Fiches Résumé - ICC-C 2025-2026

1. Semaine 1 : expressions, variables

Structure principale d'un programme :

```c
#include <stdio.h>

int main() {
  // code du programme ici

  return 0;
}
```

Les *commentaires* sont ignorés par le compilateur. Ils servent à communiquer de l'information aux humains :

```c
code; // commentaire sur une ligne, à partir des slashes

/* Long commentaire
 * qui s'étend
 * sur plusieurs lignes.
 */
```

Afficher du texte à l'écran :

```c
printf("Hello world!\n");
```

Les " délimitent du *texte* (voir plus tard ce que c'est réellement dans un programme). Dans du texte, le \n est un seul caractère spécial qui représente le retour à la ligne (imaginez que cela représente l'appui sur la touche Enter).

Types de variables

Type C	Vague correspondant en maths	Vraies valeurs possibles
`int`	$\mathbb{Z}$	$[-2^{31}, 2^{31}-1]$
`double`	$\mathbb{R}$	Valeurs en « virgule flottante » (approximations)

Dans ce cours, lorsque nous utilisons la notation d'intervalle $[a, b]$, il s'agit toujours *d'entiers*. On a donc $[a, b] = \{n \in \mathbb{Z} : n \geq a \wedge n \leq b\}$. On utilisera aussi très souvent (plus souvent même !) des intervalles d'entiers *semi-ouverts* : $[a,\ b[= \{n \in \mathbb{Z} : n \geq a \wedge n < b\}$ (on peut avoir $a = b$, auquel cas l'ensemble est vide). Une propriété intéressante des intervalles d'entiers semi-ouverts est que $|[a,\ b[| = b - a$.

Définition d'une variable :

```c
int age = 19;
double pi = 3.141592653589793;
```

Afficher une variable :

```c
printf("Age = %d\n", age);
printf("PI = %g\n", pi);
```

Afficher du texte avec plusieurs valeurs insérées :

```c
printf("Age = %d, PI = %g\n", age, pi);
```

Chaque `%` correspond à une variable en argument de `printf`, dans l'ordre. Les `%` s'appellent des « spécificateurs de format ». Ils doivent correspondre au *type* des variables qu'on veut y insérer. En voici quelques uns qui vous seront utile :

Spécificateur	Type associé	Format
`%d`	`int`	Notation **d**écimale (c'est-à-dire en base 10)
`%x`	`int`	Notation he**x**adécimale (c'est-à-dire en base 16)
`%f`	`double`	Notation à virgule **f**ixe
`%.5f`	`double`	Notation à virgule **f**ixe, avec 5 chiffres derrière la virgule
`%e`	`double`	Notation scientifique (**e**xponentielle)
`%.6e`	`double`	Notation scientifique (**e**xponentielle), avec 6 chiffres significatifs
`%g`	`double`	Notation **g**énérale (virgule fixe ou scientifique, au besoin)
`%.4g`	`double`	Notation **g**énérale, avec 4 chiffres significatifs

On peut former des *expressions* avec les constantes et les variables :

```
int annee_naissance = 2001;
int annee_courante = 2026;
int age = annee_courante - annee_naissance;

double radius = 3.5;
double perimeter = 2.0 * pi * radius;
```

Les parenthèses `()` permettent de forcer la priorité des opérateurs :

```
double base1 = 6.0;
double base2 = 4.3;
double height = 3.5;
double trapezoid_area = ((base1 + base2) * height) / 2.0;
```

Les opérateurs suivants sont disponibles sur les `double`. Attention, chaque opérateur est suivi d'une approximation (à environ 16 chiffres décimaux significatifs). Les résultats des calculs sur des `double` sont donc rarement exacts.

Opérateur C sur `double`	Opération mathématique
`-a`	$-a$
`a + b`	$a + b$
`a - b`	$a - b$
`a * b`	$a \cdot b$
`a / b`	a/b

Les opérateurs suivants sont disponibles sur les `int`. En cas de dépassement des bornes $[-2^{31}, 2^{31}[$, le comportement est *indéfini* (pire qu'une approximation). Par contre, si on reste dans les bornes, le résultat est toujours *exact*.

Opérateur C sur `int`	Opération mathématique
`-a`	$-a$
`a + b`	$a + b$
`a - b`	$a - b$

Opérateur C sur `int`	Opération mathématique
`a * b`	$a \cdot b$
`a / b`	quotient de la division euclidienne de a par b ($= \lfloor a/b \rfloor$ pour des positifs)
`a % b`	reste de la division euclidienne de a par b ($= a - b \cdot \lfloor a/b \rfloor$ pour des positifs)

⚠️ L'opérateur `/` a donc une signification très différente s'il est appliqué sur des `int` ou sur des `double` !

Dans les programmes informatiques, les `int` sont beaucoup plus fréquents que les `double`. N'utilisez des `double` que si vous devez manipuler des nombres non-entiers (ou de très grands entiers). Préférez toujours les `int` lorsque vous manipulez des nombres entiers.

On peut *réassigner* le contenu d'une variable avec l'opérateur `=`.

```c
int x = 5;
int y = x * 2;
printf("x = %d, y = %d\n", x, y); // affiche x = 5, y = 10
x = 6;
printf("x = %d, y = %d\n", x, y); // affiche x = 6, y = 10
```

On *évitera* autant que possible de réassigner les variables. En effet, cela est contraire aux intuitions que nous avons construites en mathématiques. En maths, si je dis « Soit $x = 5$ », je ne peux pas dire plus tard « Maintenant $x = 6$ ». Cependant, en C nous devrons le faire lorsque nous utiliserons des *structures de contrôles* (voir plus tard).

2. Semaine 2 : booléens, branchements conditionnels

2.1. Booléens

Le type `bool` représente un *booléen* ou « valeur de vérité ». Il a exactement 2 valeurs possibles : `true` (vrai) et `false` (faux). Il tire son nom de **l'algèbre de Boole**, bien que nous y pensons plutôt en termes de logique propositionnelle.

En C, son usage requiert l'utilisation de l'include

```c
#include <stdbool.h>
```

Type C	Vague correspondant en maths	Vraies valeurs possibles
`int`	$\mathbb{Z}$	$[-2^{31}, 2^{31} - 1]$
`double`	$\mathbb{R}$	Valeurs en « virgule flottante » (approximations)
`bool`	$\{\text{FAUX}, \text{VRAI}\}$	$\{\texttt{false}, \texttt{true}\}$

Les opérateurs suivants sont disponibles sur les `bool`.

Opérateur C sur `bool`	Opération mathématique
`!a`	$\neg a$
`a || b`	$a \vee b$
`a && b`	$a \wedge b$

Les *opérateurs de comparisons* sur les `int` et les `double` prennent des nombres comme opérandes, mais résultent en une valeur de vérité, donc un `bool` :

```c
int a = 5;
int b = 7;
bool x = a < b; // x vaut true car "a < b" est vrai
bool y = a >= b; // y vaut false car "a >= b" est faux
bool vrai = true; // le mot-clef true représente la valeur "vrai"
bool faux = false; // le mot-clef false représente la valeur "faux"
```

Opérateur C sur `int` ou `double`	Relation mathématique, vrai ou faux (`bool`)
`a == b`	$a = b$
`a != b`	$a \neq b$
`a < b`	$a < b$
`a <= b`	$a \leq b$
`a > b`	$a > b$
`a >= b`	$a \geq b$

Il peut aider de voir les opérateurs de comparaison comme des *questions*. Le code C `a < b` peut être lu comme « est-ce que $a < b$? ». Il est alors clair que la *réponse* à cette question est soit oui (`true`) soit non (`false`).

⚠️ Une erreur courante est de confondre `a = b` (*réassignation*, pas d'équivalent mathématique) et `a == b` (*test d'égalité*, équivalent à $a = b$ en maths).

On peut aussi comparer des booléens. En revanche, cela n'a de sens qu'avec == et `!=`. De plus, c'est souvent utilisé à mauvais escient. Si vous avez la tentation d'écrire « `calcul == true` », écrivez seulement « `calcul` ». De même, « `calcul == false` » est équivalent à « `!calcul` ».

On peut créer des expressions complexes mêlant entiers, flottants et booléens. Par exemple, l'expression `(a >= 5) && (a < 10)` teste si $a \in [5, 10[$. Remarquez que `(a >= 5)` est un `bool`, que `(a < 10)` aussi, et donc qu'on peut les combiner avec && pour former un autre `bool`.

2.2. Branchement conditionnel (`if`)

La structure `if..else` est appelée « branchement conditionnel » ou plus simplement « condition », voire même « un `if` ». Elle demande une expression booléenne. Si elle vaut `true`, la première *branche* est exécutée. Sinon (si elle vaut `false`), la branche après le `else` est exécutée.

```c
if (expression_booleenne) {
  code_si_true;
} else {
  code_si_false;
}
```

La partie `else { ... }` peut être omise. Dans ce cas, c'est équivalent à `else {}` (donc « sinon, ne rien faire »).

On peut enchaîner plusieurs conditions avec le format suivant :

```c
if (x == 0) {
  // ...
} else if (x > 0) {
  // ...
} else {
```

```
  // ...
}
```

2.3. Lecture au clavier (`scanf`)

La fonction prédéfinie `scanf` permet de demander la valeur d'une variable à entrer au clavier. Sa syntaxe doit être considérée magique pour l'instant :

```
int x;
scanf("%d", &x); // ne pas oublier le &, et ne pas mettre de \n
```

lit un entier au clavier, tandis que

```
double x;
scanf("%lf", &x);
```

lit un `double`.

2.4. Exemple complet

Exemple complet mêlant tous les concepts principaux de la semaine :

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

int main() {
  printf("Entrez un entier : "); // pas de \n, intentionnellement
  int n;
  scanf("%d", &n);

  if (n == 0) {
    printf("n vaut 0.\n");
  } else (n > 0) {
    printf("%d est strictement positif\n", n);
  } else {
    printf("%d est strictement négatif\n", n);
  }
}
```

3. Semaine 3 : fonctions et boucles

3.1. Fonctions

La définition d'une fonction se fait avec la syntaxe suivante :

```
type_retour nom_fonction(type_param1 nom_param1, type_param2 nom_param2) {
  // corps de la fonction
  return expression_retour;
}
```

Il peut y avoir de 0 à plusieurs paramètres. S'il n'y a « rien à renvoyer », on utilise le type de retour fictif `void` (vide).

Exemple :

```
double poly2degre(double a, double b, double c, double x) {
  return a*x*x + b*x + c;
}
```

L'*appel* ou *utilisation* d'une fonction ressemble à la syntaxe mathématique :

```
double resultat = poly2degre(5.0, 4.0, 3.0, 2.5);
```

Une erreur courante est de vouloir répéter les types de retour et les types de paramètres quand on appelle la fonction. Par exemple, vouloir écrire

```
double resultat = double square(double a);
```

Cela n'a pas de sens en C, et le compilateur vous le fera savoir. Il faut bien distinguer *définition* (avec les types) de l'*appel* (sans les types). Il faut écrire

```
double resultat = square(a);
```

Contrairement aux fonctions mathématiques, les fonctions en C peuvent **faire** des choses. On peut par exemple écrire une fonction qui demande un entier au clavier. C'est utile parce que c'est une séquence d'opérations que l'on fait souvent.

```
int read_int() {
  printf("Entrez un entier : ");
  int n;
  scanf("%d", n);
  return n;
}
```

On peut ensuite l'utiliser pour demander plus facilement des entiers au clavier.

```
int x = read_int();
int y = read_int();
```

3.2. Boucles

Les boucles répètent une séquence d'instructions un nombre de fois arbitrairement grand.

La boucle `while` est la plus fondamentale. Elle correspond aux boucles « Tant que » vues en théorie.

```
int i = 0;
int sum = 0;
while (i < 10) { // Tant que i < 10, répéter :
  sum += i;
  i++;
}
```

La boucle `do..while` est une variante (beaucoup) plus rare, qui exécute au moins une fois son corps. Elle correspond aux boucles « Répéter … Tant que » vues en théorie.

```
int n;
do {
  n = read_int();
} while (n < 0);
```

La boucle `for` est la plus commune. Elle rassemble *initialisation*, *condition* et *étape* en une structure. Elle correspond aux boucles « Pour » vues en théorie.

```
for (int i = 0; i < n; i++) {
  sum += i;
}
```

De manière générale, une boucle

```
for (init; cond; step) {
  body;
}
```

est équivalente à

```
init;
while (cond) {
  body;
  step;
}
```

mais est souvent plus facile à lire (avec un peu d'entraînement).

4. Semaine 4 : tableaux

Les tableaux sont des séquences de n éléments d'un même type, auxquels on peut accéder par indice. Les indices commencent à 0, et sont donc dans l'ensemble $[0,\ n[$. Un tableau a une taille fixée à sa création. Elle ne peut jamais être modifiée ultérieurement.

```
int entiers[10]; // tableau de 10 entiers
entiers[0] = 5; // écrit la valeur 5 dans la 1ère case du tableau
printf("%d\n", entiers[0]);
```

Le type des éléments d'un tableau peut être n'importe quoi, comme `double flottants[n]`, `bool booleans[n]`, etc.

La taille et les indices des tableaux sont des valeurs de type `size_t`. Un `size_t` est un entier *non signé* (sans signe), qui est donc toujours positif ou nul.

Type C	Vague correspondant en maths	Vraies valeurs possibles
`int`	$\mathbb{Z}$	$[-2^{31}, 2^{31}-1]$
`size_t`	$\mathbb{N}$	$[0,\ 2^{64}[$
`double`	$\mathbb{R}$	Valeurs en « virgule flottante » (approximations)
`bool`	$\{\text{FAUX}, \text{VRAI}\}$	$\{\texttt{false}, \texttt{true}\}$

On itère souvent sur tous les éléments d'un tableau avec une boucle `for`. Il est commun d'utiliser le nom `i` (index) pour la variable qui passe par tous les indices du tableau. On utilise le spécificateur de format `%lu` si on veut afficher un `size_t` (**l**ong **u**nsigned).

```
for (size_t i = 0; i < 10; i++) {
  printf("Indice %lu, valeur %d\n", i, entiers[i]);
}
```

Un tableau ne connaît que là où il commence. Il **ne connaît pas sa propre taille** ! C'est à vous de transmettre la taille du tableau en plus du tableau lui-même.

En paramètre de fonction, on note `int param[]` un paramètre qui est un tableau d'entiers. Il faut donc aussi transmettre sa taille en paramètre supplémentaire.

```
int sum(int entiers[], size_t len) {
  int result = 0;
  for (size_t i = 0; i < len; i++) {
    result += entiers[i];
  }
  return result;
}
```

On ne peut pas (pour l'instant) *renvoyer* un tableau d'une fonction. Cependant, si la fonction modifie les éléments d'un tableau, ils seront aussi modifiés à l'appel. C'est le *même* tableau (ce sont les mêmes cases de mémoire).

```c
void read_ints(int entiers[], size_t len) {
  for (size_t = 0; i < len; i++) {
    printf("Entrez l'élément %lu :", i);
    scanf("%d", &entiers[i]);
  }
}

int main() {
  int valeurs[10];
  read_ints(valeurs, 10);
  printf("%d\n", valeurs[0]); // affiche le premier élément demandé
  return 0;
}
```

Une fonction qui n'a pas l'intention de modifier les éléments d'un tableau devrait le déclarer comme `const`. On aurait donc dû écrire :

```c
int sum(const int entiers[], size_t len) {
  ...
}
```

On peut créer un tableau d'une taille connue seulement à l'exécution du programme :

```c
int main() {
  printf("Combien d'éléments ? ");
  size_t len;
  scanf("%lu", &len);
  double elems[len];
  ...
}
```

Cependant, une fois créé, il ne peut toujours pas changer de taille. Modifier `len` après coup n'y changera rien.

5. Semaine 5 : chaînes de caractères

Le type `char` représente *plus ou moins* un *caractère*. En fait, pour être exact, il représente exactement un *octet*. Cela suffit à représenter les lettres latines non accentuées, les chiffres arabes, et quelques signes de ponctuation des langues occidentales. Les autres « caractères » de la vie réelle ont besoin de plusieurs `char` pour être stockés en C.

Du « texte » est une *chaîne de caractères* (*string* en anglais) que nous stockons dans un *tableau* de `char`. Un caractère supplémentaire spécial, `'\0'`, indique la fin de la chaîne. La notation d'une chaîne avec les guillemets `"..."` ajoute implicitement le `\0` requis.

```c
// les deux lignes suivantes sont équivalentes
char hello1[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
char hello2[] = "hello";
```

Le `\0` final permet à une chaîne de connaître sa longueur, alors que ce n'est en général pas vrai pour les tableaux. La longueur d'une chaîne ne compte pas le `\0` final. `hello1` et `hello2` ont donc bien une longueur de 5, et non de 6.

L'include `<string.h>` contient plusieurs fonctions utiles à la manipulation des chaînes de caractères. Renseignez-vous notamment sur `strlen`, `strncpy` et `strncat`.

Pour lire une chaîne de caractères au clavier, il faut d'abord réserver suffisamment d'espace dans un tableau, puis utiliser `fgets`. On peut imprimer une chaîne de caractères avec le spécificateur de format `%s` dans `printf`.

```c
char name[100];
printf("Quel est votre nom ? ");
fgets(name, 100, stdin);
printf("Bonjour %s !\n", name);
```

Remarquez que le morceau de code ci-dessus affichera

```
Bonjour Sébastien
 !
```

La fin du message « ! » est la ligne suivante ! C'est parce que `fgets` stocke aussi le `\n` correspondant à l'appui sur la touche Enter. Que ferez-vous pour régler ce problème ?

6. Semaine 6 : Pointeurs et tableaux à 2 dimensions

6.1. Mémoire et pointeurs

La mémoire (RAM) de l'ordinateur, mise à disposition pour votre programme, est comme un immense tableau d'octets.

Si votre programme dispose de 8 Go de RAM, le « grand tableau » de sa mémoire possède $8 \cdot 2^{30} = 8\,589\,934\,592$ cases, chacune représentant un octet.

Un octet contient 8 bits. Il peut donc représenter $2^8 = 256$ valeurs différentes. Par exemple, les entiers naturels dans l'intervalle $[0,\ 256[$ ou des entiers relatifs dans l'intervalle $[-128,\ 128[$.

> Le type `char` est d'ailleurs le véritable type d'un *octet* en C. Et c'est donc un entier dans l'intervalle $[-128,\ 128[$. Leur interprétation comme « caractères » comme des lettres ou des chiffres n'est qu'une convention. Pour les positifs, cette convention est régie par le **standard ASCII**.
>
> **Hors cours** : Les négatifs sont plus compliqués ; ils sont utilisés par groupes de 2 à 4 pour encoder les autres code points du **standard Unicode**.

Chaque variable de votre programme est « rangée » dans la mémoire, où elle prend plus ou moins de place en fonction de son type.

Type C	# octets	Vague correspondant en maths	Vraies valeurs possibles
`int`	4	$\mathbb{Z}$	$[-2^{31}, 2^{31}-1]$
`size_t`	8	$\mathbb{N}$	$[0,\ 2^{64}[$
`double`	8	$\mathbb{R}$	Valeurs en « virgule flottante » (approximations)
`bool`	1	$\{\text{FAUX}, \text{VRAI}\}$	$\{\texttt{false}, \texttt{true}\}$
`char`	1		$[-128,\ 128[$

Pour stocker un `int x` dans la mémoire, on aura besoin de *4 cases consécutives*. Par exemple, elle pourrait être rangée dans les cases numérotées 1028–1031. Un `char c` pourrait aussi être rangé dans la case 1032.

…	1026	1027	1028	1029	1030	1031	1032	1033	…
…	?	?	`int x`				`char c`	?	…

Le nombre 1028 est alors *l'adresse* de `x`. On peut l'obtenir avec `&x`. Le type de `&x` est un *pointeur vers `int`*, noté `int *`. Ce n'est pas un simple `size_t`, sinon on oublierait qu'à cet endroit se trouve un `int`.

```
int *adresse_de_x = &x;
```

Fondamentalement, un *pointeur* est donc l'adresse d'une variable, c'est-à-dire le numéro de la (première) case où est stockée cette variable dans le grand tableau de la mémoire.

On peut accéder au contenu qui se cache derrière un pointeur `p` avec l'opérateur de *déréférencement* `*p`, que ce soit pour le lire ou y écrire.

```
int x = 543;
int *p = &x;
printf("*p = %d\n", *p); // 543

*p = 789;
printf("*p = %d\n", *p); // 789
printf("x = %d\n", x); // 789 aussi !
```

De manière générale, pour `v` de type `T`, l'adresse de `v`, notée `&v`, est de type `T*`. En sens inverse, pour `p` de type `T*`, la valeur *pointée par `p`*, notée `*p`, est de type `T`.

Les pointeurs sont des valeurs aussi ! Il faut donc les stocker dans la mémoire ! 🤯 Avec 8 Go de RAM, nous avons besoin d'au moins $\log_2(8 \cdot 2^{30}) = 33$ bits pour donner un numéro à chaque case. Comme en informatique, on aime les puissances de deux, on va monter directement jusqu'à 64 bits, soit 8 octets. La taille d'un pointeur est donc de 8 octets. Et ce, quel que soit le type de la valeur *pointée* !

Type C	# octets	Vague correspondant en maths	Vraies valeurs possibles
`int`	4	$\mathbb{Z}$	$[-2^{31}, 2^{31} - 1]$
`size_t`	8	$\mathbb{N}$	$[0, 2^{64}[$
`double`	8	$\mathbb{R}$	Valeurs en « virgule flottante » (approximations)
`bool`	1	$\{\text{FAUX}, \text{VRAI}\}$	$\{$`false`, `true`$\}$
`char`	1		$[-128, 128[$
`T*`	8		adresses de tous les octets de la mémoire

Les pointeurs sont souvent utilisés pour passer l'adresse d'une variable à une fonction, de sorte que cette puisse *modifier* le contenu de la variable.

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

int main() {
  int x = 5, y = 7;
  swap(&x, &y);
  printf("x = %d, y = %d\n", x, y); // x = 7, y = 5
}
```

C'est précisément cette technique qu'utilise `scanf` ! Vous savez donc maintenant pourquoi il faut écrire `&x` dans un appel à `scanf`.

On appelle souvent cette technique le « passage par référence », par opposition au « passage par valeur ». Cette appellation peut cependant induire en erreur. Les pointeurs sont aussi des valeurs ! En réalité, on passe le pointeur lui-même par valeur.

En C, il n'existe donc que le passage par valeur. D'autres langages, tels que C++ ou Pascal, ont des mécanismes dédiés pour faire du réel passage par référence.

⚠️ L'*associativité* du * dans la définition de variable est étrange. Si vous voulez définir deux pointeurs sur des entiers, il faut écrire

```
int *p, *q;
```

et non

```
int *p, q;
```

La deuxième notation définirait `q` comme un `int`, et non comme un `int*`.

C'est pour se rappeler cette étrangeté qu'on place l'espace *devant* *. On évitera de définir des pointeurs et non-pointeurs en même temps, pour préserver nos cellules grises.

Notez que dans les *paramètres* d'une fonction, il faut de toutes façons répéter le type à chaque fois, comme `...(int *p, int *q)`. Il n'y a donc pas d'ambiguïté à cet endroit.

6.2. Tableaux à 2 dimensions

En mathématique, une matrice A de taille $m \times n$ est un tableau à *2 dimensions*. Ses éléments sont identifiés au moyens de 2 indices : A_{ij}, où $i \in [0,\, m[$ et $j \in [0,\, n[$ (oui, en maths, les indices commenceraient à 1 ; c'est une matrice qui s'est adaptée au monde informatique avec des indices commençant à 0). Comment représente-t-on cela en C ?

Une manière de faire est « d'étaler » tous les éléments sur une seule ligne. Par exemple, pour la matrice de taille 3×4

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 7 & 8 & 9 & 10 \\ 65 & 66 & 67 & 68 \end{pmatrix}$$

on peut écrire toutes les lignes à suite, comme

$$(1\ \ 2\ \ 3\ \ 4\ \ 7\ \ 8\ \ 9\ \ 10\ \ 65\ \ 66\ \ 67\ \ 68)$$

On peut stocker ces éléments dans un tableau de $3 \cdot 4 = 12$ éléments :

```
int matrice[12] = { 1, 2, 3, 4, 7, 8, 9, 10, 65, 66, 67, 68 };
```

Mais alors où se trouve l'élément $A_{2,3} = 67$? À partir du début du tableau, il faut « sauter » 2 lignes de 4 cases chacune, puis encore compter 3 cases. Il est donc à l'indice `idx` $= 2 \cdot 4 + 3 = 11$.

De manière générale, l'élément de matrice A_{ij} se trouve à l'indice `idx` $= i \cdot n + j$, où n est le nombre de *colonnes* de la matrice.

Hors cours : C permet en fait de faire des tableaux à plusieurs dimensions avec réellement plusieurs indices. Cependant, cela est limité à des tableaux dont la taille est entièrement connue quand on compile, y compris pour les paramètres des fonctions. Leur usage est donc assez limité, et nous ne les présenterons pas dans ce cours.

7. Semaine 7 : Structures

Jusqu'à présent, nous avons vu deux grandes catégories de types de données.

D'une part, les types *simples* ou *scalaires* (ou sens mathématique des termes). Dans une variable d'un tel type, on stocke *une* valeur. On y trouve par exemple `int`, `double`, `boolean` ou `char`. En font partie aussi les types pointeur, comme `int *`, puisqu'essentiellement on y stocke une *adresse*, qui est un entier naturel.

D'autre part, les types *tableaux*. Un tableau aggrège un nombre arbitrairement grand de valeurs. Le nombre de valeurs est d'ailleurs souvent connu seulement à l'exécution du programme, par exemple après avoir été lu sur l'entrée standard. En revanche, tous les éléments du tableau sont d'un même type T. On dit que c'est une collection *homogène* à taille *variable*.

Les *structures* remplissent un autre rôle d'aggrégation : c'est un aggrégat *hétérogène* à taille *fixe*. Une structure possède un nombre fixé d'éléments, chacun pouvant être d'un type différent.

Par exemple, on peut vouloir stocker diverses informations sur un personnage dans un jeu vidéo : son nom (une chaîne de caractères), son niveau (un entier), si il a déjà trouvé un bouclier (un booléen), etc.

En C, canoniquement, on définit une structure séparément de son type :

```
struct player {
  char nom[100];
  int niveau;
  bool a_bouclier;
}
```

On dit que `nom`, `niveau` et `a_bouclier` sont les *champs* de `player` (*fields* en anglais). On peut ensuite déclarer des variables de type `struct player`, et accéder à leurs champs avec la syntaxe `struct_var.nom_champ`.

```
struct player mon_player;
mon_player.niveau = 1;
int n = mon_player.niveau;
```

Il est cependant très fastidieux d'écrire `struct player` à chaque fois. On définit donc généralement un *alias de type* associé :

```
typedef struct player player_t;
```

qui permet d'utiliser `player_t` à la place de `struct player`. On veut faire cela tellement souvent que C nous permet de faire les deux choses en une fois :

```
typedef struct player {
  char nom[100];
  int niveau;
  bool a_bouclier;
} player_t;
```

Les alias de types ne sont pas réservés aux structures. Comme son nom l'indique, un alias peut faire référence à n'importe quel autre type. Vous pourriez définir

```
typedef int niveau_t;
```

et utiliser `niveau_t` pour le type d'un niveau. C'est parfois utile pour communiquer un sens plus précis aux données. La syntaxe pour définir un alias vers un type tableau est de mettre les crochets après le nom de l'alias :

```
typedef char nom_t[];
```

Pour se rappeler : c'est toujours la même syntaxe que pour définir une *variable*, mais on remplace le nom de la variable par le nom de l'alias et on ajoute `typedef` devant.

On initialise une variable de type structure comme on initialise un tableau. Soit directement avec `{}` :

```
player_t mon_player = { "Link", 1, false };
```

soit on assignant ses champs un à un :

```
player_t mon_player;
strncpy(mon_player.nom, "Link", 100);
mon_player.niveau = 1;
mon_player.a_bouclier = false;
```

On peut assigner une structure entière avec `=` :

```
player_t mon_player = ...;
player_t autre_player = mon_player;
```

Cela recopie tous les champs de la structures.

Rappelez-vous qu'en C, tout transfert de paramètre se fait *par valeur*. Si l'on donne une variable structure en argument à une fonction, on obtient aussi une copie de la structure. Pour les scalaires, cette copie est gratuite ; et pour les tableaux, elle n'est tout bonnement pas autorisée. Mais pour les structures ? Cette copie a souvent un coût plus élevé que nécessaire.

On voudra donc souvent transmettre les structures aux fonctions *via* un pointeur, pour des questions de performances. Cependant cela donne un pouvoir trop grand à la fonction, qui peut maintenant modifier la structure ! On compense cela en ajoutant `const`, comme avec les tableaux.

Donc, *dans la grande majorité des cas*, les structures seront transmises aux fonctions de la manière suivante :

```
void show_player(const player_t *p) {
  printf("%s, niveau %d, bouclier=%d\n", (*p).nom, (*p).niveau, (*p).a_bouclier);
}
...
show_player(&mon_player);
```

Remarquez que le `.` est plus fort que `*`, et qu'il faut donc écrire `(*p).nom` pour d'abord suivre le pointeur avant de lire le champ `nom`. C'est tellement courant et tellement ennuyeux que C nous offre l'opérateur « flèche » `->` exprès pour ça : `ptr_struct->nom_champ` est équivalent, par définition, à `(*ptr_struct).nom_champ`. On écrira donc plutôt

```
void show_player(const player_t *p) {
  printf("%s, niveau %d, bouclier=%d\n", p->nom, p->niveau, p->a_bouclier);
}
```

Remarque finale : `player_t` est un type comme les autres. On peut créer un tableau de `player_t`, ou mettre `player_t` dans une autre structure, etc. Si on devait ajouter la position dans le plan à `player`, on définirait certainement une `struct point` avec un alias `point_t`, et nous aurions un champ `point_t position;` dans `struct player`.

8. Semaine 8 : Allocation dynamique

En semaine 6, nous avons découvert les pointeurs. On peut y stocker l'adresse d'une variable, par exemple pour la donner à une fonction qui voudrait la modifier (comme dans `scanf`).

Les variables sont cependant toujours « créées », jusqu'à présent, comme des variables locales de fonctions. Même les paramètres d'une fonction sont essentiellement des variables locales à cette fonction (qui reçoivent leur *valeur* depuis l'extérieur).

L'*allocation dynamique de mémoire* nous permet d'échapper à ces contraintes. On va pouvoir « créer » dynamiquement de l'espace mémoire en dehors des fonctions. Bien sûr, la mémoire totale est fixe (elle dépend du hardware de votre ordinateur), donc on ne *crée* pas vraiment de mémoire.

On dit qu'on *alloue* de la mémoire lorsqu'on réserve un petit bout de la mémoire totale avec la fonction `malloc` (**m**emory **alloc**ation). On reçoit alors un pointeur vers un bout de mémoire distinct de toutes les variables locales, et de tous les autres bouts de mémoire déjà renvoyés par `malloc`. On peut alors utiliser ce pointeur pour manipuler ce bout de mémoire.

Il faut donner en paramètre à la `malloc` la taille du morceau de mémoire que l'on veut, exprimé en octets. Comme on veut en principe y stocker un certain type de données, on utilisera souvent `sizeof(T)` pour que C calcule la bonne taille. Par exemple, `sizeof(int)`.

> Bien que `sizeof(T)` ressemble à un appel de fonction, ce n'est **pas** une fonction ; c'est magique ! Remarquez que l'« argument » de `sizeof` est un *type*, et non pas une *expression*. On n'écrit pas `sizeof(5)`, mais bien `sizeof(int)`.

Puisque la quantité de mémoire totale est limitée, il faut *libérer* le bout de mémoire lorsqu'on n'en a plus besoin. Sinon, on finira par épuiser la mémoire disponible (et votre ordinateur deviendra tellement lent que vous n'aurez d'autre choix que l'éteindre brutalement). On libère un bout de mémoire avec la fonction `free`. Une fois libéré, ce bout de mémoire redevient disponible pour un futur appel à `malloc`.

`malloc` et `free` requièrent un `#include <stdlib.h>`.

```
int *px = (int *) malloc(sizeof(int)); // allouer un bloc de la taille d'un int
*px = 5;
printf("*px = %d\n", *px);
free(px); // libérer le bloc, pour ne pas épuiser la mémoire
// On n'a plus le droit d'utiliser *px ici !
```

Mais pourquoi faire ça ? Ne serait-il pas plus simple de déclarer `int x;` puis `int *px = &x;` ? *Si, absolument.* Mais il y a certaines situations où ce n'est pas possible. Nous en donnons deux pour l'instant.

La première, c'est si vous voulez définir une fonction qui crée un tableau, puis *renvoie* ce tableau en résultat. Puisque tableau = pointeur, cela veut dire qu'il faut renvoyer un pointeur. Mais si on a créé le tableau normalement, comme variable locale, le contenu est détruit quand la fonction se termine :

```
int *make_squares(size_t n) {
  int squares[n];
  for (size_t i = 0; i < n; i++) {
    squares[i] = i * i;
  }
  return squares;
}

int main() {
  int *squares = make_squares(5);
  printf("%d\n", squares[3]); // catastrophe, le contenu a été détruit !

  return 0;
}
```

D'ailleurs, `gcc` nous avait prévenu (il est fort !) :

```
$ gcc -Wall sandbox.c -o sandbox
sandbox.c: In function ‘make_squares’:
sandbox.c:9:10: warning: function returns address of local variable [-Wreturn-local-
addr]
    9 |   return squares;
      |          ^~~~~~~
$ ./sandbox
Segmentation fault (core dumped)
```

Puisque les variables locales de la fonction seront détruites quand elle se termine, nous n'avons pas le choix : il faut allouer de la mémoire en dehors des variables locales :

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

int *make_squares(size_t n) {
  int *squares = (int *) malloc(n * sizeof(int));
  for (size_t i = 0; i < n; i++) {
    squares[i] = i * i;
  }
  return squares;
}

int main() {
  int *squares = make_squares(5);
  printf("%d\n", squares[3]); // OK, la mémoire est toujours là
  free(squares); // par contre il faut absolument la libérer après !

  return 0;
}
```

La seconde, c'est si vous voulez définir une structure qui contient un tableau dont la taille n'est connue qu'à l'exécution du programme. Par exemple, on pourrait définir une `struct train` représentant un train de voyageurs. On veut stocker le nombre de personnes dans chaque voiture du train. Comme tous les trains n'ont pas la même longueur, on ne peut pas faire ceci :

```c
typedef struct train {
  size_t carriage_count;
  int persons_by_car[???]; // que met-on ici ?
} train_t;
```

On va alors plutôt définir `persons_by_car` comme un `int *`, et on va dynamiquement allouer la mémoire nécessaire.

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

typedef struct train {
  size_t carriage_count;
  int *persons_by_car;
} train_t;

train_t create_train(size_t carriage_count) {
  train_t train;
  train.carriage_count = carriage_count;
  train.persons_by_car = (int *) malloc(carriage_count * sizeof(int));
```

```c
  // Ne pas oublier d'initialiser le contenu du tableau !
  for (size_t i = 0; i < carriage_count; i++) {
    train.persons_by_car[i] = 0;
  }

  return train;
}

void free_train(const train_t *train) {
  free(train->persons_by_car);
}

int main() {
  train_t t1 = create_train(5);
  t1.persons_by_car[3] += 10;
  printf("%d\n", t1.persons_by_car[3]);
  free_train(&t1);

  return 0;
}
```

Notez qu'on a défini une fonction `free_train` pour préserver une certaine symétrie : `create_train` alloue un nouveau `train_t` ; `free_train` fait ce qu'il faut pour libérer ce que `create_train` a alloué. On veillera à garder ce type de symétrie, pour ne pas faire des `free` en trop ou trop peu.

9. Semaine 9 : Tests

« Un programme non testé est un programme qui ne fonctionne pas. », dit-on. Certes, nous avons déjà testé nos programmes en les lançant à la main, me direz-vous. En réalité, cela n'est pas considéré comme « tester ».

Nous voyons ici deux formes de *tests automatiques*, que l'on peut relancer à souhait, sans avoir à faire de manipulations.

On s'attendra à ce que tous vos programmes soient testés automatiquement, en particulier votre projet.

9.1. Tests de « désinfection »

Les tests de « désinfection » (*sanitizing test*) en anglais sont un moyen automatique de vérifier que vos programmes évitent les pires problèmes. Il s'agit des problèmes de mauvaises manipulation de la mémoire, ou de *comportement indéfini*.

Voici quelques problèmes majeurs que l'on peut détecter :

- Faire une opération arithmétique qui déborde de la capacité d'un `int` (*overflow*). Par exemple, additionner 2 000 000 000 avec lui-même devrait donner 4 millions, mais cela n'est pas représentable dans un `init`.
- Tenter d'utiliser un pointeur après l'avoir libéré avec `free`.
- Dépasser des bornes d'un tableau.
- Utiliser une variable sans l'avoir initialisée.

Toutes ces erreurs sont graves en C, car elles donnent lieu à de l'*undefined behavior*. Votre programme est susceptible de faire *n'importe quoi* quand elles arrivent. Pas seulement calculer un résultat aberrant ou déclencer une « Segmentation fault ». Il peut réellement se passer n'importe quoi.

Votre système d'exploitation empêchera votre programme de déraper tellement qu'il brûle votre ordinateur, bien sûr. Mais en théorie, il pourrait quand même effacer tous les fichiers de votre ordinateur !

En pratique, les bugs de cette sorte représentent 40% **des failles de sécurité**. Il est donc impératif de s'en protéger.

Vous pouvez activer les tests de désinfection en ajouter `-fsanitize=undefined -fsanitize=address` (et pourquoi pas `-O0 -g` en plus) aux options que vous donnez à `gcc`. Par exemple :

```
$ gcc -Wall -fsanitize=undefined -fsanitize=address -O0 -g foo.c -o foo
```

Lors de l'exécution de votre programme, cela affichera des messages d'erreur alarmants si votre programme fait des bêtises.

Par exemple, le programme suivant cause un overflow sur des entiers :

```
int times2(int x) {
  return 2 * x;
}

int main() {
  int a = 2000000000;
  printf("%d\n", times2(a));
  return 0;
}
```

Si on le compile avec les options ci-dessus, on ne reçoit pas de message du compilateur pendant la compilation. En revanche, à l'exécution du programme, nous avons :

```
$ ./foo
foo.c:4:12: runtime error: signed integer overflow: 2000000000 * 2 cannot be
represented in type 'int'
-294967296
```

Réessayons sans les options de « sanitize », et nous observons :

```
$ ./foo
-294967296
```

Le compilateur **clang** propose de meilleurs tests. En particulier, son option `-fsanitize=memory` est capable de détecter si vous utilisez de la mémoire que vous n'avez pas initialisée. Que ce soit une variable locale, ou un bout de mémoire renvoyé par `malloc` !

9.2. Tests unitaires

Les tests unitaires sont une discipline de développement de programme. Il faut les écrire spécifiquement pour tester la logique de vos programmes ; on ne les reçoit pas gratuitement. Mais cela en vaut la peine.

Par exemple, supposons que vous ayez écrit une fonction

```
int factorial(int x) {
  ...
}
```

Bien sûr, vous pouvez l'appeler depuis `main`, tester quelques valeurs, et vérifier qu'elle tient la route. Mais ensuite, si vous la modifiez (pour corriger un bug ou la rendre plus rapide), vous devrez tout refaire !

Pour gagner du temps, on écrit des tests unitaires. On définit une fonction `test_factorial()` qui appelle `factorial` avec des valeurs connues, et vérifie qu'on obtient le bon résultat. On utilise la

fonction `assert(...)` (dans l'include `<assert.h>`) pour crasher intentionellment le programme si la condition est fausse.

```
void test_factorial() {
  assert(factorial(0) == 1);
  assert(factorial(1) == 1);
  assert(factorial(3) == 6);
  assert(factorial(5) == 120);
}
```

On n'oublie pas d'appeler ces fonction au début de notre `main`, avant le déroulement normal du programme. Comme vous aurez plusieurs fonctions à tester, une bonne pratique sera de définir une fonction `test_all` qui les invoque toutes.

```
void test_all() {
  test_factorial();
  // ... appeler toutes les autres test_*()
}

int main() {
  test_all();

  // ... programme normal

  return 0;
}
```

Si `factorial` contient un bug et que nos tests le détectent, on obtiendra un message comme le suivant :

```
$ ./foo
foo: foo.c:35: test_factorial: Assertion `factorial(3) == 6' failed.
```

Le gros avantage des tests unitaires est qu'ils sont tous retestés à chaque fois. Si on introduit un bug en voulant améliorer une partie de notre code qu'on avait déjà écrite, on le saura tout de suite !

Les tests unitaires font partie intégrante d'un programme. Ils doivent être conservés même quand le programme est « fini ». En particulier, **vous devez rendre vos tests dans votre projet**. Ils seront évalués.

10. Semaine 10 : Liste chaînée

10.1. Pointeur `NULL`

La valeur `NULL` est une valeur de type pointeur (n'importe quel `T*`). Elle représente le pointeur nul, qui ne pointe vers aucune mémoire valide.

Elle sert souvent à marquer « pas de pointeur ici ». On peu tester si un pointeur est nul avec `p == NULL` ou `p != NULL`.

Il est interdit (comportement indéfini) de déréférencer un pointeur nul, donc de faire `*p` ou `p->foo` si `p == NULL`.

10.2. Structures récursives

Une structure `struct foo` ne peut pas contenir de champ de type `struct foo`. Une telle structure aurait une taille infinie. En revanche, elle peut contenir des champs de type `struct foo *`, c'est-à-dire des pointeurs vers d'autres `foo`.

```c
typedef struct foo {
  struct foo *other;
} foo_t;
```

Notez qu'on ne peut pas utiliser `foo_t *other`, car à l'intérieur de la définition de `foo`, on ne « sait » pas encore ce qu'est `foo_t`.

Ces structures récursives génèrent des « chaînes » de `foo_t` liées les unes aux autres. Souvent, ces chaînes se « finissent » avec un champ `other` qui vaut `NULL`.

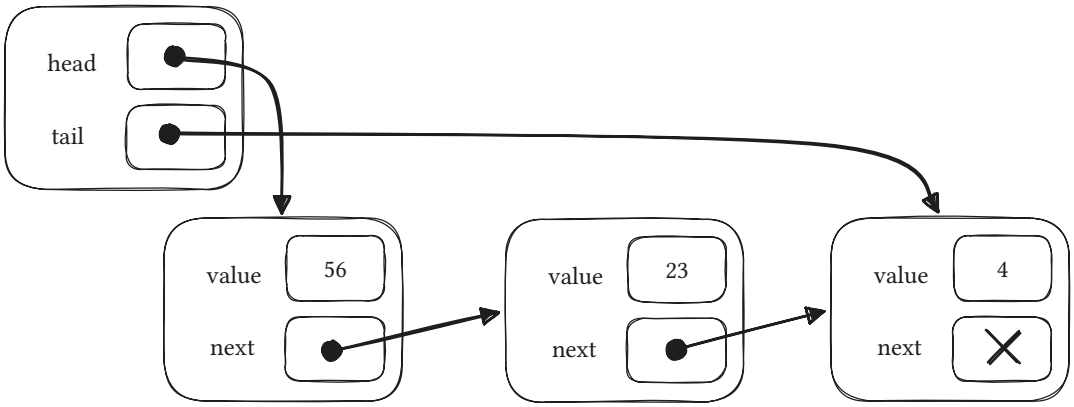
10.3. Listes chaînées

Une liste chaînée est représentée par les structures suivantes :

```c
typedef struct cell {
  int value;
  struct cell *next;
} cell_t;

typedef struct intlist {
  cell_t *head;
  cell_t *last;
} intlist_t;
```

Les différentes cellules sont liées entre elles *via* leur champ `next`. Le champ `next` de la dernière cellule d'une liste vaut `NULL`.



Le champ `head` de `intlist_t` est le plus important. Il pointe vers la première cellule. Sans lui, on perdrait toute référence aux cellules, et donc à tout le contenu de la liste.

Le champ `last` permet d'ajouter efficament (en $\Theta(1)$) un élément à la fin d'une liste.

Toutes les opérations sur une liste doivent bien entendu maintenir les relations entre les différents pointeurs. Elles doivent aussi souvent allouer une nouvelle `cell_t` avec `malloc`, ou en détruire une avec `free`.

11. Semaine 11 : Divers

Cette semaine, il n'y a pas vraiment de sujet « cohérent ». On vous présente une série de diverses petites choses que l'on peut faire en C, et dont on n'a pas encore parlé.

11.1. Prototypes de fonctions

Si on essaye d'appeler une fonction qui est définie *plus loin* dans le programme, on se heurte à une erreur :

```c
int main() {
  int x = foo(5, true);
  printf("%d\n", x);
}

int foo(int a, bool b) {
  if (b) {
    return a + 1;
  } else {
    return a - 1;
  }
}
```

```
$ gcc -Wall sandbox.c -o sandbox
sandbox.c: In function ‘main’:
sandbox.c:7:11: error: implicit declaration of function ‘foo’ [-Wimplicit-function-
declaration]
    7 |   int x = foo(5, true);
      |           ^~~
sandbox.c: At top level:
sandbox.c:11:5: error: conflicting types for ‘foo’; have ‘int(int,  _Bool)’
   11 | int foo(int a, bool b) {
      |     ^~~
sandbox.c:11:1: note: an argument type that has a default promotion cannot match an
empty parameter name list declaration
   11 | int foo(int a, bool b) {
      | ^~~
sandbox.c:7:11: note: previous implicit declaration of ‘foo’ with type ‘int()’
    7 |   int x = foo(5, true);
      |           ^~~
```

On peut éviter ce problème en ajoutant la *déclaration* de la fonction avant de l'utiliser, sans sa *définition*. On utilise pour cela son *prototype* : toute la première ligne, mais pas le corps de la fonction :

```c
int foo(int a, bool b);

int main() {
  int x = foo(5, true);
  printf("%d\n", x);
}

int foo(int a, bool b) {
  ...
}
```

Maintenant, on peut compiler et exécuter le problème sans erreur.

11.2. Opérateur ternaire `a ? b : c`

Pas matière d'examen (mais parfois utile quand même)

Il arrive souvent qu'on fasse un `if/else` dont les deux branches sont très courtes. Elles calculent toutes les deux une seule expression, qu'elles mettent dans la même variable, ou retournent avec `return`. Par exemple :

```c
int r;
if (x > 10) {
  r = a + 1;
} else {
```

```
  r = a + 2;
}
```

Cela prend beaucoup de place. En plus, ça nous force à déclarer `int r` sans lui donner de valeur initiale, ou alors en lui donnant une valeur initiale qui ne sert à rien.

À la place, on peut utiliser l'opérateur ternaire `_ ? _ : _`.

```
int r = (x > 10) ? (a + 1) : (a + 2);
```

Cet opérateur se lit « si `x > 10` alors `a + 1` sinon `a + 2` ». Comme un `if/else`, la branche qui n'est pas choisie n'est pas évaluée.

Autre exemple, la fonction `foo` ci-dessus peut s'écrire :

```
int foo(int a, bool b) {
  return b ? (a + 1) : (a - 1);
}
```

On peut utiliser cet opérateur partout où une expression est attendue. Par exemple, en argument d'une fonction, ou comme opérande d'un autre opérateur. On veillera à mettre explicitement des parenthèses pour lever le moindre doute quant à l'associativité.

11.3. Arguments du programme

Pas matière d'examen

Vous l'avez vu avec le mini-projet, on peut « appeler » un programme en lui donnant des « arguments », un peu comme une fonction :

```
$ ./programme argument1 "long argument 2" 3 encore-un-argument
```

Le programme peut récupérer ces arguments, sous forme de tableau de chaînes de caractères, avec une signature alternative de `main`.

```
int main(int argc, char *argv[]) {
  // Affiche tous les arguments
  for (size_t i = 0; i < argc; i++) {
    printf("%lu: '%s'\n", i, argv[i]);
  }
}
```

`argv` est un tableau de chaînes de caractères. `argc` est la *taille* de ce tableau. Pour des raisons historiques, il doit être un `int` ; même si `size_t` serait clairement plus approprié.

Lancé avec la ligne de commande ci-dessous, ce programme affiche :

```
0: './programme'
1: 'argument1'
2: 'long argument 2'
3: '3'
4: 'encore-un-argument'
```

`argv[0]` est toujours le nom du programme, tel qu'on l'a écrit dans la ligne de commande. On devra souvent l'ignorer. Les réels arguments commencent à l'indice 1.

Si on veut interpréter un argument comme un nombre, on peut utiliser la fonction `strtol` ou `strtod` de C. On vous laissera regarder leur documentation.

11.4. Énumérations

Il arrive très souvent qu'on veuille représenter une donnée qui est membre d'un petit ensemble fini de possibilités. Par exemple, les couleurs (*suits*) des cartes à jouer : cœur, carreau, trèfle et pique.

Bien sûr, on pourrait utiliser un `int` (même avec un `typedef`) assorti de quelques constantes :

```
typedef int suit_t;

const suit_t HEARTs = 1;
const suit_t DIAMONDS = 2;
const suit_t CLUBS = 3;
const suit_t SPADES = 4;
```

Ce motif arrive tellement souvent que C propose les *énumérations* pour ce schéma de données. Plutôt que ce qu'on a écrit ci-dessus, on écrira :

```
typedef enum suit {
  HEARTS,
  DIAMONDS,
  CLUBS,
  SPADES,
} suit_t;
```

Cela a l'avantage de mieux communiquer sur le fait q'une `suit_t` n'est pas n'importe quel entier. C'est l'une des quatre couleurs possibles, et rien d'autre.

> Avez-vous remarqué que `bool` peut être vu comme un `enum` avec deux valeurs : `false` et `true` ?

11.5. Switch

Les `enum` amènent au `switch`. Très souvent, on devra faire des tests en série pour gérer chaque cas possible d'un `enum`. Par exemple, si on veut écrire une fonction qui affiche une `suit_t` à l'écran, elle ressemblera probablement à ceci :

```
void show_suit(suit_t s) {
  if (s == HEARTS) {
    printf("cœur\n");
  } else if (s == DIAMONDS) {
    printf("carreau\n");
  } else if (s == CLUBS) {
    printf("trèfle\n");
  } else if (s == SPADES) {
    printf("pique\n");
  } else {
    assert(false && "huh, what did you do?");
  }
}
```

Ce n'est pas très lisible. En plus, on se sent obligé-e de mettre ce `else` inutile à la fin. Avec un `switch`, on écrira :

```
void show_suit(suit_t s) {
  switch (s) {
    case HEARTS:
      printf("cœur\n");
      break;
    case DIAMONDS:
      printf("carreau\n");
```

```
      break;
    case CLUBS:
      printf("trèfle\n");
      break;
    case SPADES:
      printf("pique\n");
      break;
  }
}
```

Si on n'a qu'une partie des cas à traiter, et que « tous les autres » se traitent de la même manière, on peut utiliser `default:` pour « tous les autres ».

```
void show_suit(suit_t s) {
  switch (s) {
    case HEARTS:
      printf("cœur\n");
      break;
    case DIAMONDS:
      printf("carreau\n");
      break;
    default:
      printf("les autres\n");
  }
}
```

On peut rassembler plusieurs cas qui doivent être traités de la même manière en enchaînant plusieurs `case`.

```
void show_suit(suit_t s) {
  switch (s) {
    case HEARTS:
    case DIAMONDS:
      printf("rouge\n");
      break;
    case CLUBS:
    case SPADES:
      printf("noir\n");
      break;
  }
}
```

Le `break` est nécessaire pour que l'exécution ne continue pas au `case` suivant. Pour des raisons historiques, en C, l'exécution saute au `case` qui correspond, puis *continue jusqu'au bout du `switch`*. C'est une mauvaise idée en 2026 ; les nouveaux langages ne font plus ça. Du coup, en C, on doit écrire `break` à la fin de chaque `case` pour « sortir du `switch` ». Par convention, on n'écrit pas `break` dans le `default`, puisqu'il est toujours à la fin.

On peut utiliser un `switch` avec n'importe quel type d'`enum`, mais aussi avec n'importe quel type entier (`int`, `size_t`, etc.). Par exemple, on pourrait écrire Fibonacci comme suit (si on veut subir un $\Theta(2^n)$ pour rien, en tous cas) :

```
int fibonacci(int n) {
  switch (n) {
    case 0:
      return 0;
    case 1:
      return 1;
```

```
    default:
      return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```

Pourquoi n'a-t-on pas mis de `break` ici ?

11.6. Opérateurs bit-à-bit (*bitwise*)

Pas matière d'examen

Vous savez désormais que les entiers sont représentés par une suite de bits. Les processeurs sont dotés d'instructions très efficaces pour manipuler tous les bits d'un entier d'un coup. Bien sûr, souvent on les exploite pour faire des opérations arithmétiques (addition, multiplication, comparaisons, etc.).

Il peut arriver qu'on souhaite exploiter les bits eux-mêmes, un peu comme un « mini-tableau » de bits (valeurs 0 ou 1). Si on fait ça, on aura intérêt à utiliser explicitement un `unsigned long` (32 bits non signés) ou `unsigned long long` (64 bits non signés).

On aura aussi tendance à écrire les constantes sous forme hexadécimale. Par exemple, `0x1af` pour représenter $431 = 1 \cdot 256 + 10 \cdot 16 + 15 \cdot 1$. Chaque chiffre hexadécimal (de `0` à `f`) correspond exactement à 4 bits.

Les opérateurs `&`, `|` et `^` font des « et », « ou » et « ou exclusif » bit-à-bit. Par exemple, 1001_b `&` $1100_b = 1000_b$ (où le suffix i_b indique la base 2). Le premier bit du résultat est 1 car les premiers bits des deux opérandes sont 1. Le second bit est 0 car le second bit du premier opérande est 0 et celui du deuxième opérande est 1. Etc. On peut mieux le voir sous forme de tableau :

	1	0	0	1
	1	1	0	0
&	1	0	0	0

L'opérateur unaire `~a` inverse tous les bits a `a`.

Les opérateurs `a << n` et `a >> n` décalent les bits de `a` de n cases vers la gauche/droite. Les bits qui sortent « tombent ». On remplit les trous avec des 0. Par exemple, si on a un entier 8 bits `a` qui vaut 10011010_b, alors `a << 2` vaut 01101000_b et `a >> 3` vaut 00010011_b.

En supposant que tous les bits qui « tombent » sont des 0, quelle est l'interprétation mathématique de `a << 3` ?

Étant donnés deux `unsigned long (long) a` et `b`, un un `int n`, on a :

Opération	Significantion	Exemple
`a & b`	« et » bit-à-bit	1001_b `&` $1100_b = 1000_b$
`a | b`	« ou » bit-à-bit	1001_b `|` $1100_b = 1101_b$
`a ^ b`	« ou exclusif » bit-à-bit	1001_b `^` $1100_b = 0101_b$
`~a`	« non binaire »	`~`$1001_b = 0110_b$
`a << n`	décalage à gauche	10011010_b `<<` $2 = 01101000_b$
`a >> n`	décalage à droite	10011010_b `>>` $3 = 00010011_b$

Une application typique des manipulations de bits est la représentation efficace d'un *ensemble* de petits entiers. Si vous devez représenter un ensemble d'entiers compris dans l'intervalle $[0, 40]$, par exemple, vous pouvez le faire avec 41 bits, donc un `unsigned long long`. L'élément i appartient à l'ensemble représenté par x si le bit i de x vaut 1. Avec cette représentation, on peut faire les calculs efficaces suivants :

Mathématiques	Opérations bit-à-bit
$x \cup y$	`x | y`
$x \cap y$	`x & y`
$x \setminus y$	`x & ~y`
$\{i\}$	`1 << i`
donc, $x \cup \{i\}$	`x | (1 << i)`
test $i \in x$, équiv. $(x \cap \{i\}) \neq \emptyset$	`(x & (1 << i)) != 0`

Vous trouverez souvent ce type d'encodage pour représenter des « flags » : des ensembles d'options booléennes, avec un nombre fini et petit d'options possibles. Il est donc bon de connaître cette technique afin de ne pas être totalement perdu-e quand vous tomberez dessus.

12. Semaine 11 : fichiers

En très très résumé cette semaine. J'améliorerai ça plus tard.

12.1. Fichiers texte

Ouvrir un fichier en lecture :

```
FILE *f = fopen(path, "rb");
```

Vérifier que le résultat n'est pas `NULL`. Si c'est `NULL`, on n'a pas pu ouvrir le fichier (il n'existe pas, ou on n'a pas le droit de le dire, par exemple) :

```
if (f == NULL) {
  printf("Cannot open file\n");
  return; // par exemple
}
```

Ne jamais oublier de fermer le fichier quand on en a fini avec lui :

```
fclose(f);
```

Si le fichier est un fichier texte, on peut le lire comme comme on a appris à lire au clavier. Utiliser `fscanf` à la place de `scanf` :

```
int x;
fscanf(f, "%d", &x);
```

Pour `fgets`, c'est *déjà* la « version `f` ». Remplacer `stdin` par le pointeur de fichier :

```
char str[100];
fgets(str, 100, f);
```

La fonction `feof` indique si on est à la fin du fichier :

```
while (!feof(f)) {
  // do something
}
```

Pour écrire, ouvrir le fichier en écriture avec

```c
FILE *f = fopen(path, "wb");
```

puis utiliser `fprintf` à la place de `printf`. Ne pas oublier de tester si on a pu ouvrir le fichier, et de le fermer à la fin.

12.2. Fichiers binaires

Pour lire/écrire un fichier binaire, il faut absolument bien définir son *format*. Dans ce cours, on le définit avec la notation suivante. Ce n'est pas un schéma universel. Cependant, il suit les règles d'une **grammaire BNF**

```
content ::= int32 x:int64 uint32[4] string
```

représente un fichier dans lequel sont enchaînés : un entier signé 32 bits, un entier signé 64 bits, 4 entiers non signés 32 bits, et une chaîne de caractères. On a nommé le second entier `x`.

Une `string` est définie comme

```
string ::= len:uint32 char[len]
```

c'est-à-dire un entier 32 bits non signé `len` qui représente sa longueur, suivi de `len` caractères.

On ouvre et on ferme le fichier comme ci-dessus. Par contre, on n'utilise pas `fscanf`/`fgets`/`fprintf`.

On peut lire/écrire un tableau de `char` de taille connue avec

```c
fread(tableau, 1, n, f);
fwrite(tableau, 1, n, f);
```

où n est le nombre d'éléments du tableau.

On construit le reste par-dessus ces fonctions. Le fichier `binary-file-utils.c` sur Moodle vous donne des primitives pour lire et écrire les types de données suivants :

- `int8`, `int16`, `int32`, `int64` : entiers signés.
- `uint8`, `uint16`, `uint32`, `uint64` : entiers non signés.
- `double` : `double` au format binaire `binary64` (celui que vous avez vu en Théorie).

On évitera d'utiliser `fread` et `fwrite` avec d'autres types que `char` et `unsigned char`.

Ah oui, j'allais oublier : quelques nouveaux types C utiles :

- `unsigned char` : un entier 8 bits non signé
- `unsigned int` : un entier 32 bits non signé
- `long long` : un entier 64 bits signé
- `unsigned long long` : un entier 64 bits non signé

Sauf qu'on vous a un peu menti depuis le début. Les types entiers en C ne garantissent pas vraiment une taille bien spécifique. À part `char` et `unsigned char`, qui sont garantis avoir 1 octet.

Lorsqu'on veut manipuler des entiers de taille vraiment spécifique, on utilisera les types suivants, définis dans `<stdint.h>` :

- `int8_t`, `int16_t`, `int32_t`, `int64_t` : entiers signés.
- `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` : entiers non signés.

13. Semaine 13 : récursivité

Il n'y a pas grand chose à dire sur la récursivité en C. Une fonction peut s'appeler elle-même. On peut donc appliquer toutes les notions théoriques directement.

On s'intéresse cependant aux *structures récursives* : des `struct` dont certains champs sont des pointeurs vers la même `struct`.

Nous avons déjà vu une forme de structure récrusive : la liste chaînée. Une autre structure intéressante est *l'arbre binaire*. Un arbre binaire est soit vide (ou une *feuille*), soit une branche avec des sous-arbres gauche et droit.

Une forme particulière d'arbre binaire est *l'arbre binaire de recherche*, ou *binary search tree*. On définit sa structure mathématique comme suit :

$$T ::= \texttt{NULL} \mid (T, v, T)$$

c'est-à-dire : un arbre est soit l'arbre vide `NULL`, soit un triplet (T, v, T) avec une valeur et deux sous-arbres.

En C, cela donne la structure

```
typedef struct btree {
  struct btree *left;
  int value;
  struct btree *right;
} btree_t;
```

où un sous-arbre est un `btree_t*`. Un `btree_t *t` est vide si et seulement si `t == NULL`.

Pour pouvoir être qualifié d'arbre binaire *de recherche*, un tel arbre binaire doit avoir les propriétés suivantes :

- toutes les valeurs dans le sous-arbre de gauche sont $\leq v$;
- toutes les valeurs dans le sous-arbre de droite sont $\geq v$.

On peut le définir encore plus formellement si on définir la fonction $S(t)$ qui est l'ensemble des valeurs dans un sous-arbre :

$$S(t) = \begin{cases} \emptyset & \text{si } t = \texttt{NULL} \\ S(l) \cup \{v\} \cup S(r) & \text{si } t = (l, v, r) \end{cases}$$

Les propriétés d'un arbre binaire de recherche s'écrivent alors mathématiquement comme : pour un sous-arbre $t = (l, v, r)$, on a

- $\forall x \in S(l) : x \leq v$ et
- $\forall x \in S(r) : x \geq v$.

On peut écrire certaines fonctions importantes comme relations de récurrence.

L'exemple le plus important est la fonction `contains`, qui teste $x \in S(t)$. Formellement, la *spécification* de `contains` est que

$$\texttt{contains}\ (t, x) = (x \in S(t))$$

On « l'implémente » comme

$$\texttt{contains}\ (t, x) = \begin{cases} \texttt{false} & \text{si } t = \texttt{NULL} \\ \texttt{true} & \text{si } t = (l, v, r) \wedge x = v \\ \texttt{contains}\ (l, x) & \text{si } t = (l, v, r) \wedge x < v \\ \texttt{contains}\ (r, x) & \text{si } t = (l, v, r) \wedge x > v \end{cases}$$

Quand on veut ajouter une valeur dans un arbre, on doit l'insérer au bon endroit pour respecter les propriétés. La spécification est cette fois que

$$\texttt{insert}\ (t, x) = t' \text{ tel que } S(t') = S(t) \cup \{x\}$$

Techniquement, ce n'est pas suffisant, parce qu'on ne dit rien de ce qui se passe quand on ajoute un élément qui est déjà présent. On pourrait modéliser un ensemble simple, ou alors un « multi-set », qui sait aussi *combien* d'occurrences de x sont dans t. On ignore ces détails.

Cela donne lieu à « l'implémentation » récursive suivante :

$$\texttt{insert}\ (t, x) = \begin{cases} (\texttt{NULL}, \text{x}, \texttt{NULL}) & \text{si } t = \texttt{NULL} \\ (\texttt{insert}\ (l, x), v, r) & \text{si } t = (l, v, r) \wedge x < v \\ (l, v, \texttt{insert}\ (r, x)) & \text{sinon} \end{cases}$$

13.1. Preuves

Pas matière d'examen

On peut *prouver* que les « implémentations » que nous avons données sont correctes.

Par exemple, pour `contains`, on peut établir le lemme suivant.

Lemme 13.1.1: Pour tout t arbre binaire de recherche, x entier, $\texttt{contains}\ (t, x) = (x \in S(t))$.

Preuve: Par induction sur la taille de t, puis par analyse de cas pour t.

Cas $t = \texttt{NULL}$:

On a $x \in S(\texttt{NULL}) = x \in \emptyset = \texttt{false}$, et $\texttt{contains}\ (\texttt{NULL}, x) = \texttt{false}$.

Cas $t = (l, v, r)$:

Par hypothèse d'induction, puisque l et r sont « plus petits » que t, on a $\texttt{contains}\ (l, x) = (x \in S(l))$ et $\texttt{contains}\ (r, x) = (x \in S(r))$.

Par définition de $S(t)$, on a $S(t) = S(l) \cup \{v\} \cup S(r)$.

Si $x = v$, alors $x \in S(t) = \texttt{true}$ et $\texttt{contains}\ ((l, v, r), x) = \texttt{true}$.

Si $x < v$, alors par les propriétés respectées par un arbre binaire de recherche, on sait que $\forall x \in S(r) : x \geq v$. Donc $x \notin S(r)$ et donc $x \in S(t)$ ssi $x \in S(l)$. Or $\texttt{contains}\ ((l, v, r), x) = \texttt{contains}\ (l, x) = x \in S(l)$ (rappelez-vous l'hypothèse d'induction).

Si $x > v$, symmétrique au cas $x < v$. □

Challenge : Pouvez-vous prouver, en utilisant une technique similaire, que `insert` est correcte ? Attention, `insert` *produit* un nouvel arbre binaire de recherche. Il faut donc aussi prouver que son résultat satisfait bien aux propriétés d'un arbre binaire de recherche (pas seulement que $S(t') = S(t) \cup \{x\}$). L'énoncé complet du lemme à prouver est :

Lemme 13.1.2: Pour tout t arbre binaire de recherche, x entier, $t' = \texttt{insert}\ (t, x) = (l', v', r')$, on a :

- $S(t') = S(t) \cup \{x\}$ (on a ajouté x)
- $\forall y \in S(l') : y \leq v'$ et $\forall y \in S(r') : y \geq v'$ (on a produit un arbre binaire de recherche valide).