Boucles

Instructions itératives

ICC-C: Cours 4

Compléments

Instructions conditionnelles

Types pour nombres réels
et comment les lire

- `float` = type *moins précis* pour les nombres réels
  - pour lecture il faut privilégier la chaîne de formatage `"%f"`
- `double` = type *plus précis* pour les nombres réels (2x `float`)
  - pour lecture il faut privilégier la chaîne de formatage `"%lf"`
  - signifie “long float”
- sur certaines machines `"%f"`, `"%g"` fonctionnent aussi pour lire des `double`
- ⚠️ regardez les “warnings” — messages d’avertissements de gcc

Options d’affichage

- On peut rajouter des options autour du marqueur %
  - Le nombre de chiffres “réservés”: %6d
  - Précision d’un nombre réel: %.2f
  - Si on veut que les espaces vides soient remplis de 0: %04d

https://learn.microsoft.com/fr-ch/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions?view=msvc-170

%[*flags*][*width*][.*precision*][*size*]*type*

```c
printf("Je réserve 6 espaces: %6d\n",
    123);
// Affiche: Je réserve 6 espaces:    123

printf("Précision 2: %.2f\n",
       8.11912);
// Affiche: Précision 2: 8.12

printf("Padding %04d\n", 3);
// Affiche: Padding 0003

printf("Nous sommes le %04d-%02d-%02d\n",
       2024, 3, 12);
// Affiche: Nous sommes le 2024-03-12
```

Les caractères spéciaux

Instructions composées

et portée (*scope*)

```
{
  int test = 8;
  {
    printf("C'est une instruction composée “
           "dans une instruction composée\n");
    test = 12;
  }
  printf("test vaut %d\n", test);
  // Affiche: test vaut 12
}

printf("test? C’est qui test??")
```

`test` est connue ici et peut y être utilisée!

`test` est inconnue ici!

Portée imbriquée

Nested scope

```
{
    // a est inconnue
    int a = 10; // La variable ‘a’ est définie ici
    ...
    {
        printf("a vaut %d\n", a); // Affiche: a vaut 10
        ...
        int a = 20; // Une nouvelle variable ‘a’ est définie ici
        // la deuxième variable ‘a’ “cache” la première!
        printf("a vaut %d\n", a); // Affiche: a vaut 20
        ...
    } // La deuxième variable ‘a’ n’est plus définie, la première reprend la main!
    printf("a vaut %d\n", a); // Affiche: a vaut 10
    ...
} // La variable ‘a’ n’est plus définie
```

a est inconnue

La première variable a est connue ici et peut y être utilisée

La deuxième variable a est connue ici *pas la première!*

La première variable a est de nouveau connue

Instruction conditionnelle `if`

```
int points_lausanne = 0, points_basel = 0;

int buts_lausanne = 2, buts_basel = 1;

if (buts_lausanne > buts_basel)
    points_lausanne += 3;

if (buts_lausanne < buts_basel)
    points_basel += 3;

if (buts_lausanne == buts_basel) {
    points_lausanne++;
    points_basel++;
}
// points_lausanne = 3, points_basel = 1  0
```

On l'aurait vu
si on avait formaté notre code

Dans VS Code
- Windows: Alt+Shift+F
- Linux: Alt+Shift+I
- macOS: ⌥+⇧+F

Boucles

Instructions itératives

L’instruction `while`

`while` *statement*

- Nous voulons maintenant pouvoir executer le même code autant de fois que nécessaire jusqu’à ce qu’une **condition** soit satisfaite
- Exemples:
  - Lire une vidéo — tant qu’on n’est pas à la fin, afficher un nouveau cadre
  - Chercher un mot donné dans un document — tant qu’il n’est pas trouvé, essayer le mot suivant
  - Tout programme interactif — tant que l’utilisateur ne veut pas quitter, répondre aux instructions de l’utilisateur

L’instruction `while`

`while` *statement*

- Une boucle `while` a la syntaxe suivante:

  ```
  while (expression) instruction
  ```

- L’expression sera évaluée comme valeur booléenne, donc la boucle sera terminée dès que l’expression s’évalue à `0`

- On peut juste “lire” le code:

Tant qu’<u>`expression`</u> est vraie execute l’<u>`instruction`</u>. Dès qu’<u>`expression`</u> n’est plus vraie, continue à l’instruction d’après.

- L’instruction peut être une instruction composée (souvent), mais peut être une instruction simple, un `if`, un autre `while`, etc.

Un exemple simple

Le programme minuteur

- Tant que `nombre` est supérieur à `0`, imprime & enlève `1` de `nombre`
- L'expression `(n--)` a la valeur de `n` *avant* l'évaluation
- Effet secondaire: soustrait `1` à la variable `nombre`

```
int main()
{
    printf("Bonjour, je suis “
           "le programme minuteur.\n");
    printf("Entrez un nombre positif : ");
    int n;
    scanf("%d", &n);

    printf("Je vais compter de %d à 0.\n”, n);

    while (n >= 0)
        printf("%d\n", n--);
}
```

Un exemple simple

Le programme minuteur

Ligne active	Expression à évaluer	Valeur	n après
10	printf("Je vais ...	–	3
11	n >= 0	1	3
12	printf("%d\n", n--);	–	2
11	n >= 0	1	2
12	printf("%d\n", n--);	–	1
11	n >= 0	1	1
12	printf("%d\n", n--);	–	0
11	n >= 0	1	0
12	printf("%d\n", n--);	–	-1
11	n >= 0	0	-1
	On sort de la boucle while		
13	Fin du programme		

```
1  #include <stdio.h>
2
3  int main()
4  {
5    printf("Bonjour, "
            "je suis le programme minuteur.\n");
6    printf("Entrez un nombre positif : ");
7    int n;
8    scanf("%d", &n); // On entre 3
9
10   printf("Je vais compter de %d à 0.\n", n);
11   while (n >= 0)
12     printf("%d\n", n--);
13 }
```

```
Je vais compter de 3 à 0.
3
2
1
0
```

Un autre exemple simple

Le programme perroquet 🦜

```c
#include <stdio.h>

int main()
{
    printf("Bonjour, je suis le programme perroquet.\n");
    printf("Je répète les nombres que vous entrez.\n");
    printf("Tapez \"0\" pour arrêter le programme.\n");

    int nombre = 1;
    while (nombre)
    {
        printf("Entrez un nombre : ");
        scanf("%d", &nombre);
        printf("Vous avez entré : %d\n", nombre);
    }
    printf("Ciao.\n");
}
```

On aurait pu écrire

```c
while (nombre != 0)
```

mais **c'est la même chose!**

Un autre exemple simple

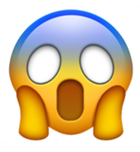
Le programme perroquet 🦜

Tant qu’on entre un entier non-nul (différent de 0), ce programme va juste répéter ce qu’on lui dit

```
Bonjour, je suis le programme perroquet.
Je répète les nombres que vous entrez.
Tapez "0" pour arrêter le programme.
Entrez un nombre : 12
Vous avez entré : 12
Entrez un nombre : -3
Vous avez entré : -3
Entrez un nombre : 0
Vous avez entré : 0
Ciao.
```

```
int nombre = 1;
while (nombre)
{
    printf("Entrez un nombre : ");
    scanf("%d", &nombre);
    printf("Vous avez entré : %d\n", nombre);
}
printf("Ciao.\n");
```

Entrée invalide



- Que se passe-t-il si on lui dit de répéter “coucou” plutôt qu’un entier? 😈

```
Entrez un nombre : -3
Vous avez entré : -3
Entrez un nombre : coucou
Entrez un nombre : Vous avez entré : -3
Entrez un nombre : Vous avez entré : -3
Entrez un nombre : Vous avez entré : -3
Entrez un nombre : Vous avez entré : -3
Entrez un nombre : Vous avez entré : -3
Entrez un nombre : Vous avez entré : -3
Entrez un nombre : Vous avez entré : -3
Entrez un nombre : Vous avez entré : -3
Entrez un nombre : Vous avez entré : -3
```

```c
int nombre = 1;
while (nombre)
{
    printf("Entrez un nombre : ");
    scanf("%d", &nombre);
    printf("Vous avez entré : %d\n", nombre);
}
printf("Ciao.\n");
```

Boucle infinie!

∞

- Le perroquet est coincé dans une **boucle infinie**…
- Dans le **terminal** pour arrêter un programme
  - qui ne termine pas,
  - ou qui prend trop longtemps à terminer…

`Ctrl-C`

Pourquoi?

```
scanf("%d", &nombre);
```

- La fonction scanf retourne le **nombre d'objets lus**
- `%d` — elle attend un entier
- Si on écrit autre chose qu'un entier, elle va juste retourner 0:
  "j'ai lu 0 entiers"
- Elle ne va pas modifier `nombre`
- Elle ne va pas consommer depuis `stdin`
- Pas de nouvelle entrée requise de l'utilisateur, car **<u>il y a déjà qqch</u>** dans `stdin`

stdin nous bloque…

-3↵

stdin “-3\n”

```
scanf("%d", &nombre); // => j’ai lu 1 objet; nombre vaut -3
```

stdin “”

coucou↵

stdin “coucou\n”

```
scanf("%d", &nombre); // => j’ai lu 0 objets; nombre vaut toujours -3
```

stdin “coucou\n”

etc.

How do we fix it?

- On peut tester si on a bien lu un entier
- Dans le cas contraire, on peut ***casser*** la boucle avec l'instruction `break`
- `break` nous fait sortir de la boucle et passer à l'instruction après le `while`

```
int nombre = 1;
while (nombre)
{
  printf("Entrez un nombre : ");
  if (scanf("%d", &nombre) != 1)
  {
    // Si la lecture échoue
    printf("Erreur: pas un nombre.\n");
    break;
  }
  printf("Vous avez entré : %d\n", nombre);
}
```

Les instructions `break` et `continue`

- `break` nous fait sortir tout de suite de la boucle
- L'instruction `continue` interrompt l'itération courante et passe à l'itération suivante
- Considérons un perroquet capricieux 🦜 qui n'aime pas les nombre pairs

```c
int nombre = 1;
while (nombre)
{
  printf("Entrez un nombre : ");
  if (scanf("%d", &nombre) != 1)
  {
    // Si la lecture échoue
    printf("Erreur: pas un nombre.\n");
    break;
  }
  if (nombre % 2 == 0) {
    printf("Je n'aime pas les nombres pairs.\n");
    continue;
  }
  printf("Vous avez entré : %d\n", nombre);
}
```

Perroquet capricieux

```
Entrez un nombre : -3
Vous avez entré : -3
Entrez un nombre : 16
Je n’aime pas les nombres pairs.
Entrez un nombre : 13
Vous avez entré : 13
Entrez un nombre : 0
Je n’aime pas les nombres pairs.
Ciao.
```

```c
int nombre = 1;
while (nombre)
{
  printf("Entrez un nombre : ");
  if (scanf("%d", &nombre) != 1)
  {
    // Si la lecture échoue
    printf("Erreur: pas un nombre.\n");
    break;
  }
  if (nombre % 2 == 0) {
    printf("Je n'aime pas les nombres pairs.\n");
    continue;
  }
  printf("Vous avez entré : %d\n", nombre);
}
```

Les instructions break et continue

```
{
    while (ma_condition)
    {  ...
       if (casser)
       {

          break;
       }
       else if (continuer)
       {

          continue;
       }
       instruction_après_if
    }// Fin du while
    instruction_après_while
}
```

Saute instruction_après_if et recommence la boucle en réévaluant ma_condition

Sort de la boucle et passe à l'instruction_après_while

Lire un tableau d’entiers

Enfin! 😮‍💨

- Définir le tableau
- Lire le nombre de valeurs
- Vérifier qu’il n’est pas trop grand
- Lire les valeurs une par une

```
double heures[20]; // sommeil par nuit
int nuits;
printf("Combien de nuits ?\n”);
scanf("%d", &nuits);
if (nuits > 20)
{
    printf("Erreur: trop de nuits.\n");
    return 1;
}

int indice = 0;
printf("Entrez %d valeurs : \n”, nuits);
while (indice < nuits)
{
    scanf("%f", &heures[indice]);
    indice++;
}
```

La somme des éléments

- Une variable pour calculer la somme avec valeur initiale 0
- On parcourt chaque élément et on l’ajoute à la somme

```
double heures[20]; // heures de sommeil
int nuits, indice;

double total_heures = 0;
indice = 0;
while (indice < nuits)
{
    total_heures += heures[indice];
    indice++;
}
printf("Vous avez dormi en tout %g heures.\n",
       "Donc en moyenne %g heures par nuit.\n",
       total_heures,
       total_heures / nuits);
```

La somme des éléments

total_heures 29.6

```
indice = 0;
while (indice < nuits)
{
    total_heures += heures[indice];
    indice++;
}
```

indice: 0 indice: 1 indice: 2 indice: 3

6.5 7.2 6.7 9.2

heures[]

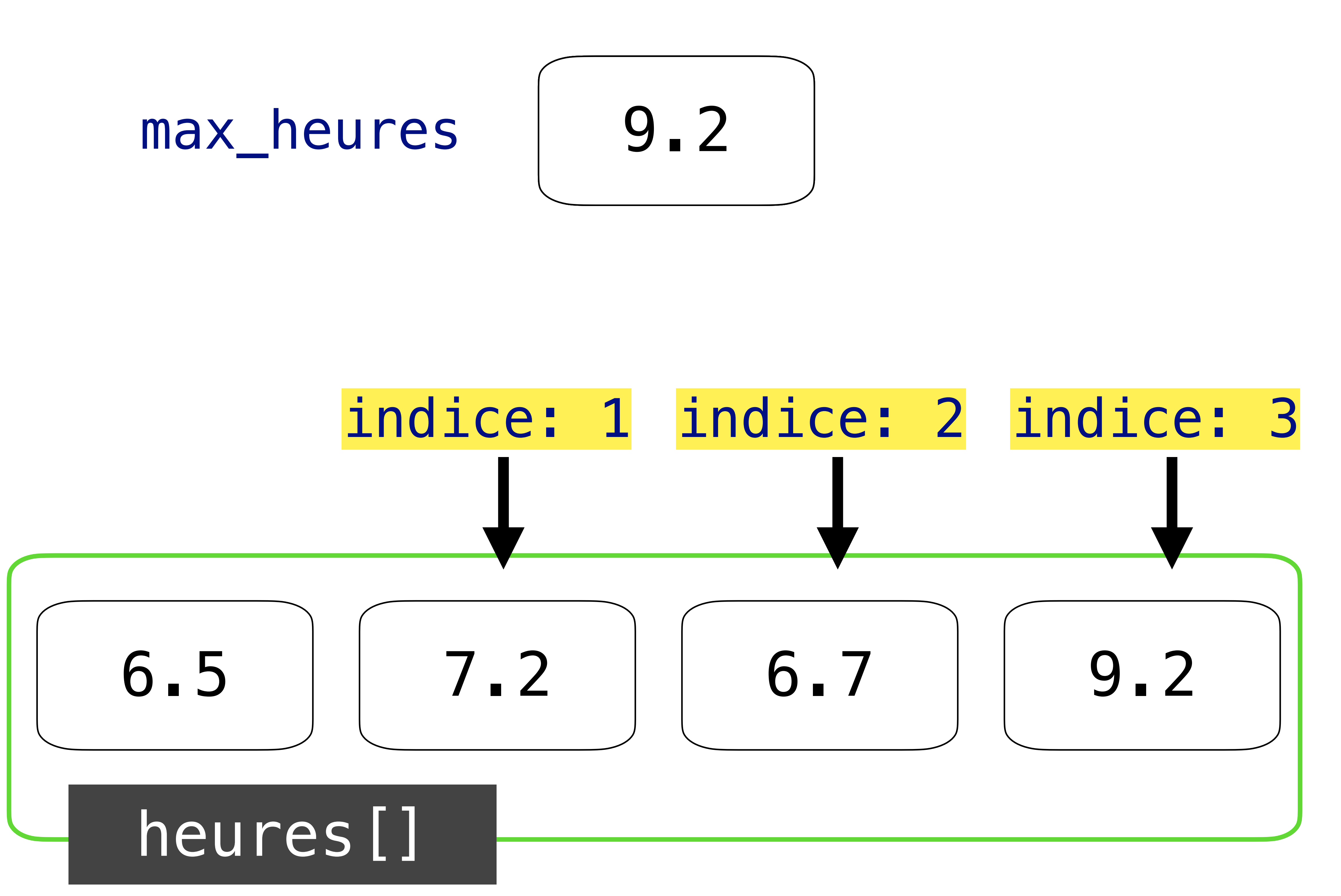
Trouver la valeur max

- C’est pareil!
- Valeur initiale = 1er élément
- Comparer à chaque valeur et mettre à jour si nécessaire

```
double heures[20]; // heures de sommeil
int nuits, indice; // nuits >= 1

double max_heures = heures[0];
indice = 1;
while (indice < nuits)
{
    if (heures[indice] > max_heures)
        max_heures = heures[indice];
    indice++;
}
printf("Vous avez dormi au plus %g heures.\n”,
       max_heures);
```

Le max des éléments




```
double max_heures = heures[0];
indice = 1;
while (indice < nuits)
{
    if (heures[indice] > max_heures)
        max_heures = heures[indice];
    indice++;
}
```

Lire un tableau d’entiers

- Ce n’est pas amusant de rentrer des valeurs à la main
- Nous avons un fichier texte avec les valeurs désirées
  - La première valeur représente le nombre total de nuits
- Utilisons la *redirection de l’entrée standard* dans le terminal
- Syntaxe:
  `./executable < fichier`

```
❯ head ~/data/sommeil.in
12
5
7
6.3
7.1
4.3
8
8.1
8.1
6.45
```

```
❯ ./sommeil < ~/data/sommeil.in
```

```
Combien de nuits voulez-vous enregistrer?
Entrez 12 valeurs :
Vous avez dormi 83.15 heures en tout.
Vous avez dormi en moyenne 6.92917 heures par nuit.
Vous avez dormi au plus 9 heures.
```

L’instruction for

```
for (initialisation; condition; itération) instruction
```

- Est équivalente à:

```
{
 initialisation
 while (condition) {
   instruction
   itération
 }
}
```

L’instruction for

- Lire un tableau d’entiers

```
int indice = 0;
printf("Entrez %d valeurs : \n”, nuits);
while (indice < nuits)
{
    scanf("%f", &heures[indice]);
    indice++;
}
```

Initialisation — Exécuté une seule fois

La condition pour continuer

L’expression pour itérer

```
int indice;
printf("Entrez %d valeurs : \n”, nuits);
for (indice=0; indice < nuits; indice++)
{
    scanf("%f", &heures[indice]);
}
```

Boucles imbriquées

Nested loops

- On peut mettre une boucle dans une boucle!
- Par exemple, afficher toutes les paires

```
int vec[] = {1, 2, 3};
int size = 3;
int i, j;

for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
        printf("(%d, %d)\n", vec[i], vec[j]);
    }
}
```

```
// Affiche :
// (1, 1)
// (1, 2)
// (1, 3)
// (2, 1)
// (2, 2)
// (2, 3)
// (3, 1)
// (3, 2)
// (3, 3)
```

Boucle infinie

- Parfois nous **voulons** qu'un programme ne termine pas
  - Par exemple MS Word, ou Keynote, ou Chrome
- Ces programmes ont quelque part une boucle infinie, mais qui peut être cassée sous certaines conditions

```
while (1)
{
    interaction avec l'utilisateur
    if (terminer le programme) {
        break;
    }
}
```