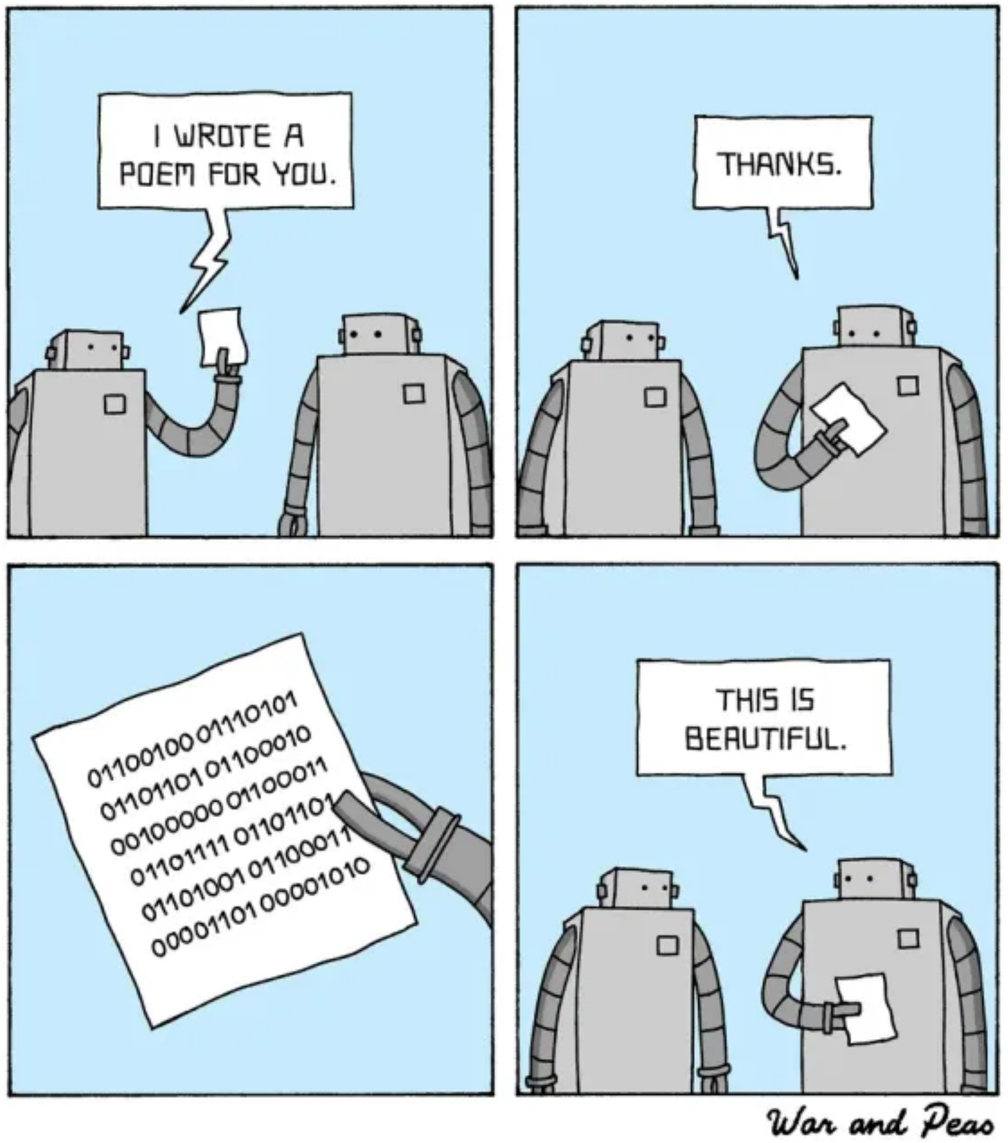
Fichiers



ICC-C Cours 12

Rappels

- Ouvrir un fichier

le chemin (*path*) vers le fichier qu'on veut lire

en quel "mode" on veut ouvrir le fichier

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

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 (Ajouter): "a"

- On écrit à la fin d'un fichier existant

Lecture en mode texte
`fscanf`

- On lit avec `fscanf`
- Comme `scanf` mais le 1er paramètre = un objet fichier valide (“ouvert”)

```
char buffer[100];
int items_read;

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

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

Lecture depuis stdin
`fscanf`

- `stdin` est aussi un “fichier” qui est déjà ouvert —
  c’est une variable globale dans `stdio.h` de type `FILE*`

```
char buffer[100];
int items_read;

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

- Même chose que

```
items_read = scanf("%s", buffer);
```

`fgets` marche aussi

- On veut lire une ligne de texte (jusqu'au `'\n'`, y compris):

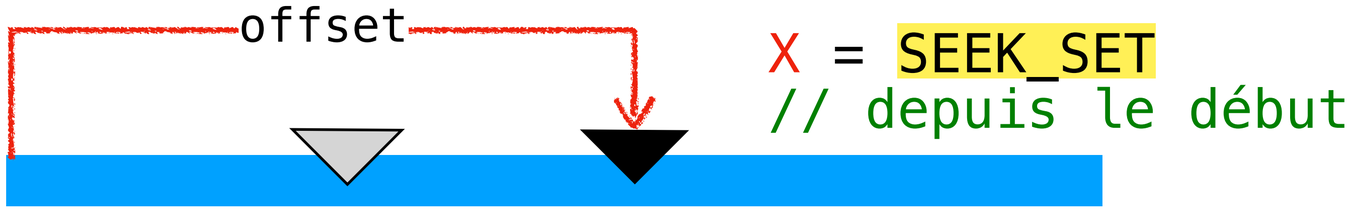
```
char buffer[100];

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

fgets(buffer, 99, file_read);
// Lit au plus 99 caractères ou jusqu’au \n suivant
```

Curseur

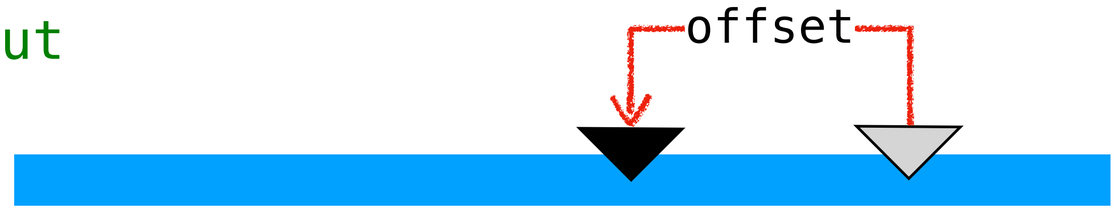
- Indique où on se trouve dans le fichier par rapport au début

```
int where = ftell(file_read);
```

- On peut bouger le curseur avec `fseek(file_read, offset, X)`

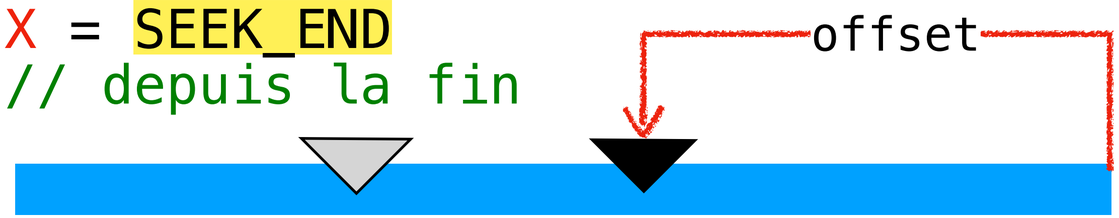



```
X = SEEK_SET
// depuis le début
```




```
X = SEEK_CUR
// depuis la position actuelle
```

```
X = SEEK_END
// depuis la fin
```



Fin du fichier

- Si on est à la fin du fichier, alors `feof` retourne vrai

  ```
  feof(file_read)
  ```

- D'autres indices que nous sommes arrivés à la fin:
  - Lire avec `fgets` retourne `NULL`
  - `fscanf` retourne `0`

"Can you be more specific?"

Écriture en mode texte
`fprintf, fputs`

- On écrit dans des fichiers en mode texte avec `fprintf`:

```
int items_written;

FILE* file_write = fopen("message.txt", “w"); // Write mode – fichier écrasé

items_written = fprintf(file_write, “ICC %d\n”, 2025);
```

- … ou avec `fputs`:

```
fputs(“Je peux écrire dans des fichiers!”, file_write);
// ⚠️ l’objet file_write est le dernier argument
```

Buffering

- L’écriture n’est pas toujours instantanée

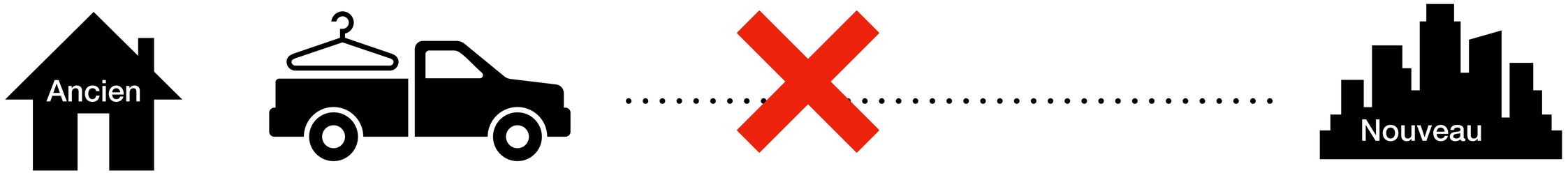
```c
#include <unistd.h>
#include <stdio.h>

int main()
{
    printf("Un deux trois ");
    sleep(1);
    printf("Nous irons au bois\n");
}
```

```
❯ ./flush

⌛ 1 seconde

Un deux trois Nous irons au bois
```

- L’affichage à l’écran se fait **tout à la fois** après 1 seconde

Buffering

- Souvent les mécanismes d’entrée/sortie utilisent des “buffers” = mémoire tampon
- C’est *inefficace* d’envoyer le moindre octet écrit tout de suite
  - E.g., réseau, disque dur, …
  - ~ Le camion de déménagement ne part pas sans être bien rempli



Buffering

Le buffer
de la sortie standard

stdout

“”

```
printf("Un deux trois ");
```

stdout

"Un deux trois "

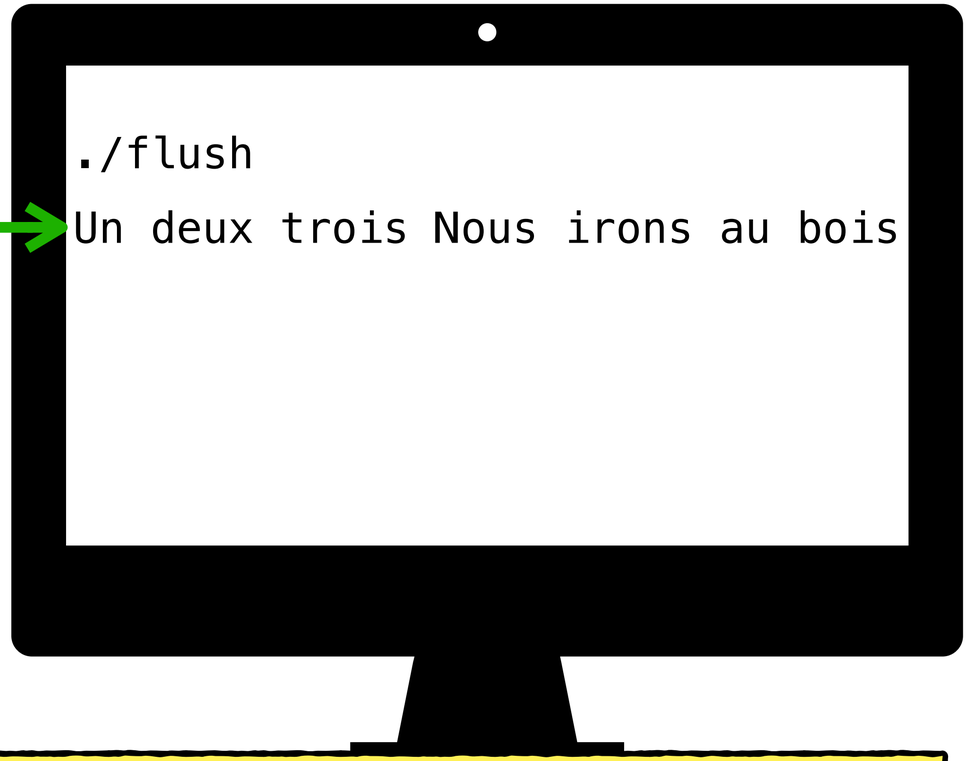
```
printf("Nous irons au bois\n");
```

stdout

"Un deux trois Nous
irons au bois\n"

stdout

“”



Le buffer stdout se vide (=on affiche)
quand un caractère \n y est écrit

Flush and close

- On peut déclencher l’écriture du contenu du buffer pour le fichier `file` avec

  ```
  fflush(file);
  ```

  - **Flush** = tirer la chasse
  - On fait partir le camion avec un cintre dedans, mais bon
- C’est fait automatiquement quand on **ferme** le fichier avec

  ```
  fclose(file);
  ```

- **N’oubliez pas de fermer le fichier!**

Fichiers binaires

- On peut lire/écrire “en mode binaire” (= des suites d’octets) depuis un fichier
- Ingrédients:
  - Un objet fichier obtenu par `fopen`
  - Un tableau (dynamique) d’objets (`int`, `float`, `struct`, etc.)
  - Une méthode pour convertir ce tableau en suite d’octets (et vice-versa)
  - **Variante lecture:** on veut lire depuis le fichier vers le tableau
  - **Variante écriture:** on veut écrire dans le fichier le contenu du tableau

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, structs, etc.
- … et des objets qui **ne sont pas** sérialisables (spécifiques à l'exécution):
  - Des pointeurs (adresses), des descripteurs de fichiers ouverts, etc.

Sérialisation des types

- Types numériques = “facile”

```
int valeur = 1234;
char *octets = (char *)&valeur;
```

- Automatiquement converti en binaire

```
|11010010|00000100|00000000|00000000|
```

- “Little Endian” pour Intel x86 et ARM
- Pareil pour structs, tableaux, etc.

Lecture binaire

fread

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

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

int nombre_objets_lus = fread(tableau,
                              taille_objet,
                              n_objets,
                              file);
```

Objets lus avec succès

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

Taille d'un seul objet (bytes)

Combien d'objets on veut lire

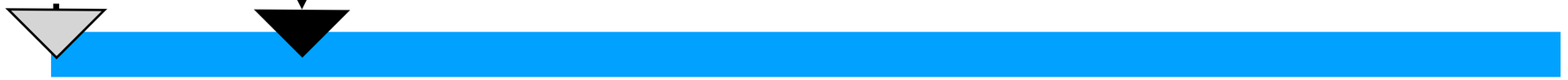
Lecture binaire
`fread`

```c
char buffer[100];
int items_read;

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

items_read = fread(buffer, 1, 7, file);
```

taille d’un objet (ici 1 byte)

nombre d’objets

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



Pas de `0` ajouté à la fin!
On a juste lu des octets, `fread` ne connait pas les strings!

buffer: |’D'(68)|'e'(101)|'a'(97)|'r'(114)|' '(32)|'M'(77)|’r’(114)|???|...
items_read: 7

Écriture binaire
`fwrite`

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

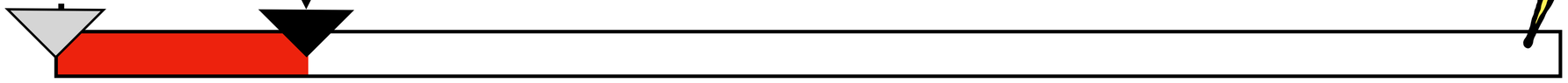
```
char buffer[100] = “coucou”;
int items_written;
FILE* file = fopen("message.txt", “w"); // Write mode

items_written = fwrite(buffer, 1, 7, file);
```

taille d'un objet

nombre d'objets

Fichier écrasé par l'ouverture en mode “w”

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



buffer: |'c'(99)|'o'(111)|'u'(117)|'c'(99)|'o'(111)|'u'(117)|'x00'(0)|
items_written: 7

Sérialiser & écrire des struct

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

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

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

fwrite(records, sizeof(struct call_record), 2, rec_file);

fclose(rec_file);
```

Désérialiser et lire des struct

Deserialize

```c
struct call_record records[10];

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

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

fclose(rec_file);
```

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

On ne sait pas combien de struct sont dans le fichier!

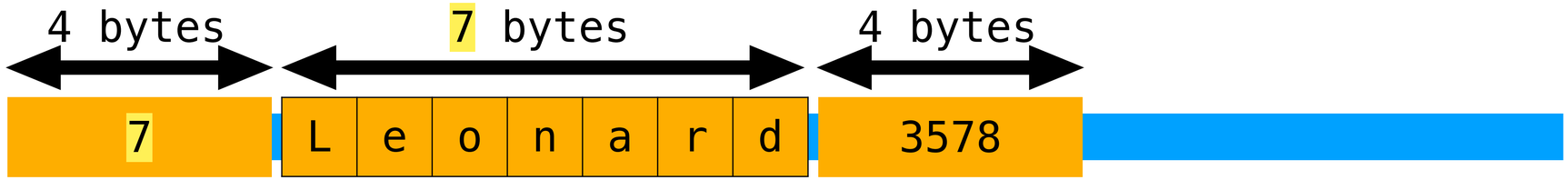
On en lit 10 à la fois (au plus)

Sérialiser des struct de taille variable

```
struct info
{
    char *nom;
    int temps_sec;
};
```

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

```
void to_file(FILE *out, const struct info *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);
}
```



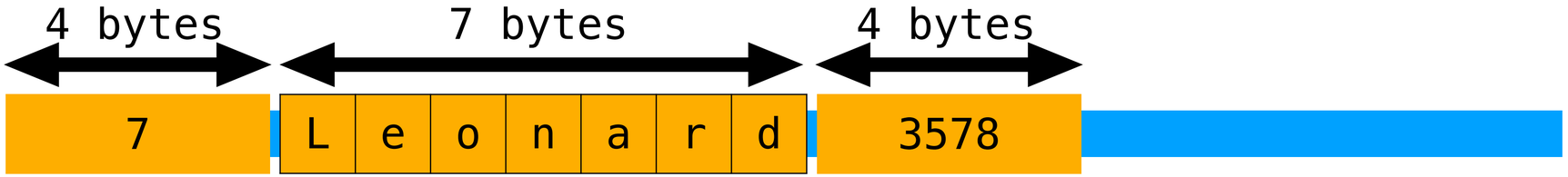
https://www.20min.ch/fr/story/canton-de-vaud-pres-de-33-000-coureurs-aux-20km-de-lausanne-103336310
Dimanche, le Kényan Leonard Bor a réussi en 59’ 38 le deuxième meilleur chrono de l’histoire sur l’épreuve des 20 km [de Lausanne]

Désérialiser des struct de taille variable

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

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

```c
void from_file(FILE *in, struct info *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);
}
```



Fonctions pour gérer des blocs de mémoire

Copier

- Souvent pour gérer des blocs de mémoire on utilise le type `void*`
- Pour copier un bloc de mémoire: `memcpy` (dans `string.h`)

```
void *memcpy(void *dst, const void *src, size_t n);
```

- Exemple:

```
int tableau[] = {10, 20, 30};

int *tableau_copie = malloc(sizeof(tableau));

memcpy(tableau_copie, tableau, sizeof(tableau));
```

Allouer la mémoire pour la copie

Faire la copie octet par octet

Fonctions pour gérer des blocs de mémoire

Comparer

- On veut savoir si deux blocs sont identiques: la fonction `memcmp`

```c
int memcmp(const void *s1, const void *s2, size_t n);
```

- Comme si on faisait la **différence** entre les deux blocs, retourne
  - un nombre négatif si `s1 < s2` (ordre lexicographique)
  - `0` si `s1 == s2`
  - Un nombre positif si `s1 > s2` (ordre lexicographique)

Fonctions pour gérer des blocs de mémoire

Comparer deux strings

- On veut savoir si deux strings sont identiques: les fonctions `strcmp`, `strncmp`

```
int strcmp(const void *s1, const void *s2);
int strncmp(const void *s1, const void *s2, size_t n); // premiers n
```

- Comme si on faisait la **différence** entre les deux strings, retourne
  - un nombre négatif si `s1 < s2` (ordre lexicographique)
  - `0` si `s1 == s2`
  - Un nombre positif si `s1 > s2` (ordre lexicographique)

Fonctions pour gérer des blocs de mémoire

Copier string

- Pour copier un string: `strcpy`, `strncpy`

```
char *strcpy(char *dst, const char *src);
```

- La longueur n'est pas spécifiée, car il y a le marqueur de fin 0
- **Dangereux**, sauf si nous sommes 100% sûrs d'avoir assez de mémoire dans `dst`
- Sinon (mieux) utiliser

```
char *strncpy(char *dst,
              const char *src,
              size_t n);
```

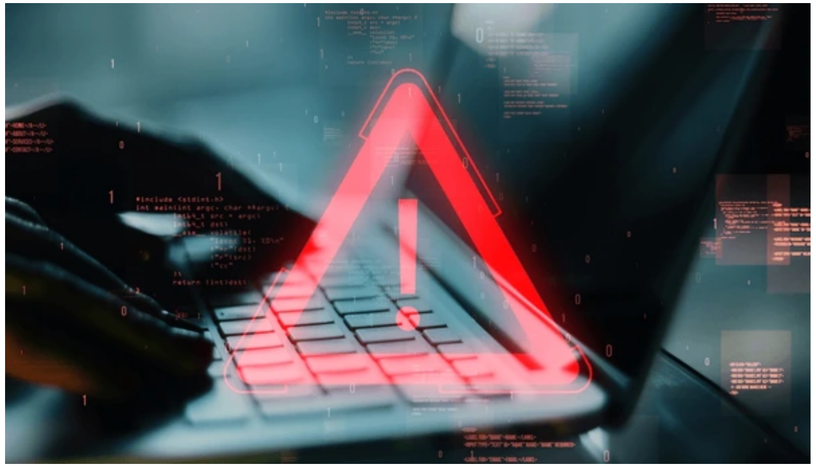
```
char *salut = "Salut";

char *salut_copie =
  malloc(strlen(salut) + 1);

strncpy(salut_copie, salut,
        strlen(salut) + 1);
```

Allouer la mémoire pour la copie — ok!

Faire la copie avec le char 0 à la fin

Buffer overflow

- Attaque informatique qui exploite la gestion de la mémoire interne d’un programme
- Peut arriver si on écrit des octets dans un tableau sans faire trop attention à sa taille
- L’écriture déborde sur des variables qui se trouvent dans le même voisinage



Exemple de buffer overflow

```c
int auth(const char *secret)
{
    int result = 0;
    char mdp[9];

    printf("Mot de passe (max 8 chars):\n");
    scanf("%s", mdp);

    if (!strcmp(mdp, secret))
    {
        result = 1;
    }

    return result;
}
```

```c
auth("ICC2025");
```

- Lit `mdp` du clavier
- Compare avec `ICC2025`
- Si égal, alors retourne `1`
- Sinon retourne `0`

Exemple de buffer overflow

```c
int auth(const char *secret)
{
    int result = 0;
    char mdp[9];

    printf("Mot de passe (max 8 chars):\n");
    scanf("%s", mdp);

    if (!strcmp(mdp, secret))
    {
        result = 1;
    }

    return result;
}
```

```
auth("ICC2025");

> ./do_auth
Mot de passe (max 8 chars):
coucou
Invalide!

> ./do_auth
Mot de passe (max 8 chars):
ICC2025
Correct!

> ./do_auth
Mot de passe (max 8 chars):
Hackerman!
Correct!
```

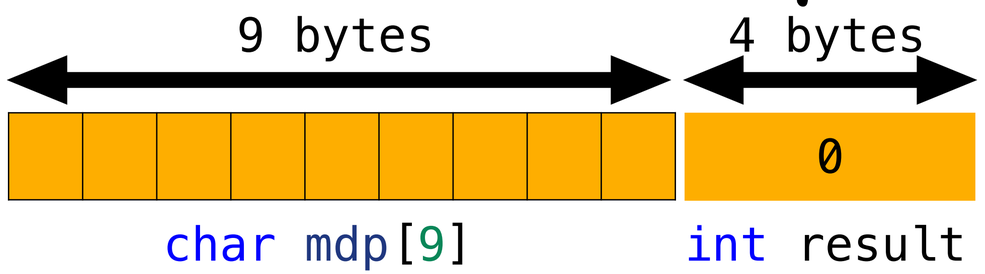
Exemple de buffer overflow

```c
int auth(const char *secret)
{
    int result = 0;
    char mdp[9];

    printf("Mot de passe (max 8 chars):\n");
    scanf("%s", mdp);

    if (!strcmp(mdp, secret))
    {
        result = 1;
    }

    return result;
}
```

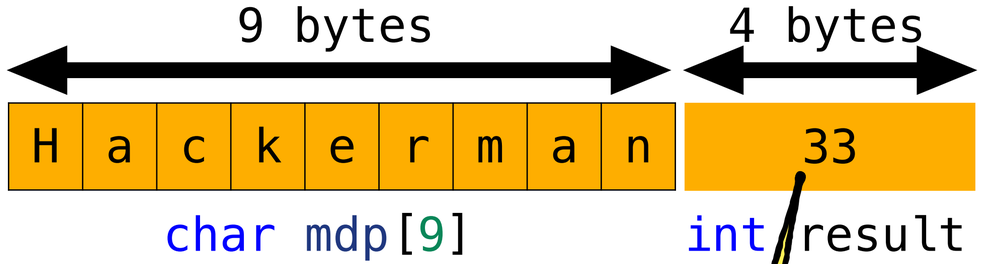
On n'arrive jamais ici

Pourtant, la fonction retournera 33, pas 0

Se trouve juste après physiquement…




```
Mot de passe (max 8 chars):
Hackerman!
```



Le code ASCII de '!', l'écriture a débordé

Exemple de buffer overflow

```
int auth(const char *secret)
{
    int result = 0;
    char mdp[9];

    printf(“Mot de passe (max 8 chars):\n");
    scanf("%s", mdp);

    if (!strcmp(mdp, secret))
    {
        result = 1;
    }

    return result;
}
```

Qu’est-ce qui est faux?

- Pas de limite imposée sur le nombre de caractères lus!
- Résolution:

```
scanf("%8s", mdp);
```

Les variantes récentes de compilateur ne placent plus les variables locales à proximité.

Désactiver la protection (**bad idea!**):

```
gcc -fno-stack-protector \
    -o do_auth do_auth.c
```