Institut Galilée
Architecture, système, réseaux
Licence 2 (tronc commun)
2005-2006

10 octobre 2005

Adresses, pointeurs, allocation mémoire

1 Adresses

Chaque donnée stockée en mémoire principale (variable, fonction) a un emplacement unique, représenté par son *adresse*. En C, `&x` est l'adresse de `x` et `*a` est la donnée stockée à l'adresse `a`.

Un *pointeur* est une variable de type adresse. On déclare un pointeur `p` sur des données de type `type` en composant : `type *p`.

Manipulation 1 (Appel par valeur)**.** *Soit le programme :*

```
#include <stdio.h> /* pour printf() et scanf() */

void carre(int i){
  i = i * i;
}

int main(){
  int j = 0;
  printf("Donner un nombre entier\n");
  scanf("%d", &j);
  carre(j);
  printf("Son carre est %d !\n", j);
  exit(0);
}
```

Ce programme se trouve dans le fichier `~boudes/TP3/p1.c`*, le copier dans votre répertoire.*

Compiler, exécuter. Que se passe-t-il ? Quelle était l'intention du programmeur ? Corriger, en un minimum de modifications. Pouvez-vous corriger le programme en insérant seulement cinq caractères ? Pourquoi faut-il passer l'adresse de `j` *à scanf ?*

Correction 1. Le programme rend le nombre entré et non son carré. C'est parce que le C fonctionne en appel par valeur et non en appel par nom. On peut corriger d'au moins trois manières :

- **La plus directe : remplacer la ligne** `carre(j);` **par** `j = j * j;` **(on efface au moins 7 caractères et on en insère au moins 4).**
- `j = carre(j);`, `int carre(int i){return(i * i)}`.
- **la solution attendue : dans le** `main()` **:** `carre(&j)` **et dans** `carre()`, `carre(int *i)` **et** `*i = *i * *i;`.

En appel par valeur, on simule l'appel par nom en passant les adresses des variables. La question sur `scanf` **doit juste servir à les mettre sur la voie. (Au cas où ils s'attardent la dessus, on peut rappeler que le passage par adresse dans** `scanf` **permet la lecture de plusieurs données comme dans** `scanf("%d/%d", &p, &q)`**, et rend le nombre de données reconnues...).**

Les adresses sont ordonnées linéairement : on peut représenter la mémoire comme un tableau à une dimension (une suite de cases) dont les cases contiennent les données et où les adresses sont les indices. Une case mémoire est faite de plusieurs octets (4 sur un système 32 bits). Une donnée peut elle-même occuper plusieurs cases. Son adresse est alors celle de la première case (et le nombre de cases occupées se déduit de son type).

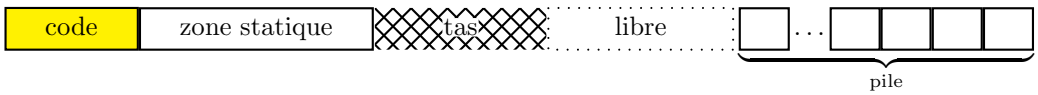
L'opérateur `sizeof` du C donne la taille d'une donnée en nombre d'octets. Appliqué à un type avec la syntaxe `sizeof(`*type*`)` il donne la taille en octets de n'importe quelle donnée de ce type. Appliqué à une expression avec la syntaxe `sizeof` *expression*, il donne le nombre d'octets qu'occuperait une donnée de même type que *expression*.

Manipulation 2. *Sur votre système, quelle est la taille du type* `char` *?* `short` *?* `int` *?* `long int` *?*

Correction 2. On obtient 1, 2, 4, 4.

Pour afficher une adresse, on peut utiliser le format `%p` de `printf` comme dans l'instruction `printf("Adresse de x : %p \n", &x)`. L'adresse est donnée en hexadécimal.

Zones mémoires. Un processus utilise quatre zones mémoires. La *zone de code* contient les instructions du programme. Sa taille et son contenu ne varient pas au cours de l'exécution. La *zone d'allocation statique* stocke les variables qui durent tout le temps de l'exécution du processus : les variables globales et les variables locales statiques. Sa taille est fixée, son contenu peut changer. La *pile* est la zone mémoire qui stocke les variables ayant une durée de vie limitée à une fonction ou un bloc (les variables locales). La pile forme un bloc contiguë de mémoire qui varie en taille. Elle est gérée selon le principe du dernier arrivé, premier sortie d'où le nom de pile (une pile est une structure de données très courante en informatique, pas uniquement pour la gestion de la mémoire). Lorsque le processus fait appel à une fonction, les adresses mémoires nécessaires au stockage de ses variables locales sont prises dans l'espace « au dessus » de la pile : la pile croît. Une fois la fonction terminée elles sont libérées (pour pouvoir être réutilisées ensuite). Le *tas* est la zone des emplacements mémoires obtenus par allocation dynamique (fonction `malloc()`, vue en détails plus loin). Les données stockées dans le tas ne forment pas nécessairement un bloc contiguë de mémoire. Un tas comme une pile est une structure de données qui tire son nom de la manière dont y sont gérées l'ajout et la suppression d'éléments. Nous ne détaillons pas ici son fonctionnement.



La pile croît des adresses hautes (à droite) vers les adresses basses (à gauche).

Manipulation 3. *Soit le programme (*`~boudes/TP3/p2.c`*) :*

```
#include <stdlib.h> /* pour malloc() et free() */
#include <stdio.h> /* pour printf() */

int k = 5;

void unefonction(int i){
  int j = 0;
  static int compteur = 0;
  j++;
  compteur++;
  printf("unefonction(%d) : compteur = %d (%p), j = %d (%p)\n",
 i, compteur, &compteur, j, &j);
  if (compteur == 1) { /* C'est le premier appel */
    unefonction(k); /* appel imbrique */
  }
}

int main(){
  void (*f)(int); /* variable de type fonction */
  char *p;
```

```
  f = unefonction;
  p = (char *) f;
  *p = 0;
  f(0);
  printf("&f : %p \n", &f);
  printf("f : %p \n", f);
  exit(0);
}
```

À l'exécution, ce programme affiche des valeurs et des adresses de variables. Lesquelles ? Lesquelles correspondent à quelle zone ?

Correction 3. **Les valeurs de `j` et `compteur`, deux fois. L'adresse de `f`, et celle de `j` (deux fois), dans la pile. La valeur de `f`, une adresse dans la zone de code, correspondant au code machine de `unefonction`, l'adresse de `compteur` (deux fois), dans la zone statique.On peut éventuellement leur faire afficher l'adresse de `k` pour comparer.**

Questions sur `unefonction()`. *La fonction* `unefonction()` *s'appelle elle-même : pourquoi cet appel récursif ne continue-t-il pas indéfiniment ? Les instances de la variable* `j` *au premier et au deuxième appel ont elles la même adresse ? Et les instances de la variables* `compteur` *? Que compte-t-on avec la variable* `compteur` *? Modifier le programme pour rendre l'appel récursif infini (*`CTRL-C`*, pour arrêter). Pour la suite, reprendre le programme non modifié.*

Correction 4. **L'appel récursif s'arrête car `compteur` prend la valeur 2 au deuxième appel. La condition pour un nouvel appel n'est donc plus satisfaite. Les deux instances de `j` ont des adresses différentes (et stockent des valeurs différentes). Par contre les deux instances de `compteur` sont identiques puisqu'elles ont la même adresse. Rappeler qu'une variable statique n'est initialisée qu'une fois (dès la compilation). La variable `compteur` compte le nombre d'appels de `unefonction()`. Pour rendre l'appel infini on supprime le test sur `compteur`.**

Questions sur `main()`. *La variable locale* `f` *est un pointeur. Après initialisation, quelle est sa valeur ? Cette valeur est une adresse, d'après vous dans quelle zone mémoire est-elle située ? Peut-on écrire à cette adresse ?*

Correction 5. **le pointeur `f` pointe à l'adresse du code de `unefonction()` dans la zone de code. On ne peut pas écrire à cette adresse (`*f = *f ;` provoque une erreur de compilation et si on contourne en faisant `char *p ; p = (char *) f ; f = unefonction ; *p = 0 ;` on a cette fois une erreur de bus à l'exécution).**

Manipulation 4 (Optionnelle)**.** *Écrire un programme qui dépasse les capacités de la pile d'exécution et termine prématurément sur une erreur. Indication : on pourra employer une fonction ayant un tableau de grande taille parmi ses variables locales et, si nécessaire, l'appeler récursivement.*

Correction 6.

```
#include <stdio.h> /* pour printf() */

void foo(){ /* On essaie d'empiler... */
  char t[1024 * 1024]; /* ...1 mega octet sur la pile... */
  static int c = 0;
  c++;
  printf("foo %4d\n", c);
  if (c < 1024) foo(); /* ...mille fois. */
}
/* Ca plante la huitieme fois : la pile ne peut pas contenir plus de 8 Mo. */

int main(){
  foo();
  exit(0);
}
```

Tableaux. Si `t` est une variable de type tableau (par exemple déclarée par `int t[3] ;`) alors `t` est un pointeur sur le premier élément du tableau (`*t = t[0]`) ; l'adresse *de même type* suivant `t`, `t + 1`, est l'adresse du deuxième élément du tableau (`*(t + 1) = t[1]`) ; etc.

Manipulation 5 (Erreur d'adressage)**.** *Soit le programme suivant (*`~boudes/TP3/p3.c`*) :*

```
#include <stdio.h>

/* Les lignes commentees ont ete utilisees pour trouver l'indice du
   tableau qui permet de declencher l'erreur.  */

// int *p = NULL;

void erreur(){
  int tableau[16];
  tableau[36] = 5; /* mac os 10.3.9 G3 */
  tableau[25] = 5; /* environnement F206, F207, G215*/
  // printf("%d\n",p - tableau);
}

int main(){
  int note;
  note = 18;
  // p = &note;
  erreur();
  printf("note = %d\n", note);
  exit(0);
}
```

Qu'est censée afficher la fonction `main` *de ce programme ? Tester – en le compilant avec la commande* `gcc p3.c -o p3` *(sans autres options). Expliquer l'erreur.*

Correction 7. Avec un peu de chance l'exécution donne `note = 5`. En fait, comme le suggèrent les commentaires on a d'abord fait afficher la différence entre l'adresse (sur la pile) de `note` et celle de `t` lors de l'appel de `erreur()`. Puis, sans modifier la pile (variables locales, etc.), on a adressé `note` à partir de `t` en utilisant cette différence (ici 25). On aurait pu écrire quelque chose comme `t[(int) (note - t)]` au lieu de `t[25]`. La morale est que : 1) en C, on peut très bien déclarer un tableau de taille n et y écrire en dehors de l'intervalle d'indices $0, \ldots, n-1$; 2) on peut provoquer des erreurs très difficiles a détecter en manipulant les pointeurs ; 3) comme la pile croît du *haut* vers le *bas*, un dépassement d'indice dans un tableau affecte une donnée empilée avant le tableau.

2 Allocation dynamique de la mémoire

En général, avant l'exécution d'un programme, il y a des données dont on ne connaît pas à l'avance la taille qu'elles occuperont en mémoire principale. Il faut donc pouvoir réserver de nouvelles zones mémoire au cours de l'exécution. En C, cette allocation dynamique de la mémoire est à la charge du programmeur. On dispose pour cela des fonctions `malloc()` (allocation d'une portion contiguë de mémoire), `calloc()` (allocation d'une portion contiguë de mémoire et mise à zéro), `realloc()` (redimensionnement) et `free()` (libère une zone dont on a plus besoin) de la bibliothèque standard `stdlib.h`.

Manipulation 6. *Lire la page de manuel de malloc (*`man malloc`*).*

Manipulation 7. *Écrire un programme C qui :*

1. *demande un entier n à l'utilisateur ;*

2. *alloue dynamiquement un tableau* `t` *de* n *cases et le remplie avec les entiers de* 1 *à* n *;*
3. *alloue dynamiquement un second tableau* `u` *de* n *entiers, remplie de* 0 *;*
4. *demande une nouvelle taille de tableau* k *à l'utilisateur ;*
5. *avec* `realloc()`*, change la taille du tableau* `t` *à* k*.*
6. *libère la mémoire prise par* `t` *et* `u`*.*

Selon les cas ($n > k$ *et* $n < k$*, essayer plusieurs valeurs) :*
- *quelle est le contenu du tableau* `t` *(aux indices* $0, \ldots, k-1$*) ?*
- *L'adresse du tableau* `t` *(c'est à dire, la valeur du pointeur* `t`*) change-t-elle en cours d'exécution ? (Expliquer.)*

Correction 8. Le tableau `t` contient les entiers de 1 **à** n **puis des** 0 **jusqu'à l'indice** k **lorsque** $n < k$ **(attention ces zéros ne proviennent pas forcément du `calloc()`). Lorsque** k **est suffisamment grand, `t` change d'adresse. Parce qu'il n'y a plus la place d'étendre `t` de manière contiguë, à partir de son adresse courante.**

Les fonctions `malloc()` et `calloc()` peuvent échouer à allouer l'espace mémoire demandé. Dans ce cas, elles renvoient l'adresse spéciale `NULL`, qui ne correspond à aucun emplacement mémoire utilisable.

Manipulation 8 (Tester ses malloc)**.** *Imaginer un petit programme qui fasse échouer* `malloc()` *et le tester. Lorsque* `malloc()` *échoue (en rendant* `NULL`*) est-ce que le programme s'arrête ?*

Correction 9.
```
#include <stdlib.h>
#include <stdio.h>

#define Go 1024 * 1024 * 1024

int main(){
  printf("malloc : %p\n", malloc(Go));
  printf("malloc : %p\n", malloc(Go));
  printf("malloc : %p\n", malloc(Go));
  printf("malloc : %p\n", malloc(Go));
  printf("malloc : %p\n", malloc(Go));
  return(0);
}
```
Le programme ne s'arrête pas sur les échecs de `malloc`.

Lorsqu'on alloue de la mémoire dynamiquement il faut :
- vérifier qu'on obtient effectivement la zone demandée, et prévoir une action dans le cas où on ne l'obtient pas, comme, par exemple, dans la portion de code suivante :
```
#include <assert.h>
...
  char * p = malloc(1024*1024*1024*sizeof(char));
  assert(p != NULL);
...
```
- penser à libérer la mémoire allouée lorsqu'on en a plus besoin, avec `free()`.

Manipulation 9 (Piles d'entiers)**.** *On se propose d'écrire des fonctions pour manipuler des piles d'entiers (sur le principe des listes simplement chaînées). Soient les types :*

```
typedef struct cellule {
              int contenu; /* entier */
  struct cellule *suivant; /* pointeur vers la cellule suivante */
} cellule;

typedef struct pile {
```

```
       int cardinal; /* nombre d'elements */
  cellule *premier; /* pointeur vers la cellule du premier element */
} pile;
```

Écrire : une fonction qui crée une pile vide ; une fonction qui rend le cardinal d'une pile ; une fonction `estvide()` *qui teste si une pile est vide ; une fonction* `empiler()` *qui permet d'empiler un entier sur une pile existante ; une fonction* `depiler()` *qui supprime le dernier entier empilé sur la pile et rend sa valeur ; une fonction qui libère l'espace mémoire occupé par une pile ; une fonction qui affiche les éléments d'une pile du dernier au premier et une qui les affiche dans l'ordre d'empilement. On pourra s'aider du fichier* `~boudes/TP/p4.c` *et compléter les trous.*

Fichier p4corr.c :

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

typedef struct cellule {
              int contenu; /* lignes */
  struct cellule *suivant; /* pointeur vers la cellule suivante */
} cellule;

typedef struct pile {
       int cardinal; /* nombre d'elements */
  cellule *premier; /* pointeur vers la cellule du premier element */
} pile;

pile *nouvelle_pile(){
  pile *p = malloc(sizeof(pile));
  assert(p != NULL);
  p->cardinal = 0;
  p->premier = NULL;
  return p;
}

int cardinal(pile *p){
  return(p->cardinal);
}

int estvide(pile *p){
  /* rend 1 si la pile est vide, 0 sinon */
  if (p->cardinal == 0) return 1;
  else return 0;
}


void empiler(pile *p, int n){
  /* Ajoute n en haut de la pile p */
  cellule *c = malloc(sizeof(cellule));
  assert(c != NULL);
  c->suivant = p->premier;
  c->contenu = n;
  p->premier = c;
  p->cardinal++;
}
```

```
int depiler(pile *p){
  /* Enleve le dernier element de la pile et rend sa valeur */
  int n;
  cellule *c;
  assert(estvide(p) == 0);
  c = p->premier;
  n = c->contenu;
  p->premier = c->suivant;
  p->cardinal--;
  free(c);
  return(n);
}

void liberer(pile *p){
  /* libere toute la memoire utilisee par p */
  while (estvide(p) != 1) depiler(p);
  free(p);
}

int afficher(pile *p){
  /* Affiche la liste des elements de la pile du dernier au premier */
  cellule *c;
  if (estvide(p)){
    printf("pile vide\n");
    return(0);
  }
  c = p->premier;
  while (c != NULL) {
    printf("%d, ", c->contenu);
    c = c->suivant;
  }
  printf("\n");
}

void afficher2(pile *p){
  /* Affiche la liste des elements de la pile du premier au dernier */
}

int main(){
  /* on teste ces fonctions */
  pile *p, *q;
  p = nouvelle_pile();
  empiler(p, -5);
  empiler(p, 3);
  empiler(p, 8);
  empiler(p, 1);
  printf("estvide(p) = %d\n", estvide(p));
  afficher(p);
  //  afficher2(p);
  q = nouvelle_pile();
  printf("estvide(q) = %d\n", estvide(q));
  afficher(q);
  empiler(q, depiler(p));
```

```
  empiler(q, depiler(p));
  afficher(q);
  afficher(p);
  liberer(q);
  liberer(p);
  exit(0);
}
```