Programmation impérative

Cours 3 : Allocation de mémoire en C

Catalin Dima

Utilisation des pointeurs

- Pointeur pointant sur une adresse déjà en utilisation par une autre variable (individuelle, tableau, etc.)
  - Astucieux, mais est-ce tellement utile ?...
- Pointeur pointant sur une zone de mémoire nouvellement réservée.
  - Idée : on ne connaît pas à l'avance (c.à.d. à la compilation !) si on aura besoin de zones supplémentaires de zone pour stocker des valeurs calculées.
  - Exemple : tableau de taille variable.
  - Exemple 2 : liste de valeurs dans laquelle on insère ou on supprime au gré de l'utilisation du programme.
  - Exemple 3 : matrices creuses.
  - Exemple 4,5,6,7,8... : arbres/hastables/diverses structures de données.

Allocation de mémoire : `malloc`

- Supposons qu'on veut écrire un programme qui manipule des séquences de valeurs entières saisies par l'utilisateurs, suite de taille variable, donnée par l'utilisateur et qui peut être différente entre deux saisies différentes.
- On peut le faire avec un tableau `int tab[2000]` et une variable supplémentaire `int taille` qui sera initialisée à chaque saisie avec le nombre de valeurs que l'utilisateur veut saisir.
  - Mais à chaque exécution il y aura `2000-taille` cases mémoire inutilisées !
  - Et, en plus, il faut contraindre l'utilisateur à saisir une valeur `taille <= 2000` !
  - Il ne sera pas content s'il doit travailler avec 3000 valeurs !

Allocation de mémoire : `malloc`

- Supposons qu'on veut écrire un programme qui manipule des séquences de valeurs entières saisies par l'utilisateurs, suite de taille variable, donnée par l'utilisateur et qui peut être différente entre deux saisies différentes.
- On peut le faire avec un tableau `int tab[2000]` et une variable supplémentaire `int taille` qui sera initialisée à chaque saisie avec le nombre de valeurs que l'utilisateur veut saisir.
  - Mais à chaque exécution il y aura `2000-taille` cases mémoire inutilisées !
  - Et, en plus, il faut contraindre l'utilisateur à saisir une valeur `taille <= 2000` !
  - Il ne sera pas content s'il doit travailler avec 3000 valeurs !
- Solution : `malloc` – fonction qui réserve une zone de mémoire (contiguë) de taille donnée en paramètre et renvoie sa première adresse en résultat, de sorte à ce qu'on puisse l'affecter à un pointeur :

```
ptr = malloc(3000);
```

  - On dit qu'on alloué 3000 octets au pointeur `ptr`.

Utilisation de `malloc`

Déclaration `void *malloc(size_t dimension)` :

- `size_t` : dimension de la zone de mémoire à réserver, en octets.
- `void *` : par défaut, `malloc` renvoie un pointeur sans type, c'est à nous d'utiliser l'espace réservé avec un type particulier
- ... par exemple en l'affectant à un pointeur `char` :

```
char * ptr;
ptr = (char*) malloc(3000);
```

- Bibliothèque qui contient la fonction `malloc` : stdlib.h (à inclure avec `#include`!).

Exemple

Pour notre exemple de réservation d'une zone de mémoire pour un tableau d'entiers :

```
int * tbl, taille;
scanf("%d",&taille);
tbl = (int *)malloc(taille * sizeof(int));
```

- Fonction `sizeof` (opérateur !) nous donne la taille d'un type de données (ici `int`) – elle peut varier d'un compilateur à l'autre !
- Résultat (pour un compilateur pour lequel un `int` est codé sur 4 octets) : réservation de 12000 octets, dont la première adresse est pointée par `ptr`.
- Les 12000 octets pourront être utilisés comme un tableau de 3000 entiers.
- Donc `ptr[4] =4567` stocke la valeur 4567 sur les 4 octets qui se trouvent à l'adresse `ptr + 16`(octets).
  (dessin au tableau)

Attention !

- Ne pas "perdre" l'adresse de la zone allouée avec `malloc`, car elle ne peut plus être retrouvée !
- Donc si vous jouez avec la valeur de `tbl`, faites en sorte que vous pouviez faire le calcul inverse et revenir à la valeur que `malloc` vous a envoyée juste après le `scanf` ci-dessus.
- Un autre appel à `malloc` ne vous enverra pas l'adresse antérieure que vous avez perdue !

calloc

- Une allocation mémoire (un peu) plus structurée se fait avec `void *calloc(size_t nbre, size_t tailledunecase)`, équivalente à `malloc(nbre * tailledunecase)` :

```
int *tbl, taille;
scanf("%d",&taille);
tbl = (int *)calloc(taille, sizeof(int));
```

- Une petite différence : `calloc` initialise à 0 toutes les cases allouées.
- ... à la différence de `malloc` qui fournit les cases mémoire telles quelles (avec les éventuels restes des calculs précédents).
- ... ce qui peut être embêtant si on oublie d'instancier les cases d'un tableau, en considérant qu'avant toute utilisation elles ont la valeur 0.

`realloc` : changer d'avis

- Si jamais on se rend compte que la zone de mémoire allouée est trop petite ou trop grande, il faut allouer une zone de taille différente.
- On peut bien-sûr le faire à l'aide de `malloc/calloc`, mais...
- ... il faut pas oublier de copier dedans toutes les valeurs qu'on avait stockées dans la première zone !
- `realloc` fait ce boulot pour nous :
  - `void *realloc(void *ptrancien, size_t nvlletaille)` : on donne le pointeur ancien (qui identifie la zone de mémoire utilisée jusqu'alors), et la nouvelle taille qu'on veut voir allouer au pointeur ancien.
- Effet de l'instruction `ptr = realloc(ptr, nvlletaille)` :
  - Une nouvelle zone de mémoire est réservée.
  - Le contenu de la zone de mémoire allouée à `ptr` est copié (octet par octet), au début de cette nouvelle zone.
  - Le résultat retourné est un pointeur qui pointe sur le premier octet de cette nouvelle zone de mémoire.
  - L'ancienne zone est déclarée *inutilisée*.
- Si la taille de la nouvelle zone est plus grande que celle de l'ancienne zone, tout le contenu de l'ancienne est copié.
- Si la taille est plus petite, on recopie que ce qui rentre dans la nouvelle zone, à partir du premier octet.
- Si, pour une raison ou une autre, on ne peut pas réallouer la mémoire, la valeur renvoyée sera `NULL` (ou 0 sur 4 octets) et l'ancienne zone ne sera pas déclarée libérée.
  - Donc faire bien attention ! ne jamais faire `ptr = realloc(ptr,nvval)`, car on risque de se retrouver avec un pointeur null et de perdre la référence de l'ancienne zone de mémoire, et tout son contenu !

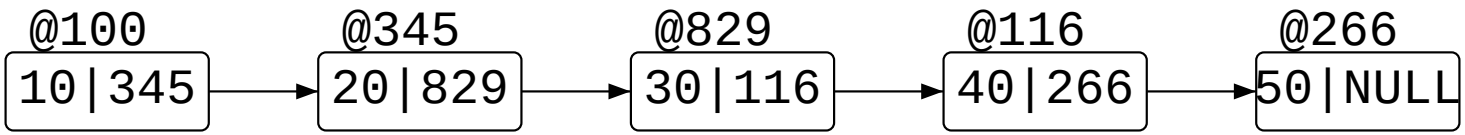
`free` : prendre soin de ne pas avoir de fuites de mémoire

- Lorsqu'une zone de mémoire allouée cesse d'être nécessaire, que fait-on avec ?
- Il faut la marquer comme étant disponible pour des allocations ultérieures !
- `void free(void *ptr)` fait le boulot : prend un pointeur en paramètre et, *en considérant que sa valeur représente une zone de mémoire allouée*, la marque comme étant désalouée et, donc, utilisable par un `malloc/calloc/realloc` ultérieur.
- Si `ptr` ne pointe pas sur une zone précédemment allouée (avec un `malloc` ou compagnie), le comportement peut devenir bizarre... !
- Si `ptr` est `NULL` (c.à.d. 0), il ne se passe rien.
- Donc, pour éviter des problèmes, la suggestion est d'enchaîner un `free` avec une affectation de valeur `NULL` :

```
int *ptr;
ptr = (int *) malloc(sizeof(int));
.....
free(ptr);
ptr = NULL;
```

Listes chaînées

- Les tableaux sont utiles pour le stockage simultané des valeurs d'un type donné.
- Mais parfois on est amené à prévoir le rajout, une par une, d'une suite de valeurs du même type.
- **Liste chaînées** : suite de cellules, chacune composée d'une valeur et d'un pointeur indiquant la place de la prochaine cellule dans la suite :



- La position des cellules en mémoire peut ne pas correspondre à leur place dans la suite !

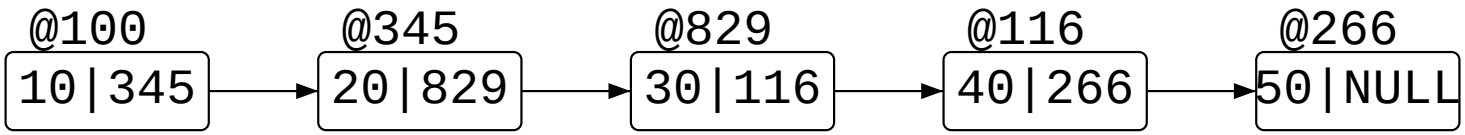
Déclaration de listes chaînées

- Supposons que la valeur à stocker est un entier sur 4 octets :

```
struct liste{
  int val;
struct liste *ptr;
};
struct liste *x;
```

- En tant que pointeur, `x` contient une adresse, et `*x` est la donnée stockée à cette adresse.
- En tant que struct, `*x` possède deux champs : `(*x).val` et `(*x).ptr`.
- Raccourci de notation : `x->val`, respectivement `x->ptr`.
  - `x->val` est la valeur stockée dans la cellule pointée par la variable `x`.
  - `x->ptr` est le pointeur stocké dans la cellule pointée par la variable `x`.
- À noter que `x->ptr` est lui-même un pointeur !
  - `x->ptr->val` est la valeur stockée dans la cellule pointée par la variable `x->ptr`.
  - `x->ptr->ptr` est le pointeur stocké dans la cellule pointée par la variable `x->ptr`.

Exemple de liste chaînée



- Supposons que la variable `x`, déclarée en tant que `struct liste *x`, stocke la valeur (adresse !) `@100`.
- Combien vaut `x->val` ?
- Combien vaut `x->ptr` ?
- Combien vaut `x->ptr->val` ?
- Combien vaut `x->ptr->ptr->ptr` ?
- Combien vaut `x->ptr->ptr->ptr->ptr->ptr->ptr` ?

Exemple, suite

- Combien vaut `x->val` ? `10`
- Combien vaut `x->ptr` ? `@345`
- Combien vaut `x->ptr->val` ? `20`
- Combien vaut `x->ptr->ptr->ptr` ? `@116`
- Combien vaut `x->ptr->ptr->ptr->ptr->ptr->ptr` ? Erreur sémantique ! (l'exécution d'une instruction contenant cette expression peut crasher le programme !)

Exemple, création d'une liste chaînée

- Une liste chaînée est créée à l'aide de `malloc/calloc` !
  - Mais combien doit-on réserver pour une cellule de type `struct liste` ?
  - Rien de plus simple ! `sizeof(struct liste)` nous renvoie la bonne réponse !

```
struct liste *cell;   // on peut réutiliser la déclaration du type liste !
cell = (struct liste*) malloc(sizeof(struct liste));
cell->val = 10;
cell->ptr=NULL;
```

Création d'une liste chaînée, 1er essai

- Et maintenant créons une liste contenant 3 valeurs lues à l'entrée std :

```
struct liste *cell1, *cell2, *cell3;
cell1 = (struct liste*) malloc(sizeof(struct liste));
scanf("%d",&cell1->val); // eh oui ! cell1->val n'est pas un pointeur !
cell2 = (struct liste*) malloc(sizeof(struct liste));
scanf("%d",&cell2->val);
cell3 = (struct liste*) malloc(sizeof(struct liste));
scanf("%d",&cell3->val);
   // et il faut connecter les 3 cellules !
cell1->ptr = cell2;
cell2->ptr = cell3;
cell3->ptr = NULL;
```

Création “algorithmique” d’une liste chaînée

- Mouais, mais c’est pas joli ! On veut le faire pour n’importe quel nombre de cellules !
- Il faut faire une boucle !

Création “algorithmique” d’une liste chaînée

- Créer la première cellule d’abord, puis les autres en allouant le pointeur dans chaque cellule :

```
struct liste *cell;
int i;
cell = (struct liste*) malloc(sizeof(struct liste));
scanf("%d",&cell->val);
for (i=0,i<2;i++){
   cell->ptr = (struct liste*) malloc(sizeof(struct liste));
   scanf("%d",&cell->ptr->val); // c’est là qu’il faut mettre la valeur !
   cell->ptr->ptr = NULL;       // le champ pointeur de la nouvelle cellule !
   cell = cell->ptr ;           // on passe au suivant !
}
```

- Mais... gros problème !
- Comment on revient à la tête de liste ?... ben, on l’a perdue !!

Création “algorithmique” d’une liste chaînée (2)

```
struct liste *cell, *tete;
int i;
cell = (struct liste*) malloc(sizeof(struct liste));
tete = cell;                        // et maintenant on la perd plus !
scanf("%d",&cell->val);
for (i=0,i<2;i++){
   cell->ptr = (struct liste*) malloc(sizeof(struct liste));
   scanf("%d",&cell->ptr->val); // c’est là qu’il faut mettre la valeur !
   cell->ptr->ptr = NULL;
   cell = cell->ptr ;               // on passe au suivant !
}
```

Création “algorithmique” d’une liste chaînée (3)

On peut aussi créer cette liste dans l’ordre inverse :

```
struct liste *cell, *cellprec;
int i;
cellprec = NULL;
for (i=0,i<3;i++){
   cell = (struct liste*) malloc(sizeof(struct liste));
   scanf("%d",&cell->val);
   cell->ptr = cellprec;
   cellprec = cell;     // et la prochaine fois on insère en tête de liste !
}
```

Fonctionnement de `malloc/calloc/realloc/free`

- La mémoire du programme est découpée en plusieurs parties :
  - Zone de code (text).
  - Zone des données initialisées (variables déclarées globales).
  - Tas = zone d'allocation dynamique de mémoire.
  - Pile - zone d'allocation de mémoire pour les paramètres et les variables déclarées dans les fonctions, et pour la valeur de retour.
- Tout `malloc` va chercher dans le tas une zone de mémoire de taille suffissante pour l'allouer.
- Les zones allouées sont marquées dans des octets spéciaux (qui sont réservés en supplément des octets demandés).
- Les zones libres sont mises elles aussi dans une liste chaînée.
- Chaque allocation diminue une zone libre, ou la supprime carrément de la liste des zones libres, et remplit les octets d'information de la zone réservée.
- Chaque `free` fait le contraire, en fusionnant éventuellement des zones libres si elles deviennent contiguës.
- Différentes implémentations de ces principes existent – celle sous *Linux* est la bibliothèque `glibc`.