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`.

```
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 ?