

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`?

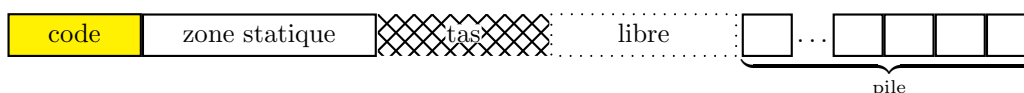
Les adresses sont ordonnées linéairement : on peut représenter la mémoire comme un tableau à une dimension (une suite de cases) dont les cases contiennent les données et où les adresses sont les indices. Une case mémoire est faite de plusieurs octets (4 sur un système 32 bits). Une donnée peut elle-même 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`?*

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 imbriqué */
    }
}

int main(){
    void (*f)(int); /* variable de type fonction */
    f = unefonction;
    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 ?

Questions sur `mafonction()`. La fonction `mafonction()` 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 variable `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.

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 ?

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.

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 */
    tableau[25] = 5; /* environnement F206, F207 */
    // 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.

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 remplit 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 $0, \dots, k - 1$) ?

L'adresse du tableau t (c'est à dire, la valeur du pointeur t) change-t-elle en cours d'exécution ? (Expliquer.)

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 ?*

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 p4.c :

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

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;

pile *nouvelle_pile(){
    /* Cree une pile vide et renvoie son adresse */
}

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

int estvide(pile *p){
    /* rend 1 si la pile est vide, 0 sinon */
}

void empiler(pile *p, int n){
    /* Ajoute n en haut de la pile p */
}

int depiler(pile *p){
    /* Enleve le dernier element de la pile et rend sa valeur */
}

void liberer(pile *p){
    /* libere toute la memoire utilisee par p */
}

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

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