Langage C avancé
Allocation dynamique de mémoire

Samuel KOKH

samuel.kokh@cea.fr

MACS 1 – Institut Galilée

Insuffisances de la gestion mémoire du stack en C (iso 89)

Problème à résoudre

On considère des vecteurs de `double` de taille fixe `VECTSIZE`.
Ecrire une fonction `SumVect` qui réalise la somme de deux vecteurs et retourne le résultat.

Proposition de réponse

```c
#define VECTSIZE 5

#define VECTNULL {0,0,0,0,0}

double* SumVect(double *U, double *V){
  double W[VECTSIZE]=VECTNULL;
  int i = 0;

  for(i = 0; i < VECTSIZE; i++){
     W[i] = U[i] + V[i];
  }
  return W;
}
```

Ca ne fonctionne pas !

La gestion automatique de la mémoire de la pile (stack) ne permet pas de créer un espace de stockage d'information au sein d'une fonction et de transmettre son adresse d'une fonction vers une autre de manière pérenne.

Autre problème à résoudre (en ANSI C ISO 89)

On considère des vecteurs de `double` de taille fixe mais non-déterminée *a priori*

Ecrire une fonction`OnesVect` qui crée un vecteur rempli de `1` de taille `size` où . `size` est le paramètre de la fonction `OnesVect`

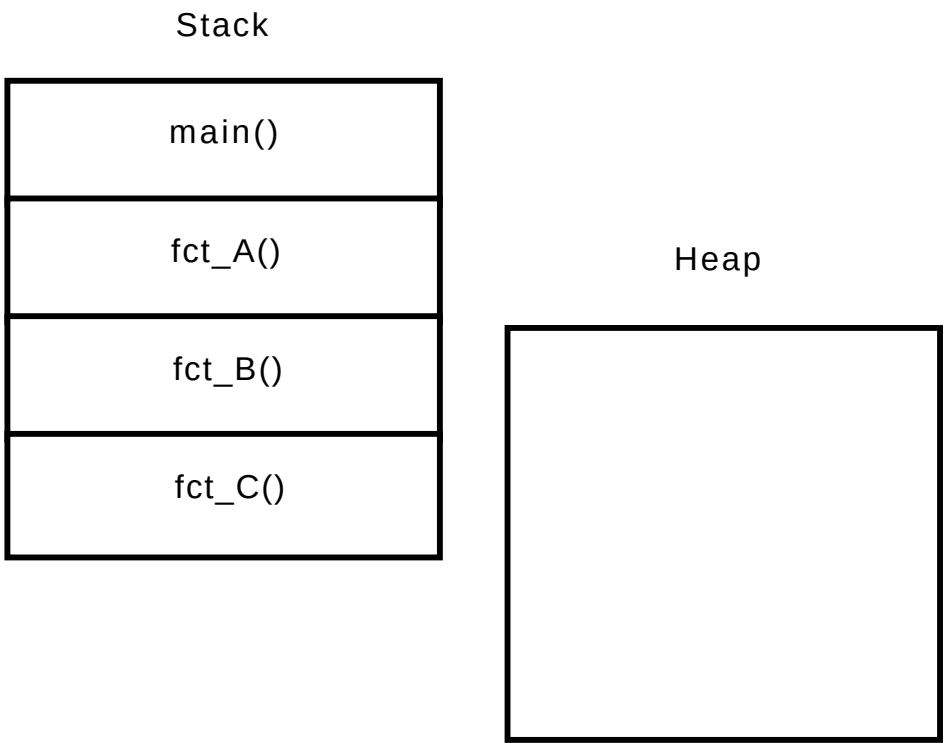
Que faire ?

On a besoin d'accéder à de la mémoire hors de la pile (stack).

Allocation de mémoire dans le « tas » (Heap)

Grâce à des commandes spécifiques, on peut allouer de la mémoire dans un autre espace mémoire appelé « tas » (Heap), hors de la pile.

Accéder au tas



L'accès est totalement manuel :

- allocation : « à la main »
- désallocation : « à la main »

Jeu de fonctions associées

- `malloc()`
- `calloc()`
- `realloc()`
- `free()`

Allocation de mémoire avec `malloc()`

```
#include <stdlib.h>

    void *malloc(size_t size);
```

Appel de la fonction malloc

- alloue un bloc memoire d'une taille size octets dans le tas (heap)
- retourne l'adresse du début de ce bloc mémoire grâce à une valeur de type (void*)
- en cas d'échec de son appel, malloc retourne la valeur d'adresse `NULL`

Deux ingrédient nécessaire à l'utilisation de `malloc()`

- pointeur générique (`void*`)
- spécification de taille mémoire en octets

Décrire des tailles d'objets en mémoire : opérateur `sizeof()`

Utilisation de `sizeof()` avec les types

`sizeof(aType_t)` : retourne la taille en octets de `aType_t`

`sizeof(int)` : retourne la taille en octets d'un `int`

`sizeof(double*)` : retourne la taille en octets d'un pointeur vers `double`

`sizeof(struct mystruct_t)` : retourne la taille en octets d'un `struct mystruct_t`

Utilisation de sizeof() avec des variables

```c
int i;
double *dptr;
char ch;
```

```c
sizeof(i);      // valeur = taille d'un int
sizeof(&i);     // valeur = taille d'un ptr vers int
sizeof(dptr);   // valeur = taille d'un ptr vers double
sizeof(*dptr);  // valeur = taille d'un double
sizeof(ch);     // valeur = 1 (octet)
```

Utilisation de sizeof() avec des variables

```
long array[10];

struct MyStruct_t *var;
```

```
sizeof(array[10]); // valeur = 10*(taille d'un long)

sizeof(var);       // valeur = taille d'un
                   // ptr vers un (struct MyStruct_t)
sizeof(*var);      // valeur = taille
                   // d'un (struct MyStruct_t)
```

Libération de mémoire avec `free()`

```
#include <stdlib.h>

    void free(void *ptr);
```

Appel de la fonction `free()`

- `free()` libère l'espace mémoire du Heap vers lequel pointe `ptr`.
- après appel de `free()`, cet espace est de nouveau mobilisable dans le Heap par appel à une fonction de type `malloc()`, `calloc()` ou `realloc()`.

Mémoire dynamique : exemple simple

De manière générale

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

Type_t *ptr=NULL;
...
ptr=malloc(sizeof(Type_t)); // allocation
/* test du resultat de l'allocation */
if (NULL==ptr){
  printf("problem!\n");
  exit(EXIT_FAILURE);
}
...
fct(*ptr); // cette fonction prend un parametre
           // Type_t en argument
...
free(ptr); // liberation de la memoire pointee par ptr
ptr = NULL; // mesure de securite
```

Mémoire dynamique : exemple simple

Un exemple avec des entiers

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

int *ptr=NULL;
...
/* formulation condensee:
allocation + test du resultat  */
if ( NULL==( ptr = malloc(sizeof(int)) ) ){
  printf("problem!\n");
  exit(EXIT_FAILURE);
}
...
*ptr=14;
fct(*ptr); // cette fonction prend un int en argument
...
free(ptr); // liberation de la memoire pointee par ptr
ptr = NULL; // mesure de securite
```

Allocation de mémoire avec calloc()

```
#include <stdlib.h>

   void *calloc(size_t nmemb, size_t size);
```

Appel de la fonction calloc()

- alloue un bloc memoire d'une taille permettant de stocker `nmemb` éléments de taille `size` octets dans le tas $\Rightarrow$`nmemb`$\times$`size` octets
- met à zéro tous les bits du bloc mémoire alloué
- retourne l'adresse du début de ce bloc mémoire grâce à une valeur de type `(void*)`
- en cas d'échec de son appel, `malloc()` retourne la valeur d'adresse `NULL`

Spécificité de calloc

C'est la mise à zéro des bits : `calloc` = Clear ALLOC

Réallocation de mémoire avec `realloc()`

```
#include <stdlib.h>

   void *realloc(void *ptr, size_t size);
```

Appel de la fonction `realloc()`

- alloue en nouveau bloc mémoire de taille `size` octets, si c'est possible l'opération est réalisée en modifiant la taille du bloc alloué vers lequel pointe `ptr`.
- si la nouvelle taille est plus petite que la précédente : les size premiers octets de l'ancien blocs sont identiques aux size octets du nouveau bloc
- si la nouvelle taille est plus grande que la précédente : les premiers octets de l'ancien bloc sont identiques aux premieres octets du nouveau bloc. Les octets suivants dans le nouveau bloc ne sont pas initialisés
- la fonction retourne l'adresse de démarrage du nouveau bloc

A propos de l'appel de `malloc()`

Ne pas oublier

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

Dans les fichiers où sont effectués les appels à `malloc()`, `calloc()`, `realloc()`, `free()`. Sinon une compilation avec par exemple le flag `-Wall` retourne des warnings

```
main.c:7: warning: implicit declaration of function
malloc
main.c:7: warning: incompatible implicit declaration of
built-in function malloc
main.c:10: warning: implicit declaration of function free
```

A propos de l'appel de malloc

Exemples d'appel

```
1 int *ptr=NULL;
2 ...
3 ptr = malloc(sizeof(int)); // ok
4 ptr = malloc(sizeof(*ptr));// ok
5 ptr = malloc(sizeof(8));   // DANGER: passe
6                            // a la compilation!
```

A propos de l'appel de malloc

Exemples d'appel

```c
#define NB_ELTS 12
int *A=NULL;

/* versions avec malloc */
A = malloc(12*sizeof(int)); // bof
A = malloc(NB_ELTS*sizeof(int)); // ok

A = malloc(12*sizeof(*A)); // bof
A = malloc(NB_ELTS*sizeof(*A)); // ok

/* versions avec calloc */
A = calloc(12, sizeof(int)); // bof
A = calloc(NB_ELTS, sizeof(int)); // ok

A = calloc(12, sizeof(*A)); // bof
A = calloc(NB_ELTS, sizeof(*A)); // ok
```

Retour sur la fonction SumVect()

```c
#include<stdio.h>

#define VECTSIZE 5

double* SumVect(double *U, double *V){
    double *W = NULL;
    int i = 0;

    W=malloc(VECTSIZE*sizeof(*W));
    if(NULL==W){
            // gestion de l'erreur
        ...
    }

    for(i = 0; i < VECTSIZE; i++){
        W[i] = U[i] + V[i] ;
    }

    return W;
}
```