Introduction à la Programmation des Algorithmes

2.4. Langage C – Tableaux et structures

François Fleuret

https://fleuret.org/11x001/

UNIVERSITÉ
DE GENÈVE

Nous n'avons vu et utilisé jusqu'ici que des types de base, et nous nous sommes limités à trois d'entre eux

- `int`,
- `float`, et
- `char`.

Nous n'avons vu et utilisé jusqu'ici que des types de base, et nous nous sommes limités à trois d'entre eux

- `int`,
- `float`, et
- `char`.

Dans de nombreuses situations, il est nécessaire de créer de nouveaux types pour représenter des valeurs ou des objets plus complexes:

- grandeurs mathématiques (vecteur, matrice, polynôme, etc.),
- médias (échantillons sonore, image, etc.),
- éléments d'une base de donnée (client, article, etc.)

Nous allons passer en revue les deux principales manière de créer des variables qui combinent plusieurs éléments de types de base:

- les **tableaux**, qui sont simplement composés de plusieurs éléments de même type, qui peuvent être référencés à l'aide d'un indexe entier, et
- les **structures**, composées de plusieurs champs de types différents et nommés.

Nous utiliserons dans certains exemples l'opérateur `sizeof` qui retourne le nombre d'octets qu'une variable occupe en mémoire (la valeur retournée est du type `size_t` qui est un entier positif).

```
1  int n;
2  float x;
3  char c;
4
5  printf("%d %d %d\n", sizeof(n), sizeof(x), sizeof(c));
```

affiche

```
4 4 1
```

Nous utiliserons dans certains exemples l'opérateur `sizeof` qui retourne le nombre d'octets qu'une variable occupe en mémoire (la valeur retournée est du type `size_t` qui est un entier positif).

```
1  int n;
2  float x;
3  char c;
4
5  printf("%d %d %d\n", sizeof(n), sizeof(x), sizeof(c));
```

affiche

```
4 4 1
```

Cet opérateur peut aussi être directement appliqué à un type, par exemple `sizeof(int)` retourne le nombre d'octets qu'occupe en mémoire une variable de type `int`.

Les tableaux

Un **tableau** est défini par un type et un nombre d'éléments et ces derniers peuvent ensuite être référencés à l'aide d'un indexe entier. La déclaration et la manipulation d'un tableau se fait avec l'opérateur `[]`.

Un **tableau** est défini par un type et un nombre d'éléments et ces derniers peuvent ensuite être référencés à l'aide d'un indexe entier. La déclaration et la manipulation d'un tableau se fait avec l'opérateur `[]`.

Par exemple, pour déclarer un tableau de 10 entiers et remplir ensuite ce tableau des valeurs de 0 à 9, il suffit de faire.

```
1  int a[10];
2
3  for(int k = 0; k < 10; k++) {
4    a[k] = k;
5  }
```

La ligne 1 est une déclaration d'un tableau a contenant 10 éléments de type `int`. Les lignes 3–5 définissent une boucle qui répète la ligne 4 qui met la valeur k dans l'élément d'indice k.

Pour la déclaration d'une variable, on a utilisé la syntaxe

```
truc nom_de_variable;
```

qui indique à l'ordinateur de réserver une zone mémoire à laquelle on fera référence avec `nom_de_variable` et qui sera manipulée comme étant du type `truc`.

Pour la déclaration d'une variable, on a utilisé la syntaxe

```
truc nom_de_variable;
```

qui indique à l'ordinateur de réserver une zone mémoire à laquelle on fera référence avec `nom_de_variable` et qui sera manipulée comme étant du type `truc`.

Pour la déclaration d'une variable de type tableau, une déclaration

```
truc nom_de_variable[nombre];
```

indique à l'ordinateur de réserver une zone mémoire à laquelle on fera référence avec `nom_de_variable`, et correspondant à `nombre` variables de type `truc` rangées en mémoire consécutivement.

Étant donné un tableau déjà déclaré, la syntaxe `nom_de_variable[k]` où `k` est un entier, fait référence à la variable d'indexe k dans le tableau.

Là encore le C est très primitif: il va associer en interne à `nom_de_variable` l'adresse du premier élément du tableau et l'opérateur `[]` va ajouter à cette adresse k multiplié par le nombre d'octets qu'occupe un élément.

L'opérateur [] pour indexer une valeur peut évidemment être combiné avec les opérateurs que nous avons vus précédemment. Il a une très haute priorité, la même que ()

```
() []
++ -- - (unaire)

* / %
+ - (binaire)
< <= > >=
== !=
&&
||
= += -= *= /= %=
```

```c
int nb[7];

for(int k = 0; k < 7; k++) {
  nb[k] = k;
}

for(int k = 0; k < 7; k++) {
  if(nb[k] % 2 == 0) nb[k]++;
}

for(int k = 0; k < 7; k++) {
  printf("nb[%d]=%d\n", k, nb[k]);
}
```

affiche

```c
int nb[7];

for(int k = 0; k < 7; k++) {
  nb[k] = k;
}

for(int k = 0; k < 7; k++) {
  if(nb[k] % 2 == 0) nb[k]++;
}

for(int k = 0; k < 7; k++) {
  printf("nb[%d]=%d\n", k, nb[k]);
}
```

affiche

```
nb[0]=1
nb[1]=1
nb[2]=3
nb[3]=3
nb[4]=5
nb[5]=5
nb[6]=7
```

Lorsqu'il est appliqué à un tableau, l'opérateur `sizeof` retourne l'occupation totale en mémoire, qui est égale à la taille en octets d'un élément multipliée par le nombre d'éléments.

```
1  int mes_entiers[10];
2  char mes_caracteres[10];
3  printf("%d %d\n", sizeof(mes_entiers), sizeof(mes_caracteres));
```

affiche

Lorsqu'il est appliqué à un tableau, l'opérateur `sizeof` retourne l'occupation totale en mémoire, qui est égale à la taille en octets d'un élément multipliée par le nombre d'éléments.

```
1  int mes_entiers[10];
2  char mes_caracteres[10];
3  printf("%d %d\n", sizeof(mes_entiers), sizeof(mes_caracteres));
```

affiche

```
40 10
```

Lorsqu'il est appliqué à un tableau, l'opérateur `sizeof` retourne l'occupation totale en mémoire, qui est égale à la taille en octets d'un élément multipliée par le nombre d'éléments.

```
1  int mes_entiers[10];
2  char mes_caracteres[10];
3  printf("%d %d\n", sizeof(mes_entiers), sizeof(mes_caracteres));
```

affiche

```
40 10
```

Une manière simple de calculer le nombre d'éléments dans un tableau est de diviser sa taille par celle d'un de ses élément.

```
1  float v[250];
2  printf("%d\n", sizeof(v)/sizeof(float));
```

affiche

Lorsqu'il est appliqué à un tableau, l'opérateur `sizeof` retourne l'occupation totale en mémoire, qui est égale à la taille en octets d'un élément multipliée par le nombre d'éléments.

```
1  int mes_entiers[10];
2  char mes_caracteres[10];
3  printf("%d %d\n", sizeof(mes_entiers), sizeof(mes_caracteres));
```

affiche

```
40 10
```

Une manière simple de calculer le nombre d'éléments dans un tableau est de diviser sa taille par celle d'un de ses élément.

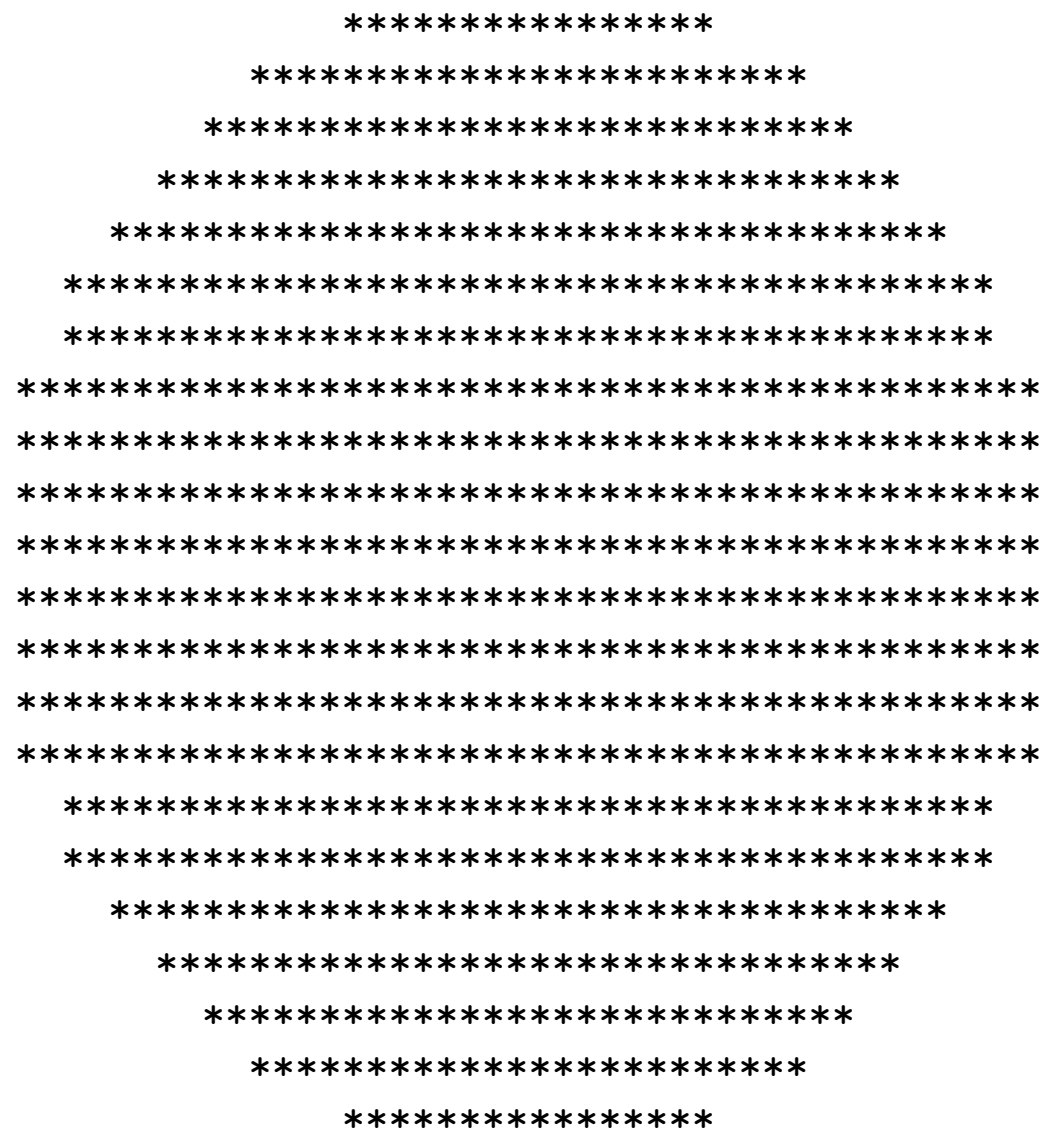
```
1  float v[250];
2  printf("%d\n", sizeof(v)/sizeof(float));
```

affiche

```
250
```

L'opérateur [] peut être utilisé plusieurs fois successivement, permettant ainsi de faire des tableaux à plusieurs dimensions.

```
1   int n = 25;
2   int img[n][n];
3
4   for(int i = 0; i < n; i++) {
5     for(int j = 0; j < n; j++) {
6       float x = 2 * (float) i / (float) n - 1.0;
7       float y = 2 * (float) j / (float) n - 1.0;
8       img[i][j] = x * x + y * y <= 0.9 * 0.9;
9     }
10  }
11
12  for(int i = 0; i < n; i++) {
13    for(int j = 0; j < n; j++) {
14      if(img[i][j]) printf("**");
15      else printf("  ");
16    }
17    printf("\n");
18  }
```



Le C ne contrôle rien. Des erreurs extrêmement graves se produisent si par accident un programme essaye d'accéder à un élément en dehors du tableau, avec un indexe trop grand ou négatif.

Un tel accès en dehors d'un tableau peut:

- ne rien faire,
- accéder à / modifier une autre variable située à côté en mémoire,
- crasher le programme au niveau du système d'exploitation ("segmentation fault"),
- se comporter différemment sur différentes machines et/ou avec différents compilateurs.

Un cas simple

```
1   int main(void) {
2     int a[2], b = 1;
3
4     a[0] = 2;
5     printf("b=%d\n", b);
6     a[1] = 2;
7     printf("b=%d\n", b);
8     a[2] = 2;
9     printf("b=%d\n", b);
10
11    return 0;
12  }
```

affiche

```
b=1
b=1
b=2
```

Comme la zone mémoire allouée à un programme est par défaut assez large, le système ne détecte pas de problème si on reste dans cette zone.

```c
int main(void) {
  int a[2];

  a[1000] = 2;
  printf("ping\n");
  a[2000] = 2;
  printf("pong\n");

  return 0;
}
```

affiche

```
ping
Segmentation fault (core dumped)
```

L'opérateur d'affectation permet d'initialiser les valeurs dans un tableau au moment de sa déclaration en utilisant l'opérateur {}:

```
1    int k[3] = { 8, 4, 2 };
```

et le compilateur est même capable de déterminer seul la taille du tableau

```
1    int k[] = { 8, 4, 2 };
```

Pour les tableaux de caractères, la même syntaxe avec {} est possible:

```
1    char word[] = { 'b', 'l', 'a', 'h' };
```

mais une syntaxe avec une chaîne de caractères entre " est plus compacte:

```
1    char word[] = "blah";
```

Par convention dans ce dernier cas, un caractère avec le code zéro est rajouté à la fin. Le tableau `word` ici contient donc 5 éléments.

En revanche, l'opérateur d'affectation ne permet pas de recopier un tableau dans un autre:

```c
#include <stdio.h>

int main(void) {
  int t1[3], t2[3];
  t1 = t2;

  return 0;
}
```

```
test.c: In function ‘main’:
test.c:7:6: error: assignment to expression with array type
   t1 = t2;
      ^
```

Exemple: le **Crible d'Ératosthène**

Cet algorithme permet de déterminer les nombres premiers plus petits qu'un entier maximum par éliminations successives.

On va utiliser un tableau avec un booléen par entier qui indique quel nombre est premier:

- on l'initialise avec "vrai" pour toutes les entiers sauf 0 et 1,
- on le parcours, et pour chaque entier n qui n'a pas encore été mis à "faux" (et qui est donc premier) on met à "faux" toutes les entiers du tableau de la forme $k \cdot n,\ n \geq 2$.

Exemple: le **Crible d'Ératosthène**

Cet algorithme permet de déterminer les nombres premiers plus petits qu'un entier maximum par éliminations successives.

On va utiliser un tableau avec un booléen par entier qui indique quel nombre est premier:

- on l'initialise avec "vrai" pour toutes les entiers sauf 0 et 1,
- on le parcours, et pour chaque entier n qui n'a pas encore été mis à "faux" (et qui est donc premier) on met à "faux" toutes les entiers du tableau de la forme $k \cdot n,\ n \geq 2$.

En pratique, lors de cette seconde boucle:

- 0 et 1 sont "faux",
- 2 est "vrai", on met donc à "faux" tous ses multiples,
- 3 est "vrai", donc on met à "faux" tous ses multiples,
- 4 est "faux" car c'est un multiple de 2,
- 5 est "vrai" donc on met à "faux" tous ses multiples, etc.
- 6 est "faux" car c'est un multiple de 2 (et 3), etc.

Exemple: le **Crible d'Ératosthène**

```
1   int max = 100;
2   int est_premier[max];
3
4   for(int n = 0; n < max; n++)
5     est_premier[n] = (n >= 2);
6
7   for(int n = 0; n < max; n++)
8     if(est_premier[n])
9       for(int k = 2 * n; k < max; k += n)
10        est_premier[k] = 0;
11
12  for(int n = 0; n < max; n++)
13    if(est_premier[n]) printf("%d\n", n);
```

Lignes 4–5, on initialise le tableau en mettant à "vrai" toutes les valeurs plus grandes que 2.

Exemple: le **Crible d'Ératosthène**

```
1   int max = 100;
2   int est_premier[max];
3
4   for(int n = 0; n < max; n++)
5     est_premier[n] = (n >= 2);
6
7   for(int n = 0; n < max; n++)
8     if(est_premier[n])
9       for(int k = 2 * n; k < max; k += n)
10        est_premier[k] = 0;
11
12  for(int n = 0; n < max; n++)
13    if(est_premier[n]) printf("%d\n", n);
```

Lignes 4–5, on initialise le tableau en mettant à "vrai" toutes les valeurs plus grandes que 2. Lignes 7–10 on parcourt tout le tableau, et si une valeur est "vrai" (test ligne 8) on met à "faux" tous ses multiples (lignes 9–10).

Exemple: le **Crible d'Ératosthène**

```
1   int max = 100;
2   int est_premier[max];
3
4   for(int n = 0; n < max; n++)
5     est_premier[n] = (n >= 2);
6
7   for(int n = 0; n < max; n++)
8     if(est_premier[n])
9       for(int k = 2 * n; k < max; k += n)
10        est_premier[k] = 0;
11
12  for(int n = 0; n < max; n++)
13    if(est_premier[n]) printf("%d\n", n);
```

Lignes 4–5, on initialise le tableau en mettant à "vrai" toutes les valeurs plus grandes que 2. Lignes 7–10 on parcourt tout le tableau, et si une valeur est "vrai" (test ligne 8) on met à "faux" tous ses multiples (lignes 9–10). On affiche finalement tous les entiers "vrai" lignes 12–13.

Les structures

Une **structure** permet de déclarer des variables composées de plusieurs champs, chacun avec un type et un nom. Ce type de déclaration se fait en C avec le mot clé `struct`, et la référence à un champ particulier se fait avec l'opérateur `.`

Par exemple, pour créer une variable `ma_maison` qui représente un bâtiment avec un champ `surface` et un champ `nb_de_pieces`, et ensuite mettre les valeur 110 et 4 dans les champs, on peut faire:

```
1  struct {
2    float surface_habitable;
3    int nb_de_pieces;
4  } ma_maison;
5
6  ma_maison.surface_habitable = 110.0;
7  ma_maison.nb_de_pieces = 4;
```

Dans l'exemple précédent, nous aurions pu définir deux variables `ma_maison_surface_habitable` et `ma_maison_nb_de_pieces`.

En pratique, les structures sont utilisées quand on a plusieurs variables similaires, pour imposer que tous les champs sont présents et identifiés de manière identique.

Cela peut se faire en définissant un **modèle de structure** d'abord, et en déclarant ensuite des variables selon ce modèle:

```
1  struct Batiment {
2    float surface_habitable;
3    int nb_de_pieces;
4  };
5
6  struct Batiment ma_maison, la_maison_du_voisin;
```

Donc nous utiliserons `struct` des deux façons suivantes:

```
struct NomDeStructure {
  type identifiant;
  ...
};
```

pour définir un **modèle** de structure avec une liste de champs, chacun avec un type et un identifiant

Donc nous utiliserons `struct` des deux façons suivantes:

```
struct NomDeStructure {
  type identifiant;
  ...
};
```

pour définir un **modèle** de structure avec une liste de champs, chacun avec un type et un identifiant, et

```
struct NomDeStructure nom_de_variable;
```

pour indiquer à l'ordinateur de réserver une zone mémoire où placer les champs définis par `NomDeStructure` et auxquels on fera référence dans la suite avec `nom_de_variable.identifiant`.

Comme attendu, la taille en mémoire d'une structure est la somme des tailles de ses champs.

```
1  struct Utilisateur {
2    int numero_de_client;
3    char identifiant[12];
4    char mot_de_passe[32];
5  };
6
7  struct Utilisateur u;
8
9  printf("%d\n", sizeof(u));
```

affiche

Comme attendu, la taille en mémoire d'une structure est la somme des tailles de ses champs.

```
1  struct Utilisateur {
2    int numero_de_client;
3    char identifiant[12];
4    char mot_de_passe[32];
5  };
6
7  struct Utilisateur u;
8
9  printf("%d\n", sizeof(u));
```

affiche

48

Les tableaux et les structures peuvent être combinés: on peut faire un tableau de structures, et les champs d'une structure peuvent être des tableaux ou des structures. Dans le cas de structures imbriquées, on peut enchaîner les opérateurs `.`

```
1   struct Utilisateur {
2     int numero_de_client;
3     char identifiant[12];
4     char mot_de_passe[32];
5   };
6
7   struct Equipe {
8     int nom[12];
9     struct Utilisateur membres[8];
10  };
11
12  struct Equipe e;
13
14  e.membres[0].numero_de_client = 123;
```

L'opérateur . pour faire référence à un champ a une très haute priorité, la même que ()

```
() [] .
++ -- - (unaire)

* / %
+ - (binaire)
< <= > >=
== !=
&&
||
= += -= *= /= %=
```

L'opérateur d'affectation permet d'initialiser les valeurs dans une structure lors de sa déclaration, en listant les valeurs des champs entre {}:

```
1   struct Rectangle {
2     float hauteur, largeur;
3   };
4
5   struct Rectangle r = { 3.0, 2.5 };
6
7   printf("%f %f\n", r.hauteur, r.largeur);
```

affiche

```
3.000000 2.500000
```

Et l'opérateur d'affectation permet de recopier une structure dans une autre (y compris si elles ont des champs tableaux!):

```
1  struct Rectangle {
2    float hauteur, largeur;
3  };
4
5  struct Rectangle r = { 3.0, 2.5 }, t;
6
7  t = r;
```

Le mot clé `typedef` permet d'associer un identifiant à un type pré-existant pour clarifier le code source. Par ex.

```
typedef float temperature;
```

Le mot clé `typedef` permet d'associer un identifiant à un type pré-existant pour clarifier le code source. Par ex.

```
typedef float temperature;
```

Il peut être utilisé en particulier pour simplifier l'utilisation des `struct`:

```
1  typedef struct {
2    float rouge, vert, bleu;
3  } couleur;
4
5  int main(void) {
6    couleur jaune = { 1.0, 1.0, 0.0 };
7
8    return 0;
9  }
```

Donc on notera la différence entre

```
1  struct {
2    float rouge, vert, bleu;
3  } couleur;
```

qui déclare une variable `couleur` qui est une structure avec trois champs de type `float`, et

```
1  typedef struct {
2    float rouge, vert, bleu;
3  } couleur;
```

qui définit un nouveau type `couleur` qui est une structure avec trois champs de type `float`, mais ne déclare pas de variable.

Fin