Structures et listes chaînées en C

Lycée Louis-le-Grand

Alias de types en C

```c
typedef int age;
```

```c
bool can_vote(age a) {
    return a >= 18;
}
```

But cosmétique (documentation) et modularité

Cas des booléens

Dans `stdbool.h` :

```c
typedef _Bool bool;
```

Remarque : depuis C23, **`bool`** est un type officiel !

Structures

```c
struct displacement {
    double n_s;
    double e_w;
};
```

```c
struct displacement d;
d.n_s = 10.2;
d.e_w = 4.5;
```

Initialisation immédiate, copie

```c
struct displacement d = { .n_s = 10.2, .e_w = 4.5 };
```

Mais seulement pour l'initialisation :

```c
d = { .n_s = 10.2, .e_w = 4.5 }; // ILLEGAL
```

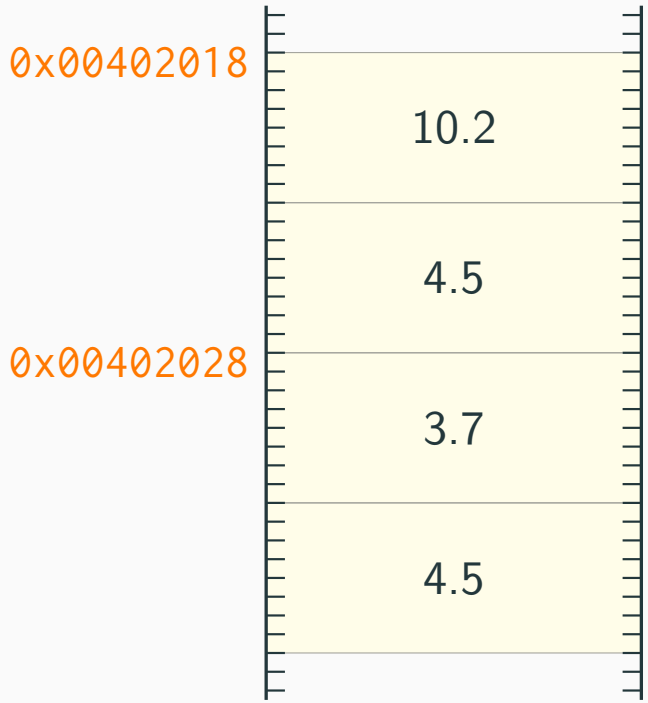
Totalement hors programme :

```c
d = (struct displacement){ .n_s = 10.2, .e_w = 4.5 };
```

Copie

```c
struct displacement d = { .n_s = 10.2, .e_w = 4.5 };

struct displacement d2 = d;    // <- données *copiées*
d2.n_s = 3.7;                  // <- n'affecte pas dep !
```



Mémoire

```
d : struct displacement
  0x00402018
d2 : struct displacement
  0x00402028
```

Test d'égalité

comparaison illégale avec ==

```c
bool displ_equals(struct displacement d1,
                  struct displacement d2) {
    return d1.n_s == d2.n_s && d1.e_w == d2.e_w;
}
```

Structures et alias

```c
struct displacement {
    double n_s;
    double e_w;
};

typedef struct displacement displacement;
```

```c
typedef struct displacement {
    double n_s;
    double e_w;
} displacement;
```

Structures anonymes

```c
typedef struct {
    double n_s;
    double e_w;
} displacement;
```

Complètement anonyme :

```c
struct { double n_s; double e_w; } d;
```

Énumérations (HP)

```c
enum direction { north, south, east, west };
```



```c
enum direction dir;
dir = south;
```

Variantes d'écriture

```c
enum direction { north, south, east, west };
typedef enum direction direction;
```

```c
typedef enum direction { north, south, east, west }
                                                direction;
```

```c
typedef enum { north, south, east, west } direction;
```

```c
enum { north, south, east, west } dir;
```

Types complexes

```
type mytype = I of int | F of float
```

Peut se traduire en

```c
typedef struct {
    enum { integer, floatingpoint } type;
    int i;
    double f;
} mytype;
```

Types complexes

Ou plus économe en mémoire :

```c
typedef struct {
    enum { integer, flotingpoint } type;
    union {
        int i;
        double f;
    } data;
} mytype;
```

Listes chaînées

```c
struct int_list {
    int value;
    struct int_list* next;
};
```

Déclaration avec :

```c
struct int_list mylist;
```

Listes chaînées

```c
struct int_list {
    int value;
    struct int_list* next;
};

typedef struct int_list int_list;
```

Déclaration avec :

```c
int_list mylist;
```

Identifier l'extrémité

Plusieurs solutions !

Identifier la tête

Encore plus de solutions !

Dans la suite

```c
struct cell {
    int value;
    struct cell* next;
};

typedef struct cell int_list;
```

Ajout à gauche

```c
int_list* add_left(int elem, int_list* p_list) {
    int_list* p_res = malloc(sizeof(int_list));
    (*p_res).value = elem;
    (*p_res).next = p_list;
    return p_res;
}
```

Obtenir la tête

```c
int head(int_list* p_list) {
    return (*p_list).value;
}
```

```c
int head(int_list* p_list) {
    assert(p_list != NULL);

    return (*p_list).value;
}
```

Obtenir la queue

```c
int_list* tail(int_list* p_list, bool free_head) {
    assert (p_list != NULL);

    int_list* p_res = (*p_list).next;
    if (free_head) {
        free(p_list);
    }
    return p_res;
}
```

Calculer la longueur

```c
int length(int_list* p_list) {
    if (p_list == NULL) {
        return 0;
    }
    return 1 + length((*p_list).next);
}
```

Calculer la longueur


```c
int length(int_list* p_list) {
    int count = 0;
    int_list* ptr = p_list;
    while (ptr != NULL) {
        count = count + 1;
        ptr = (*ptr).next;
    }
    return count;
}
```

Calculer la longueur

```c
int length(int_list* p_list) {
    int count = 0;
    for (int_list* ptr = p_list; ptr != NULL;
                                 ptr = (*ptr).next) {
        count = count + 1;
    }
    return count;
}
```

Présence d'un élément

```c
bool member(int_list* p_list, int elem) {
    for (int_list* ptr = p_list; ptr != NULL;
                                 ptr = (*ptr).next) {
        if ((*ptr).value == elem) {
            return true;
        }
    }
    return false;
}
```

Index d'un élément


```c
int index(int_list* p_list, int elem) {
    int count = 0;
    for (int_list* ptr = p_list; ptr != NULL;
                                 ptr = (*ptr).next) {
        if ((*ptr).value == elem) {
            return count;
        }
        count = count + 1;
    }
    return -1;
}
```

Retournement


```c
int_list* reverse(int_list* p_list) {
    int_list* p_res = NULL;
    for (int_list* ptr = p_list; ptr != NULL;
                                 ptr = (*ptr).next) {
        int_list* tmp = malloc(sizeof(int_list));
        (*tmp).value = (*ptr).value;
        (*tmp).next = p_res;
        p_res = tmp;
    }
    return p_res;
}
```

Destruction

```c
void delete(int_list* p_list) {
    while (p_list != NULL) {
        int_list* tmp = p_list;
        p_list = (*p_list).next;
        free(tmp);
    }
}
```

```c
void delete(int_list* p_list) {
    if (p_list != NULL) {
        delete((*p_list).next);
        free(p_list);
    }
}
```

Les listes sont mutables !

```c
void increment(int_list* p_list) {
    for (int_list* ptr = p_list; ptr != NULL;
                                 ptr = (*ptr).next) {
        (*ptr).value = (*ptr).value + 1;
    }
}
```

```
# let rec increment = function
    | [] -> []
    | h::t -> h+1::increment t;;

val increment : int list -> int list = <fun>
```

Ajout... à droite !


```c
void add_right(int elem, int_list* p_list) {
    assert( p_list != NULL )

    while ((*p_list).next != NULL) {          // On recherche le
        p_list = (*p_list).next               // dernière cellule
    }
    int_list* tmp = malloc(sizeof(int_list)); // On crée une
    (*tmp).value = elem                       // nouvelle cellule
    (*tmp).next = NULL                        // et on l'accroche
    (*p_list).next = tmp;                     // en bout de liste
}
```

Copie de liste


```c
void copy(int_list* p_list) {
    int_list* p_copy = NULL;
    int_list* p_last = NULL;
    for (int_list* ptr = p_list; ptr != NULL; ptr = (*ptr).next) {
        int_list* tmp = malloc(sizeof(int_list));
        (*tmp).value = (*ptr).value;
        (*tmp).next = NULL;
        if (p_copy == NULL) { // Si c'est la première cellule
            p_copy = tmp;
        } else {
            (*p_last).next = tmp;
        }
        p_last = tmp;
    }
    return p_copy;
}
```