# Organisation et utilisation des systèmes d'exploitation 2

3. Rappels sur le langage C

Guillaume Pierre

Université de Rennes 1

Hiver 2014




## Table of Contents





## Table of Contents





## Gestion de la mémoire

- Jusqu'à présent la mémoire était allouée de façon statique
  - On peut savoir facilement combien de mémoire sera allouée en lisant le programme
  - La mémoire est réservée au moment de la compilation
- Mais on veut souvent décider de la quantité de mémoire nécessaire **pendant l'exécution!**
  - On a besoin d'une chaîne de caractères, mais on ne sait pas encore sa taille
  - On a besoin d'un tableau, mais on ne sait pas encore combien d'éléments il nous faut
  - Les tailles dépendent de résultats de calculs, des entrés clavier de l'utilisateur, etc.

## Allocation de mémoire dynamique

- `malloc()` alloue la quantité de mémoire que vous demandez:

```
#include <stdlib.h>
void *malloc(size_t size);
```

  - `malloc` prend a taille (en octets) comme paramètre
    - Par exemple: pour stocker 3 entiers il nous faut `3*sizeof(int)` octets
  - `malloc` retourne un pointeur vers la mémoire qu'il nous a alloué
    - Ce pointeur est un `void *`, c'est-à-dire "pointeur vers n'importe quel type de données"
    - Ne le sauvez pas comme un `void *`! Il faut le "transformer" en pointeur utilisable:

```
#include <stdlib.h>
int *i = (int *) malloc(3*sizeof(int));
i[0] = 12;
i[1] = 27;
i[2] = 42;
```



## Désallocation de mémoire dynamique

- Après avoir alloué de la mémoire avec `malloc`, cette mémoire restera allouée jusqu'à ce que vous la désallouiez

```
#include <stdlib.h>
void free(void *pointer);
```

- Après avoir fini d'utiliser de la mémoire dynamique, **il faut absolument la désallouer!**
  - Sinon elle restera allouée (et inutilisée) jusqu'à la fin du programme

```
int main() {
  int *i = (int *) malloc(3*sizeof(int));
  /* Utiliser i */
  free(i);
  /* Faire quelque chose d'autre */
}
```



## Evitez les autres fonctions de mémoire dynamique!

- Dans le bons livres de C on vous parle de fonctions "avancées" pour gérer la mémoire dynamique
  - `calloc()`, `realloc()` etc.

**Ne les utilisez pas!**

- Ces fonctions sont plus difficiles à utiliser qu'elles n'en ont l'air
  - Il est très facile de faire des erreurs difficiles à débogguer
  - De toute façon **elles sont très rarement nécessaires...**
  - ... et certainement dans dans vos programmes pour SYR2



## Utilisation de la mémoire dynamique

- On peut retourner de la mémoire dynamique comme résultat d'une fonction

```
int *createIntArrayWrong() {
  char tmp[32];
  return tmp;                          /* FAUX! */
}

int *createIntArray(int size) {
    return (int *) malloc(size*sizeof(int)); /* CORRECT */
}

int main() {
    int *array = createIntArray(10);
    /* ... */
    free(array);
    return 0;
}
```

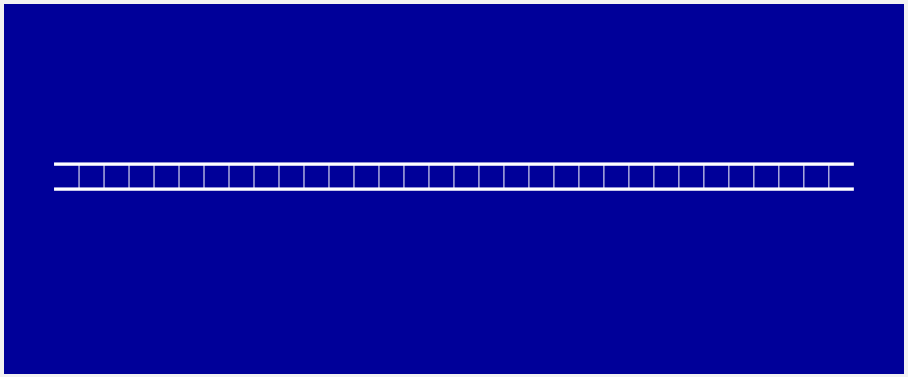
## Allouer un struct [1/4]

- Que se passe-t-il si on utilise malloc pour allouer un struct?

```
#include <stdlib.h>

struct complex {
    int i;
    double d;
    char string[10];
}

int main() {
    struct complex *c = (struct complex *) malloc(sizeof(struct complex));

    /* Faire quelque chose d'utile avec c */

    free(c);
    return 0;
}
```

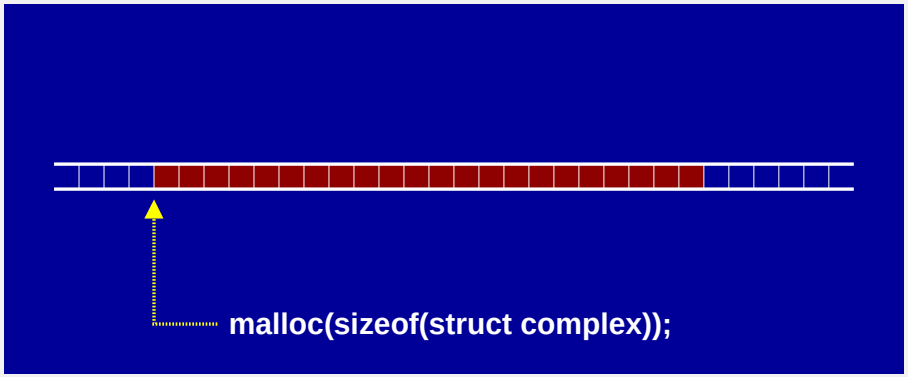


## Allouer un struct [2/4]



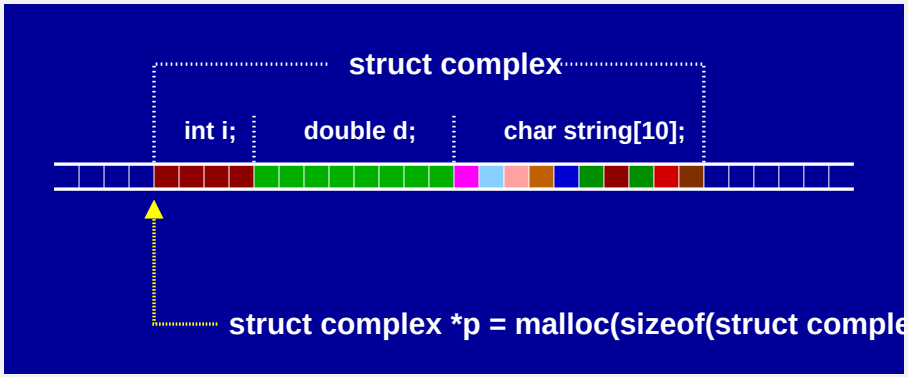


## Allouer un struct [3/4]






## Allouer un struct [4/4]



## Fuites de mémoire [1/2]

- Il faut **toujours** garder un pointeur vers la mémoire allouée dynamiquement
  - C'est indispensable pour pouvoir utiliser cette mémoire, puis pour la désallouer
  - Si vous ne le faites pas vous obtenez une **fuite de mémoire**
  - Les fuites de mémoire réservent lentement toute la mémoire de votre machine, la menant droit vers le crash!

```
int main() {
  int *i = (int *) malloc(3*sizeof(int));
  i = 0;      /* Oups, j'ai perdu le pointeur vers la mémoire dynamique */
  free(???);  /* Je ne peux plus la désallouer */
```

**Les fuites de mémoire sont des bugs très graves!**
**Si vos programmes ont des fuites de mémoire,**
**vos TPs auront des fuites de notes :-(**



## Fuites de mémoire [2/2]

- Si la machine n'a plus de mémoire disponible, `malloc` retourne un pointeur `NULL`

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

int main() {
    int *array = (int *) malloc(10*sizeof(int));

    if (array == NULL) {
        printf("Pas de mémoire disponible!\n");
        return 1;
    }

    /* Faire quelque chose d'utile ici */
    return 0;
}
```



## Quand ne faut-il **PAS** utiliser malloc()?

- `malloc()` est utile si:
  - **Vous ne connaissez pas la taille** du buffer au moment de la compilation
  - Ou **vous ne savez pas combien de buffers** seront nécessaires
  - Ou **vous voulez garder le buffer** après la fin de la fonction

- Dans tous les autres cas, **utilisez de la mémoire statique!**

```
int foo() {
  struct timeval *t;
  t = (struct timeval *)
      malloc(sizeof(struct timeval));
  ...
  gettimeofday(t);
  ...
  free(t);
}
```

⇒

```
int foo() {
  struct timeval t;
  ...
  gettimeofday(&t);
  ...
  /* Pas besoin de free() */
}
```



## Table of Contents

## Les buffers et les chaînes de caractères

- Toutes les chaînes de caractères sont des buffers, mais tous les buffers ne sont pas des chaînes de caractères
- **Buffers:**
  - Un buffer est un espace mémoire d'une certaine taille
  - Il peut contenir n'importe quelles valeurs, y compris des '\0'
  - Les fonctions orientées buffer ne tiennent pas compte du contenu des buffers. Ils manipulent X octets à partir d'un pointeur
- **Chaînes:**
  - Une chaîne est contenue dans un buffer. La chaîne peut grandir jusqu'à atteindre la taille du buffer - 1
  - Une chaîne peut contenir n'importe quelle valeur, **sauf '\0'**. Le caractère '\0' veut dire: "la chaîne s'arrête ici"
  - Les fonctions orientées chaînes lisent le contenu de chaque octet et s'arrêtent dès qu'elles voient une valeur '\0'



## Fonctions orientées buffers ou chaînes

| | **Buffers** | **Chaînes** |
|---|---|---|
| Créer | char buf[1024]; | char str[1024]; |
| Effacer | bzero(buf,1024); | str[0]='\0'; |
| Copier | memcpy(buf2,buf,1024); | strncpy(buf2,buf,1024); buf2[1023]='\0'; |
| Comparer | memcmp(buf,buf2,1024); | strncmp(buf,buf2,1024); |
| Mesurer | sizeof(buf) | strlen(buf) |



## Be careful whether you manipulate strings or buffers

- Suivant l'usage il vaut mieux utiliser des buffers ou des chaînes
  - Données binaires ⇒ buffers
    (par ex: une image JPG contient des '\0')
  - Données texte ⇒ chaînes...



## Table of Contents

## Les pointeurs de fonctions

- Les pointeurs de fonctions sont une sorte spéciale de pointeurs
  - Un pointeur désigne une adresse en mémoire
  - Quand un programme s'exécute, les fonctions sont stockées en mémoire

  $\Rightarrow$ Il doit être possible de créer un pointeur qui pointe vers une fonction!
- Les pointeurs de fonctions sont utiles pour passer une fonction comme paramètre d'une autre fonction
  - Par exemple, la fonction `signal()` indique au système d'exploitation quelle fonction il doit lancer quand il reçoit un certain signal
  - Il faut être capable de désigner une fonction



## Typage des pointeurs de fonction

- Les pointeurs de fonction sont typés
  - Exemple: un pointeur vers une fonction qui prend deux `int` comme paramètres et qui retourne un `float`
- Désolé, la syntaxe est assez laide...

```
result_type (*pointer_name) (parameter_list);
```

  - Par exemple: un pointeur vers une fonction qui prend deux `int` comme paramètres et qui retourne un `float`
    `float (*mon_pointeur) (int p1, int p2);`
  - Ici `mon_pointeur` est le nom de la variable de type pointeur
- On peut utiliser les pointeurs de fonction comme des variables, des paramètres de fonction ou comme des types de valeurs retournés par les fonctions



## Exemples [1/3]

```
int max(int a, int b) {
    return (a > b ? a : b);
}
int mul(int x, int y) {
    return x*y;
}

int main() {
    int result;
    int (*func) (int p1, int p2);    /* func est un pointeur de fonction */

    func = max;                      /* func pointe vers la fonction max */
    result = func(1, 2);
    printf("%d\n", result);

    func = mul;                     /* func pointe vers la fonction mul */
    result = func(3,4);
    printf("%d\n", result);
    return 0;
 }
```



## Exemples [2/3]

```
#include <stdio.h>

int max(int a, int b) {
    return (a > b ? a : b);
}

/* foo prend comme paramètres deux ints et un pointeur de fonction */
int foo(int a, int b, int (*func) (int p1, int p2)) {
    return func(a, b);
}

int main() {
     int result = foo(11, 22, max);
     printf("%d\n", result);
     return 0;
}
```

## Exemple [3/3]

```
#include <stdio.h>

int max(int a, int b) {
    return (a > b ? a : b);
}

/* faa() ne prend aucun paramètre et il retourne un pointeur de fonction */
int (*faa()) (int p1, int p2) {
  return max;
}

int main() {
     int result;
     int (*func) (int p1, int p2);
     func = faa();
     result = func(11, 22);
     printf("%d\n", result);
     return 0;
}
```



## Table of Contents





## Manipuler les fichiers

- Pour manipuler un fichier il faut:
  - Ouvrir le fichier
  - Lire/écrire
  - Fermer le fichier
- Les fonctions pour lire/écrire dans les fichiers sont orientées **buffer**
  - Elles ne regzardent pas la valeur de chaque octet, elles lisent/écrivent simplement X octets



## Ouvrir un fichier [1/2]

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- Paramètres:
  - `pathname`: le nom du fichier
  - `flags`:
    - `O_RDONLY`: accès read-only
    - `O_WRONLY`: accès write-only
    - `O_RDWR`: accès read-write
    - `O_CREAT`: si le fichier n'existe pas, alors le créer
  - `mode`: Spécifie les droits d'accès des fichiers que vous créez
    - 0600: accès read-write pour vous, rien pour les autres
    - 0644: accès read-write pour vous, read-only pour les autres

## Ouvrir un fichier [2/2]

- `open()` retourne un entier:
  - $-1$ signifie "erreur:" l'ouverture a échoué
  - $\geq 0$: c'est le "descripteur de fichier" qui représente le fichier ouvert. Il faudra le passer à toutes les autres fonctions qui manipulent les fichiers.
- Inutile de spécifier un "mode" si vous ne créez pas un nouveau fichier

```
fd = open("this_file_already_exists", O_RDONLY);
```

- On peut combiner plusieurs flags ensemble:

```
fd = open("foo", O_RDWR|O_CREAT, 0644);
```



## Lire un fichier

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

- Paramètres:
  - `fd`: le descripteur du fichier à lire
  - `buf`: un buffer où copier les données lues
  - `count`: le nombre d'octets à lire
- `read()` retourne un entier:
  - $-1$ signifie "erreur:" la lecture a échoué
  - $\geq 0$: ce nombre indique le nombre d'octets effectivement lus (si ce nombre est différent du nombre d'octets que vous avez demandé alors vous avez atteint la fin du fichier)



## Ecrire dans un fichier

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

- Parameters:
  - `fd`: le descripteur du fichier
  - `buf`: un buffer contenant les données à écrire
  - `count`: le nombre d'octets à écrire
- `write()` retourne un entier:
  - $-1$ signifie "erreur:" l'écriture a échoué
  - $\geq 0$: le nombre d'octets effectivement écrits (normalement identique au nombre demandé, ou alors vous avez trouvé une exception bizarre)



## Fermer un fichier

```
#include <unistd.h>

int close(int fd);
```

- Paramètres:
  - `fd`: le descripteur du fichier
- `close()` retourne un entier:
  - $-1$ signifie "erreur:" la fermeture a échoué
  - 0 veut dire OK
- N'oubliez pas de fermer vos descripteurs de fichiers quand vous n'en avez plus besoin!

## Fichier spéciaux

- Quand votre programme démarre, 3 descripteurs de fichiers sont créés pour vous automatiquement
  - Vous pouvez les utiliser comme vous voulez
  - Vous n'êtes pas obligés de les fermer après usage
- Les trois descripteurs standard:
  - **0: l'entrée standard du programme (stdin).** Quand l'utilisateur tape sur le clavier, le message peut être lu dans le descripteur 0. Vous ne pouvez pas écrire dans ce descripteur.
  - **1: la sortie standard du programme (stdout).** Ce que vous écrivez dans ce descripteur apparaîtra à l'écran.
  - **2: la sortie d'erreur de votre programme (stderr).** Ce que vous écrivez dans ce descripteur apparaîtra à l'écran. A utiliser pour les messages d'erreur.



## Que faire quand une fonction échoue?

- Il faut **toujours** vérifier les valeurs retournées par les fonctions standard, et vérifier si elles indiquent des erreurs
  - Sinon déboguer vos programmes sera infernal...
- Il existe une variable globale standard appelée `errno`
  - Elle est définie dans `<errno.h>`
- Quand une fonction standard échoue, elle écrit un code d'erreur dans `errno`
- Pour convertir `errno` en anglais:

```
int fd = open("wrongfilename",O_RDONLY);
if (fd<0) {
  perror("Erreur d'ouverture du fichier");
  exit(1);
}
```



## Vous pouvez également écire directement dans stderr

- `printf()` écrit normalement dans **stdout**.
- Pour écrire dans stderr:

```
fprintf(stderr,"Ce message est ecrit dans stderr\n");
```

- Par exemple:

```
int main(int argc, char **argv) {
   if (argc!=2) {
      fprintf(stderr,"Usage: blablabla\n");
   }

   [...]
}
```



## Table of Contents

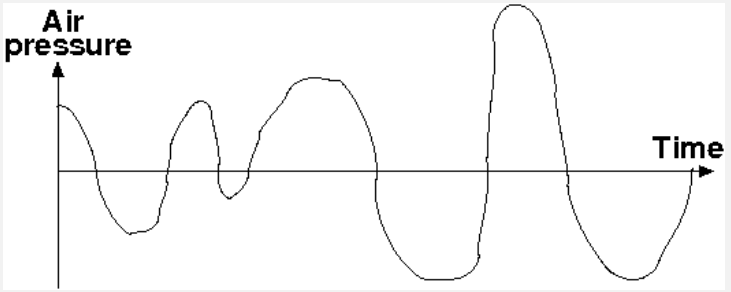
## Gestion du son

- Votre projet à la fin de SYR2 manipulera des fichiers audio
- Pas de panique, **ce n'est pas aussi difficile qu'il n'y paraît!**
  - Vous disposerez d'une petite librairie qui fait les parties difficiles pour vous
  - Mais il faudra comprendre la structure des fichiers WAV pour réaliser la fin du projet



## Les codages audio

- Le son est une onde mécanique

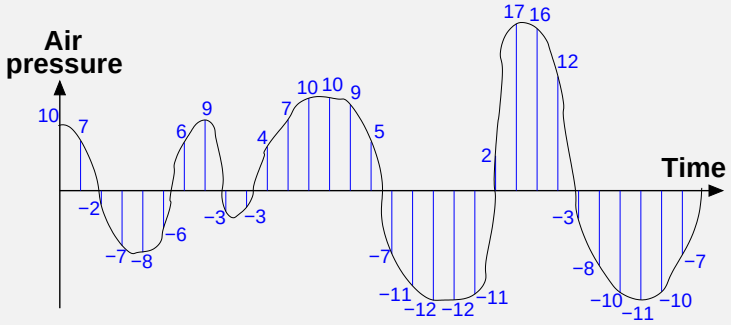



- Comment représente-t-on une onde analogique en une représentation numérique?



## Les codages audio

- Solution: échantillonner l'onde à intervalles réguliers




- Un fichier audio contient la suite des mesures effectuées
  - Ici: 10, 7, -2, -7, -8, -6, 6, 9, -3, -3, 4, etc.
- La plupart des formats audio compressent les valeurs pour prendre moins de place
  - Mais les fichiers WAV contiennent simplement les échantillons sans aucune compression (plus facile!)



## Les en-têtes des fichiers WAV

- Tous les fichiers audio commencent par un **en-tête** pour indiquer comment il a été codé:
  - Combien de **canaux?** (1=mono; 2=stéréo)
  - **Fréquence d'échantillonnage?** (les CDs audio sont échantillonnés à 44100Hz; d'autres fichiers WAV peuvent utiliser d'autres fréquences)
  - Quelle est la **représentation d'un échantillon?** (entiers 8 bits non-signés ou entiers 16 bits signés)

## Lire un fichier WAV [1/2]

- La librairie contient une fonction pour lire l'en-tête du fichier:

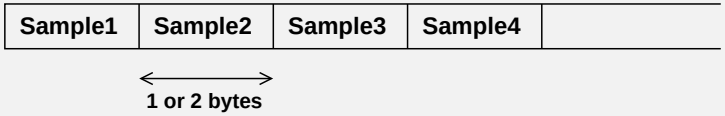
```
#include "audio.h"
int aud_readinit(char *filename, int *sample_rate,
                 int *sample_size, int *channels);
```

  - Indiquer le nom du fichier à ouvrir (filename)
  - Passer des pointeurs vers trois entiers
  - La fonction va:
    1. Ouvrir le fichier
    2. Lire l'en-tête WAV et écrire la fréquence d'échantillonnage, la taille des échantillons et le nombre de canaux dans vos variables
    3. Retourner le descripteur du fichier (qui pointe juste après l'en-tête WAV)

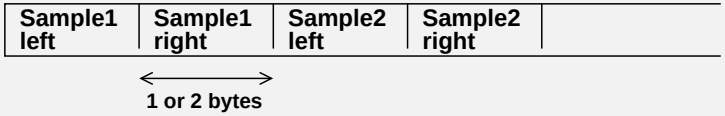


## Lire un fichier WAV [2/2]

- Vous pouvez alors lire le fichier WAV vous-même:
  - Mono:




  - Stéréo:




- N'oubliez pas de fermer le fichier après usage!



## Jouer un fichier WAV

- Vous devez d'abord ouvrir le "device audio" et déclarer quel genre de fichier WAV vous allez fournir:

```
#include "audio.h"
int aud_writeinit(int sample_rate, int sample_size, int channels);
```

  - Il faut passer la fréquence d'échantillonnage, la taille des échantillons et le nombre de canaux
  - Attention: vous ne pouvez pas passer n'importe quelles valeurs et espérer que cela va marcher. Si vous passez des valeurs "bizarres" le système choisira une valeur proche à votre place...

- La fonction retourne un descripteur de fichier
  - Il suffit d'écrire vos échantillons dedans!
  - Et ne pas oublier de fermer le fichier ensuite...



## Manipuler un fichier audio

- Je vous demanderai de programmer des petits *filtres* qui transforment le signal sonore

1. **Facile:** mentir au device audio
   - ⋆ Exemple: le fichier est codé à 44100Hz mais vous prétendez qu'il est à 22050Hz
   - ⇒ La carte son va jouer le son deux fois moins vite!
2. **Plus intéressant:** manipuler les échantillons!
   - ⋆ Echanger les deux canaux
   - ⋆ Pour ajuster le volume: multiplier tous les échantillons par une constante
   - ⋆ Ajouter de l'écho
   - ⋆ Etc.