Compsys C

Intro

Pour les entrées on utilise `scanf("%lf", &variable)`, le `&` permet de donner l'adresse de la variable à `scanf` pour qu'il la modifie.

Pour utiliser `M_PI` (= pi),

```
/* ligne pour avoir M_PI (= pi). A mettre AVANT le include de math.h. */
#define _USE_MATH_DEFINES
#include <math.h>
```

Syntaxe pour initialiser une structure :

```
struct Person p1 = {"Alice", 25};

Person p2;
p2.age = 30;
strcpy(p2.name, "Bob");
```

> **à ne pas oublier**
>
> tout ce qu'il n'est pas évident de faire car ne va pas casser le code
>
> - ajouter les `const` à tous les endroits possibles
> - vérifier qu'à chaque `malloc` un `free` est mis dans le code
> - vérifier que chaque fichier ouvert est bien fermé

Les pointeurs

Les pointeurs :

- référence (éviter la duplication, pointeur A vers objet X, pointeur B vers objet X)
  - ne pas oublier de protéger l'argument avec un `const`
- généricité (on veut que le pointeur A pointe vers objet X puis vers objet Y)
  - `typedef double *Fonction(double)`
- portée (pour éviter que l'objet ne soit enlevé de la mémoire)

> **Cas d'utilisation d'un pointeur (portée)**

En retour de fonction : `Complexe* resultat = malloc(sizeof(Complexe));` puis `return resultat` **et non pas &resultat!**
En effet C considère que comme un `ptr` est renvoyé, la variable `resultat` n'est plus utilisée et donc détruite (et le pointeur ne pointe plus vers la bonne valeur)!

allouer un pointeur : créer une valeur puis en garder l'adresse dans le pointeur
libérer un pointeur : supprimer la valeur de la mémoire (mais l'adresse dans le pointeur est toujours la même). une bonne pratique est d'effacer aussi l'adresse du pointeur.

`nullptr` en C++ c'est `NULL` en C

déclaration d'un pointeur : `int* ptr = NULL;` ou `int* ptr = &i;`
pour lire la valeur d'un pointeur `printf("%d", *ptr);`
pour les structures `p->x` est équivalent à `(*p).x` si `p` est un pointeur sur une structure

Pointeur constant, et pointeur vers un objet constant

TLDR

`const type * ptr` objet constant
`type * const ptr` pointeur constant

Déclaration	Pointeur externe constant ?	Pointeur intermédiaire constant ?	Objet final constant ?
`int **ptr`	❌	❌	❌
`const int **ptr`	❌	❌	✅
`int * const *ptr`	❌	✅	❌
`const int * const *ptr`	❌	✅	✅
`int ** const ptr`	✅	❌	❌
`const int ** const ptr`	✅	❌	✅
`int * const * const ptr`	✅	✅	❌
`const int * const * const ptr`	✅	✅	✅

JCC écrit plutôt `type const* ptr` (identique à `const type* ptr`) → déclare un pointeur sur un objet constant de type `type` (on ne pourra pas modifier la valeur de l'objet au travers de `ptr` mais on pourra faire pointer `ptr` vers un autre objet)

```
*ptr = 9; // impossible !
ptr = &j; // possible!
```

et `type* const ptr` → déclare un pointeur constant sur un objet (on ne pourra pas faire pointer `ptr` vers autre chose mais on pourra modifier la valeur de `obj` au travers de `ptr`)

```
*ptr = 9; // possible !
ptr = &j; // impossible
```

Allocation dynamique (malloc, calloc, realloc)

allouer de la mémoire en C :

- allocation statique, à la compilation : sur le stack, variables locales
- allocation dynamique, pendant l'exécution : sur le heap, indépendamment du fait qu'il y ait une variable ou non

`ptr = malloc(taille);` pour allouer dans le `heap`
`calloc(3, sizeof(double))` pour allouer 3 double à la suite dans le `heap` on ne fait pas `malloc(3 * sizeof(double))` car il pourrait y avoir un débordement avec la multiplication
ces fonctions renvoient `NULL` si l'allocation n'a pas pu se faire

`calloc` initialise aussi la mémoire à zéro, tandis que `malloc` n'initialise rien
on peut utiliser `memset` pour initialiser la mémoire `memset(ptr, 0, sizeof(*ptr))` pour initialiser à `0`

avec `free(ptr);` on libère la mémoire allouée
bonne pratique : ajouter aussi un `ptr = NULL` après.

⚡ On ne peut free qu'un pointeur qui a été alloué dynamiquement !

On ne peut appeler `free` que sur un pointeur `ptr = malloc` ou un `calloc`, etc. Le code suivant **ne fonctionnera pas**!

```
int* ptr = calloc(2, sizeof(int));
ptr[0] = 1;
ptr++;
ptr[0] = 1;
free(ptr); // ne fonctionne pas ! car ptr[1] a été alloué à partir de
ptr[0]
```

```
typedef struct {
    size_t size; // nombre d'éléments dans le tableau
```

```
    size_t allocated; // taille allouée en mémoire
    int* content;
} vector;
```

`realloc(ptr_old, nouvelle_taille)` (comme si c'était `re(m)alloc`) permet de réallouer des zones déjà allouées (en **augmentation** ou **diminution**). Le pointeur va être déplacé si nécessaire (si par exemple dans la zone mémoire initiale il n'y a plus la place de rajouter des éléments). Si le `realloc` échoue, la zone mémoire initiale sera inchangée (et `NULL` sera renvoyé).
attention : bien vérifier avec `realloc` qu'on vérifie qu'il n'y a pas de débordement!

> **Valeur de retour de calloc, realloc**
>
> Les valeurs de retour `ptr` de `calloc` et `realloc` sont la valeur du **premier** élément de la zone mémoire créée ! Pour accéder au deuxième élément, on fait `ptr[1]`, au troisième `ptr[2]`, etc.
> (et donc `*ptr` ≡ `ptr[0]`).

Les chaînes de caractères

En C ce sont des tableaux de caractères. Ils se terminent par le caractère nul (`\0` ou `(char) 0`).

```
char nom[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
// en pratique on écrit juste
char nom[6] = "Hello";
```

en allocation dynamique : `char *nom;` + `malloc` / `calloc`. il faut allouer `n+1` caractères ! à cause du `\0`.

l'utilisation du = avec une valeur littérale n'est à faire qu'avec les tableaux.

on peut **copier** une chaîne avec `strncopy(char* dest, char const* src, size_t n)` copie les `n` premiers caractères de `src` dans `dest`. Retourne `dest`. Attention, ça n'ajoute pas `\0` à la fin si `src` a plus de `n` caractères !

ajoute au plus `n` caractères de `src` à la fin de `dest` ! retourne `dest`
`char* strncat`.

on peut comparer des chaînes avec `strncmp` !

> **Pièges : pointeur ou pas pointeurs ?**
>
> ```
> char* s = "bonjour";
> ```

En C, le code ci-dessus fonctionne mais n'est **pas correct**! En effet `s` va pointer vers un emplacement mémoire en lecture seule. il faut bien écrire `const char* s = "bonjour";` si on veut que ce soit juste.

Sinon, on doit `calloc` puis utiliser `strncopy` !

```
char* s = calloc(TAILLE+1, 1); strncpy(s, "bonjour", TAILLE);
```

Pointeurs sur les fonctions

On utilise `(*ptr)` à la place du nom de la fonction. Par exemple :

```
g = &f; // ou g = f;
z = (*g)(i); // ou g(i);
```

Généricité

- **pointeurs génériques** : `void* ptr`, on ne sait pas sur quoi pointe `ptr`
- **fonctions génériques** : `int compare_int(const void* a, const void* b);` on est obligés d'utiliser des pointeurs !

Casting

On peut changer le type `(type) expression;` p. ex. pour aller des double vers des int. Casting de pointeur : ça ne va pas changer la valeur pointée mais son interprétation !

```
double x = 5.4;
int* i = (int*) &x;

(int) x; // 5. ici c'est la valeur x qui est convertie en double (beaucoup
de travail)
*i; // -1717986918. là on lit directement le double stocké en
l'interprétant comme un int!
```

le casting est utile quand on utilise des pointeurs génériques (il se fait tout seul `int* ptr1;`, `void* ptr2;` , `ptr2 = ptr1;`).

on pourrait réécrire `compare_int` comme `int compare_int(int const* a, int const* b);` et ensuite caster cette fonction en `(int (*)(void const*, void const*))` quand on veut l'utiliser de façon générique !

Les tableaux multi-dimensionnels

? Quelle différence entre `int tab[][N]` et `int** tab` ?

`int** tab` :

- n'est pas continu en mémoire
- n'est pas alloué au départ
- les lignes n'ont pas forcément le même nombre d'éléments

ⓘ **à noter : `ptr[0]` ≡ `*(ptr)` , on a aussi `ptr[0][0]` ≡ `*(ptr[0])` , etc.**

`int p1[N][M]`

C'est un **tableau de N tableaux de int de taille M.**

`int* p2 [N]`

C'est un **tableau de `N` pointeurs**, où chaque pointeur pointe vers un **int**. Ainsi, `p2[0]` est un pointeur vers un premier entier, `p2[1]` vers un autre entier, etc. jusqu'à `p2[N-1]` .

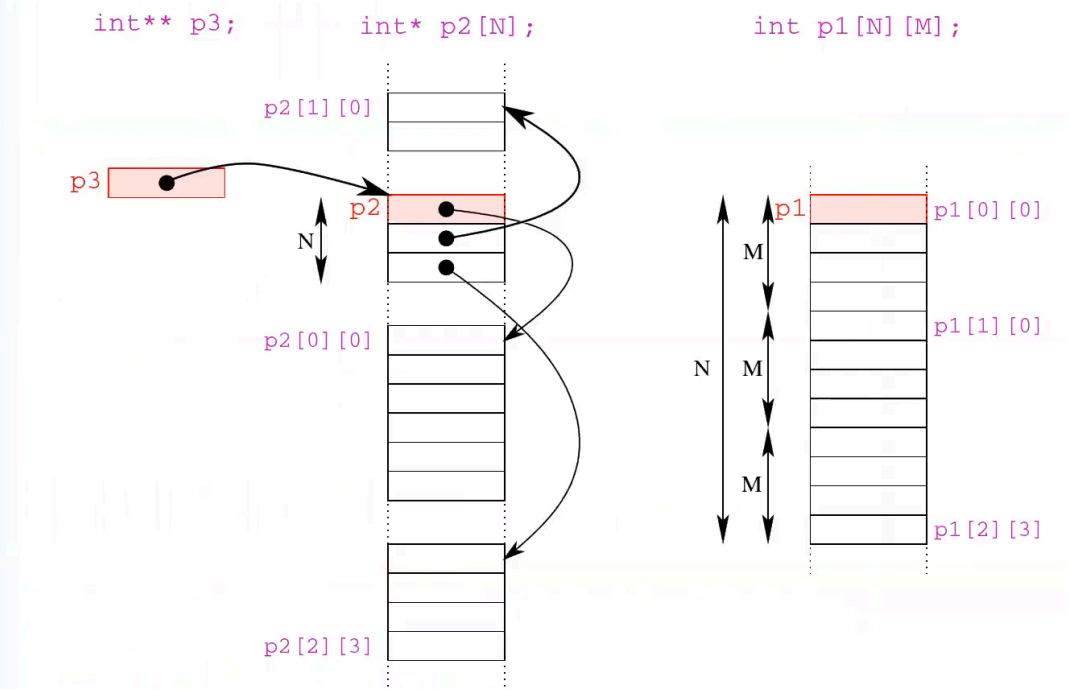
Cependant rien n'empêche `p2[0]` d'être un tableau de taille arbitraire ! par exemple `p2[0][0]` (qui pour rappel ≡ `*(p2[0])`) est un `int` certes, mais `p2[0][1]` peut aussi être un `int` ! de même que `p2[0][2]` , ... jusqu'à `p2[0][M]` !

On obtient donc un **tableau de `N` tableaux de int de tailles arbitraires!**

`int** p3`

C'est un **pointeur vers un pointeur de int** (donc `*(ptr)` ou `ptr[0]` est un pointeur vers un `int`). Cependant comme avant, rien n'empêche `p3[1]` de pointer aussi vers un pointeur de `int` !

et on se retrouve avec un **tableau de taille arbitraire de tableaux de int de tailles arbitraires!**



Note : `p2` est un tableau, stocké dans le stack ! mais comme on remplit chaque élément de p2 avec un `calloc` (dans le heap), on a des différences d'adresses importantes.

⑦ Quand utiliser `[SIZE]`, `ptr*` ou `ptr**` ?

Par ordre de préférence :

On utilise `type array[SIZE]` quand on veut stocker une liste de taille fixe d'éléments (que ce soient des adresses, des double, des struct, peu importe).

Sinon, ce sont **des tableaux dynamiques** et on utilise `type*` ou `type**`. Maintenant, est-ce qu'on veut un tableau de valeurs ou un tableau d'adresses, de références vers les valeurs ?

On utilise `type* ptr` puis `ptr[0]`, `ptr[1]` ? → quand on veut stocker directement les valeurs éléments dans notre tableau, ce qui n'est pas toujours le cas.

On utilise `type**` uniquement dans le cas où on est obligés d'avoir un tableau de références vers des éléments (typiquement si on créé des `Node` qui se référencent entre eux, comme avec un champ `neighbors`), on **doit** avoir un tableau de références et non pas plein de copies des `Node`.

Arithmétique des pointeurs

- `+1` à un pointeur fait avancer le pointeur de la taille d'un objet pointé
- `*p++` ≡ `*(p++)` ≡ accéder à la valeur de `p`, et augmenter `p` de 1
- `(*p)++` ≡ augmenter la valeur stockée à `p` de 1
- **attention**! `p2 - p1` n'est **pas** de type `int` mais de type `ptrdiff_t` !

ⓘ `while(lu = *p++) {}` ≡ `while((lu = *p++) != '\0') {}`

comme les pointeurs ont un type, quand on fait `+1` , C ajoute automatiquement le bon offset pour arriver au prochain élément en fonction de la taille du type !

en résumé, `(int) p+1` est égal à `(int) p + sizeof(Type)` !

Sizeof

`sizeof` n'est pas évaluée si on lui donne une expression (et non un type)

⚡ les erreurs avec `sizeof`

Le code suivant ne fonctionne pas !

```
void f(int t[N]) { // équivalent à f(int* t) à cause du decay !
    ...sizeof(t)/sizeof(int)...
    // évalué comme la taille d'un pointeur sur la taille d'un entier
!
}
```

Rappel : un tableau n'a **jamais** connaissance de sa taille ! Voir https://www.geeksforgeeks.org/array-decay-in-c/.

```
    int tab[1000];
    const int* const end = tab + sizeof(tab);
// erreur ! sizeof(tab) c'est 1000 * sizeof(int) !
// on ne veut que 1000!
// on peut faire sizeof(tab)/sizeof(int)
```

(bonus) Flexible array member

l'idée c'est d'avoir une struct

```
struct vector_double { int size; double a[1]; }
```

sans pointeur donc, un tableau a un élément

et on initialise une zone mémoire **continue** de taille `sizeof(struct vector_double) + (size-1) * sizeof(double)` !

intérêt :

- au lieu d'avoir un `struct` d'un côté et un `double *` qu'il faut allouer séparément, on a une seule allocation et donc un seul `free` à gérer
- on a une zone mémoire continue

Plusieurs fichiers ?

On sépare la partie **déclaration** de notre API et la partie **implémentation** de notre API.

Partie déclaration

- fichiers headers, `.h`
- s'importent avec `#include`
- ils doivent commencer par `#pragma once;`

pour n'importer qu'une fois le fichier `.h`, on peut utiliser l'inclusion conditionnelle (avec `if not define`) :

```
#ifndef MONFICHIERAMOI_H
    #define MONFICHIERAMOI_H
    // le fichier comme d'habitude
#endif
```

Pour qu'un module C puisse être utilisé en C++, il faut rajouter quelques lignes :

```
#pragma once

#ifdef __cplusplus
extern "C" {
#endif

void function(int* ptr);

#ifdef __cplusplus
}
#endif
```

Partie implémentation

- fichiers sources, `.c`

Compilation séparée

Compilation séparée, pour chaque fichier, on créé un fichier `.o` puis on les lie ensemble en une fois !

- à faire `N` fois : `gcc -c monFichierJ.c -o monFichierJ.o`

- à faire une 1 fois à la fin : `gcc monFichier1.o ... monFichierN.o -o monProgramme`

et `monProgramme` est le fichier exécutable final

Makefile

C'est long, la solution → le **Makefile**

Il se construit de la façon `but: dépendances`, p.ex: `questionnaire: qcm.o demander_nb.o`

On ajoute `LDLIBS = -lm` pour la lib mathématique.

```
CFLAGS += -g -std=c17
LDLIBS = -lm

all: questionnaire

questionnaire: demander_nombre.o qcm.o questionnaire.o

questionnaire.o: questionnaire.c qcm.h
qcm.o: qcm.c qcm.h demander_nombre.h
```

> ⓘ **Les dernières lignes peuvent être générées avec `gcc -MM *.c`**

On peut préciser la commande à exécuter pour passer des dépendances au but :

```
questionnaire.o questionnaire.c qcm.h
    gcc -o questionnaire questionnaire.o qcm.o
```

Ou on peut utiliser les variables prédéfinies :

```
questionnaire.o questionnaire.c qcm.h
    gcc -o $@ $<
```

- `$@` : le but
- `$?` : les dépendances qui ne sont plus à jour
- `$<` : dépendances telles que définies par les règles par défaut
- `$^` : liste des dépendances
- `$(CC)` : le nom du compilateur
- `$(CFLAGS)` : options de compilation
- `$(LDFLAGS)` : options du linker
- `$(LDLIBS)` : bibliothèques à ajouter

On peut définir nos propres variables.

Édition des liens

Un code objet c'est du code partiel et des **tables d'adressage** (pour retrouver les morceaux fournis dans les autres fichiers `.o`).

- table d'**exportation** des objets globaux (tous les objets qu'un `.o` propose aux autres)
- table d'**importation** (par exemple quand on importe `sqrt`, on ne sait pas où elle est, on n'a pas l'adresse)
- table des **tâches** : liste des endroits dans le code où il y a des adresses à résoudre

L'édition de liens ne peut pas tout résoudre ! L'OS va charger le programme (dans le `loader`), et va réécrire les adresses qu'il ne connaît pas pour résoure les dernières ambiguïtés.

par exemple, pour les tables **d'exportation** :

```
qcm.o
affiche        | code | 0
poser_question | code | 342

questionnaire.o
main           | code | 0
```

avec nom, type, et adresse relative (adresse de la première instruction de la fonction par rapport à tout le code du module)

par exemple, pour les tables **d'importation** :

```
qcm.c -- aucune table

questionnaire.c : affiche, poser_question, sqrt
```

par exemple pour les tables **des tâches** :

```
qcm.c : tous les sauts de mémoire (jump)

questionnaire.c : tous les sauts de mémoire, plus tous les endroits où un
appel à du code importé existe.
```

Le **chargeur** modifie toutes les adresses des sauts de mémoire (dont appels, code importé, etc.) en fonction de l'adresse de l'entrypoint du programme (on "translate" le code).

argc/argv

On peut aussi utiliser ce prototype pour main :

```
int main(int argc, char* argv[]) {
    // argc est le nombre d'arguments
    // argv est un tableau de chaîne de caractères avec les arguments
}
```

`monprogramme -v fichier`

- `argv[0]` --> le nom du programme `monprogramme`
- `argv[1]` --> `-v`
- `argv[2]` --> `fichier`

et donc, ici `argc = 3`

Exemple de traitement d'arguments :

```
int traite_arguments(int* nb, char** argv)
{
  int required = 0; // nb d'arguments obligatoires déjà traités
  char const * const pgm_name = argv[0]; // le nom du programme

  ++argv; --(*nb); // passe à l'argument suivant

  while ((*nb) > 0) { // tant qu'il y a des arguments
    if (!strcmp(argv[0], "-P")) { //  option -P
      /* par exemple, avec option_P une variable globale,
       * ou mieux : le champ d'une structure passée en paramètre  */
      option_P = 1;

    } else if (!strcmp(argv[0], "-i")) {
      // une option avec 1 argument : par exemple -i nom
      option_I = 1;
      ++argv; --(*nb); // passe à l'argument suivant
      if (*nb == 0) { // si l'argument de l'option n'est pas là...
        fprintf(stderr,
                "ERREUR: pas d'argument pour l'option -i\n");
        return ERREUR_I; // une constante définie globalement
      } else {
```

GDB

- `layout src`
- `run` ou `r`

help

break NUMERO_DE_LIGNE ou br NUMERO_DE_LIGNE

break NOM_DE_FONCTION ou br NOM_DE_FONCTION

delete

info br

where ou bt (ou backtrace)

print ou p

display

cont ou c

next ou n

step ou s