) {

  char *old_path=getenv("PATH");

  print_var("PATH");

  if(unsetenv("PATH")!=0) {
    fprintf(stderr,"Erreur unsetenv\n");
    exit(EXIT_FAILURE);
  }
```

```
  print_var("PATH");

  if(setenv("PATH",old_path,1)!=0) {
    fprintf(stderr,"Erreur setenv\n");
    exit(EXIT_FAILURE);
  }

  print_var("PATH");

  return(EXIT_SUCCESS);
}
```

La pile (ou stack)

La **pile** ou **stack** en anglais est la dernière zone de mémoire utilisée par un processus. C'est une zone très importante car c'est dans cette zone que le processus va stocker l'ensemble des variables locales mais également les valeurs de retour de toutes les fonctions qui sont appelées. Cette zone est gérée comme une pile, d'où son nom. Pour comprendre son fonctionnement, nous utiliserons le programme **`/C/S3-src/fact.c`** qui permet de calculer une factorielle de façon récursive.

```
// retourne i*j
int times(int i, int j) {
  int m;
  m=i*j;
  printf("\t[times(%d,%d)] : return(%d)\n",i,j,m);
  return m;
}
// calcul récursif de factorielle
// n>0
int fact(int n) {
  printf("[fact(%d)]: Valeur de n:%d, adresse: %p\n",n,n,&n);
  int f;
  if(n==1) {
    printf("[fact(%d)]: return(1)\n",n);
    return(n);
  }
  printf("[fact(%d)]: appel à fact(%d)\n",n,n-1);
  f=fact(n-1);
  printf("[fact(%d)]: calcul de times(%d,%d)\n",n,n,f);
  f=times(n,f);
  printf("[fact(%d)]: return(%d)\n",n,f);
  return(f);
}

void compute() {
  int nombre=3;
  int f;
  printf("La fonction fact est à l'adresse : %p\n",fact);
  printf("La fonction times est à l'adresse : %p\n",times);
  printf("La variable nombre vaut %d et est à l'adresse %p\n",nombre,&nombre);
  f=fact(nombre);
  printf("La factorielle de %d vaut %d\n",nombre,f);
}
```

Lors de l'exécution de la fonction `compute()`, le programme ci-dessus produit la sortie suivante.

```
La fonction fact est à l'adresse : 0x100000a0f
La fonction times est à l'adresse : 0x1000009d8
La variable nombre vaut 3 et est à l'adresse 0x7fff5fbfe1ac
[fact(3)]: Valeur de n:3, adresse: 0x7fff5fbfe17c
[fact(3)]: appel à fact(2)
[fact(2)]: Valeur de n:2, adresse: 0x7fff5fbfe14c
[fact(2)]: appel à fact(1)
[fact(1)]: Valeur de n:1, adresse: 0x7fff5fbfe11c
[fact(1)]: return(1)
[fact(2)]: calcul de times(2,1)
        [times(2,1)] : return(2)
[fact(2)]: return(2)
[fact(3)]: calcul de times(3,2)
        [times(3,2)] : return(6)
[fact(3)]: return(6)
La factorielle de 3 vaut 6
```

Il est intéressant d'analyser en détails ce calcul récursif de la factorielle car il illustre bien le fonctionnement du stack et son utilisation.

Tout d'abord, il faut noter que les fonctions `fact` et `times` se trouvent, comme toutes les fonctions définies dans le programme, à l'intérieur du **segment text**. La variable `nombre` quant à elle se trouve sur la pile en haut de la mémoire. Il s'agit d'une variable locale qui est allouée lors de l'exécution de la fonction `compute`. Il en va de même des arguments qui sont passés aux fonctions. Ceux-ci sont également stockés sur la pile. C'est le cas par exemple de l'argument `n` de la fonction

`fact`. Lors de l'exécution de l'appel à `fact(3)`, la valeur `3` est stockée sur la pile pour permettre à la fonction `fact` d'y accéder. Ces accès sont relatifs au sommet de la pile comme nous aurons l'occasion de le voir dans la présentation du langage d'assemblage. Le premier appel récursif se fait en calculant la valeur de l'argument (`2`) et en appelant la fonction. L'argument est placé sur la pile, mais à une autre adresse que celle utilisée pour `fact(3)`. Durant son exécution, la fonction `fact(2)` accède à ses variables locales sur la pile sans interférer avec les variables locales de l'exécution de `fact(3)` qui attend le résultat de `fact(2)`. Lorsque `fact(2)` fait l'appel récursif, la valeur de son argument (`1`) est placée sur la pile et l'exécution de `fact(1)` démarre. Celle-ci a comme environnement d'exécution le sommet de la pile qui contient la valeur `1` comme argument et la fonction retourne la valeur `1` à l'exécution de `fact(2)` qui l'avait lancée. Dès la fin de `fact(1)`, `fact(2)` reprend son exécution où elle avait été interrompue et applique la fonction `times` avec `2` et `1` comme arguments. Ces deux arguments sont placés sur le pile et `times` peut y accéder au début de son exécution pour calculer la valeur `2` et retourner le résultat à la fonction qui l'a appelé, c'est-à-dire `fact(2)`. Cette dernière retrouve son environnement d'exécution sur la pile. Elle peut maintenant retourner son résultat à la fonction `fact(3)` qui l'avait appelée. Celle-ci va appeler la fonction `times` avec `3` et `2` comme arguments et finira par retourner la valeur `6`.

La pile joue un rôle essentiel lors de l'exécution de programmes en C puisque toutes les variables locales, y compris celles de la fonction `main` y sont stockées. Comme nous le verrons lorsque nous aborderons le langage assembleur, la pile sert aussi à stocker l'adresse de retour des fonctions. C'est ce qui permet à l'exécution de `fact(2)` de se poursuivre correctement après avoir récupéré la valeur calculée par l'appel à `fact(1)`. L'utilisation de la pile pour stocker les variables locales et les arguments de fonctions a une conséquence importante. Lorsqu'une variable est définie comme argument ou localement à une fonction `f`, elle n'est accessible que durant l'exécution de la fonction `f`. Avant l'exécution de `f` cette variable n'existe pas et si `f` appelle la fonction `g`, la variable définie dans `f` n'est plus accessible à partir de la fonction `g`.

En outre, comme le langage C utilise le passage par valeur, les valeurs des arguments d'une fonction sont copiés sur la pile avant de démarrer l'exécution de cette fonction. Lorsque la fonction prend comme argument un entier, cette copie prend un temps très faible. Par contre, lorsque la fonction prend comme argument une ou plusieurs structures de grand taille, celles-ci doivent être entièrement copiées sur la pile. A titre d'exemple, le programme ci-dessous définit une très grande structure contenant un entier et une zone permettant de stocker un million de caractères. Lors de l'appel à la fonction `sum`, les structures `one` et `two` sont entièrement copiées sur la pile. Comme chaque structure occupe plus d'un million d'octets, cela prend plusieurs centaines de microsecondes. Cette copie est nécessaire pour respecter le passage par valeur des structures à la fonction `sum`. Celle-ci ne peut pas modifier le contenu des structures qui lui sont passées en argument. Par comparaison, lors de l'appel à `sumptr`, seules les adresses de ces deux structures sont copiées sur la pile. Un appel à `sumptr` prend moins d'une microseconde, mais bien entendu la fonction `sumptr` a accès via les pointeurs passés en argument à toute la zone de mémoire qui leur est associée.

```
#define MILLION 1000000

struct large_t {
  int i;
  char str[MILLION];
};

int sum(struct large_t s1, struct large_t s2) {
  return (s1.i+s2.i);
}

int sumptr(struct large_t *s1, struct large_t *s2) {
  return (s1->i+s2->i);
}

int main(int argc, char *argv[]) {
  struct timeval tvStart, tvEnd;
  int err;
  int n;
  struct large_t one={1,"one"};
  struct large_t two={1,"two"};

  n=sum(one,two);
  n=sumptr(&one,&two);
```

Certaines variantes de Unix et certains compilateurs permettent l'allocation de mémoire sur la pile via la fonction **alloca(3)**. Contrairement à la mémoire allouée par **malloc(3)** qui doit être explicitement libérée en utilisant **free(3)**, la mémoire allouée par **alloca(3)** est libérée automatiquement à la fin de l'exécution de la fonction dans laquelle la mémoire a été allouée. Cette façon d'allouer de la mémoire sur la pile n'est pas portable et il est préférable de n'allouer de la mémoire que sur le tas en utilisant **malloc(3)**.

Les versions récentes du C et notamment **[C99]** permettent d'allouer de façon dynamique un tableau sur la pile. Cette fonctionnalité peut être utile dans certains cas, mais elle peut aussi être la source de nombreuses erreurs et difficultés. Pour bien comprendre ce problème, considérons à nouveau la fonction `duplicate` qui a été définie précédemment en utilisant **malloc(3)** et des pointeurs.

```
#include <string.h>

char *duplicate(char * str) {
  int i;
  size_t len=strlen(str);
  char *ptr=(char *)malloc(sizeof(char)*(len+1));
  if(ptr!=NULL) {
    for(i=0;i<len+1;i++) {
        *(ptr+i)=*(str+i);
      }
  }
```

```
  return ptr;
}
```

Un étudiant pourrait vouloir éviter d'utiliser **malloc(3)** et écrire plutôt la fonction suivante.

```
char *duplicate2(char * str) {
  int i;
  size_t len=strlen(str);
  char str2[len+1];
  for(i=0;i<len+1;i++) {
    str2[i]=*(str+i);
  }
  return str2;
}
```

Lors de la compilation, **gcc(1)** affiche le **warning** `In function ‘duplicate2’: warning: function returns address of local variable`. Ce warning indique que la ligne `return str2;` retourne l'adresse d'une variable locale qui n'est plus accessible à la fin de la fonction `duplicate2`. En effet, l'utilisation de tableaux alloués dynamiquement sur la pile est équivalent à une utilisation implicite de **alloca(3)**. La déclaration `char str2[len];` est équivalente à `char *str2 = (char *)alloca(len*sizeof(char));` et la zone mémoire allouée sur la pile pour `str2` est libérée lors de l'exécution de `return str2;` puisque toute mémoire allouée sur la pile est implicitement libérée à la fin de l'exécution de la fonction durant laquelle elle a été allouée. Donc, une fonction qui appelle `duplicate2` ne peut pas récupérer les données se trouvant dans la zone mémoire qui a été allouée par `duplicate2`.

Footnotes

[1] Pour des raisons de performance, le compilateur C ne génère pas de code permettant de vérifier automatiquement qu'un accès via un pointeur pointe vers une zone de mémoire qui est libre. Il est donc parfois possible d'accéder à une zone mémoire qui a été libérée, mais le programme n'a aucune garantie sur la valeur qu'il y trouvera. Ce genre d'accès à des zones mémoires libérées doit bien entendu être complètement proscrit.

[2] Sur de nombreuses variantes de Unix, cette limite à la taille du stack dépend du matériel utilisé et peut être configurée par l'administrateur système. Un processus peut connaître la taille maximale de son stack en utilisant l'appel système **getrlimit(2)**. L'administrateur système peut modifier ces limites via l'appel système **setrlimit(2)**. La commande `ulimit` de **bash(1)** permet également de manipuler ces limites.

[3] Dans de nombreuses variantes de Unix, il est possible de connaître le sommet du segment **text** d'un processus grâce à la variable *etext*. Cette variable, de type `char` est initialisée par le système au chargement du processus. Elle doit être déclarée comme variable de type `extern char etext` et son adresse (`&etext`) correspond au sommet du segment text.

[4] Nous verrons ultérieurement que grâce à l'utilisation de la mémoire virtuelle, il est possible pour un processus d'utiliser des zones de mémoire qui ne sont pas contiguës.

[5] Il existe différentes alternatives à l'utilisation de **malloc(3)** pour l'allocation de mémoire comme **Hoard** ou **gperftools**

[6] Il possible de lister les définitions actuelles des variables d'environnement via la commande **printenv(1)**. Les interpréteurs de commande tels que **bash(1)** permettent de facilement modifier les valeurs de ces variables. La plupart d'entre elles sont initialisées par le système ou via les fichiers qui sont chargés automatiquement au démarrage de l'interpréteur comme `/etc/profile` qui contient les variables fixées par l'administrateur système ou le fichier `.profile` du répertoire d'accueil de l'utilisateur qui contient les variables d'environnement propres à cet utilisateur.