Compilation, fichiers d'en-tête, et make

CS221 — CM #2

Sébastien MICHELLAND

Grenoble INP — Esisar

Votre enseignant/TD-man du jour

Sébastien MICHELLAND
- Doctorant avec Mme. GONNORD et M. DELEUZE
- Bureau D203 (Bâtiment D, 2ème étage) (… pas toujours là)
- sebastien.michelland@lcis.grenoble-inp.fr

- Compilation FTW
- Ravi d'être là
- Fera du C votre langage préféré

Dans ce cours…

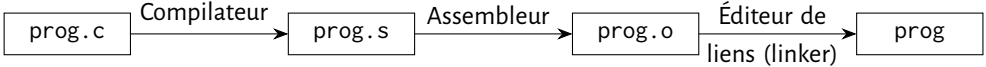
- Vue rapide du processus de compilation, pour comprendre :
  1. Comment marchent les en-têtes (`#include <stdio.h>`) ;
  2. Comment séparer un programme en plusieurs fichiers `.c` ;
  3. Comment utiliser des bibliothèques.
- Introduction au système de build `make`, pour automatiser la compilation.

Spoilers / révisions express :

1. Le compilateur compile *un seul fichier* `.c` *à la fois* et ne "connaît" les contenus des autres fichiers que grâce aux en-têtes (`.h`).
2. Les en-têtes sont juste une promesse publique sur ce que les `.c` / bibliothèques contiennent.
3. Pour utiliser une bibliothèque, il faut à la fois un `#include` *et* un `-l` (`<math.h>`, `-lm`).
4. Dans `make`, les en-têtes sont des dépendances.

Quand on compile…

Étapes de compilation, vues de loin



Code source C

```c
int f(int x);

int g(int x)
{
  return f(x)+1;
}
```

Code assembleur

```
_g:
  addi  sp, sp, -16
  sw    ra, 12(sp)
  auipc ra, 0
  jalr  ra, 0(ra)
  addi  a0, a0, 1
  lw    ra, 12(sp)
  addi  sp, sp, 16
  ret
```

Fichier objet

```
411106c6 97000000
82900505 b2404101
8280...
```

Exécutable

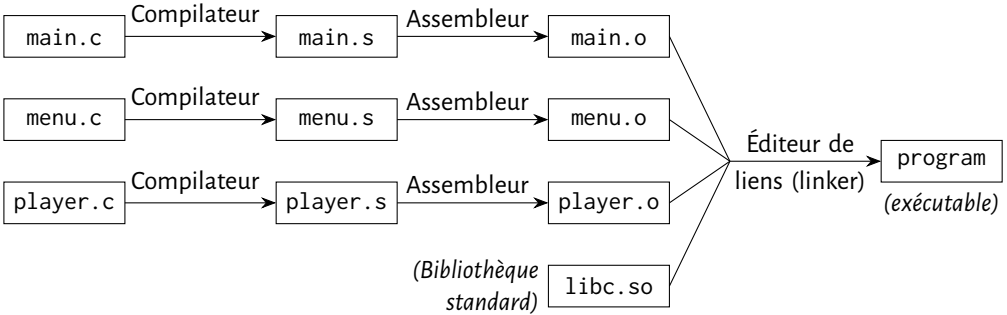
```
411106c6 e53f0505
b2404101 8280...
```

Lancé avec : `./prog`

La compilation de plusieurs fichiers est *individuelle* !

Dans un programme avec plusieurs fichiers, **les fichiers `.c` sont compilés individuellement**. Ils ne sont réunis qu'au tout dernier moment pour créer le fichier exécutable.

Illustration : `gcc main.c menu.c player.c -o program -Wall -Wextra`



Comment se fait la communication, alors ?

Les variables et fonctions du programmes viennent :

1. Des fichiers `.c`/`.o` ;
2. Des bibliothèques liées avec `-l` (lettre "L" minuscule) ;
   - Exemple : `-lc` (automatique !), la bibliothèque standard (`libc.so`)
   - Exemple : `-lm`, la bibliothèque mathématique (`libm.so`)
   - Exemple : `-lGL`, la bibliothèque OpenGL pour les graphismes 3D (`libGL.so`)

Et donc le code de `printf()` est dans `libc.so` (**pas** <stdio.h>), celui de `sqrt()` dans `libm.so` (**pas** <math.h>).

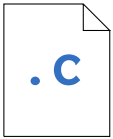
Problème : Comment le compilateur peut savoir si nos appels de fonctions sont valides s'il ne va pas regarder les autres `.c` ou les bibliothèques ?

Solution : On lui annonce avec un `.h` ce que les autres `.c` et les bibliothèques contiennent.

Les fichiers d'en-tête (.h)

Principe des fichiers d'en-tête

Chaque fichier `.c` est accompagné d'un fichier `.h` ("header") du même nom décrivant ses contenus.



Fichier source :

- Contient le code et les variables
- Tout ce qui est utilisé à l'exécution est dans ce fichier
- Privé : vu uniquement par le compilateur

Fichier d'en-tête :

- Liste des variables et fonctions du fichier `.c`
- Ne contient aucun code (uniquement un outil de communication)
- Public : inclus par les autres fichiers du projet (`#include`)

C'est différent par exemple de Python, où un module `.py` fait les deux à la fois.

Formule des fichiers d'en-tête

- Pour chaque fichier comme `player.c` du projet utilisé par un autre fichier, créer un en-tête `player.h` et y lister la vitrine de `player.c` :

Objet du programme	**Dans `player.c`**	**Dans `player.h`**
Fonction publique	`void spawn(void) { /* ... */ }`	`void spawn(void);` (1)
Variable globale non init.	`int pos_x;`	`extern int pos_x;`
Variable globale initialisée	`int skills[10] = { 0 };`	`extern int skills[10];`
Fonction privée	`static int exp(int x) { /* ... */ }`	*(rien)*
Variable globale privée	`static int anim = 42;`	*(rien)*
Définition de structure	*(rien)*	`struct s { ... };`

- (1) s'appelle le **prototype** d'une fonction.
- Inclure `player.h` dans `player.c` et tous les fichiers qui utilisent `player.c`.

Inclusion de fichiers

On peut inclure littéralement un `.h` dans un `.c` avec `#include`.

```c
// main.c
#include "player.h"
void main(void)
{
    spawn();
}
// player.h
void spawn(void);
```

```c
// main.c après l'inclusion
void spawn(void);
void main(void)
{
    spawn();
}
```

Deux délimiteurs selon le type d'en-tête :

- <> pour les en-têtes standard / bibliothèques
- "" pour les en-têtes du projet

Fiche pratique : écrire un projet avec plusieurs fichiers

Un programme en 4 temps 3 fichiers

Pour écrire un projet avec plusieurs fichiers :

1. Séparer le code en plusieurs `.c`.
2. Créer un en-tête `X.h` pour chaque fichier `X.c`.
   - (sauf le fichier principal avec `main`, qui n'est utilisé par personne)
3. Inclure `X.h` dans `X.c` et tous les utilisateurs de `X.c`.
4. Compiler tous les fichiers d'un coup :
   - `gcc main.c X.c Y.c Z.c -o main -Wall -Wextra -lm`
   - Tous les fichiers `.c` doivent être listés
   - Toutes les bibliothèques listées avec `-l` (sauf la libc est liée par défaut)
   - Bien activer `-Wall -Wextra`

Demo time !

Résumé de la démo

- Dans `player.c`, on écrit une fonction pour calculer la distance entre le joueur en `x,y` et l'origine du repère (exemple arbitraire)

```c
// player.c
#include <math.h>
#include "player.h"

float distance(float x, float y)
{
    return sqrtf(x*x + y*y);
}
```

Comme on utilise `sqrtf()`, une fonction de la bibliothèque mathématique, on inclut `<math.h>` (pour qu'elle soit annoncée) et on compilera avec `-lm` (pour avoir son code).

- On crée le fichier d'en-tête associé `player.h` dans lequel on écrit le prototype de `distance()` :

```
// player.h
float player(float x, float y);
```

Et *on inclut* `player.h` *dans* `player.c` pour que le compilateur puisse vérifier qu'on ne s'est pas trompés.

- Enfin on écrit `main.c`, dans lequel on inclut `player.h` pour pouvoir appeler `distance()` :

```c
// main.c
#include <stdio.h>
#include "player.h"

int main(void)
{
    float x = 1.25, y = 4.50;
    printf("distance = %f\n", distance(x, y));
    return 0;
}
```

Compilation et et test :

```
michelse@realm ~$ gcc main.c player.c -o program -Wall -Wextra -lm
michelse@realm ~$ ./main
distance = 4.670385
```

Résumé des fichiers dans un projet

```
michelse@realm ~$ ls -l
total 32
-rwxr-xr-x 1 michelse michelse 15536 Sep 17 22:25 main
-rw-r--r-- 1 michelse michelse   150 Sep 17 21:41 main.c
-rw-r--r-- 1 michelse michelse    78 Sep 17 22:25 Makefile
-rw-r--r-- 1 michelse michelse   105 Sep 17 21:33 player.c
-rw-r--r-- 1 michelse michelse    34 Sep 17 22:25 player.h
```

- Le code du projet : `main.c`, `player.c`, `player.h`
- Le programme exécutable : `main`
- De quoi automatiser la compilation : `Makefile`

Piège : les en-têtes sont une *promesse* et on doit la tenir !

Si on ment ou qu'on se trompe, c'est de notre faute !

```c
// pointeur.c
#include <stdio.h>
void afficher_pointeur(int *p)
{
    printf("%d\n", *p);
}
// pointeur.h
void afficher_pointeur(int p);
```

```c
// main.c
#include "pointeur.h"
int main(void)
{
    afficher_pointeur(73);
    return 0;
}
```

- Ce programme **crashe** et il n'y a pas d'erreur de compilation !

Solutions :

1. Inclure `pointeur.h` dans `pointeur.c`, comme ça le compilateur remarque l'erreur.
2. Et au passage `-Wall -Wextra` aide aussi.

Ça marche pas, je fais quoi ?

warning: implicit declaration of function `distance'

```
main.c: In function ‘main’:
main.c:7:31: warning: implicit declaration of function ‘distance’ [-Wimplicit-...
    7 |     printf("distance = %f\n", distance(x, y));
      |                               ^~~~~~~~
```

Traduction : **la fonction est appelée mais n'est pas annoncée par un `.h`.**

Raisons courantes :

1. Un `#include` a été oublié (ici `#include "player.h"`);
2. Le prototype a été oublié dans le `.h`.

fatal error: player.h: No such file or directory

```
main.c:2:10: fatal error: player.h: No such file or directory
    2 | #include <player.h>
      |          ^~~~~~~~~~
```

Traduction : **le `.h` n'existe pas ou n'a pas été trouvé.**

Raisons courantes :

1. Confusion entre `#include <>` et `#include ""` (utiliser `""` quand le `.h` fait partie du projet) ;
2. Typo dans le nom du fichier.

error: conflicting types for `distance'

```
player.c:4:7: error: conflicting types for ‘distance’; have ‘float(float,  float)’
    4 | float distance(float x, float y)
      |       ^~~~~~~~
In file included from player.c:2:
player.h:1:7: note: previous declaration of ‘distance’ with type ‘float(float,  int)’
    1 | float distance(float x, int y);
      |       ^~~~~~~~
```

Traduction : **le .h et le .c ne sont pas d'accord sur le type de la fonction.**

Solution : corriger la déclaration incorrecte.

undefined reference to `sqrtf'

```
/usr/bin/ld: /tmp/ccOrmxAW.o: in function `distance':
player.c:(.text+0x34): undefined reference to `sqrtf'
collect2: error: ld returned 1 exit status
```

(Fun fact : c'est une erreur de l'éditeur de liens, pas du compilateur !)

Traduction : **la fonction `sqrtf` est appelée mais son code n'est pas fourni.**

Raisons courantes :

1. Cette fonction vient d'une bibliothèque et l'option `-l*` appropriée n'a pas été donnée (ici `-lm`) ;
2. Un des fichiers `.c` du projet a été oublié dans la commande `gcc` ;
3. Typo dans le nom de la fonction.

Automatisation avec make

Rédaction de règles dans un Makefile

- Objectif de `make` : automatiser le lancement des commandes de compilation.
- Le fichier `Makefile` contient les commandes sous forme de *règles*.

```
# Forme générale
<CIBLE>: <DÉPENDANCES...>
        <COMMANDES...>

# Par exemple
main: main.c
        gcc main.c -o main
```

- *Cible* : Fichier produit par la commande (sortie)
- *Dépendances* : Fichiers lus par la commande (entrée)
- *Commandes* : Liste de commandes à lancer pour produire la cible

- Au lieu de taper les commandes dans le terminal, on tape "`make`" ou "`make <CIBLE>`".
- (Note : avant les commandes il faut une *tabulation*, pas des espaces.)

Cas simple : compiler tout le projet d'un coup

- Dans ce cas simple, on lance la compilation de tous les fichiers et l'édition des liens avec une seule commande. La cible est directement le programme exécutable, et tous les fichiers du projet sont des dépendances.

Dans le fichier `Makefile` :

```
program: main.c player.c player.h
	gcc main.c player.c -o program -Wall -Wextra -lm
```

- On compile le programme avec la commande `make` (qui affiche la commande et la lance).

```
michelse@realm ~$ make
gcc main.c player.c -o program -Wall -Wextra -lm
michelse@realm ~$ ./program
# (...)
```

Cas de la compilation séparée

- Dans ce cas, on compile chaque fichier `.c` en `.o` individuellement (avec l'option `-c`) et ensuite on crée le programme exécutable à partir des `.o`.
- Chaque fichier `.o` a comme dépendances son `.c` et les en-têtes ; le fichier exécutable a comme dépendance les `.o`.

Règle pour compiler `player.c` en `player.o` (à répliquer pour chaque `.o`) :

```
player.o: player.c player.h
	gcc -c player.c -o player.o -Wall -Wextra
```

Règle pour lier les `.o` en le programme final (une seule nécessaire) :

```
program: main.o player.o
	gcc main.o player.o -o program -Wall -Wextra -lm
```

- *Attention : Chaque `.o` doit avoir tous les `.h` dans ses dépendances, sinon le programme risque d'être mal compilé (silencieusement) et de ne pas fonctionner !*

Bonus

Bonus

Réflexions métaphysiques sur le code C qui ne marche pas.

S'il y a le temps.

La vraie nature du C — la mémoire

CS221 — CM #3

Sébastien MICHELLAND

Grenoble INP — Esisar

Dans ce cours…

- La gestion de la mémoire vue comme une (non-)abstraction unique au langage C ;
- Comment les variables sont créées et stockées dans la mémoire ;
- Contrats et responsabilités du programmeur sur la gestion de la mémoire.

Spoilers / révisions express :

1. En C, les programmeurs sont en fait responsables de gérer l'allocation et la libération de la mémoire ; ce n'est automatique que pour les variables locales.
2. Ça se voit particulièrement sur les tableaux parce qu'ils ne sont pas copiés quand on les passe en paramètres / valeurs de retour des fonctions (c'est les pointeurs y menant qui sont copiés).
3. On peut créer des variables de deux façons : locales (vivent jusqu'à la fin de la fonction, taille fixe, libérées automatiquement) ou avec `malloc()` (vivent aussi longtemps qu'on veut, taille qu'on veut, libérées par `free()`).
4. La *propriété* des variables (ie. la responsabilité de surveiller et libérer la mémoire associée) est très importante et doit être notée en commentaire dans la description de chaque fonction. La gestion de la mémoire fait partie du *contrat* de chaque fonction.

L'abstraction mémoire

Culture — l'abstraction en programmation

Abstraction : *acte de cacher un détail technique derrière une interface simplifiée.*

Les entiers en assembleur : registres.

- Les valeurs sont limitées par la taille des registres (16, 32, 64 bits)…
- … qui change d'une architecture à l'autre.

Le langage C fournit les types `int8_t`, `int16_t`, `int32_t`, etc.

- `intX_t` *abstrait* la taille des registres (on ne s'en soucie plus).
- Par contre on a toujours des limites sur le nombre de chiffres.

Les entiers en Python.

- `int` n'a pas de limite de taille, sur toutes les architectures.
- C'est juste l'ensemble des entiers naturels $\mathbb{N}$.

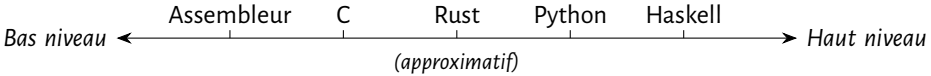
Vocabulaire

Bas niveau *(… d'abstraction) : monde où les détails techniques sont visibles.*

- "D'accord mais comment ça marche à l'intérieur ?"
- Très contrôlé et très puissant, mais fastidieux.

Haut niveau *(… d'abstraction) : monde où les détails sont cachés, on l'on présente uniquement des notions simples/mathématiques.*

- "Ok mais c'est compliqué, moi je veux juste X"
- Rapide et facile, mais moins de libertés.



La vraie nature du C

Thèse de votre enseignant :
La caractéristique unique du C est que la gestion de la mémoire n'est pas abstraite.

Conséquences :

- Les programmeurs C doivent comprendre comment la mémoire marche.
- Vous avez des responsabilités en plus (*allouer, libérer* la mémoire).
- Et ce même si en Rust/Python/Javascript c'est automatique.

Avantages :

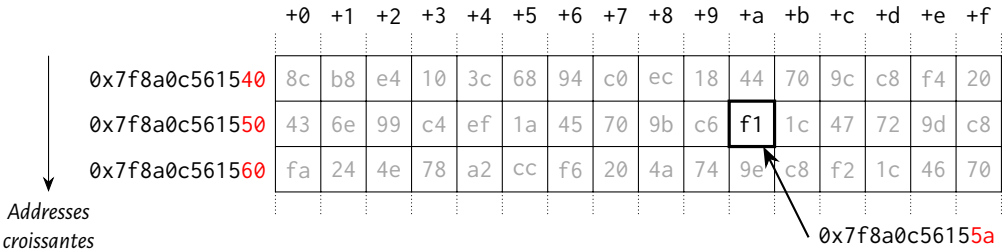
- Capacité d'écrire du code bas niveau extrêmement fin et optimisé, ou super léger.
- … oui en fait à votre niveau y'a pas d'avantage.

Stockage en mémoire

C'est quoi la mémoire au juste ?

- La RAM de votre ordinateur / téléphone / etc.

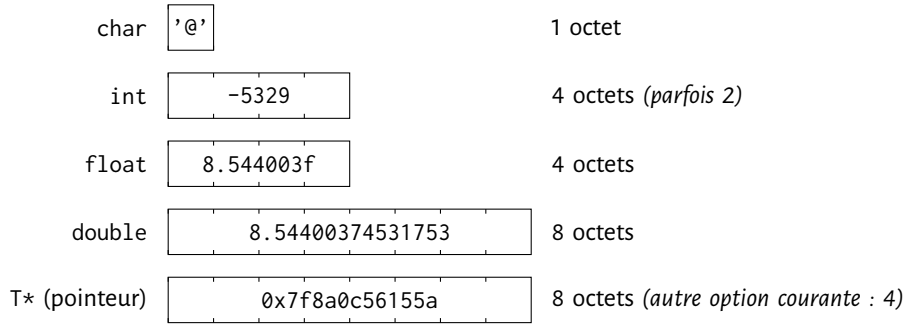
C'est un tableau géant *d'octets*. En C l'octet (8 bits) est la plus petite unité de mémoire possible.



- Le numéro d'un octet en mémoire s'appelle une **adresse** (comme dans l'opérateur &).
- Quand on a une adresse, on peut aller lire la mémoire à cet endroit.
- … les pointeurs sont en fait juste des adresses.

Les variables sont représentées par des octets dans la mémoire

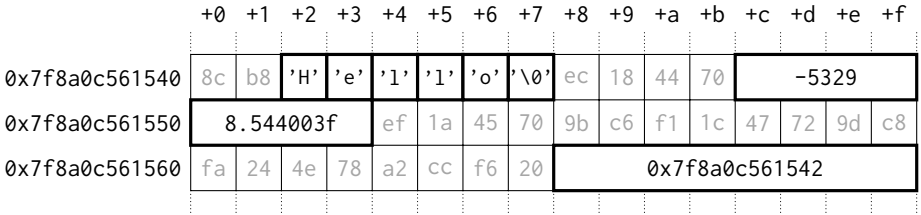
- Chaque type de donnée a une taille fixe.



- Ici on ne se préoccupe pas de la *représentation* (correspondance entre les octets stockés dans la mémoire et la valeur de la variable).

Stockage des variables

Toutes les variables sont mélangées dans la mémoire.



La mémoire elle-même ne sait pas identifier les variables ni leur type.
Seul le programme sait qui est où ! (Et le debugger quand vous compilez avec -g.)

Stockage des tableaux

Tableau : série de valeurs les unes à la suite des autres dans la mémoire.

```
int T[4] = { 42, -5329, 2023, 777 };
```

42	−5329	2023	777

C'est tout ! Les tableaux en C sont très primitifs. En particulier *la taille n'est pas stockée donc il faut la traquer séparément.*

Pour prendre un tableau en paramètre il faut donc typiquement *deux* arguments :

```
int maximum(int T[], int taille) { /* ... */ }
/* ou bien */
int maximum(int *T,  int taille) { /* ... */ }
```

Note : crochets vides (`int T[]`) est une notation pour étoile (`int *T`) en paramètre d'une fonction.

Stockage des chaînes de caractères

Chaîne de caractères (convention) : tableau de char avec une sentinelle (`'\0'`) pour indiquer la fin.

```
char str[6] = "Hello";
```

'H'	'e'	'l'	'l'	'o'	'\0'

Dans ce cas pas besoin de connaître la taille, on peut la trouver en cherchant la sentinelle. Et donc typiquement :

```
#include <string.h>
int nombre_majuscules(char *str)
{
  int taille = strlen(str);
  /* ... */
}
```

La mémoire ne connaît vraiment pas les tailles des tableaux !

```
int T[4] = { 42, -5329, 2023, 777 };  /* à l'adresse 0x7f8a0c56155c */
char str[6] = "Hello";                /* à l'adresse 0x7f8a0c561542 */
```

- Il n'y a de protection contre les accès en-dehors des bornes du tableau.
- Par exemple `T[4]` est la valeur en rouge (entier 1883643122) :

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+a	+b	+c	+d	+e	+f
0x7f8a0c561540	8c	b8	'H'	'e'	'l'	'l'	'o'	'\0'	ec	18	44	70	9c	c8	f4	20
0x7f8a0c561550	43	6e	99	c4	ef	1a	45	70	9b	c6	f1	1c	42			
0x7f8a0c561560	-5329				2023				777				f2	1c	46	70

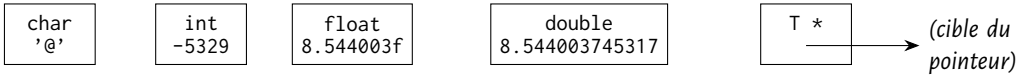
- Le/la programmeur·se est *responsable* de s'assurer que les bornes sont respectées !

Diagrammes mémoire

Diagrammes mémoire : variables et pointeurs

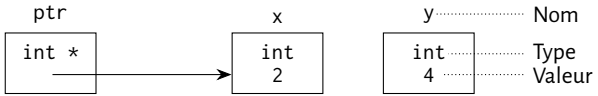
La représentation en grille cache le fait que les *pointeurs* mènent à d'autres variables.

On va donc dessiner les variables comme des blocs "flottants" et les pointeurs comme des flèches.



Exemple :

```
int x = 2;
int y = 4;
int *ptr = &x;
```

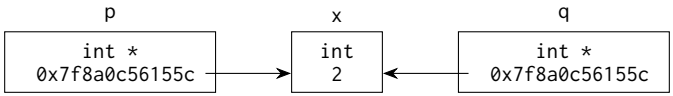


Les pointeurs n'ont qu'une destination

Le dessin suggère que les pointeurs ont une "source", mais ce sont juste des flèches vers des cibles.

```
int x = 2;
int *p = &x;
int *q = &x;
```

p et q ont la même cible donc `p == q` (ce sont des *alias*).

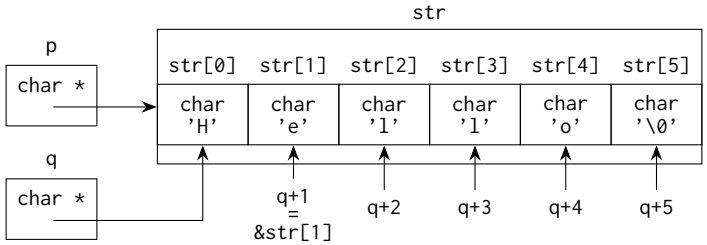


On dit que les deux pointeurs p et q *pointent vers* x (leur valeur est &x).

Diagrammes mémoire : tableaux

Utiliser un tableau dans un calcul, pour initialiser une variable ou comme paramètre d'une fonction, le transforme automatiquement en un pointeur vers son premier élément.

```
char str[6] = "Hello";   // str est de type char[6]
char *p = str;           // Transformation automatique char[6] -> char *
char *q = &str[0];       // Identique à p
```



*p c'est la variable str[0], et *q aussi.

Exercice (au tableau) !

Dessiner le diagramme à chaque étape de ces fonctions.

```
void swap(int *p, int *q)
{
  int i = *p;
  *p = *q;
  *q = i;
}
int maximum(int *T, int N)
{
  int max = 0;
  for(int i = 0; i < N; i++)
    max = (T[i] > max) ? T[i] : max;
  return max;
}
```

Allocation et durée de vie

Allocation et durée de vie

Allocation (verbe : allouer) : *acte de réserver de la mémoire pour y stocker une variable.*

Il existe exactement deux façons d'allouer de la mémoire.

1. Créer une variable locale ou un argument de fonction.

```c
int x = 2;                    /* Alloue 4 octets */
char *s = "Hello";            /* Alloue 8 octets (un pointeur) */
char r[6] = "Hello";          /* Alloue 6 octets */
int f(int x, int *y) { ... }  /* Alloue 4 (x) + 8 (y) octets */
```

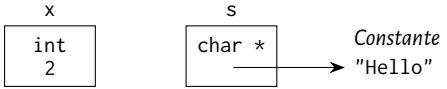
2. Utiliser `malloc()` (“*allocation dynamique*”).

```c
/* Alloue 5 * 4 = 20 octets pour le tableau, et 8 octets pour le pointeur T. */
int *T = malloc(5 * sizeof(int));
```

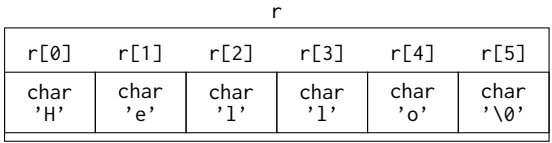
Diagramme mémoire d'une allocation

Avec `malloc()` on alloue toujours aussi un pointeur pour stocker l'adresse de la mémoire fournie.

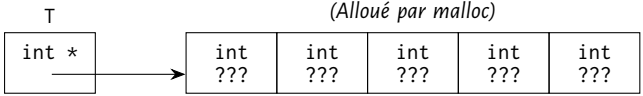
```
int x = 2;
char *s = "Hello";
```




```
char r[6] = "Hello";
```




```
int *T =
  malloc(5*sizeof(int));
```



Note : la mémoire allouée par `malloc()` n'est pas initialisée !

Remarque sur le placement de l'étoile

En C, les espaces ne comptent pas : `char *p` et `char* p` c'est pareil.

Cependant, dans une définition de variable, l'étoile "s'accroche" au nom, pas au type.

```
int* p, q; /* Crée un pointeur p et un entier q ! */
```

Coller l'étoile au nom nous rappelle qu'il faut la répéter si on veut plusieurs pointeurs.

```
int *p, *q; /* Crée deux pointeurs p et q */
```

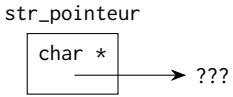
Une astuce pour repérer les allocations ratées

Dans une allocation, on connaît toujours la taille au moment où on alloue.

- Variable locale : le type (et/ou nombre d'éléments du tableau) détermine la taille.
- `malloc()` : la taille est donnée en paramètre.

Donc **si vous n'avez pas donné de taille à votre tableau/chaîne, vous ne l'avez pas alloué·e.**

```c
char str_locale[8];
char *str_pointeur;
strcpy(str_locale, "Hello");   /* OK, on a réservé 8 > 6 octets */
strcpy(str_pointeur, "Hello"); /* Erreur ! */
```



Le pointeur n'est pas initialisé, il ne mène à aucune mémoire !

Libération des allocations dynamiques

Libération : acte de rendre de la mémoire qui avait été réservée (inverse de l'allocation).

Les variables locales sont libérées automatiquement à la fin de la fonction.

Pour `malloc()`, la mémoire est libérée quand on appelle `free()` dessus.

Conséquences :

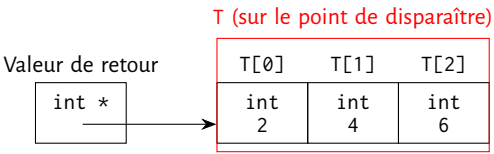
1. Les variables créées avec `malloc()` vivent aussi longtemps que nécessaire.
2. Il y a une *responsabilité de libérer la mémoire*.
3. C'est important de savoir qu'est-ce qui est alloué avec `malloc()` ou pas.

Durée de vie (et limites de taille)

Pourquoi on a besoin de `malloc()` ?

- Les variables locales sont détruites à la fin de la fonction !
- Les valeurs de base (`int`, etc) sont copiées quand on les renvoie : OK.
- Mais on ne peut pas renvoyer un tableau : problème !

```c
int *renvoie_tableau(void) {
  int T[3] = { 2, 4, 6 };
  return T;
}
```



On renvoie un pointeur vers une variable qui se fait immédiatement libérer !!

- `warning: function returns address of local variable [-Wreturn-local-addr]`

Conclusion sur l'allocation

Méthode	**Variable locale**	**`malloc()`**

Allocation	`int x;`
	`char *s;`
	`int T[8] = { /* ... */ };`
		`ptr = malloc(taille);`

Libération	Automatique à la fin de la fonction	`free(ptr);`
Durée de vie	Uniquement la fonction	Aussi longtemps qu'on veut
Taille des tableaux	Fixe (et petite)	Dynamique

Quelle méthode utiliser ?

- Pour allouer une valeur de base (`int`, `float`, etc) : variable locale.
- Pour allouer un tableau ou une chaîne de caractères :
  - S'il est d'une taille non-fixe ou doit exister après la fin de la fonction : `malloc()`.
  - Sinon : variable locale.

Contrats sur la propriété

Qui libère la mémoire ?

```c
char str[20];
char *p = strcpy(str, "Hello");
char *q = strdup("Hello");
```

- Faut-il `free(p)` ?
- Faut-il `free(q)` ?

Certaines fonctions renvoient des pointeurs qui proviennent de leurs paramètres (ici `p == str`) et d'autres renvoient des pointeurs alloués avec `malloc()` (ici `q`).

Cette information importante fait partie du *contrat* de la fonction.

Contrat mémoire d'une fonction

Deux fonctions de copie de chaîne de caractères. `strcpy()` laisse l'appelant allouer, `strdup()` le fait automatiquement avec `malloc()`.

```
/* Copie la chaîne src vers dst. src doit être une chaîne valide et dst doit
   pointer vers au moins strlen(src)+1 octets alloués. Renvoie dst. */
char *strcpy(char *dst, char *src);

/* Duplique la chaîne src. Renvoie un pointeur alloué avec malloc(). Libérer le
   pointeur renvoyé avec free() après utilisation. */
char *strdup(char *src);
```

Le contrat d'une fonction n'est pas que son type ! Un commentaire qui explique ce qu'elle calcule et comment la mémoire est gérée est indispensable !

Exercices (au tableau) !

Dessiner les diagrammes mémoire de `strcpy()` et `strdup()`.

Bonus

Bonus

- Les chaînes de caractères litérales sont des constantes
  - `char *str = "Hello";`
  - Modifier `str` est une segfault
  - Indice : on n'a pas spécifié de taille, c'est louche (il n'y a pas d'allocation)
- Interprétation des notations des types C comme l'usage des variables
  - `int *p` indique que si on écrit `*p` c'est de type `int`
  - `int *p[10]` indique que `p[...]` est de type `int *` et `*p[...]` est un `int`
- Adresse d'un tableau lors d'un appel de fonction
  - `int T[10];`
  - `&T` est de type `int (*)[10]` et `*T` est le tableau complet
  - Mais avoir un pointeur vers un tableau *de 10 éléments* spécifiquement est assez inutile

Algorithmique : structures de données

CS221 — CM #4

Sébastien MICHELLAND

Grenoble INP — Esisar

Dans ce cours…

- Modèles et structures de données du point de vue purement mathématique (pas de code).
- Introduction très brève à la “complexité” d’un algorithme et la notation $O()$.
- Structure et performance des tableaux et listes chaînées.
- Quelques idées de base sur les files et les piles.

Harcèlement du jour : Faites des dessins !

Révisions :

- Repasser sur les dessins des opérations sur les tableaux et les listes chaînées.
- Écrire les algorithmes en pseudo-code et retrouver leur complexité.

Introduction

Structures de données

On fait attention à séparer l'étude algorithmique (abstraite, au papier) de l'implémentation.

Modèle de données :
Description mathématique d'une organisation de valeurs et une interface pour les utiliser.

Un peu comme un `.h` : le modèle dit quelles fonctions existent mais pas comment elles sont codées. (Il y a toujours plusieurs façons de les coder avec différents compromis.)

Structure de données :
Détail du stockage des données en mémoire et algorithmes pour les manipuler.

Un peu comme un `.c` : la structure de données choisit comment on code les fonctions.

Motivation : la complexité algorithmique

Complexité algorithmique : quantification mathématique de la quantité de calculs nécessaire pour accomplir une tâche.

- Pensez-y comme une description préliminaire, mathématique, de la performance/vitesse d'une méthode de calcul.
- Passe *avant* le code : pour que le programme aille vite, il faut avant tout une bonne méthode de calcul, et *ensuite* du code optimisé.

Idée générale : on va compter le nombre d'étapes élémentaires d'une opération (accès, insertion, suppression, etc.) en fonction de la taille de la structure de données.

Plus l'opération devient lente quand la structure devient grande (ie. contient beaucoup d'éléments), plus la complexitée est élevée (mauvaise).

Modèles : séquences, piles, et files

Séquence

Séquence abstraite : suite de valeurs numérotées dans un ordre fixé.

#1	#2	#3	...	#n
$valeur_1$	$valeur_2$	$valeur_3$		$valeur_n$

Opérations d'accès classiques :

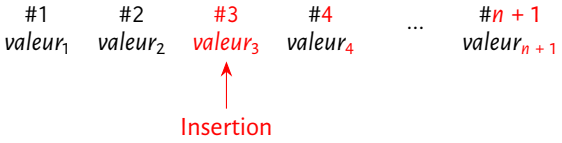
- Taille de la séquence $\rightarrow n$
- Accès au i-ème élément (pour $1 \leq i \leq n$) $\rightarrow valeur_i$

Opérations de modification classiques :

- Remplacement d'un élément à n'importe quelle position
- Insertion à n'importe quelle position
- Suppression à n'importe quelle position

Structures de données : tableau, liste chaînée

L'insertion renumérote tous les éléments suivants :



De même pour la suppression :

Pile

Pile : type limité de séquence où on n'interagit qu'avec le sommet.

Opérations d'accès classiques :

- Tester si la pile est vide $\rightarrow n \neq 0$

Opérations de modification classiques :

- *Empilage* : Ajouter un nouvel élément au sommet
- *Dépilage* : Retirer et renvoyer l'élément au sommet, requiert $n > 0$

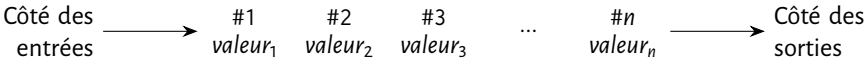
Structures de données : liste chaînée

Pourquoi on voudrait une séquence mais en moins bien ?

- "Principle of least power" : Utiliser l'outil le moins puissant nous permet de gagner en efficacité/performance.

File

Autre type limité de séquence, où cette fois on ajoute d'un côté et on retire de l'autre.



Opérations d'accès classiques :

- Tester si la file est vide $\rightarrow n \neq 0$

Opérations de modification classiques :

- *Enfilage* : Ajouter un nouvel élément du coté des entrées
- *Défilage* : Obtenir le premier élément du côté des sorties (requiert $n > 0$)

Structures de données : paire de piles, tableau circulaire

Introduction à la complexité

Culture : un sous-domaine de l'informatique théorique

La complexité est un domaine entier de l'algorithmique.

- But : caractériser à quel point il est compliqué de calculer des valeurs d'intérêt.

En CS221 on parle que de problèmes faciles, mais en fait y'a des problèmes très durs !

1. Trier un tableau ? Easy.
2. Étant donné un ensemble d'entiers, déterminer si on peut le couper en deux sous-ensembles de somme égale ? Difficile.
3. Trouver le meilleur prochain coup d'une partie d'échecs ? Encore plus dur.
4. En maths, trouver si un ensemble d'égalités entre des valeurs implique une égalité entre des formules ? Parfois totalement impossible.
5. Déterminer si un programme va crasher au lancement, d'une façon qui marche pour tous les programmes et trouve toujours la réponse ? Fondamentalement impossible.

Tout CS221 (dont ce cours) est dans la catégorie 1.

La complexité d'un algorithme dans un cas simple

Trois ingrédients :

1. Une tâche à accomplir (eg. trier une séquence).
2. Une donnée d'entrée (eg. séquence) avec une taille, souvent notée n.
3. Une définition d'opération élémentaire (eg. lire/écrire un élément).

La **complexité** de l'algorithme est le nombre d'opérations élémentaires nécessaire pour accomplir la tâche, en fonction de n, par exemple $2n^2 + 3n + 1$.

Généralement on veut un ordre de grandeur. On utilise la **notation "Grand O"**, qui garde que le terme dominant (qui grandit le plus vite) et ignore les facteurs constants.

$$2n^2 + 3n + 1 \quad \rightarrow \quad O(n^2)$$

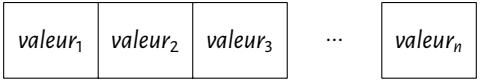
On dit "de l'ordre de n^2" ou bien "un grand O de n^2".

Dans ce cours, la complexité sera presque toujours $O(1)$, $O(n)$ ou $O(n^2)$.

Structures de données

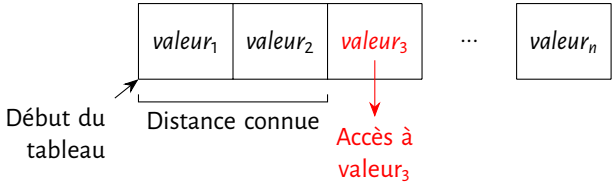
Tableau

Le structure de **tableau** représente des valeurs stockées d'un bloc en mémoire.



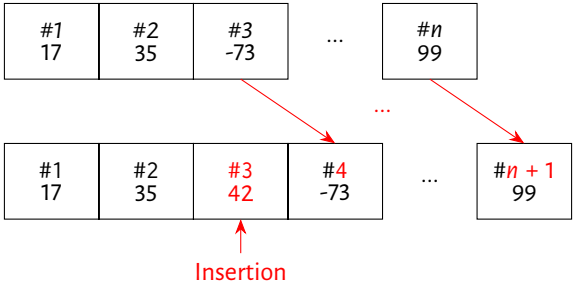
Le tableau implémente le modèle de **séquence**, mais est lent sur les insertions et suppressions.

L'accès à un élement peut se faire en une seule opération :



Nombre d'opérations (lecture/écriture d'éléments) : $O(1)$.

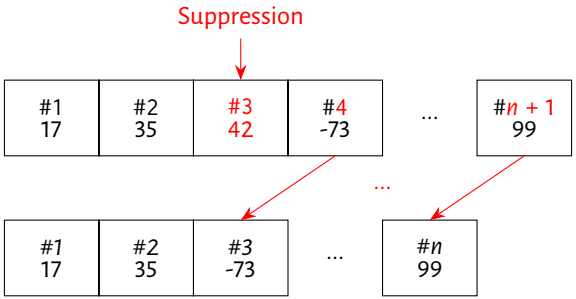
Par contre, les insertions/suppressions nécessitent de déplacer les éléments suivants en mémoire :



Algorithme : au tableau.

Nombre d'opérations (lecture/écriture d'éléments) : $O(n)$.

De même pour les suppressions :

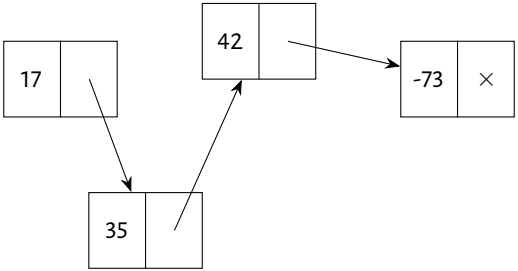


Algorithme : au tableau.

Nombre d'opérations (lecture/écriture d'éléments) : $O(n)$.

Liste chaînée

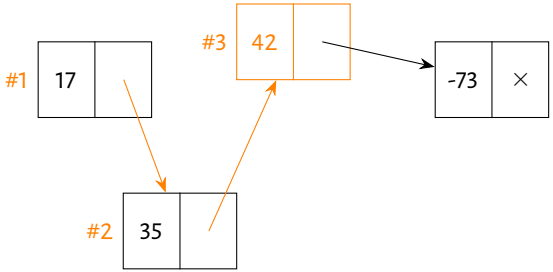
La structure de **liste chaînée** est une séquence dans laquelle l'ordre des éléments est indiqué à la main par une chaîne de références.



La liste est composées de *maillons* (ou *noeuds*) qui contiennent chacun une valeur et une référence vers le maillon suivant.

La liste chaînée dessinée représente les données $[17, 35, 42, -73]$.

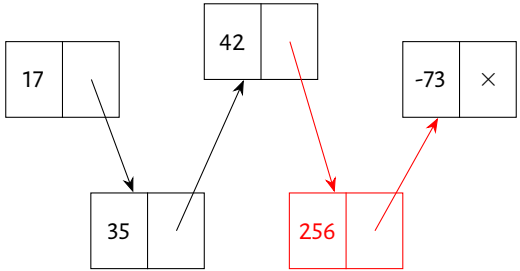
Accéder à un élément prend du temps car il faut suivre le chaînage, on ne peut pas “sauter” à la destination comme avec un tableau.



Pire cas (accès au dernier élément), il faut parcourir toute la chaîne.

Nombre d’opérations (lecture/écriture d’éléments) : $O(n)$.

L'insertion dans une liste chaînée, par contre, est très facile, il suffit de modifier les références à proximité du point d'insertion.



Nombre d'opérations (lecture/écriture d'éléments) : $O(1)$.

La suppression : au tableau (pareil).

Séquences : tableau vs. liste chaînée

Selon la tâche, on va choisir une structure différente :

- Beaucoup d'accès arbitraires, peu de modifications ? Tableau.
- Beaucoup de modifications, surtout au milieu ? Liste chaînée.

Opération	Tableau	Liste chaînée
Accès au i-ème élément	$O(1)$	$O(n)$
Remplacement d'un élément	$O(1)$	$O(1)$
Insertion	$O(n)$	$O(1)$
Suppression	$O(n)$	$O(1)$

Structures de données pour les piles et les files

On en verra surtout comme exemples en TD/TP ; ici, pour la culture.

Moralement : on a de quoi faire des séquences, on peut donc faire des piles et des files.

Mais “principle of least power” : on peut trouver des options spécialisées plus rapides.

Pile :

- La liste chaînée fait déjà toutes les opérations en $O(1)$: parfait.

File :

- On peut utiliser une double pile (pile d’entrée, pile de sortie).
- Ou bien tourner dans un “tableau circulaire”, si on a une borne sur la taille.

Résumé des structures classiques

Résumé des structures classiques

Modèle : séquence

- Tableau : accès rapide, insertion/suppressions lentes
- Liste chaînée : accès lent, insertion/suppression rapides

Modèle : pile

- Liste chaînée : tout en temps constant (yay !)

Modèle : file

- Double pile : rigolo mais pas super efficace en pratique
- “Tableau circulaire” : efficace en pratique, mais taille limitée

On en verra pas mal en TD/TP.

CS221 — TD #3B

live, laugh, libc

Grenoble INP — Esisar

Où trouver de la documentation ?!

Vous avez le nom d'une fonction (eg. `strcpy`), vous voulez savoir comment elle marche :

- `man 3 strcpy`

Vous voulez une liste de fonctions pour savoir s'il y a quelque chose d'utile :

- cppreference.com : `https://en.cppreference.com/`
- Par exemple pour `<string.h>` : `https://en.cppreference.com/w/c/string/byte`

Vous savez ce que vous voulez en anglais mais pas comment le faire :

- Votre moteur de recherche favori + Stack Overflow
- ChatGPT… mais je vous conseille d'apprendre à chercher

Catégories de fonctions dans <string.h>

Trois catégories de fonctions pour trois types de données :

- `str*` : fonctions sur les chaînes de caractères (se termine au premier `\0`)
- `strn*` : fonctions sur les *"null-padded strings"* (comme une chaîne de caractères mais le buffer est rempli de `\0` jusqu'à la fin)
- `mem*` : fonctions sur les régions de mémoire (se termine après une taille donnée)

La plupart du temps vous utiliserez les `str*`, `memcpy()`, et `memset()`. Occasionnellement, d'autres.

Chaînes de caractères

Classiques :

- `strlen()` : Longueur d'une chaîne (`\0` **exclus**).
- `strcpy()` : Copie vers un buffer pré-alloué par l'appelant.
- `strdup()` : Alloue un buffer avec `malloc()` et copie dedans.
- `strcat()` : Copie à la suite d'une chaîne de caractères (le buffer doit être pré-alloué assez grand !).
- `strcmp()` : Compare deux chaînes (**attention** renvoie `0` en cas d'égalité)

Un peu plus rares :

- `strchr()`, `strrchr()` : Cherche la première/dernière occurrence d'un caractère dans une chaîne.
- `strstr()` : Cherche une sous-chaîne dans une chaîne.
- `strspn()`, `strcspn()` : Cherche le plus long préfixe composé uniquement de caractères appartenant (ou pas) à une liste.

Entrées/sorties formatées

`printf()` et `scanf()` ont plein de variations indiquées par des lettres en préfixe.

- `s` : on manipule une chaîne de caractères (“string”) au lieu du terminal
  - `sprintf(str, "%d", 42)` génère le texte dans la chaîne `str` (qu'il faut allouer à l'avance)
  - `sscanf(str, "%d", &x)` analyse la chaîne `str`
- `n` : limite de taille
  - `snprintf(str, n, "%d", 42)` génère le texte dans la chaîne `str` de taille `n`
- `f` : on manipule un fichier
  - `fprintf(fp, "%d", 42)` écrit dans le fichier `fp` (de type `FILE *`)
  - `fscanf(fp, "%d", &x)` lit depuis le fichier `fp` (de type `FILE *`)
- `a` : allocation dynamique
  - `char *str; asprintf(&str, "%d", 42)` alloue une chaîne d'une taille appropriée avec `malloc()`, génère le texte dedans, et stocke le pointeur associé dans `str`. *Attention* : Bien penser à `free(str)`.

`printf()` a également plein d'options dans les *formats* comme `%d`. Couramment :

- Nombre entier `n` : spécifie que le résultat doit faire `n` caractères, en ajoutant des espaces à gauche.
  - `printf("%4d", 42)` → `"  42"` (deux espaces en début de chaîne)
- `'0'` : indique que le caractère de remplissage doit être `0` et pas un espace.
  - `printf("%04d", 42)` → `"0042"`
- `'.'` suivi d'un nombre entier : indique le nombre de décimales ou de chiffres significatifs
  - `printf("%.2f", 8.42453)` → `"8.42"`

Astuce : pour afficher un flottant, utilisez `%g`, ça fait le café (pas de zéros en trop, passe en notation scientifique pour les grandes ou petites valeurs automatiquement, etc).

Mentions spéciales

`<stdio.h>`

- Liste également toutes les fonctions de manipulation de fichiers

`<stdlib.h>`

- Générateur de nombre aléatoires : `rand()`, `srand()`.
- Allocation dynamique : `malloc()`, `calloc()`, `realloc()`, `free()`.
- Conversion de nombres (mini-scanf) : `atoi()`, `atof()`, pour détecter les erreurs `strto*()`.
- Tri rapide : `qsort()`.

`<string.h>`

- `strerror()` : Texte de l'erreur nommée par `errno` (aussi `%m` dans `printf`)

`<stdint.h>`

- Types entiers à taille fixe `intN_t` (signés) et `uintN_t` (non signés), pour N = 8, 16, 32, 64.

TD Chapitre 3: Gestion de mémoire (CS221)

Grenoble INP – Esisar – année 2023-24

Objectifs :

1. Dessiner des diagrammes mémoire du point de vue d'une fonction seule.
2. Différencier les fonctions qui modifient leurs arguments vs. allouent avec `malloc()`.
3. Différencier taille du texte et du buffer pour les chaînes de caractères.

Exercice 1 — Échauffements

1. Dessiner le diagramme mémoire de la fonction `swap()`. Quelles hypothèses doit-on faire sur les pointeurs `a` et `b` pour que la fonction marche ? Est-ce que la fonction marche si `a` ou `b` est `NULL` ? S'ils pointent vers la même variable ?

```c
void swap(int *a, int *b)
{
    int c = *a;
    *a = *b;
    *b = c;
}
```

2. Dessiner le diagramme mémoire de la fonction `maximum()`. `T` pointe-t-il vers une variable seule, un tableau, ou une chaîne de caractères ?

```c
int maximum(int *T, int n)
{
    int max = 0;
    for(int i = 0; i < n; i++)
        max = (max > T[i]) ? max : T[i];
    return max;
}
```

Exercice 2 — Modifier ou allouer ?

On veut écrire une fonction `doubler_tableau()` qui multiple tous les éléments d'un tableau d'entiers par 2.

1. Compte tenu des règles du langage, la fonction recevra-t-elle une copie complète d'un tableau ou un pointeur menant vers un tableau fourni par l'appelant ?
2. Combien de paramètres `doubler_tableau()` devra-t-elle prendre pour pouvoir manipuler le tableau ?
3. Écrire une première version de la fonction qui renvoie `void` et modifie le tableau original ; dessiner son diagramme mémoire.
4. Écrire une seconde version qui ne modifie pas le tableau original et alloue à la place un nouveau tableau avec `malloc()`, le remplit, puis le renvoie. Dessiner son diagramme mémoire.
5. Aurait-on pu créer le nouveau tableau comme une variable locale et le renvoyer ? Pourquoi ?
6. Écrire une fonction `main()` montrant comment appeler les deux versions.

En C on utilise plus souvent le premier style (l'appelant alloue) parce qu'il nous donne moins de travail et permet certaines optimisations. Mais les deux versions sont parfaitement valides.

Exercice 3 — Tout est problème de taille

La fonction standard `strcat()` *(**str**ing **cat**enate)* concatène deux chaînes de caractères `src` et `dst` en copiant `src` à la suite de `dst`. Son prototype (simplifié) est le suivant :

```c
char *strcat(char *dst, char *src);
```

`strcat()` modifie `dst` directement (comme dans la question 3 de l'exercice 2) et n'alloue pas de nouvelle chaîne de caractères. (La valeur de retour est juste `dst`.)

1. Dessiner le diagramme mémoire des arguments de `strcat()`. Quelle hypothèse sur la taille de la zone mémoire vers laquelle `strcat()` pointe est nécessaire pour que la copie se passe bien ?
2. Est-il possible, depuis la fonction `strcat()`, de connaître la taille de cette zone mémoire ?
   - Est-ce que `sizeof(dst)` nous donne cette information ?
   - Et `strlen(dst)` ? (Est-il légal d'appeler `strlen(dst)` déjà ?)
3. Écrire une version de la fonction.
4. Proposer une fonction `main()` qui concatène les chaînes `"Hello"` et `", world!"`. Vous allouerez deux tableaux locaux pour y stocker les chaînes. Quelle taille faut-il donner au premier tableau ?
5. Implémenter une fonction `strcat_malloc()` qui se charge d'allouer une nouvelle chaîne de la bonne taille avec `malloc()` et de la remplir avec le résultat de la concaténation. Son prototype sera identique à `strcat()` (mais cette fois la valeur de retour sera un nouveau tableau, pas `dst`).

   ```c
   char *strcat_malloc(char *dst, char *src);
   ```

   Vous pourrez utiliser pour ça la fonction `strcpy()` (**str**ing **cpy**) qui copie la chaîne `src` vers la mémoire pointée par `dst`. Tout comme `strcat()`, l'appelant est responsable d'allouer assez de mémoire pour `dst` (et la valeur de retour est aussi une copie de `dst`).

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

6. Montrer comment on appellerait `strcat_malloc()` depuis `main()`.

Exercice 4 (Bonus) — Tableaux à deux dimensions

Un piège classique quand on s'habitue à manipuler de façon équivalente des tableaux locaux (`int T[10]`) et des pointeurs menant aux tableaux (`int *T`) et de voir les deux comme complètement interchangeables. Cependant, ça ne marche que pour la première dimension.

1. En généralisant l'idée qu'un tableau est une séquence de variables stockées de façon consécutive en mémoire, dessiner le diagramme mémoire correspondant à :

   ```c
   int M[2][3] = { {1,2,3}, {4,5,6} };
   ```

2. On rappelle que les éléments d'un tableau doivent tous être de même type. Quel est le type des éléments de `M` ? Serait-il possible d'avoir une ligne plus petite que l'autre ?
3. En se rappelant que `N1` et `N2` sont automatiquement convertis en pointeurs vers leurs contenus quand ils sont mentionnés dans `{ N1, N2 }`, dessiner le diagramme mémoire des variables :

   ```c
   int N1[3] = { 1, 2, 3 };
   int N2[3] = { 4, 5, 6 };
   int *N[2] = { N1, N2 };
   ```

4. Les variables `M` (tableau de tableaux) et `N` (tableau de pointeurs) ont-elles la même représentation en mémoire ? En déduire que les deux représentations ne sont pas interchangeables.

TD Chapitre 3 : Bibliothèque standard (CS221)

Grenoble INP – Esisar – année 2023-24

Objectifs :

1. Connaître quelques fondamentaux de la bibliothèque standard (libc) ;
2. Repérer les situations dans lesquelles la libc fait le boulot pour vous ;
3. Savoir où trouver plus d'infos pour les TP et projets.

Dans les exercices ci-dessous, sauf indication vous n'avez pas droit d'utiliser de boucles : juste les fonctions standard.

Exercice 1 — Chaînes de caractères

1. On suppose qu'on a un tableau contenant une liste prédéfinie de 8 noms :

   ```c
   const char *names[8] = { "zero", "ra", "sp", "gp", "tp", "t0", "t1", "t2" };
   ```

   Implémenter une fonction

   ```c
   int find_name(char *name);
   ```

   qui renvoie la position de `name` dans la liste (0 – 7) ou -1 si `name` n'apparaît pas dans le tableau. On pourra écrire une boucle (une seule).

2. On veut découper un bloc de données en sections de 512 octets (la dernière faisant 512 ou moins). Pour copier les fragments, utilisera-t-on `strcpy()`, `strncpy()` ou `memcpy()` ?

3. Écrire une fonction

   ```c
   char *substring(char *str, int position, int size);
   ```

   qui extrait de `str` la sous-chaîne commençant à la position donnée et de taille donnée. (On suppose que la section en question est valide et dans les bornes de `str`.) On allouera une nouvelle chaîne de la bonne taille avec `malloc()` avant de copier les contenus. Que faudra-t-il faire de la sous-chaîne après usage ?

4. Remplacer `substring()` par un appel bien choisi à `strndup()` en une ligne.

5. Déterminer (en une ligne) si une chaîne `str` contient le caractère `' '` (espace).

6. Bonus : déterminer (en une ligne) si une chaîne `str` contient un caractère d'espacement (espace, tabulation, ou retour à la ligne).

Exercice 2 — Entrées/sorties formatées

1. En supposant un entier `int n` positif, calculer le nombre de chiffres dans l'écriture de `n` (par exemple 1729 → 4).
2. En supposant que `str` est une chaîne de caractères de la forme `"xN"` avec `N` un entier (par exemple `"x2"` ou `"x17"`), obtenir la valeur de `N` dans une variable de type `int` (par exemple 2 ou 17).
3. En supposant un entier positif de 32 bits `n`, afficher sa représentation en hexadécimal sur 8 chiffres précédée de `0x` (par exemple 26 → `"0x0000001a"`).

Exercice 3 — Analyse de caractères

On suppose qu'on a une chaîne de caractères non-constante du type

```
"\t Nom   5 AutreNom  +12\t Etc Etc -7\n"
```

composée de différents *noms* (séries de lettres) et *valeurs* (entiers potentiellement précédés de `'+'` ou `'-'`) séparés par des caractères d'espacement (espace, tabulation `'\t'`, retour à la ligne `'\n'`).

On veut séparer et identifier ces composants pour produire le résultat suivant :

```
nom: [Nom]
valeur: [5]
nom: [AutreNom]
valeur: [+12]
nom: [Etc]
nom: [Etc]
valeur: [-7]
```

1. Écrire une version de ce programme en parcourant à la main la chaîne de caractères. Il est recommandé d'avoir une boucle `while` principale dans laquelle on identifiera et affichera un composant à chaque tour. Pour cela, il faudra sauter les espaces, identifier le composant, et afficher tous ses caractères avec d'autres boucles internes.
2. Bonus : écrire une version deux fois plus simple avec `strtok()`. (Attention, `strtok()` est une fonction bizarre, elle se souvient des appels précédents !)

TD Chapitre 7 : Algorithmique et récursivité (CS221)

Grenoble INP – Esisar – année 2023-24

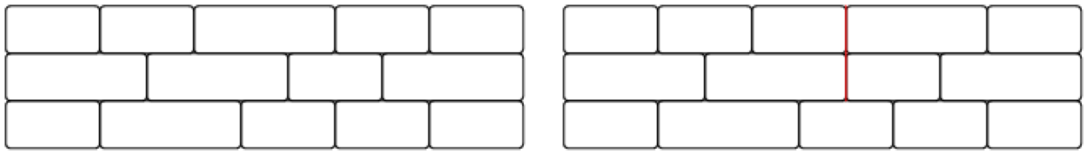
Objectifs :

1. Modélisation algorithmique d'un problème décrit informellement.
2. Étude et programmation d'une solution récursive.

Exercice 1 — Murs de briques

D'après Projet Euler 215 et Archives de 421, Polytechnique (d'où les dessins et une partie du texte ont été honteusement pompés)

On cherche à construire un mur à partir de briques horizontales de taille 2x1 et 3x1. Un mur est correctement construit si la jointure verticale entre deux briques ne se trouve jamais immédiatement au dessus d'une autre jointure verticale. Ainsi le mur de gauche ci-dessous (de hauteur 3 et de longueur 11) est correctement construit, mais pas celui de droite.



L'objectif de ce problème est de dénombrer le nombre de façons différentes de construire un mur de longueur ℓ et de hauteur h.

Modélisation du problème

Chaque rangée de briques de longueur ℓ va être codée comme un tableau d'entiers (0 ou 1) de taille $\ell - 1$. Par exemple, la rangée de taille 11 ci-dessous est codée dans le tableau de taille 10 qui suit.

0	1	0	0	1	0	1	0	1	0

Les 1 représentent les jointures entre les briques, les 0 le milieu d'une brique ou le début/la fin du mur.

Question 1 — Dessiner la rangée de briques donnée par le tableau suivant. De quelle taille est le mur ?

0	1	0	0	1	0	0

Deux rangées sont dites *compatibles* si elles peuvent être posées l'une sur l'autre (les jointures ne sont pas au même endroit).

Question 2 — Écrire une fonction `bool compatibles(int l, int *r1, int *r2)` qui détermine si deux rangées sont compatibles.

Méthode de résolution

Pour dénombrer le nombre de façons différentes de construire un mur de longueur ℓ et de hauteur h, on adopte le principe de résolution suivant :

- **Phase 1 :** Déterminer l'ensemble R des rangées possibles de longeur ℓ. Par exemple, il n'existe que deux rangées de longueur 5 : une brique de 2x1 puis une de 3x1 (`0100`) ou l'inverse (`0010`).

- **Phase 2 :** Pour chaque rangée $r \in R$, déterminer le nombre de murs de hauteurs h dont la *première rangée de briques* (celle tout en bas) est r. Cette valeur est notée $count(r, h)$.
- **Phase 3 :** Le résultat final sera $\sum_{r \in R} count(r, h)$.

On suppose pour l'instant qu'on a un tableau `int *allrows[]` de longueur `int allrows_len` contenant la liste de toutes les rangées possibles de longueur ℓ.

Dénombrement des murs

Pour dénombrer les murs, on programme la fonction récursive `count(r, h)` qui détermine le nombre de murs dont la première rangée de briques est r et dont la hauteur est h :

```
count(r, h) =
  si h vaut 1, renvoyer 1
  sinon /* renvoyer la somme pour toute rangee s COMPATIBLE de count(s, h-1) */
```

Question 3 — Expliquer l'algorithme.

Question 4 — Écrire une fonction récursive

```
int count(int *r, int h, int l, int *allrows[], int allrows_len);
```

qui implémente l'algorithme précédent. Il retourne le nombre de murs de hauteur h de première rangée r construit avec les rangées de `allrows` qui sont de longueur ℓ.

Question 5 — En déduire une fonction qui résout le problème

```
int count_all(int l, int h, int *allrows[], int allrows_len);
```

Question 6 — Expliquer pourquoi cet algorithme n'est pas optimal. (Il faudrait donc tabuler des calculs intermédiaires, un très bon exercice.)

Énumération des rangées

L'objectif est de générer toutes les combinaisons possibles de rangées de longueur ℓ.

Question 7 — Supposons qu'on a une rangée partiellement construite `base` (correctement formée de briques de taille 2x1 et 3x1), de longueur `base_len` $\leq \ell$. Sous quelles conditions existe-t-il des rangées valides qui commencent par `base` ?

Question 8 — En observant qu'une rangée qui étend `base` doit lui ajouter soit une brique de taille 2x1 soit une brique de taille 3x1, en déduire un algorithme récursif pour énumérer toutes les rangées possibles de taille ℓ.

Question 9 (à la maison) — Programmer une fonction

```
int gen_allrows_aux(int *base, int base_len, int **allrows, int l);
```

qui génère toutes les rangées valides de taille ℓ qui commencent par la rangée partiellement construite `base` de longueur `base_len` $\leq \ell$, les stocke dans `allrows` sauf si `allrows` est `NULL`, et renvoie leur nombre. (On suppose que `allrows` est déjà alloué avec la bonne taille si non nul, et chaque rangée sera allouée indépendamment avec `malloc()`.)

En déduire une fonction

```
int **gen_allrows(int *allrows_len, int l);
```

qui génère les rangées. On appelera `gen_allrows_aux()` deux fois : une avec `allrows = NULL` pour déterminer le nombre de lignes, et après avoir alloué `allrows` une deuxième fois pour remplir le tableau.

À l'aide de `gen_allrows()` et `count_all()`, résoudre le problème pour un ℓ et un h donnés.

CS221 — TDM #1A

Pointer party!

Grenoble INP — Esisar

Objectifs

Reconnaître et utiliser les pointeurs « pour le calcul » :

- Passage de variables par adresse
- Accès aux tableaux et aux chaînes de caractères

Éviter les pièges classiques :

- Toujours utiliser le contexte pour comprendre le rôle des pointeurs en paramètres de fonctions
- Ne pas essayer de retourner des tableaux

Instructions :

- Noter les réponses aux questions de chaque exercice en commentaire dans le code.
- Envoyer les programmes finaux sur Chamilo (*instructions d'archivage sur Chamilo*).

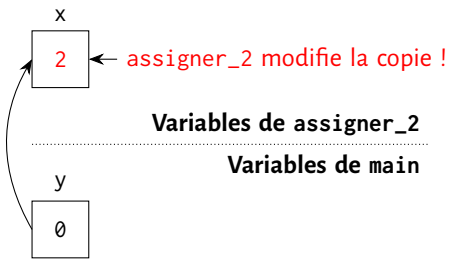
Passage de variables par adresse

Une variable passée par valeur ne peut pas être modifiée.

Les paramètres aux fonctions sont toujours **copiés lors de l'appel**, donc on ne peut pas modifier une variable passée directement (« par valeur »).

```c
/* NE MARCHE PAS ! */
int assigner_2(int x) {
    x = 2;
}
int main(void) {
    int y = 0;
    assigner_2(y);
    // y toujours égal à 0 !
}
```

Création d'une copie de y lors de l'appel

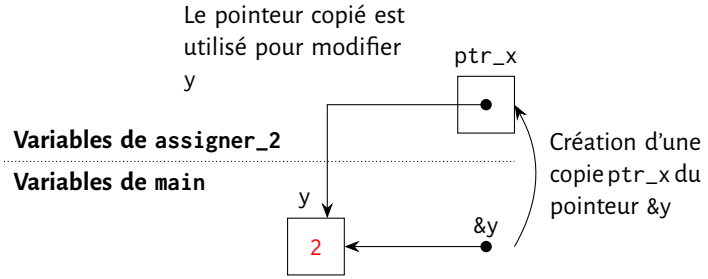


La variable x est indépendante de y donc quand `assigner_2()` modifie x ça n'affecte pas y.

Pour modifier une variable donnée en argument, on la passe par adresse.

Passer « par adresse » ça veut dire qu'on passe (une copie de) l'adresse en paramètre.

```c
int assigner_2(int *ptr_x) {
    *ptr_x = 2;
}
int main(void) {
    int y = 0;
    assigner_2(&y);
    // y = 2
}
```

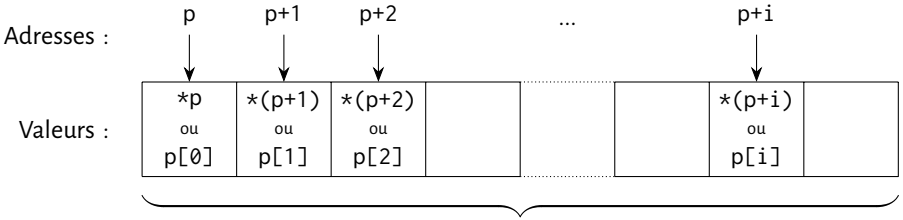


Cette fois `assigner_2` reçoit une copie du *pointeur* et elle s'en sert pour accéder à la variable originale. (Toutes les copies de &y mènent à y donc ça change rien que le pointeur soit copié.)

Accès aux tableaux et chaînes de caractères

Un pointeur peut être utilisé pour mener à un tableau.

Grâce à l'arithmétique de pointeurs, à partir d'un pointeur vers le premier élément d'un tableau on peut utiliser tout le tableau :

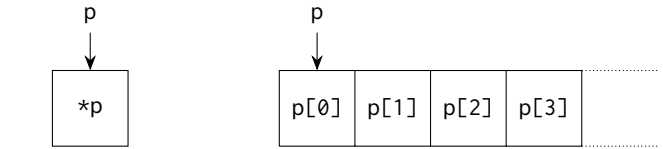


Rappel : `p[i]` *est une notation pour* `*(p+i)`.

Une fonction qui **prend un tableau d'`int` en paramètre reçoit dont généralement un `int *p`** et elle accède au tableau en écrivant `p[0]`, `p[1]`, etc.

Risque de confusion : pointeur vers une valeur vs. un tableau ! (1/2)

Attention : quand une fonction prend un `int *p` en paramètre on ne sait pas si c'est un pointeur vers un seul élément ou vers un tableau.



Un seul élément ? *Un tableau ?*

Il faut utiliser le contexte pour deviner.

- Que fait la fonction ?
- Est-ce que l'argument a un nom plutôt valeur (`x`) ou plutôt tableau (`tab`, `array`, `str`) ?
- Un `char *` ou `const char *` est presque toujours un pointeur vers une chaîne de caractères (tableau de `char` terminé par une sentinelle `'\0'`).

Risque de confusion : pointeur vers une valeur vs. un tableau ! (2/2)

```
bool multiply(int x, int y, int *result);
```

- `result` est probablement un pointeur vers *un seul élément* pour stocker le produit.

```
void sort(int *array, int N);
```

- C'est un tri ("sort" signifie « tri ») et donc `array` est probablement un pointeur vers *un tableau*.
- (En plus "array" veut dire tableau !)

```
size_t strlen(const char *str);
```

- "str" est un raccourci courant pour "string" (chaîne de caractères) ;
- `str` est donc certainement un pointeur vers *une chaîne de caractères* (tableau de `char` terminé par une sentinelle `'\0'`).

En général, se référer à l'énoncé ou à la description/le commentaire de la fonction.

Conversion automatique tableau vers pointeur

Partout où un pointeur (par exemple de type `int *`) est attendu, on peut fournir un tableau (par exemple `int array[8]`). Le compilateur comprend qu'on veut donner un pointeur vers le premier élément.

```
void sort(int *array, int N);

int array[8] = { 5, 2, 7, 0, 3, 4, 6, 1 };
sort(array, 8);
// équivalent à
sort(&array[0], 8);
```

Le tableau n'est donc pas copié (c'est le pointeur vers le premier élément qui est copié).

Exercices

Échanger les valeurs de deux variables

- **Fichier : `CS221_TDM1A/ex1_swap.c`**
- Compiler avec : `gcc ex1_swap.c -o ex1_swap -Wall -Wextra`

Un étudiant a écrit la fonction `swap()` pour échanger la valeur de deux variables.

Questions :

1. Les arguments de `swap()` sont-t-ils passés par valeur ou par adresse ?
2. Prédire le texte qui s'affichera à l'exécution de `swap.c`.
3. Corriger la fonction `swap()` en passant a et b par adresse.
4. Avez-vous changé le type de c de « `int c` » en « `int *c` » ? Pourquoi ?

Mettre une chaîne de caractères en majuscule

- **Fichier : `CS221_TDM1A/ex2_capitalize.c`**
- Compiler avec : `gcc ex2_capitalize.c -o ex2_capitalize -Wall -Wextra`

On vous fournit le squelette d'une `capitalize()`.

Questions :

1. D'après la description de la fonction, le pointeur `char *str` pointe-t-il vers un unique caractère, un tableau, ou une chaîne de caractères ?
2. Écrire et tester la fonction `capitalize()`. Quel type de parcours faut-il faire ?

Somme des éléments d'un tableau

- **Fichier : `CS221_TDM1A/ex3_sum.c`**
- Compiler avec : `gcc ex3_sum.c -o ex3_sum -Wall -Wextra`

On souhaite écrire une fonction `sum()` qui calcule la somme des éléments d'un tableau d'entiers. Pour l'instant, on a commencé par le squelette suivant :

```
int sum(int *T);
```

Questions :

1. D'après la description de la fonction, le pointeur `int *T` pointe-t-il vers un unique entier ou vers un tableau ?
2. Quel type de parcours faut-il faire pour calculer la somme ? Quelle information nous manque-t-il ?
3. Ajoutez le paramètre manquant et écrivez la fonction. Ajustez aussi l'appel dans `main()`.

Et si on veut renvoyer un tableau ?

- **Fichier : `CS221_TDM1A/ex4_increment.c`**
- Compiler avec : `gcc ex4_increment.c -o ex4_increment -Wall -Wextra`

Affirmation (simplifiée) :

- **Les fonctions C ne peuvent pas renvoyer de tableaux.**
- Option #1 : Prendre un tableau en paramètre (via un pointeur) et le modifier (cet exercice).
- Option #2 : Utiliser `malloc()` (dans les prochaines séances !).

Questions :

1. En lisant la description, déterminer un prototype pour la fonction `increment()` (qui modifie le tableau qui lui est passé en paramètre) ; expliquer votre choix.
2. Programmer la fonction `increment()` et l'appeler depuis `main()`.

CS221 — TDM #1B

Debugging dance!

Grenoble INP — Esisar

Objectifs

Acquérir les réflexes de base en cas de bug :

- "segfault" : accès à un pointeur nul, dépassement de tableau, …
- "stack smashing" : dépassement de tableau (variable locale)

Utiliser les fonctionnalités basiques d'un debugger :

- Compiler pour le debugging (avec `-O0 -g`)
- Observer la pile d'appels et les variables globales au moment d'un crash
- Bonus : poser des points d'arrêts *(breakpoints)*

Instructions :

- Noter les réponses aux questions de chaque exercice en commentaire dans le code.
- Envoyer les programmes finaux sur Chamilo (*instructions d'archivage sur Chamilo*).

Bugs classiques

Messages d'erreurs

Messages classiques et leur cause immédiate. Attention, le bug peut venir d'ailleurs, le message d'erreur n'est que le symptôme du problème !

Segmentation fault *("segfault")* : accès mémoire via un pointeur invalide.

1. Utilisation d'un pointeur nul ou non initialisé.
2. Dépassement de tableau super loin (eg. plusieurs milliers de cases).
3. Tentative de modifier une chaîne de caractères litérale.
4. Confusion sur les niveaux de pointeurs (dans ce cas `-Wall -Wextra` affiche un warning).

Stack smashing : corruption de la pile (variables locales).

1. Dépassement d'un tableau créé via une variable locale (`int array[10]`).

Double `free()` or corruption : corruption du tas (variables créées avec `malloc()`).

1. Dépassement d'un tableau créé via `malloc()`.

Résultat incorrect sans message : archéologie en perspective.

Sources habituelles des erreurs

Sources classiques de bugs liés au langage C, au niveau 2A/3A.

- Oubli d'activer, lire et résoudre les warnings (`-Wall -Wextra`)
- Dépassement de tableau (les indices doivent être entre `0` et la taille − 1)
- Oubli de gérer les sentinelles en manipulant des chaînes de caractères (en particulier ne pas allouer un octet dédié à la sentinelle)
- Allocation d'une quantité de mémoire insuffisante dans `malloc()`

Pour s'assurer qu'une boucle fait bien ce qu'il faut :

1. Vérifier que le calcul fait à chaque tour de boucle est correct.
2. Déterminer la valeur des indices au premier et dernier tour, et vérifier que les accès aux tableaux sont corrects lors de ces tours (en particulier : dans les bornes).
3. Vérifier si le résultat est correct quand la boucle ne fait aucun tour (si c'est possible).

Utilisation d'un debugger

Example typique d'utilisation de gdb

Voici le programme `example.c`, qui est buggé. Le problème est que `main()` essaie de copier `"hi"` dans une chaîne de caractères mais le pointeur `str` est nul (aucune mémoire n'a été réservée pour y stocker la chaîne).

```c
#include <stdio.h>

void write_hi(char *str)
{
    str[0] = 'h';
    str[1] = 'i';
    str[2] = '\0';
}
```

```c
int main(void)
{
    char *str = NULL;
    write_hi(str);
    printf("%s\n", str);
    return 0;
}
```

Étape 1 : Compiler avec les options `-O0 -g`.

```
% gcc example.c -o example -Wall -Wextra -O0 -g
```

Lancer le programme produit un crash (en l'occurrence une segfault) :

```
% ./example
[1]    1144895 segmentation fault (core dumped)  ./example
```

Étape 2 : Ouvrir le programme dans gdb puis le lancer avec la commande `run`.

```
% gdb -q ./example
Reading symbols from ./example...
(gdb) run
(...)
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555145 in write_hi (str=0x0) at example.c:5
5     str[0] = 'h';
```

gdb s'arrête lorsque le crash se produit et affiche la ligne de l'erreur.

Étape 3 : inspecter le programme.

La commande `bt` affiche la fonctions en cours d'exécution au moment du crash et qui l'a appelée. Chaque ligne est précédée d'un numéro : la commande `frame` permet de visiter la fonction correspondante pour savoir où elle en était.

```
(gdb) bt
#0  0x0000555555555145 in write_hi (str=0x0) at example.c:5
#1  0x000055555555517d in main () at example.c:13
(gdb) frame 1
#1  0x000055555555517d in main () at example.c:13
13    write_hi(str);
```

La commande `print` permet d'afficher la valeur des variables visibles depuis la fonction sélectionnée avec `frame`.

```
(gdb) print str
$1 = 0x0
```

Étape 4 : remonter au bug.

Utiliser les informations apprises durant l'inspection du programme dans le debugger pour remonter au bug.

Si jamais le bug se produit en amont, on peut demander à gdb de s'arrêter plus tôt en ajoutant un *breakpoint* à une certaine ligne du programme et en le redémarrant.

```
(gdb) break example.c:12
Breakpoint 1 at 0x555555555169: file example.c, line 12.
(gdb) run
(...)
Breakpoint 1, main () at example.c:12
12    char *str = NULL;
```

gdb s'arrête lorsque le programme atteint la ligne du breakpoint (ici la ligne 12 de `example.c`). Ça permet d'inspecter le programme pendant qu'il s'exécute (même s'il n'y a pas de crash à la fin).

Exercices

Erreurs basiques sur les pointeurs

- **Fichier : `CS221_TDM1B/ex1_pointers.c`**
- *Pensez à compiler avec* `-O0 -g -Wall -Wextra`.

Ce programme contient deux fonctions `integer_division()` et `first_digit()`.

1. Compiler et tester le programme. À quel moment a-t-on un problème ? Quel message du compilateur permettait de le voir venir ?
2. Corriger le problème avec `integer_division()` (avec ou sans gdb). Constater que la division marche mais qu'il y a maintenant un autre problème.
3. Que fait la fonction `first_digit()` ?
4. En utilisant gdb, exécuter le programme jusqu'à l'erreur de segmentation. À quelle ligne se produit-elle ? Que vaut `p` à ce moment ?
5. Corriger le problème en remplaçant le `printf()` final par une condition qui gère tous les cas possibles.

Dépassements de tableaux

- **Fichier : `CS221_TDM1B/ex2_arrays.c`**
- *Pensez à compiler avec* `-O0 -g -Wall -Wextra`.

1. Compiler et lancer le programme. Y a-t-il des erreurs ou des warnings ? Que se passe-t-il à l'exécution ?
2. En utilisant gdb, exécuter le programme jusqu'au moment du crash. À quelle ligne le crash se produit-t-il ? Qu'est-ce qui peut se passer de travers dans un accès de tableau comme ça ?
3. En utilisant `print` ou `info locals`, déterminer le numéro de case du tableau auquel on accède au moment du crash. L'erreur de segmentation s'est-elle produite immédiatement quand on a dépassé du tableau ?
4. En comparant avec le TDM1A, déterminer et corriger le problème avec la fonction `increment()` (il faudra modifier les arguments).

(suite de l'exercice sur le prochain slide)

5. La deuxième moitié du programme utilise une nouvelle fonction `sequence()`. Quelle partie du contrat de `sequence()` n'a pas été respectée par `main()` ? Corriger cet oubli.
6. Le changement en question 5 révèle un bug qui était caché, et qui produit (en général) une erreur `free(): invalid pointer`. Rétrospectivement, l'affichage des contenus de `S` est-il conforme à la description de `sequence()` ?
7. Avec l'aide de `gdb`, placer un breakpoint à la ligne `array[i] = i` et observer les valeurs successives de `i`. Quelle est l'origine du problème ?
8. Corriger la fonction `sequence()` et vérifier que la sortie est maintenant correcte.

Erreurs de logique

- **Fichier : `CS221_TDM1B/ex3_sort.c`**
- *Pensez à compiler avec* `-O0 -g -Wall -Wextra`.

1. Lire le programme, le compiler et constater qu'il segfault. Avec l'aide de `gdb`, déterminer dans quelle fonction l'erreur se produit. A-t-on appelé cette fonction directement ?
2. En utilisant la commande gdb `backtrace`, déterminer comment `compare()` a été appelée.

On admet que la fonction `compare()` n'est pas buggée, que la fonction standard `qsort()` ne l'est pas non plus (incroyable !) et que l'*appel* à `qsort()` est correct aussi.

3. En utilisant la commande gdb `frame`, remonter dans la fonction `print_for_year()` et inspecter les paramètres de l'appel à `qsort()`. Où se situe le problème ?

Clairement quelque chose s'est mal passé *avant* qu'on appelle `qsort()`. Malheureusement, on ne peut pas remonter dans le temps avec un debugger, donc on va relancer le programme du début en plaçant un breakpoint pour s'arrêter à un moment bien choisi.

4. Ajouter un breakpoint au niveau de la boucle `for` juste après `malloc()`, puis relancer le programme avec `run`. Inspecter la valeur de `T`. Est-ce une valeur normale ?
5. En ajoutant un autre breakpoint et en redémarrant de nouveau le programme, inspecter le paramètre passé à `malloc()` et la valeur de `prod`.
6. Identifier la source du bug (indice : il n'est pas nécessaire de décortiquer le code de la fonction `digit_count()` ni d'y mettre de breakpoint).

Maintenant que le bug a été trouvé, on comprend mieux ce qu'il s'est passé dans ce programme.

7. Pourquoi le calcul des chiffres individuels n'a-t-il pas crashé alors qu'il accède à `T` ?
8. Les résultats qui ont été affichés pour 2022 et 2023 sans provoquer de crash étaient-ils corrects ?

Bonus :

9. (Optionnel) Corriger le bug en utilisant un type d'entiers plus approprié.

CS221 — TDM #2A

Make madness!

Grenoble INP — Esisar

Objectifs

Comprendre les enjeux de la compilation séparée :

- Flot de compilation : `.c` → `.o` → exécutable
- Conséquences sur l'usage des fichiers d'en-tête (`.h`) pour partager des fonctions/variables

Automatiser la compilation d'un programme avec `make` :

- Écriture de règles basiques (cibles, dépendances, commandes)
- Ne pas oublier de `.h` dans les dépendances en cas de compilation séparée

Instructions :

- Noter les réponses aux questions de chaque exercice en commentaire dans le code.
- Envoyer les programmes finaux sur Chamilo (*instructions d'archivage sur Chamilo*).

Rappels de cours

Rappels de cours

Tous les rappels de ce TDM ont finalement été déplacés dans le cours.

Voir les slides `CS221_CM2.pdf` sur Chamilo.

Exercices

Bibliothèque mathématique, 1/3 : organisation en plusieurs fichiers

- **Dossier : `CS221_TDM2A/ex1/`**
- Fichiers : `main.c`, `mymath.c`, `mymath.h`

Les exercices de ce TDM sont filés pour construire une mini-bibliothèque mathématique.

1. Créer un fichier `mymath.c` avec deux fonctions :
   - `int my_square(int n)` calculera le carré du nombre donné.
   - `int my_fact(int n)` calculera la factorielle du nombre (positif) donné.
     (Pour rappel, la factorielle de n est le nombre $1 \times 2 \times \ldots \times n$.)
2. Créer un en-tête associé `mymath.h` qui liste les deux fonctions.
3. Écrire un programme `main.c` qui inclue `mymath.h` et utilise les fonctions de `mymath.c`.
4. Quelle commande utilisez-vous pour compiler tous les fichiers de l'exercice d'un coup en un seul programme ? Vous prendrez soin d'inclure `-Wall -Wextra` et de vérifier qu'il n'y a aucun warning.

Bibliothèque mathématique, 2/3 : automatisation avec un Makefile

- **Dossier : `CS221_TDM2A/ex2/`**
- Fichiers : `main.c`, `mymath.c`, `mymath.h`, `Makefile`

1. Copier le dossier `ex1` vers `ex2`.
2. Dans `ex2`, écrire un `Makefile` contenant la commande de compilation de l'exercice précédent. Supprimer le programme exécutable et vérifier que la commande `make` le recompile.
3. Que se passe-t-il si vous lancez `make` une deuxième fois d'affilée ? Pourquoi ?
4. Sans modifier `main.c`, ajouter une fonction à `mymath.c`
   - `int my_pow(int a, int b)` calulera a^b (avec $b \geq 0$).

   La déclarer aussi dans `mymath.h`. Vérifier que lancer `make` recompile bien le projet.
5. Modifier `main.c` pour utiliser `my_pow()`. Constater que lancer `make` recompile le projet.
6. Ré-enregistrer `mymath.h` sans changer les autres fichiers et vérifier que `make` recompile de nouveau le projet.

Bibliothèque mathématique, 3/3 : compilation séparée

- **Dossier : `CS221_TDM2A/ex3/`**
- Fichiers : `main.c`, `mymath.c`, `mymath.h`, `Makefile`

1. Copier le dossier `ex2` vers `ex3`.
2. Modifier le `Makefile` pour effectuer une compilation séparée (chaque `.c` en `.o` avec `gcc -c` et une règle pour réunir tous les `.o` avec `gcc`). Pensez à garder la règle finale en premier parce que c'est elle qui est appelée quand on tape `make` sans arguments.
3. Supprimer le programme exécutable et vérifier que `make` compile le projet en trois commandes (deux `gcc -c` puis un `gcc`).
4. Modifier `main.c` et vérifier que `make` n'exécute que *deux* commandes (car `mymath.o` est à jour).

Dans les projets avec des milliers de fichiers, cette optimisation est très importante.

5. Modifier `mymath.h` et vérifier que `make` recompile tout. *Chaque fois qu'une règle utilise un fichier* `X.c` *elle doit avoir en dépendances tous les en-têtes inclus par* `X.c` *!*

Bonus, 1/2 : variables automatiques

- **Dossier : `CS221_TDM2A/bonus1/`**
- Fichiers : `main.c`, `mymath.c`, `mymath.h`, `Makefile`

Ces exercices bonus utilisent des notions qui n'ont pas été discutées en cours ; vous êtes fortement invité·es à explorer sur Internet ou demander à votre encadrant·e.

1. Copier `ex3` vers `bonus1`, supprimer le fichier exécutable et les `.o`.
2. Réécrire la commande de la règle `main.o` en utilisant la variable `$<` (nom de la première dépendance) et la variable `$@` (nom de la cible). Testez en modifiant `main.c` que la règle marche toujours. Remarquez que maintenant la commande ne contient plus spécifiquemenet le nom "main".
3. Faire de même avec la règle `mymath.o` ; remarquez que la nouvelle commande est identique à celle de `main.o`.
4. Faire de même avec la règle `main`, en utilisant cette fois la variable `$^` (nom de *toutes* les dépendances). On rappelle que `main` ne dépend que des `.o` et pas de `mymath.h`.

5. Combiner les règles pour `main.o` et `mymath.o` sous la forme suivante

```
%.o: %.c
        # ... commande ...

main.o: mymath.h
mymath.o: mymath.h
```

   Expliquer ce que le `%` et les deux “règles” sans commandes à la fin font.

6. Ajouter une règle `clean`, marquée `.PHONY` (qu’est-ce que ça fait ?) qui nettoie le fichier exécutable et les fichiers objets avec `rm -f`. Tester avec `make clean` suivi de `make`.

Bonus, 2/2 : bibliothèque statique

- **Dossier : CS221_TDM2A/bonus2/**
- Fichiers : `main.c`, `mymath.c`, `mymath.h`, `Makefile`

Une bibliothèque statique est une archive avec un nom de la forme `libX.a`, qui contient des fichiers `.o`. Lorsqu'on l'utilise dans un programme, cela revient à ajouter les fichiers `.o` de l'archive à la commande d'édition des liens.

On peut créer l'archive `libmymath.a` à partir de `mymath.o` de la façon suivante :

```
ar rcs libmymath.a mymath.o # ... d'autres .o si besoin
```

Et ensuite on peut l'ajouter à une commande de compilation avec les options `-L` et `-l` :

```
gcc main.o -o main -L. -lmymath
```

Dans bonus2, modifier votre `Makefile` pour créer une bibliothèque statique `libmymath.a` et la lier dans `main`. (Attention, la règle `main` dépendra donc de `libmymath.a` ; vous ne pouvez donc plus utiliser `$^` dans la commande, sinon `libmymath.a` apparaîtra verbatim dans la commande gcc.)

CS221 — TDM #5

Data drama!

Grenoble INP — Esisar

Objectifs

Comprendre les mécaniques classiques des piles et files :

- Piles : aspect local des calculs (les éléments après le premier sont le contexte)
- Files : implémentation classiqu avec un tableau circulaire (exercice de structures de données)

Instructions :

- Noter les réponses aux questions de chaque exercice en commentaire dans le code.
- Envoyer les programmes finaux sur Chamilo (*instructions d'archivage sur Chamilo*).

Exercice 1 : Pile via liste chaînée, et RPN

Implémentation d'une pile

- **Fichiers : `CS221_TDM2A/ex1_pile.h`, `CS221_TDM2A/ex1_pile.c`**
- À télécharger sur Chamilo.

Dans cet exercice, on va implémenter une pile *via* une liste chaînée et s'en servir pour une mini-calculatrice en RPN.

1. `ex1_pile.h` vous est fourni avec la définition d'une structure de pile. Y a-t-il une différence entre un noeud de la pile et un noeud d'une liste chaînée ?
2. Compléter les 4 fonctions de manipulation de pile dans `ex1_pile.c`. Manipuler le début d'une liste chaînée est-il plus simple ou plus difficile que manipuler la fin ?
3. Écrire dans `main()` (avant la définition de `rpn_expr`) un test qui vérifie que vos fonctions sont correctes. Expliquez *en détail* en commentaire en quoi le test confirme que la pile marche bien.

Calcul en RPN

La RPN (“Reverse Polish Notation” ou « Notation polonaise inversée ») est une notation pour les calculs arithmétique dans laquelle les opérations vont *après* les opérandes au lieu d'aller *au milieu*.

Par exemple, l'expression « 1 2 + » représente l'opération 1+2. Ces expressions peuvent être imbriquées : « 1 (3 4 *) + » représente l'opération 1+(3*4). Un théorème drôle sur la RPN est que les parenthèses qu'on utilise habituellement pour éviter les ambiguités sont superflues. Par exemple :

- 1+(3*4) s'écrit « 1 3 4 * + » ;
- (1+3)*4 s'écrit « 1 3 + 4 * ».

L'évaluation d'une expression en RPN est également relativement simple. L'algorithme classique utilise une pile de la façon suivante :

- Commencer avec une pile vide.
- Itérer sur tous les symboles de la formule :
  - Chaque fois qu'on rencontre un nombre, le pousser sur la pile.
  - Chaque fois qu'on rencontre un opérateur, dépiler deux nombres, et pousser à la place leur somme/produit/différence.
- À la fin, il doit rester exactement un nombre dans la pile : le résultat.

Un exemple est fourni dans `ex1_pile.c` sous la forme d'un tableau `rpn_expr` où les nombres sont limités aux chiffres (0…9) et les opérateurs sont représentés par des caractères (`'+'`, `'-'`, `'*'`).

4. Implémenter l'évaluation d'expressions RPN et déterminer la valeur de `rpn_expr`.

Exercice 2 : File via tableau circulaire

Exercice 2 : File via tableau circulaire

- **Fichiers : `CS221_TDM2A/ex2_file.h`, `CS221_TDM2A/ex2_file.c`**
- À télécharger sur Chamilo.

Une description guidée d'une file dans un tableau circulaire est fournie dans les fichiers `ex2_file.h` et `ex2_file.c`.

1. Combien d'éléments une file peut-elle contenir en fonction de `size` ?
2. Exprimer le nombre d'éléments présents dans la file selon la valeur de `first` et `next`.
3. Implémenter les fonctions en suivant la description de `ex2_file.h`.
4. Écrire dans `main()` un test qui vérifie la correction de ces fonctions. Lister en commentaire, dans le plus grand détail possible, toutes les situations qu'il faudra tester pour garantir que la structure marche bien.
5. Quelle(s) fonction(s) pensez-vous qu'il serait utile d'ajouter à la file ?