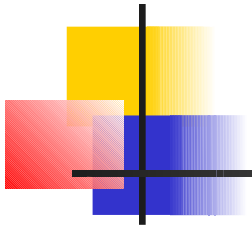
# Cours 4



# Les itérations

- Une itération est la répétition d'un traitement un certain nombre de fois
- On classe les itérations en deux catégories suivant que :
  - On connaît à l'avance le nombre de traitements à effectuer
  - On ne connaît pas ce nombre

# Moyenne des capacités d'un lot de condensateur

- Supposons que l'on désire calculer la moyenne M des capacités des 10 condensateurs issus du lot. On a M = (C1 + C2 + … + C10) / 10
- Voici un programme possible :

```
float c[10];
float moyenne;


// lecture des capacités …
moyenne = 0.1*(c[0]+ c[1]+ c[2]+ c[3]+ c[4]+ c[5]+ c[6]+ c[7]+ c[8]+ c[9]);
```

# Moyenne des capacités d'un lot de condensateur

- Mais que faire si la taille de c est 1000 !!
- Il faut trouver un moyen itératif de calculer la somme des capacités
- Pour cela on initialise une variable s (somme) à 0 et l'on accumule dans s les valeurs du tableau. En Français cela donnerait :

  *pour i de 0 à 9 faire s ← s + c[i]*

- Le traitement *s* ← s + c[i] est itéré 10 fois pour les valeurs successives de i allant de 0 à 9. A la fin de cette itération s contient la somme des 10 capacités

# Moyenne des capacités d'un lot de condensateur

- Notre itération se traduit directement en C :

```
int i;
float somme = 0;

for (i=0; i<10; i++) {
    somme = somme + c[i];
}
```

# Moyenne des capacités d'un lot de condensateur

- Au départ la variable **`i`** appelé « compteur » est initialisée à 0
- A chaque pas, si l'inégalité **`i<10`** est vérifiée le traitement est effectué puis **`i`** est incrémenté de 1
- On fait donc en tout 10 fois le traitement avec les valeurs successives 0, 1, …, 9 pour **`i`**

# Moyenne des capacités d'un lot de condensateur

- Faisons « tourner » à la main le programme :

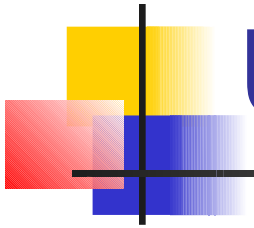
| `i` | **somme** (après l'affectation) |
|---|---|
| 0 | `c[0]` |
| 1 | `c[0] + c[1]` |
| 2 | `c[0] + c[1] + c[2]` |
| 3 | `c[0] + c[1] + c[2] + c[3]` |
| 4 | `c[0] + c[1] + c[2] + c[3] + c[4]` |
| 5 | `c[0] + c[1] + c[2] + c[3] + c[4] + c[5]` |
| 6 | `c[0] + c[1] + c[2] + c[3] + c[4] + c[5] + c[6]` |
| 7 | `c[0] + c[1] + c[2] + c[3] + c[4] + c[5] + c[6] + c[7]` |
| 8 | `c[0] + c[1] + c[2] + c[3] + c[4] + c[5] + c[6] + c[7] + c[8]` |
| 9 | `c[0] + c[1] + c[2] + c[3] + c[4] + c[5] + c[6] + c[7] + c[8] + c[9]` |

# Utilisation du `for`

- Le **`for`** de C est une structure itérative très riche mais nous en ferons une utilisation réduite dans ce cours :

```
int n;
for (n=min; n<max; n++)
      instruction;
```

```
int n;
for (n=max; n>min; n--)
      instruction;
```



# Utilisation du `for`

- Dans le premier cas (**`n++`**) :
  - *instruction* est exécuté successivement (max – min) fois
  - A la 1ère exécution de *instruction*, n contient min, à la 2ème min+1, … , à la (max – min)ème max–1

- Dans le deuxième cas (**`n--`**) :
  - *instruction* est exécuté successivement (max – min) fois
  - A la 1ère exécution de *instruction*, n contient max, à la 2ème max-1, … , à la (max – min)ème min+1

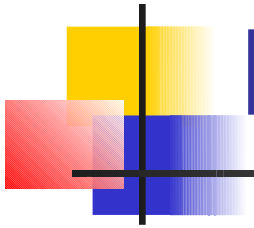
# Exemples de `for`

```
int j;
for (j=7; j>=-1; j--) {
   printf("Bonjour\n");
}
```

imprime 9 fois Bonjour à l'écran en allant à la ligne entre chaque Bonjour

```
int k; double p = 1;
double s = 0;

for (k=1; k<5; k++) {
    p = p * (1.0/k);
    s = s + p;
}
```

calcule la somme 1+ 1/2 + 1/6 + 1/24



# Remarques

- Les conditions d'arrêt **`n <= max -1`**
  (resp. **`n >= min-1`**) et **`n < max`** (resp. **`n > min`**) sont équivalentes

- Le compteur peut ou pas être utilisé dans le traitement itéré du **`for`**

- Il ne faut pas modifier le compteur à l'intérieur du traitement itéré du **`for`** . On ne peut qu'utiliser sa valeur

# `while` et `do while`

- Dans beaucoup de cas on ne connaît pas à l'avance le nombre de traitements à effectuer.

- On a deux cas de figures :

  - Tant qu'une condition est vérifiée, on répète le traitement
  - On répète le traitement jusqu'à ce qu'une condition d' arrêt soit vérifiée (ou tant qu'une condition de continuation est vérifiée)

- En C, les deux cas de figure ont leur instruction correspondante

# `while` et `do while`

```
while (expression)
    instruction
```

```
do
    instruction
while (expression);
```

Remarques :

- Dans le premier cas, on peut ne jamais effectuer de traitement
- Dans le second cas au moins un traitement est effectué

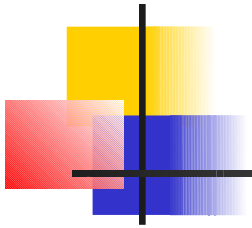
# Pb : capacité inférieure à 100 ?

- Supposons que l'on veut savoir si un condensateur a une capacité inférieure à 100 dans le lot de 10
- Le traitement à effectuer est une comparaison de la capacité à 100
- Au premier condensateur trouvé de capacité inférieure à 100, on a la réponse à notre question et il n'y a pas de raison de continuer les comparaisons

# Pb : capacité inférieure à 100 ?

Voici le programme C résolvant ce problème :

```
char reponse; // stocke le résultat
              // 'n' : non, 'o' : oui
int i = 0;
while (i<10 && c[i]>100) {
    i++;
}
reponse = (i == 10) ? 'n' : 'o';
```



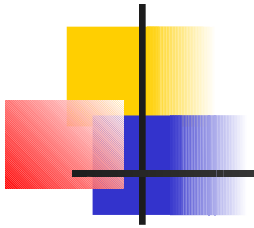
# Exercice

Questions :

1. Peut-on écrire un programme équivalent avec **`do while`** ?

2. Que se passe-t-il si le premier condensateur a une capacité inférieure à 100 ?

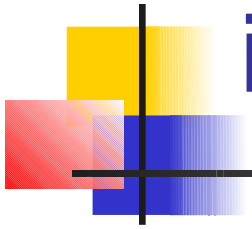
3. Pourquoi n'y a-t-il pas dépassement de tableau quand aucun condensateur n'a une capacité inférieure à 100 ?



# Exercice

Réponses :

# 3.3 Exemples de programmes itératifs



- Filtrage de fréquences
- Calcul de l'exponentielle d'un nombre réel (sans utiliser la fonction **`exp`** de **`math.h`**)
- Crible d'Eratosthène

# Filtrage de fréquences

- On dispose d'un tableau de fréquences comprises entre 0 et 1500 Hz
- On veut filtrer les « hautes »fréquences ( > 1000)
- Méthode de filtrage : si f > 1000, remplacer f par 1000

# Filtrage de fréquences : implantation du programme

- Déclarations des constantes et variables
- Génération aléatoire d'un tableau de fréquences comprises entre 0 et 1500 Hz et affichage des fréquences générées
- Filtrage
- Affichage des fréquences filtrées

# Filtrage de fréquences : implantation du programme

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

#define N 5

int main() {
  int frequences[N]; int i;
  srand(time(0));

  // generation aleatoire des frequences
  for (i=0; i<N; i++)  {
    frequences[i] = rand() % 1501;
  }

  // affichage des frequences generees
  for (i=0; i<N; i++)  {
    printf("%d%s", frequences[i], " ");
  }
  printf("\n");
  ……
```

# Filtrage de fréquences : implantation du programme

```
  // filtrage
  for (i=0; i<N; i++) {
    if (frequences[i] > 1000) {
      frequences[i] = 1000;
    }
  }
  // affichage des frequences filtrees
  for (i=0; i<N; i++) {
     printf("%d%s", frequences[i], " ");
  }
  printf("\n");

  return 0;
}
```

# Exponentielle d'un réel

- Langage C dispose d'une fonction **`exp`** qui renvoie l'exponentielle d'un nombre réel (**`#include <math.h>`**)

- On peut écrire un programme qui calcule l'exponentielle d'un réel sans utiliser cette fonction

- On utilise le fait que la suite
  $e(x)_n = 1 + x + x^2/2! + \ldots + x^n/n!$ tend vers
  exp(x) quand n tend vers l'infini

# Exponentielle d'un réel : implantation du programme

- Déclarations des constantes et variables
- Entrée de la donnée
- Calcul de l'exponentielle
- Affichage du résultat

# Exponentielle d'un réel : implantation du programme

- Déclarations des constantes et variables

```
#include <stdio.h>

const int MAX_ITERATIONS = 100; // nombre max d'iterations

int main() {
  double x; // le nombre dont on cherche l'exponentielle
  double exp_x; // son exponentielle
  double puiss_x; // x^i/i! pour i=0,..., 100
  int i;
  ……
```

# Exponentielle d'un réel : implantation du programme

- Entrée de la donnée

```
……

 // entree du nombre x

 printf("Donnez un nombre reel : ");
 scanf("%lf",&x);

 ……
```

# Exponentielle d'un réel : implantation du programme

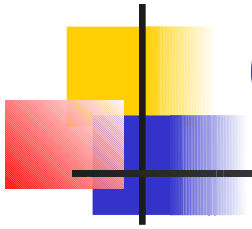
- Calcul de l'exponentielle

```
……
 // initialisation de exp_x et puiss_x
 exp_x = 1;
 puiss_x = 1;

 for (i=1; i<= MAX_ITERATIONS; i++) {

   // on calcule le nouveau terme en fct de l'ancien c-a-d x^i/i!
   puiss_x = puiss_x * (x/i);

   // on ajoute ce terme à exp_x
   exp_x = exp_x + puiss_x;
   // exp_x vaut maintenant 1 + x + ... + x^i/i!
 }
……
```

# Exponentielle d'un réel : implantation du programme

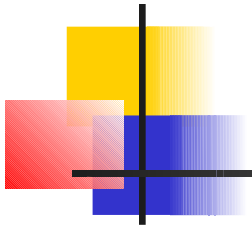
- Impression du résultat

```
……

  printf("%s%f%s%f\n", "exp(", x, ") = ", exp_x);

  return 0;
}
```



# Crible d'Eratosthène

- Un nombre *premier* est un nombre entier ayant deux diviseurs distincts exactement : 1 et lui-même
- Voici la liste des 10 premiers nombres premiers : {2,3,5,7,11,13,17,19,23,29}
- Le crible d'Eratosthène est un algorithme qui permet de donner la liste des nombres premiers inférieurs à un nombre donné N



# Crible d'Eratosthène

Principe de l'algorithme :

- On barre les multiples de 2 (à partir de 4)
- On avance au premier nombre non barré (c'est un nombre premier)
- On barre ses multiples
- On recommence jusqu'à ce que le premier nombre non barré ait son carré strictement supérieur à N
- Les nombres restants non barrés sont les nombres premiers inférieurs à N

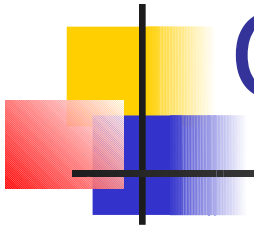
# Crible d'Eratosthène

- Pourquoi s'arrêter dès que le premier nombre non barré a son carré strictement supérieur à N ?

Si p premier tel que $p^2 > N$ alors ses multiples sont déjà barrés ou strictement supérieurs à N. En effet :

Un multiple est de la forme kp :

- $k < p$ : k a un diviseur premier inférieur strict à p donc kp a été barré grâce à ce diviseur
- $k \geq p$ : $kp \geq p^2 > N$



# Crible d'Eratosthène

Nombres premiers inférieurs à 20 :

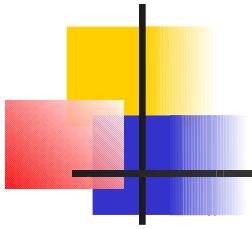
| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

# Crible d'Eratosthène

Implantation du crible :

- On utilise un tableau d'entiers **`crible`** contenant les valeurs 1 ou 0
- **`crible[i] == 0`** ⇔ i premier

# Crible d'Eratosthène : implantation du programme



- Déclaration des constantes et variables
- Initialisation du crible
- Traitement du crible
- Affichage des nombres premiers

# Crible d'Eratosthène : implantation du programme

- Déclaration des constantes et variables

```
#include <stdio.h>
 #include <math.h>

 #define N 2010 // nombres premiers inferieurs a N

 int main() {
   int crible[N+1]; // le crible
   int i,k;

   float limite = sqrt(N); // on s'arrete au premier entier
                           // superieur a la racine de N

   ……
```

# Crible d'Eratosthène : implantation du programme

- Initialisation du crible

```
……

// initialisation de Crible
for (i=2; i<=N; i++) {
  crible[i] = 0;
}

crible[0] = 1; // 0 n'est pas premier
crible[1] = 1; // 1 n'est pas premier;

……
```

# Crible d'Eratosthène : implantation du programme

- Traitement du crible

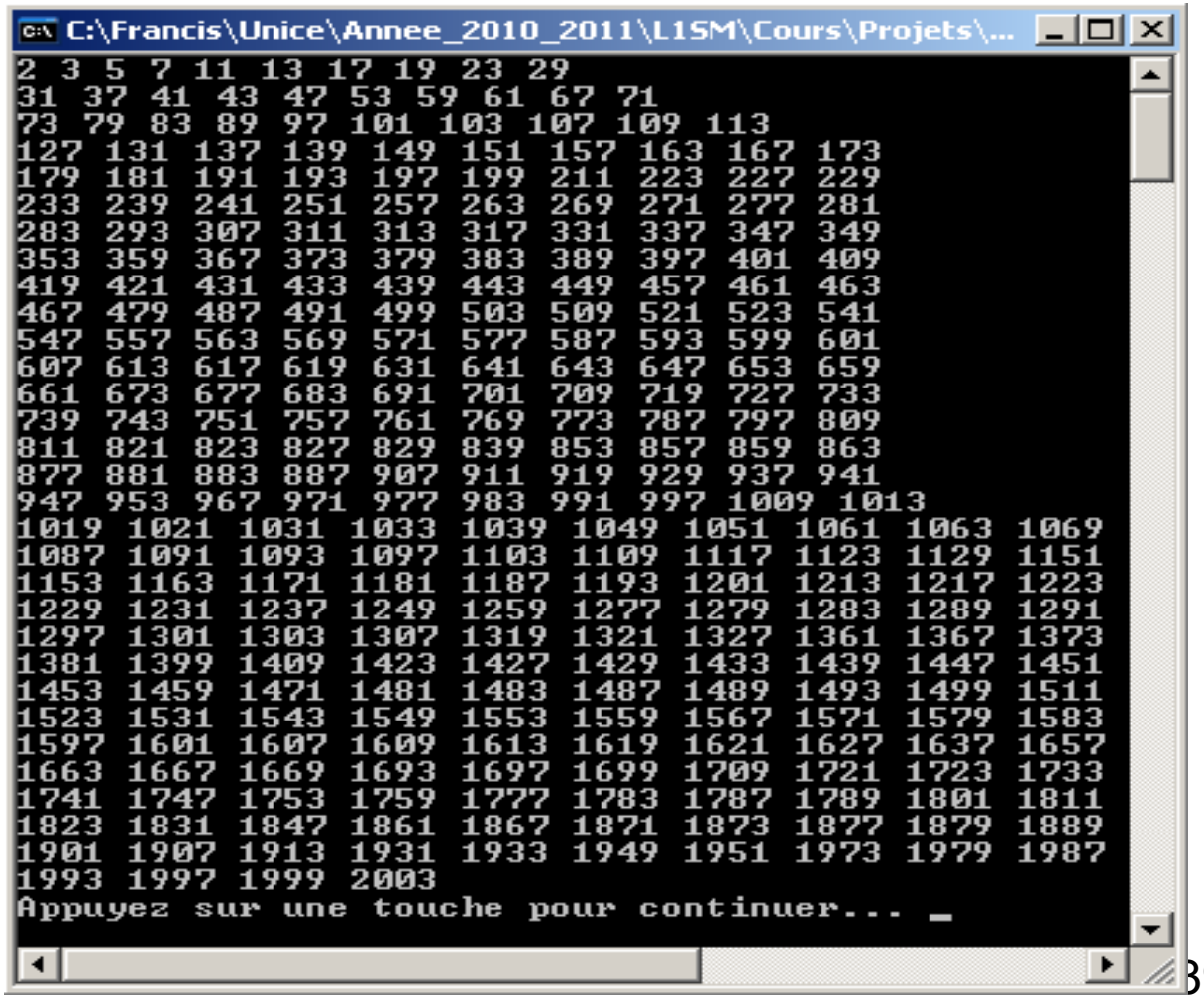
```
……
 i = 2;
 while (i <= limite) {
   if (crible[i] == 0) { // i est un nombre premier
      k = 2*i; // premier multiple de i (hors 1*i)
     // on barre les multiples de i jusqu'a N
     while (k <= N) {
       crible[k] = 1;
       k = k + i;
     }
   }
   i++; // on avance dans le tableau
 }
 ……
```

# Crible d'Eratosthène : implantation du programme

- Affichage des nombres premiers

```
 ……
  // impression des premiers inferieurs a N par ligne de 10 nombres
  k = 0; // pour compter les nombres par ligne
  for (i=2; i<=N; i++) {
    if (crible[i] == 0) {
      printf("%d%s", i, " "); // on affiche le nombre premier
      k++; // un nombre de plus affiche
      if (k%10 == 0) { // 10 ont ete affiches on va a la ligne
        printf("\n"); // retour a la ligne
      }
    }
  }
  printf("\n"); // retour a la ligne
  return 0;
}
```

# Les années « premières » depuis l'an 1 jusqu'à aujourd'hui !



# Une petite amélioration

Comment modifier la ligne

```
k = 2*i; // premier multiple de i (hors 1*i)
```

du programme pour faire moins de calcul ?