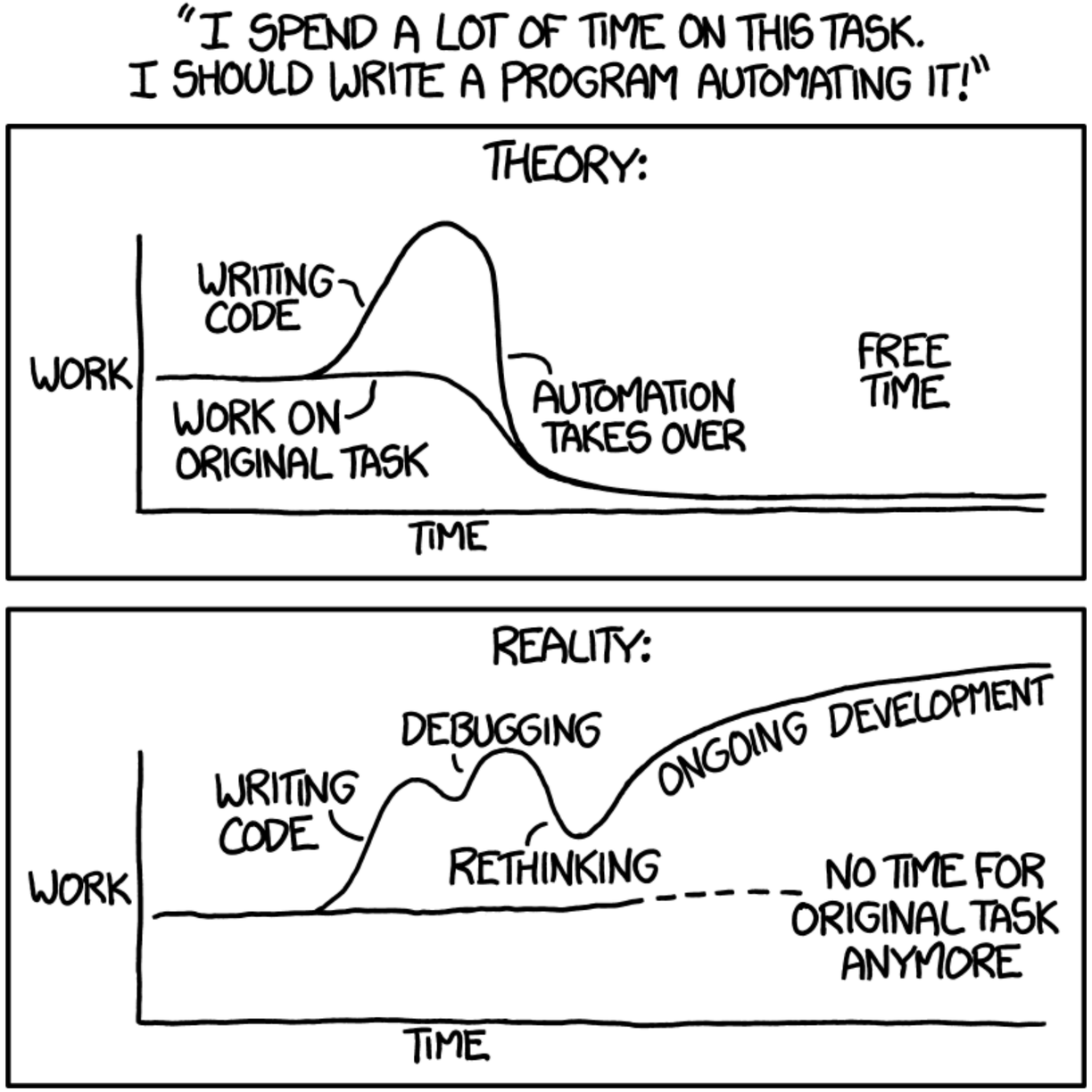
I/O

Entrée/sortie

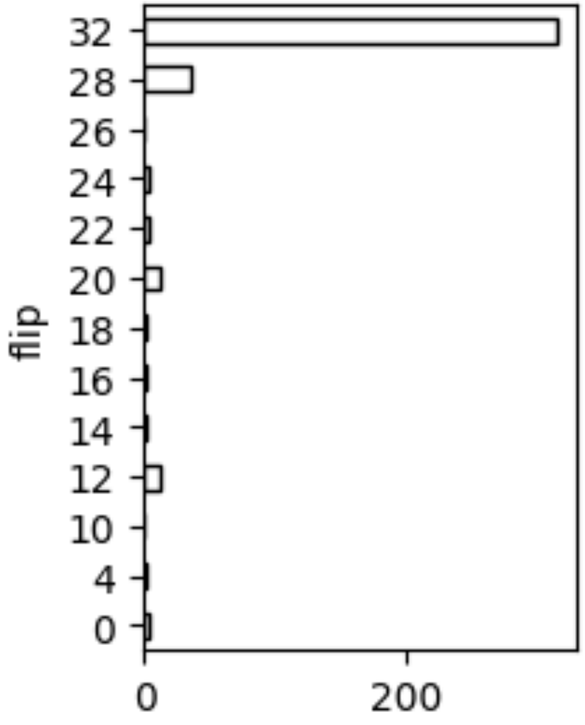


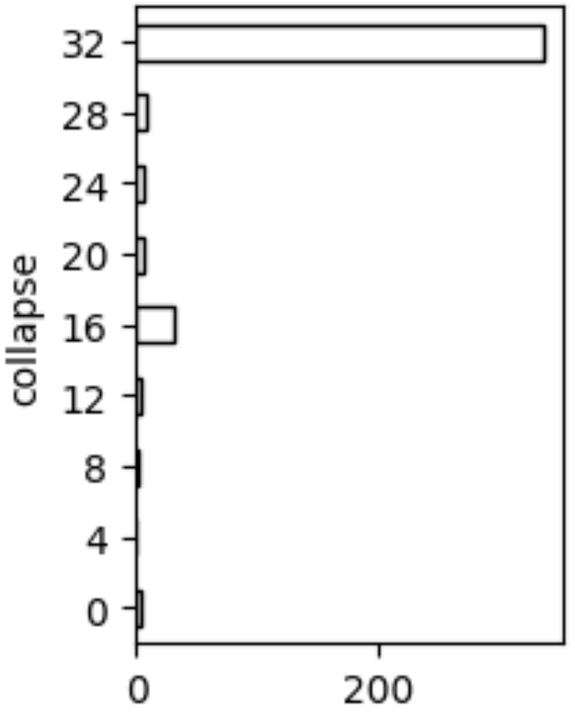
ICC-C Cours 13

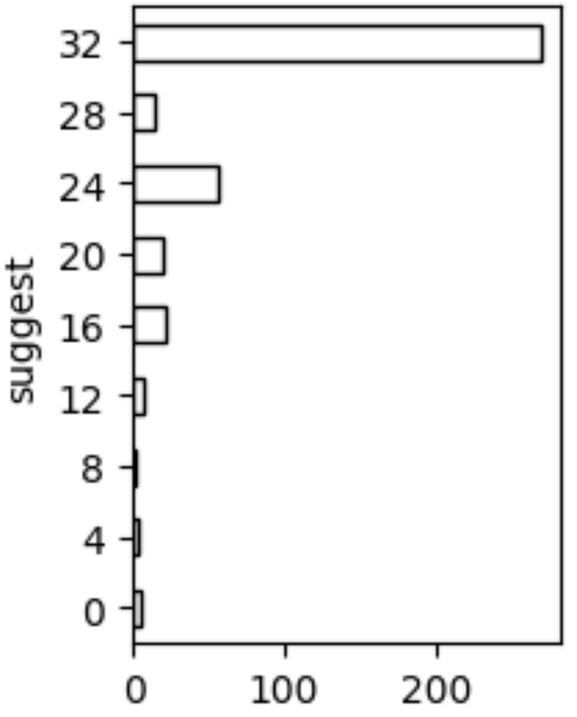
Notes des projets

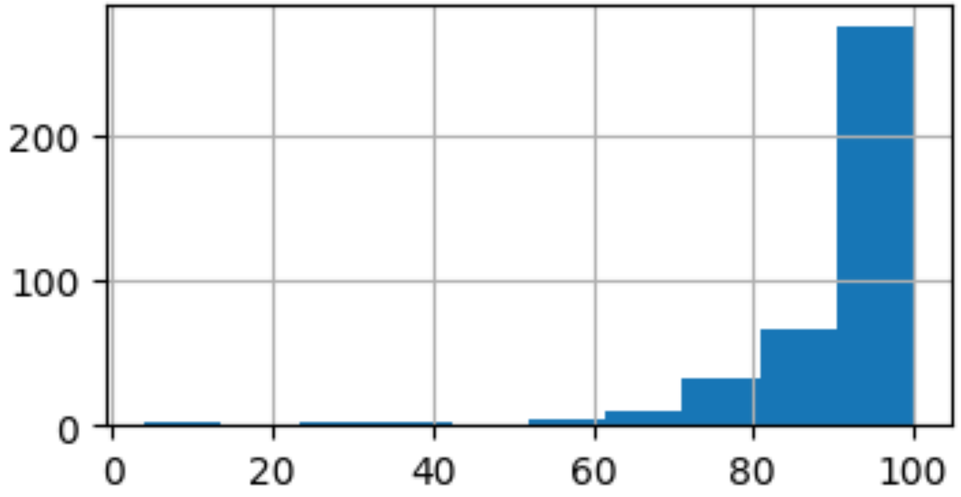
= 15% de la note finale

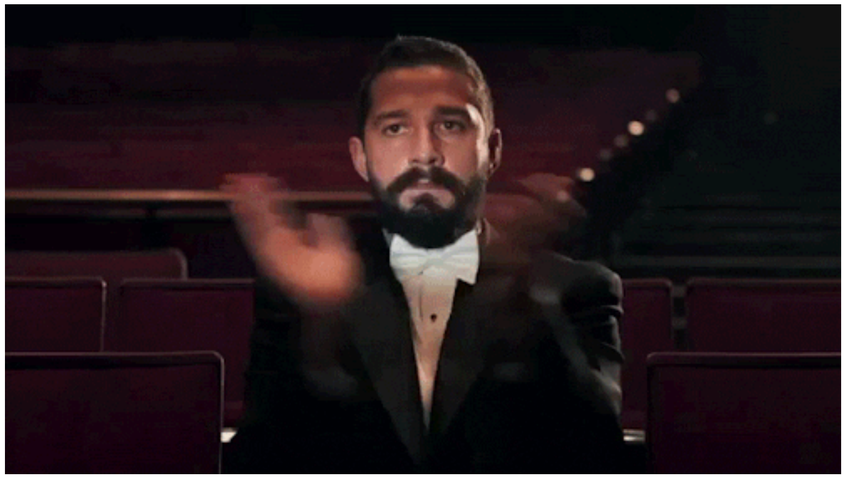
- 8 tests par fonction
- 4 points chacun
- Notes partielles par test (e.g., si mauvais score, mais bon tableau = 2)
- Note max = 100 = 4 + 32 + 32 + 32
- Un test donne 0 points si
  - SEGFAULT, mauvaise réponse, Time Limit Exceeded
- Sur moodle il y a des détails sur l’exécution des tests pour votre code


flip
32
28
26
24
22
20
18
16
14
12
10
4
0
0
200


collapse
32
28
24
20
16
12
8
4
0
0
200


suggest
32
28
24
20
16
12
8
4
0
0
100
200


200
100
0
0
20
40
60
80
100



Révision

- Expressions
  `&((*a)++)->data[3]-- + 7`
- Pointeurs
  `malloc, free, int**, ...`
- Contrôle de flux
  `while, if, for`
- Fonctions
  `void foo();`
- Listes
- Types simples et composés
  `struct foo bar;`
  `float f[20];`
- Strings
  `char *s = “coucou”;`
- Récursivité
  `f(f(f(f(...))))`
- (Fichiers) etc.
- Pour votre culture: Makefiles, détails sur le préprocesseur, …

Format de la partie prog de l’examen

= 35% de la note finale

- QCM
  - “Qu’affiche ce code?” A. B. C. D.
  - “Quel code affiche X?” A. B. C. D.
  - “Ce code échoue car…” A. B. C. D.
  - …
- Implémentation
  - 1-2 problèmes ouverts
  - Par exemple, “Écrivez une fonction récursive qui résout le problème suivant…”

Fichiers

- Un fichier réside dans le stockage “long-terme” de l’ordinateur
- SSD, Disque-dur, etc.
- Il contient une suite d’octets
- Il peut être trop gros pour être “chargé” dans la mémoire
- On peut lire, écrire, ou modifier des fichiers

Fichiers binaires vs. texte

- Tous les fichiers peuvent être lus en mode “binaire” = une suite d’octets
- Ils ont un format spécifique à l’application qui les écrit/lit
- **Exemples:** Un document Word, une présentation Keynote, la sauvegarde d’un jeu vidéo, un fichier exécutable
- Certains fichiers sont censés contenir uniquement des caractères = fichiers texte
- **Exemples:** Un fichier source `.c`, une page web `.html`, etc.

Ouvrir un fichier

- On veut lire le contenu d'un fichier
- ???
- Comme pour `malloc`, on demande au système d'exploitation…
- On appelle la fonction `fopen` qui retourne un pointeur vers un objet `FILE`

```
FILE* file = fopen("message.txt", "r");
```

- 1er paramètre = le chemin (*path*) vers le fichier qu'on veut lire
- 2e paramètre = en quel "mode" on veut ouvrir le fichier

message.txt

```
“””
Dear Mr Potter,
We are pleased to inform you that you have been accepted at Hogwarts School of
Witchcraft and Wizardry.
“””

|'D'( 68)|'e'(101)|'a'( 97)|'r'(114)|' '( 32)|'M'( 77)|'r'(114)|
|' '( 32)|'P'( 80)|'o'(111)|'t'(116)|'t'(116)|'e'(101)|'r'(114)|
|','( 44)|'\n'(10)|'W'( 87)|'e'(101)|' '( 32)|'a'( 97)|’r'(114)|
...
```

Modes

Read (Lecture): `"r"`

- On peut lire depuis le début du fichier

Write (Écriture): `"w"`

- On crée un nouveau fichier
- ⚠️ Si un fichier du même nom existe, il sera **écrasé** ⚠️

Append (Ajout?): `"a"`

- On écrit à la fin d'un fichier existant

suivi par… (optionnel, utile sous Windows)

Binary: `"b"`

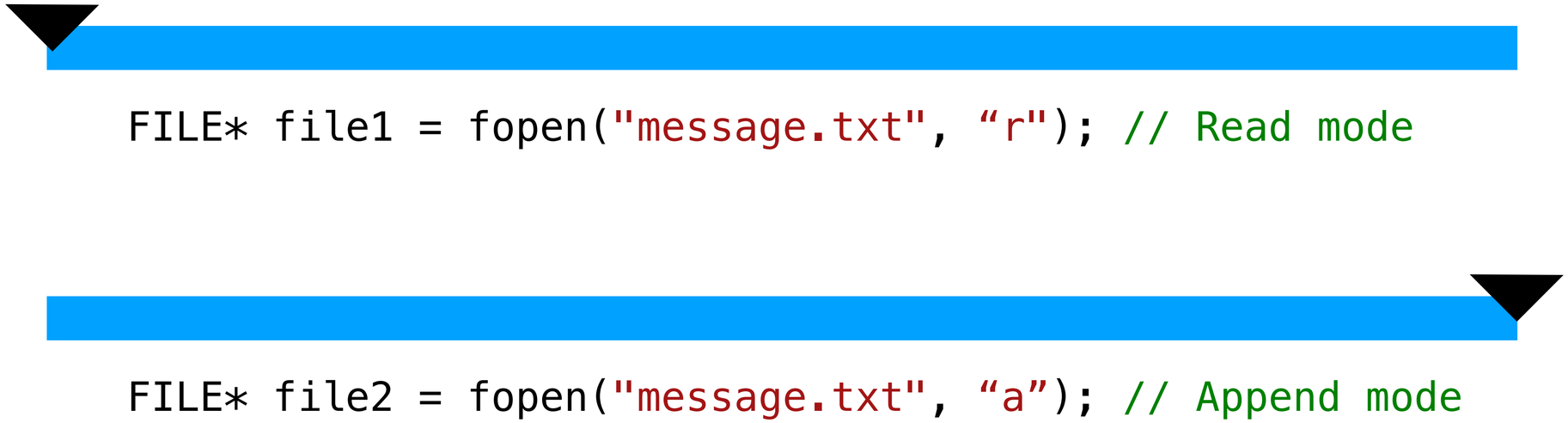
- On lit des octets

Text: `"t"`

- On lit des caractères

Le curseur

- Quand on ouvre un fichier, un curseur indique l’endroit où on se trouve



```
FILE* file1 = fopen("message.txt", “r"); // Read mode
```

```
FILE* file2 = fopen("message.txt", “a”); // Append mode
```

Lecture binaire

- On peut lire un nombre d’objets depuis le fichier avec `fread`:

```
nombre_objets_lus = fread(
                      buffer,
                      taille_objet,
                      n_objets,
                      file);
```

pointeur vers l’emplacement de mémoire qui recevra les octets lus

```
FILE* file = fopen("message.txt", “r"); // Read mode
```

Lecture binaire
`fread`

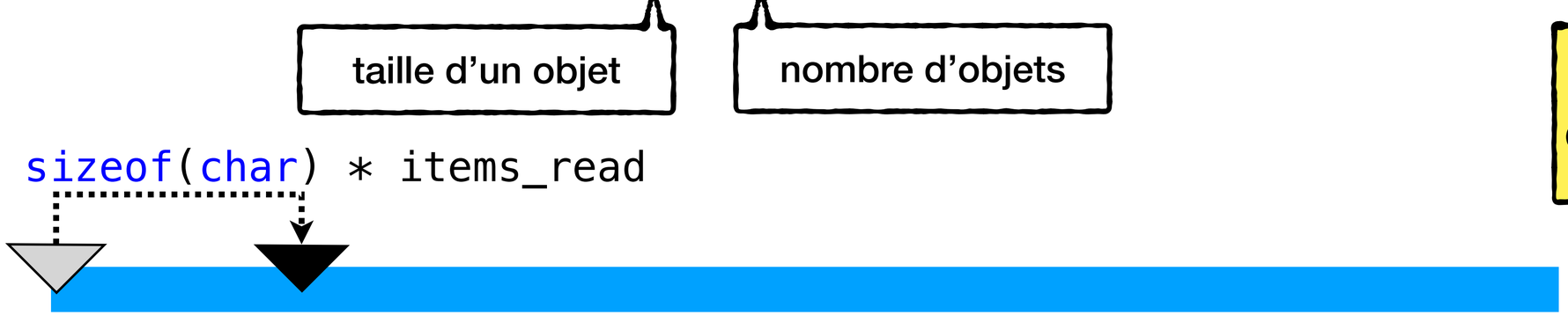
- On peut lire un nombre d'objets binaires depuis le fichier avec `fread`:

```
#define SIZE 7
char buffer[SIZE];
int items_read;

items_read = fread(buffer, 1, SIZE, file);
```

taille d'un objet

nombre d'objets

```
+ sizeof(char) * items_read
```



Pas de 0 à la fin!
On a juste lu des octets…

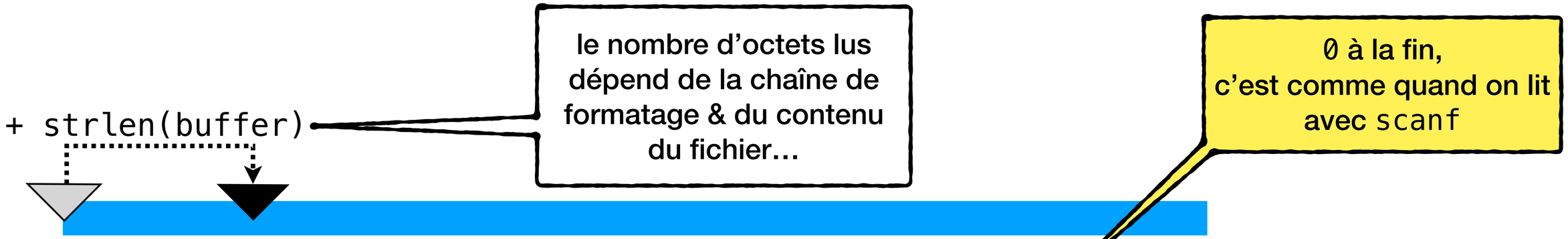
```
FILE* file = fopen("message.txt", "r"); // Read mode
buffer: |'D'( 68)|'e'(101)|'a'( 97)|'r'(114)|' '( 32)|'M'( 77)|'r'(114)|
items_read: 7
```

Lecture en mode texte
`fscanf`

Texte

- On peut aussi lire en mode texte avec `fscanf` (puis `fgets` marche aussi!)

```
#define SIZE 7
char buffer[SIZE];
int items_read;

items_read = fscanf(file2, "%s", buffer);
```




```
FILE* file2 = fopen("message.txt", "r"); // Read mode
buffer: |'D'( 68)|'e'(101)|'a'( 97)|'r'(114)|'x00'(  0)|'x00'(  0)|'x00'(  0)|
items_read: 1
```

Écriture binaire
`fwrite`

- On peut écrire un nombre d'objets binaires depuis le fichier avec `fwrite`:

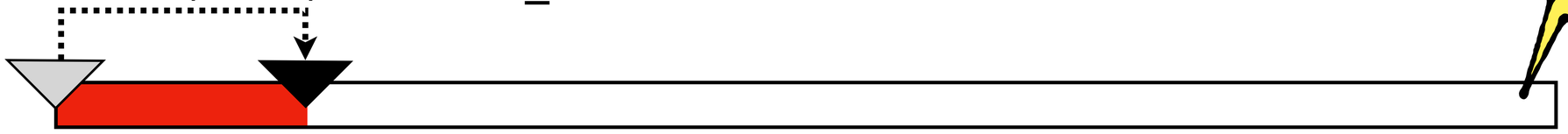
```
#define SIZE 7
char buffer[SIZE] = “coucou”;
int items_written;

items_written = fwrite(buffer, 1, SIZE, file);
```

taille d'un objet

nombre d'objets

Fichier écrasé par l'ouverture en mode “w”

```
+ sizeof(char) * items_written
```



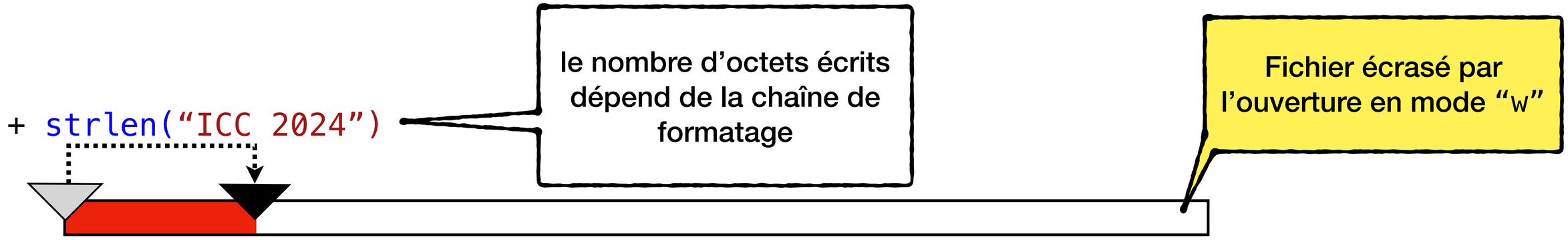
```
FILE* file = fopen("message.txt", “w"); // Write mode
buffer: |'c'( 99)|'o'(111)|'u'(117)|'c'( 99)|'o'(111)|'u'(117)|'x00'(  0)|
items_written: 7
```

Écriture en mode texte
`fprintf`

Texte

- On peut aussi écrire en mode texte avec `fprintf`:

```
int items_written;

items_written = fprintf(file2, “ICC %d", 2024);
```




```
FILE* file2 = fopen("message.txt", “w"); // Write mode
```

items_written: 1

Mais où est passé mon curseur?
`ftell`

- On aimerait retrouver la position du curseur par rapport au début du fichier
- Il faut utiliser la fonction `ftell` ~ “f-raconte”

```
items_read = fscanf(file2, "%s", buffer);

printf(“Offset = %d\n", ftell(file2));
// Affiche: Offset = 4
```


```
FILE* file2 = fopen("message.txt", “r"); // Read mode
buffer: |'D'( 68)|'e'(101)|'a'( 97)|'r'(114)|'x00'(  0)|'x00'(  0)|'x00'(  0)|
items_read: 1
```

Bouger le curseur
`fseek`

- Quand on lit/écrit le curseur bouge à l'endroit où on a fini de lire/écrire
- Parfois on aimerait aller à un endroit précis dans le fichier

```
fseek(file, offset, SEEK_SET);
```

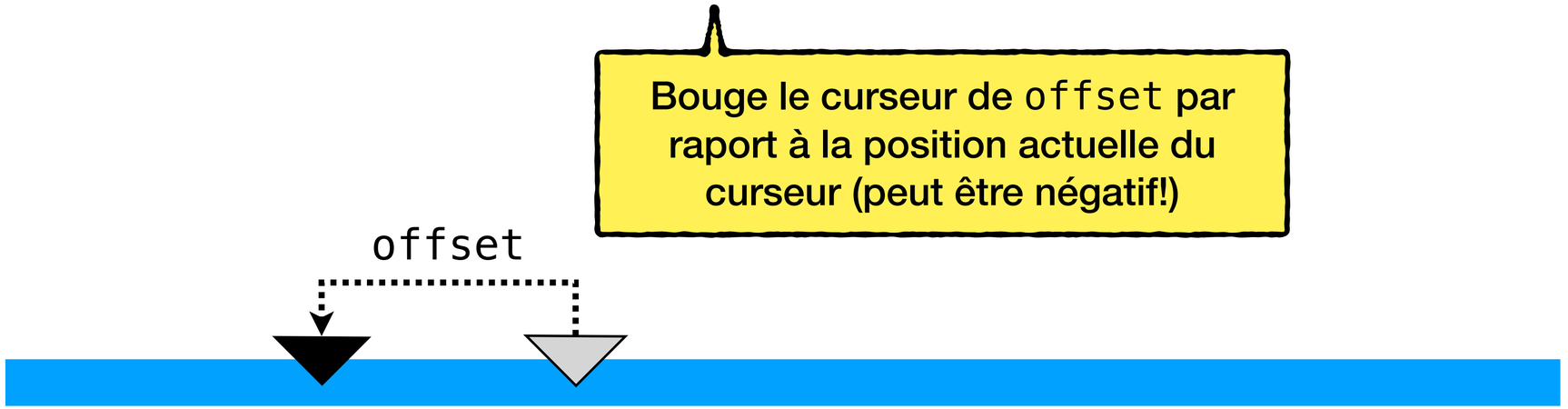
Bouge le curseur de `offset` par raport au début du fichier


```
FILE* file = fopen("message.txt", "r"); // Read mode
```

Bouger le curseur
`fseek`

- Quand on lit/écrit le curseur bouge à l'endroit où on a fini de lire/écrire
- Parfois on aimerait aller à un endroit précis dans le fichier

```
fseek(file, offset, SEEK_CUR);
```




```
FILE* file = fopen("message.txt", "r"); // Read mode
```

Bouger le curseur
`fseek`

- Quand on lit/écrit le curseur bouge à l'endroit où on a fini de lire/écrire
- Parfois on aimerait aller à un endroit précis dans le fichier

```
fseek(file, offset, SEEK_END);
```

Bouge le curseur de `offset` par raport à la fin du fichier (doit être négatif!)


```
FILE* file = fopen("message.txt", "r"); // Read mode
```

Erreurs…

- Souvent quand on interagit avec l'I/O il y a des erreurs - et il faut vivre avec!
- **"Le fichier à lire n'existe pas"**, **"Pas assez de droits pour lire/écrire"**, …

  `fopen` retournera `NULL` — il faut tester si l'objet `FILE*` retourné est **valide**
- **"On a atteint la fin du fichier, il n'y a plus rien à lire!"**

  `fscanf` / `fread` retournent moins d'éléments lus que prévu

  On peut tester si on a atteint la fin du fichier avec `feof(file)` — cette fonction retourne `1` si le fichier est terminé, ou `0` sinon

  EOF = *End Of File*

Sérialisation et désérialisation

- Sérialiser = convertir un objet qui se trouve en RAM en suite d'octets qu'on peut stocker dans un fichier
- Désérialiser = convertir une suite d'octets qu'on peut stocker dans un fichier en un objet qui se trouve en RAM
- Il y a des objets sérialisables (des données):
  - Tableaux de `int`, chaînes de caractères, etc.
- … et des objets qui **ne sont pas** sérialisables (spécifiques à l'exécution):
  - Des pointeurs (adresses), des descripteurs de fichiers ouverts, etc.

Exemple

Serialize

```c
typedef struct call_record
{
    int no_client;
    int no_tel_appel;
    int date;
    int minutes;
} call_record_t;
```

```c
call_record_t records[] = {
    {1, 798812233, 20240501, 33},
    {1, 797777777, 20240501, 12}
};

FILE* rec_file = fopen("records.bin", "w");

fwrite(records, sizeof(call_record_t), 2, rec_file);

fclose(rec_file);
```

Exemple

Deserialize


```c
typedef struct call_record
{
    int no_client;
    int no_tel_appel;
    int date;
    int minutes;
} call_record_t;
```

```c
call_record_t records[10];

rec_file = fopen("records.bin", "r");

while (!feof(rec_file))
{
    int processed = fread(records, sizeof(call_record_t), 10, rec_file);
    printf("Read %d records\n", processed);
    for (int i=0; i<processed; i++)
    {
        process_record(&records[i]);
    }
}

fclose(rec_file);
```

Attention aux pointeurs!

1101

```c
typedef struct info
{
    char *nom;
    int temps_sec;
} info_t;
```

- Le membre nom est un pointeur…
- Il faut explicitement écrire la longueur + le string

```c
void to_file(FILE *out, const info_t *record)
{
    int len = strlen(record->nom);
    fwrite(&len, sizeof(int), 1, out);
    fwrite(record->nom, sizeof(char), len, out);

    fwrite(&record->temps_sec, sizeof(int), 1, out);
}
```

Attention aux pointeurs!

1101

```c
typedef struct info
{
    char *nom;
    int temps_sec;
} info_t;
```

- Quand on lit depuis le fichier, il faut allouer le string

```c
void from_file(FILE *in, info_t *record)
{
    int len;

    fread(&len, sizeof(int), 1, in);
    record->nom = malloc(len + 1);
    fread(record->nom, sizeof(char), len, in);
    record->nom[len] = 0; // End of string

    fread(&record->temps_sec, sizeof(int), 1, in);
}
```

Qu’est-ce qu’on n’a pas vu?

- Les fonctions sur les strings (“facile”)

  `<string.h> — strlen, strcpy, strcmp, strtok, …`

- Accès au niveau des bits (“facile-moyen”)
  - Membres par bit des struct
  - Opérateurs par bit, drapeaux/flags, …
- Comment gérer la concurrence (il y a des cours entiers là-dessus)
  - Les threads, processes, etc., multiplexage avec `select`

Qu’est-ce qui manque en C?

- Les objets
- Les dictionnaires
  - “Key-value stores” — permettent de stocker des clés et des valeurs
  - On ne peut pas écrire des choses comme `age["bob"] = 42`
  - Le “module” `<search.h>` fait ça de manière très primitive
- Une meilleure gestion des chaînes de caractères
- Gestion des erreurs — soit c’est fatal, soit ça peut passer inaperçu…

Les autres langages

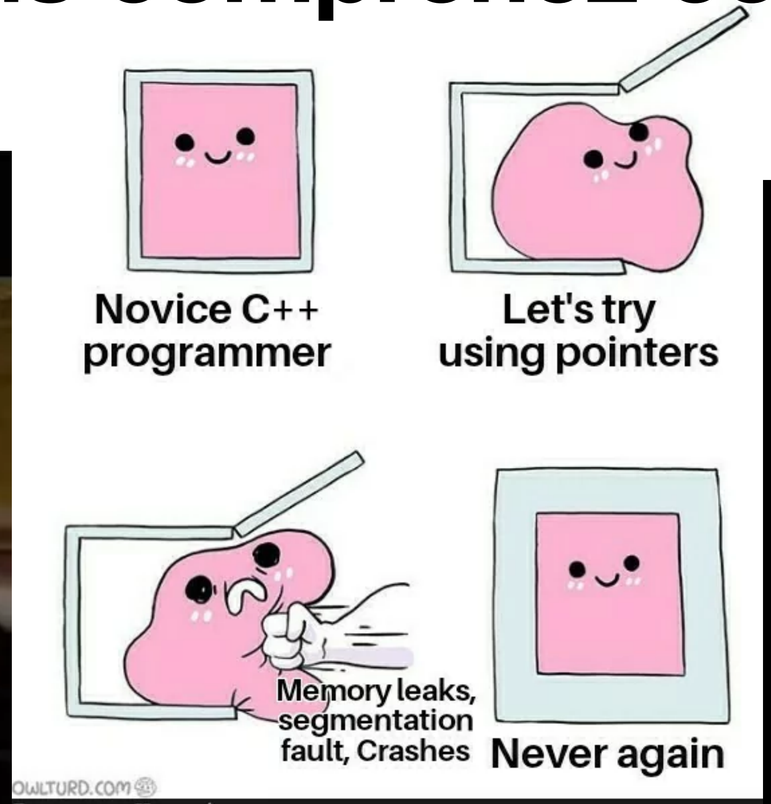
- Dans les langages comme Java, Scala, ou Python **tout est un pointeur**!

  … sauf les types de base

- La plupart des autres langages sont orientés objet — des **struct ++**
- Par contre, les autres langages ont rarement accès à la mémoire
- La gestion de la mémoire est souvent faite à l'aide d'un *Garbage Collector* = "Ramasseur de Poubelle"

Maintenant vous comprenez ce genre de blague

Sorry…



offthepack • 12m

wait can u explain pointers to me

4

Noah5900 • 8m

just keep putting & and * until it works

3

Posted in r/ProgrammerHumor

reddit

ProgrammerHumor.io


Windows
An error has occurred. To continue:
Press Enter to return to Windows, or
Press CTRL+ALT+DEL to restart your computer. If you do this,
you will lose any unsaved information in all open applications.
Error: 0E : 016F : BFF9B3D4
Press any key to continue _