Le langage C : Allocateur de mémoire

Dans cette série d'articles, nous présentons le langage C. Il ne s'agit pas de réécrire les ouvrages de référence cités en annexe, mais de donner, au travers du C, une méthode de programmation basée sur la notion d'interface.
Au fur et à mesure de notre progression, nous décrirons les éléments indispensables du langage et conduirons le lecteur à consulter les ouvrages de référence. Le but poursuivi est clairement de parvenir à construire des programmes de manière segmentée.
Ce mois-ci, nous programmerons un allocateur de mémoire qui visera à remplacer les fonctions standard de la famille de malloc(). Moins performant que la version d'origine, notre allocateur apportera une aide importante à la mise au point en détectant les blocs corrompus et les blocs non libérés.

Description

Tout au long de ces articles, nous avons présenté, puis abondamment utilisé l'allocation dynamique de la mémoire. L'allocation statique de la mémoire permet de créer des programmes rapides et simples, car la mémoire nécessaire est allouée sous la forme de tableaux au démarrage du programme. De cette manière, il n'existe pas de temps de création et libération de la mémoire. Par contre, le désavantage majeur est que le programme ne sait pas ajuster sa consommation de mémoire en fonction de ses besoins et que le programmeur est souvent obligé de surdimensionner ses tableaux.
L'allocation dynamique a permis à l'informatique en général de faire un grand bond en avant et de proposer des logiciels adaptables et évolutifs. L'allocation dynamique est en fait une vue de l'esprit car dans un ordinateur, la mémoire n'est pas fabriquée au fur et à mesure des besoins ! Il s'agit donc bien de gérer une quantité de mémoire finie en la répartissant entre les différents processus.
Chaque programme possède au démarrage une certaine quantité de mémoire allouée par le système d'exploitation et il la gère sous la forme de blocs alloués et libres. Il est possible qu'à un instant donné, le programme ait besoin de plus de mémoire, et dans ce cas, il formule une requête au système d'exploitation qui la satisfait ou non.
D'un point de vue général, la mémoire dite dynamique est constituée d'une liste de blocs libres que l'allocateur découpe en fonction des besoins et regroupe lorsque ces blocs sont "libérés".
Ce mois-ci, nous allons décrire un allocateur de mémoire simple. Mais rendons à **Ritchie** ce qui est à **Kernighan** et vice versa (ce sont les inventeurs du langage C) : l'allocateur est tiré de leur livre "*The C programming Language*", Prentice-Hall Inc, la bible du langage C qui servit pendant longtemps de norme, avant que le comité de l'ANSI ne normalise ce langage.
Cet allocateur permettra dans une certaine mesure d'aider à la mise au point des programmes en contrôlant que les blocs alloués aient été correctement libérés. Il existe des outils comme **ElectricFence** qui permettent de contrôler que l'on accède jamais en dehors des blocs alloués, que ce soit en lecture ou en écriture. Notre outil permettra dans une certaine mesure de vérifier que les écritures n'ont pas eu lieu en dehors des blocs.
Les sources de l'allocateur sont disponibles sur **http://www.erian-concept.com/pub/heap.tgz.**

En-tête de la bibliothèque

Comme à l'accoutumé, nous commençons par l'écriture de l'en-tête heap.h de la bibliothèque de fonctions qui déclare le prototype des fonctions, les constantes et les variables :

```
#ifndef ___HEAP_H
#define ___HEAP_H

void     heap_init    (unsigned   heap_size);
void     heap_end     (void);
char *   heap_malloc  (unsigned   nbytes);
char *   heap_calloc  (unsigned   size,
                       unsigned   number);
char *   heap_realloc (char     * pointer,
                       unsigned   size);
void     heap_free    (char     * pointer);
unsigned heap_size    (char     * pointer);
void     heap_stats   (unsigned * current,
                       unsigned * maximum,
                       double   * total,
                       unsigned * mallocs);
#endif /* ___HEAP_H */
```

Nous retrouvons les équivalents des fonctions *malloc()*, *calloc()*, *realloc()* et *free()*. Les fonctions *heap_init()* et *heap_end()* servent à initialiser et à terminer la bibliothèque de fonctions. Ces fonctions doivent être utilisées respectivement avant toute utilisation et après toute utilisation du tas. On trouve la fonction *heap_size()* qui renvoie la taille d'un bloc et la fonction *heap_stats()* qui permet d'obtenir des informations sur l'utilisation de la mémoire (nombre d'octets actuellement alloués, nombre maximum d'octets simultanément alloués, total de la mémoire allouée et nombre d'allocations).

Nous verrons la prochaine fois comment replacer automatiquement les appels aux fonctions standard comme par des appels aux fonctions équivalentes de cette bibliothèque, comme *heap_malloc()*.
Passons maintenant à la réalisation de la bibliothèque.

Réalisation de la bibliothèque

La réalisation de cette bibliothèque heap.c débute une fois de plus par l'inclusion des en-têtes de fonctionnalités utilisées :

```
#define ___HEAP_C

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

Nous entrons maintenant dans le vif du sujet. Nous prévoyons utiliser des fonctions d'aide à la mise au point. Pour l'instant, ces fonctions seront des macros inactives. La prochaine fois, elles seront définies :

```
#ifdef __DEBUG_HEAP

/*****************************************
* M I S E   A U   P O I N T   D U   T A S *
*****************************************/

/* Ce sera pour la prochaine fois (:-) */

#else /* __DEBUG_HEAP */
# define _debug_heap_add(p) ((char *)(p))
# define _debug_heap_del(p) ((char *)(p))
# define _debug_heap_end()
# define _debug_heap_init()
#endif /*__DEBUG_HEAP */
```

Nous déclarons ensuite un certain nombre de définitions et types de données :

```
/* Taille en octets de l'unité d'allocation */
#define NALLOC 2048

/* Type d'alignement en mémoire. Ici,
* alignement sur des mots */
#define ALIGN_TYPE int

/* En-tête des blocs mémoire */
typedef union u_header * Header;

typedef union u_header {
 struct {
   unsigned size;  /* Taille du bloc */
   Header   ptr;   /* Entête suivant */
 }          s;
 ALIGN_TYPE align; /* alignement */
} _Header;

/* Variables locales pour les statistiques */
static unsigned _current, _maximum, _mallocs;
static double   _total;

/* Variable locale des gestion du tas */
static _Header _base;       /* Base du tas */
static Header  _allocp = 0; /* Premier bloc
                            * libre */
```

La mémoire sera allouée par blocs de *NALLOC* octets. Cette granularité permettra d'éviter une trop grande fragmentation de la mémoire. En effet, au bout d'un certain temps, lorsque beaucoup de mémoire a été allouée et libérée, on observe la présence de nombreux petits blocs de mémoire libres séparés par des blocs alloués (ce qui interdit leur réunion en blocs libres plus gros). Afin d'éviter cela, la mémoire est allouée en unité, ici, de deux kilo-octets.
Les processeurs peuvent accéder n'importe quel octet de la mémoire. Cependant, la mémoire est organisée en mots de 32 bits, voire 64 bits sur la prochaine génération de processeurs. Or les accès sont plus rapides lorsque les octets sont alignés sur des mots. On déclare donc un type de donnée, ici *int*, qui forcera l'alignement des blocs alloués. On pourra forcer l'alignement sur des mots de 64 bits en associant à *ALIGN_TYPE* le type *double*.
Les blocs de mémoire possèdent tous un en-tête contenant la taille du bloc en unité d'allocation (ici, 2048 bits) et un pointeur vers le bloc suivant. Cet en-tête est déclaré comme une union entre une structure contenant ces informations et le champ forçant l'alignement.
Nous trouvons ensuite des variables privées utilisées pour maintenir des informations concernant l'allocateur, comme le nombre d'octets actuellement alloués par l'utilisateur, le maximum d'octets alloués simultanément, le nombre d'allocations effectuées et la somme de la mémoire allouée (cumul de la taille en octets des espaces alloués).
Nous trouvons ensuite les variables locales de gestion du tas. Le premier en-tête du tas, *_base*, est une variable statique qui n'est pas allouée dynamiquement. Ces champs *size* et *ptr* seront initialisés lors de l'initialisation du tas. Le pointeur *_allocp* contient l'adresse de l'en-tête du dernier bloc alloué ou zéro.

Initialisation et terminaison

L'initialisation de la bibliothèque se contente de donner aux variables locales leur valeur initiale :

```
/* I N T E R F A C E */

/* Initialisation de la bibliothèque */
void heap_init (unsigned heap_size) {
 /* Initialisation du  tas */
 _base.s.ptr  = _allocp = & _base;
 _base.s.size = 0;
 _current  = 0;
 _maximum  = 0;
 _mallocs  = 0;
 _totals   = 0.0;

 /* Intégration du premier bloc de mémoire
* dans la mémoire libre */
 heap_free (heap_malloc (heap_size));
}
```

La fonction de terminaison contrôle la cohérence du tas. Nous verrons cela la prochaine fois.

```
/* Terminaison de la bibliothèque */
void heap_end () {
 /* Vérification de l'intégrité du tas */
 _debug_heap_end();

 /* Réinitialisation des variables */
_allocp = 0;
}
```

Allocation

La fonction d'allocation est la suivante :

```
/* Équivalent de la fonction standard
* malloc() */
char *  heap_malloc (unsigned nbytes) {
 Header   p, q;
 unsigned nunits;
 if (! nbytes) {
   return 0;
 }
 /* Nombre d'unités de mémoire demandées */
 nunits = 1 + (nbytes / sizeof (_Header));
 q = _allocp;
 if (! q) {
   _base.s.ptr  = _allocp = q = & _base;
   _base.s.size = 0;
 }
 /* Recherche des blocs libres d'un bloc
* suffisamment grand ... */
 for (p = q->s.ptr; ; q = p, p = p->s.ptr) {
   /* Si la taille du bloc libre est
* suffisante ... */
    if (p->s.size >= nunits) {
     if (p->s.size == nunits) {
       /* Si la taille est exactement celle demandée
*/
        q->s.ptr = p->s.ptr;
     }
     else {
       /* Sinon, scinder le bloc ; le bloc
* alloué est placé à la fin. */
        p->s.size -= nunits;
       p         += p->s.size;
       p->s.size  = nunits;
     }
     /* Mise à jours des statistiques */
      _mallocs++;
     _current += nbytes;
     _total   += (double) nbytes;
     if (_current > _maximum) {
       _maximum = _current;
     }
     /* */
     _allocp = q;
     /* Ajout du pointeur pour la mise au point */
     _debug_heap_add ((char *)(p + 1));

     /* Retourner le pointeur */
     return (char *) (p + 1);
   }
   if (p == _allocp) {
     /* Demander de la mémoire supplémentaire
* au système d'exploitation. */
      p = _morecore (nunits);
     if (p == 0) {
       /* S'il n'y a plus de mémoire,
* positionner la variable errno, et
* retourner le pointeur nul */
        errno = ENOMEM;
       return 0;
     }
   }
 }
}
```

La fonction d'allocation commence par calculer le nombre d'unités de mémoire demandées. Si la bibliothèque vient juste d'être initialisée, *_allocp* vaut zéro. Dans ce cas, on utilise l'entête *_base* pour créer une liste de blocs libres contenant un seul élément de taille nulle ; cela obligera la fonction à réclamer plus de mémoire au système d'exploitation. On obtiendra alors la structure suivante :

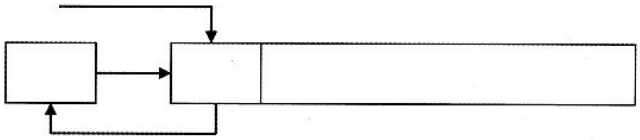

Figure 1a : Structure après réclamation de mémoire auprès du système.

Dès lors, la fonction parcourt la liste de blocs libres qui commence avec le pointeur *_allocp* à la recherche d'un bloc suffisamment grand pour accueillir la zone de mémoire demandée. Si le bloc possède exactement la taille demandée, il est utilisé intégralement et on aurait :

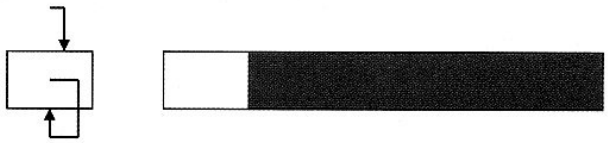

Figure 1b : Structure après allocation de toute la mémoire libre.

Dans le cas où le bloc libre est plus grand que la taille demandée, le bloc alloué est placé à la fin du bloc libre et le bloc libre est scindé en deux, ce qui donne la structure suivante :

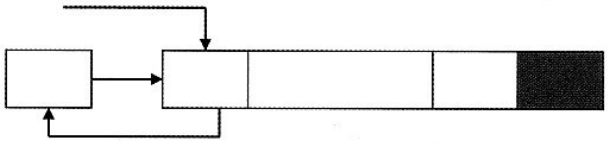

Figure 1c : Structure des données après l'allocation d'un petit bloc.

Après un certain temps de fonctionnement et des allocations et des libérations entrelacées, la structure du tas pourrait être comme dans la figure suivante :

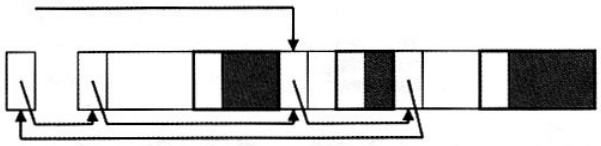

Figure 1 d :Après un certain nombre d'allocations / libérations de blocs...

La fonction utilise la conversion des pointeurs pour provoquer la granularité dans l'allocation de la mémoire. En effet, un pointeur sur caractères incrémenté de un pointe sur le caractère suivant ; de même, un pointeur sur une structure de donnée incrémenté de un pointe sur la structure suivante.

Entre les deux pointeurs, l'incrément effectif n'a pas été de un, mais de la taille de l'objet pointé, c'est-à-dire un dans le premier cas et la taille de la structure pointée dans le second cas.

Les blocs ont une taille minimum, celle de la structure *_Header*. La liste des blocs libres est maintenue par ordre croissant d'adresses dans la zone de mémoire à allouer.

Lorsqu'aucun bloc libre n'est capable d'accueillir la demande, la fonction *_morecore()* est appelée pour réclamer plus de mémoire au système d'exploitation. En cas de succès, la mémoire libre obtenue est intégrée dans la liste des blocs libres ; on procède alors à une nouvelle itération qui ne doit pas échouer. En cas d'échec, lorsque *_morecore()* retourne zéro, la constante *ENOMEM* définie dans le fichier *errno.h* est placée dans la variable globale *errno* de manière à être conforme à la norme ANSI, puis le pointeur nul est retourné.

Examinons maintenant la manière avec laquelle la mémoire est réclamée au système d'exploitation.

Demande de mémoire

D'une manière très générale, tout processus est associé à au moins trois zones de mémoire : une zone pour le code, une zone pour la pile et une zone pour les données. La fonction *sbrk()* permet d'augmenter la taille de la zone des données de *n* octets. La fonction retourne alors un pointeur sur la nouvelle zone des données de *n* octets, ou le pointeur valant -1 en cas d'échec. Nous allons utiliser cette fonction pour réclamer de la mémoire :

```
/* Demande plus de mémoire au système
* d'exploitation */
static Header _morecore (unsigned nunits) {
 Header   up;
 unsigned rnu;

 /* Nombre d'unités de mémoire */
 rnu = NALLOC * (1 + nunits / NALLOC);

 /* Demande d'extension du tas */
 up  = (Header) sbrk (rnu * sizeof (_Header));
 if (-1 == (int) up) {
   return 0;
 }
 /* Initialisation de l'en-tête */
 up->s.size  = rnu;

 /* Ajout du pointeur dans le tableau
* des pointeurs alloués. */
 _debug_heap_add ((char * ) (up + 1));

 /* Libération de ce pointeur pour
* l'ajouter dans la mémoire
* allouable. */
 heap_free ((char /* Retourner le pointeur. */
 return _allocp;
}
```

La fonction calcule donc le nombre d'unités de mémoire demandées puis invoque *sbrk()*. En cas de succès, l'en-tête du bloc nouvellement alloué est initialisé puis il est intégré dans la liste des blocs libres avec une invocation à la fonction *heap_free()*. Le pointeur *_allocp* pointe alors sur le dernier bloc alloué et c'est celui-ci qui est retourné.

La prochaine fois, nous examinerons comment libérer la mémoire, les autres fonctions d'allocation et l'aide à la mise au point. Enfin, nous terminerons en modifiant le fichier d'en-tête *heap.h* afin que les appels aux fonctions standards de la famille de *malloc()* soient automatiquement remplacés par des appels à nos fonctions au moment de la compilation.

L'auteur

Guilhem de Wailly, directeur de la société Erian Concept, partenaire NetUltra (www.netultra.net) et Getek (www.getek.fr).

Références

Langage C - norme ANSI : bon ouvrage de référence
Ph. DRIX
Masson
Programmer en C++ : une bonne référence
S.C. DEWHURST et K.T. STARK
Masson
Le Langage C : la bible!
B.W. KERNIGHAN et D.M. RITCHIE
Masson
gcc: compilateur C du GNU, et tous les outils de développement
http://www.gnu.org
Kdevel: un environnement KDE de programmation C
http://samuel.cs.uni-potsdam.de/~smeier/kdevelop_new/index.html