

# Initiation au Langage C



école supérieure de  
génie informatique

Cours de Kevin TRANCHO

Dispensé en Première et Seconde année  
À l'ESGI Paris  
Année scolaire 2022 - 2023



# Initiation au Langage C



école supérieure de  
génie informatique

Cours de Kevin TRANCHO

Dispensé en Première et Seconde année  
À l'ESGI Paris  
Année scolaire 2022 - 2023

## À propos

Ce support de cours est destiné à un usage pédagogique d'initiation au Langage C dispensé à l'ESGI Paris. Ce cours est rédigé et animé à l'ESGI Paris par Kevin TRANCHO.

## Équipe pédagogique relative à ce cours

**Kevin TRANCHO**

**ktranch@myges.fr**

Intervenant langage C. Production des supports de cours. Création et notation des sujets du contrôle continu et d'examen.

**Guénaelle BEAUJOUAN**

**gbeaujouan@esgi.fr**

Attachée de Promotion 1<sup>ère</sup> et 2<sup>ème</sup> années alternance.

**Frédéric SANANES**

**fsananes@esgi.fr**

Directeur Pédagogique 1<sup>ère</sup> et 2<sup>ème</sup> années.

**Kamal HENNOU**

**direction@esgi.fr**

Directeur de l'ESGI.

---

# Table des matières

---

<b>I</b>	<b>Modalités de ce cours</b>	<b>1</b>
<b>1</b>	<b>Préambule</b>	<b>5</b>
1.1	Comment s'organise un cours ?	5
1.1.1	Séances de cours	5
1.1.2	Support de cours	7
1.2	Quelques règles pour travailler ensemble	7
1.2.1	Politesse	8
1.2.2	Respect	8
1.2.3	Rigueur	8
1.2.4	Authenticité	8
<b>2</b>	<b>Se préparer aux évaluations</b>	<b>9</b>
2.1	Des conseils pour réussir l'UE Langage C ?	9
2.1.1	Première partie : découverte	9
2.1.2	Seconde partie : acquisition	9
2.1.3	Troisième partie : perfectionnement	9
2.1.4	L'assiduité	10
2.1.5	S'entraîner	10
2.2	Comment s'organisent les évaluations ?	10
2.2.1	Rendu des travaux pratiques en autonomie	10
2.2.2	Examen blanc	12
2.3	Planification cours et évaluations	12
2.3.1	Semestre 1	13
2.3.2	Semestre 2	14
2.3.3	Semestre 3	15
<b>II</b>	<b>Découverte</b>	<b>17</b>
<b>1</b>	<b>Introduction</b>	<b>23</b>
1.1	Algorithmique	23
1.1.1	Activité introductive à la programmation	23
1.2	Langage C	24

---

## TABLE DES MATIÈRES

---

1.2.1	Motivation . . . . .	24
1.2.2	Compilation . . . . .	25
1.2.3	Édition des liens . . . . .	26
1.3	Première application console . . . . .	26
1.3.1	Code source .c . . . . .	26
1.4	Préparer ses outils . . . . .	28
1.4.1	Installation gcc . . . . .	29
1.4.2	Compilation avec gcc . . . . .	31
1.4.3	Mac alternative : Xcode . . . . .	32
1.4.4	Online GDB : compilateur en ligne . . . . .	32
1.5	Résumé . . . . .	33
1.6	Entraînement . . . . .	35
<b>2</b>	<b>Variables</b>	<b>37</b>
2.1	Introduction et motivation . . . . .	37
2.2	Variables typées . . . . .	39
2.2.1	Déclaration . . . . .	39
2.2.2	Entiers signés . . . . .	40
2.2.3	Entiers non-signés . . . . .	41
2.2.4	Nombres à virgule flottante . . . . .	41
2.2.5	Constantes . . . . .	42
2.3	Format d’affichage . . . . .	43
2.3.1	Printf . . . . .	43
2.3.2	Formats pour entiers . . . . .	43
2.3.3	Formats pour flottants . . . . .	47
2.4	Adresse d’une variable . . . . .	49
2.5	Scanf . . . . .	50
2.6	Résumé . . . . .	53
2.7	Entraînement . . . . .	56
<b>3</b>	<b>Expressions</b>	<b>61</b>
3.1	Opérations classiques . . . . .	61
3.1.1	Addition . . . . .	61
3.1.2	Soustraction et multiplication . . . . .	63
3.1.3	Compatibilité entre entiers et flottants . . . . .	63
3.2	Division . . . . .	64
3.2.1	Division entière . . . . .	64
3.2.2	Modulo : reste de la division euclidienne . . . . .	64
3.2.3	Division fractionnaire . . . . .	65
3.3	Coercition : un changement de type . . . . .	66
3.4	Réaffectation d’une variable avec opérateur . . . . .	69
3.5	Résumé . . . . .	72

---

## TABLE DES MATIÈRES

---

3.6	Entraînement . . . . .	74
<b>4</b>	<b>Conditions</b>	<b>79</b>
4.1	Sélection d'instructions avec if . . . . .	79
4.2	Comparaisons . . . . .	82
4.3	Opérateurs booléens . . . . .	84
4.3.1	Intersection . . . . .	84
4.3.2	Union . . . . .	86
4.3.3	Négation . . . . .	87
4.4	Ternaire : expression sous condition . . . . .	89
4.5	Switch : se brancher un à résultat . . . . .	90
4.6	Résumé . . . . .	94
4.7	Entraînement . . . . .	95
<b>5</b>	<b>Boucles</b>	<b>101</b>
5.1	While : répétition sous condition . . . . .	101
5.2	Do while : répétition si besoin . . . . .	102
5.3	For : un pas après l'autre . . . . .	103
5.4	Contrôle de la boucle : break or continue . . . . .	105
5.5	Résumé . . . . .	108
5.6	Entraînement . . . . .	109
<b>III</b>	<b>Notions de base</b>	<b>111</b>
<b>6</b>	<b>Fonctions</b>	<b>115</b>
6.1	La fonction main . . . . .	115
6.2	Définition de fonctions . . . . .	117
6.3	Portée des variables . . . . .	120
6.3.1	Variables locales à une fonction . . . . .	120
6.3.2	Variables globales . . . . .	124
6.3.3	Portée des variables . . . . .	124
6.4	Déplacer sa définition de fonction . . . . .	126
6.4.1	Déclaration d'une fonction . . . . .	126
6.4.2	Appels mutuels entre fonctions interdépendantes . . . . .	127
6.5	Résumé . . . . .	128
6.6	Entraînement . . . . .	129
<b>7</b>	<b>Tableaux</b>	<b>137</b>
7.1	Tableau à une dimension . . . . .	137
7.2	Chaînes de caractères . . . . .	141
7.3	Tableau à plusieurs dimensions . . . . .	142
7.4	Résumé . . . . .	146

---

## TABLE DES MATIÈRES

---

7.5	Entraînement . . . . .	147
<b>8</b>	<b>Pointeurs</b>	<b>153</b>
8.1	Un type d'adresse . . . . .	153
8.2	Arithmétique des pointeurs . . . . .	155
8.3	Allocation dynamique . . . . .	158
8.4	Résumé . . . . .	161
8.5	Entraînement . . . . .	163
<b>9</b>	<b>Projet : labyrinthe</b>	<b>173</b>
9.1	Sujet . . . . .	173
9.2	Évaluation . . . . .	174
9.2.1	(... / 6 points) Aspects code . . . . .	174
9.2.2	(... / 6 points) Aspects fonctionnalités . . . . .	174
9.2.3	(... / 15 points) Améliorations . . . . .	175
<b>IV</b>	<b>Notions avancées</b>	<b>177</b>
<b>10</b>	<b>Consolidation des bases</b>	<b>183</b>
10.1	Remise dans le bain rapide . . . . .	183
10.1.1	Un programme en C . . . . .	183
10.1.2	Variables . . . . .	184
10.1.3	Opérations . . . . .	185
10.1.4	Tests et disjonctions . . . . .	186
10.1.5	Boucles . . . . .	187
10.1.6	Fonctions . . . . .	188
10.1.7	Tableaux . . . . .	189
10.1.8	Pointeurs . . . . .	190
10.2	Entraînement . . . . .	193
<b>11</b>	<b>Fichiers</b>	<b>197</b>
11.1	Ouverture et création de fichiers . . . . .	197
11.2	Lecture et écriture . . . . .	198
11.2.1	Avec fonctions formatées . . . . .	198
11.2.2	Par caractère . . . . .	200
11.2.3	Par bloc mémoire . . . . .	201
11.3	Se déplacer dans un fichier . . . . .	203
11.4	Flux standards d'entrées et sorties . . . . .	205
11.4.1	stdout . . . . .	205
11.4.2	stdin . . . . .	205
11.4.3	stderr . . . . .	206
11.5	Résumé . . . . .	207



---

## TABLE DES MATIÈRES

---

11.6 Entraînement . . . . .	209
<b>12 Structures . . . . .</b>	<b>213</b>
12.1 Typedef . . . . .	213
12.2 Structures . . . . .	214
12.3 Définition . . . . .	215
12.4 Avec des pointeurs . . . . .	216
12.4.1 Champs de bits . . . . .	219
12.5 Unions . . . . .	221
12.6 Énumérations . . . . .	222
12.7 Résumé . . . . .	225
12.8 Entraînement . . . . .	226
<b>13 Programmation modulaire . . . . .</b>	<b>233</b>
13.1 Retour sur la compilation . . . . .	233
13.1.1 Externaliser une fonction . . . . .	233
13.1.2 Compilation séparée . . . . .	235
13.2 Directives préprocesseur . . . . .	237
13.2.1 include . . . . .	238
13.2.2 define . . . . .	239
13.2.3 conditionnement . . . . .	244
13.2.4 Module . . . . .	245
13.3 Makefiles . . . . .	248
13.3.1 Concept . . . . .	248
13.3.2 Possibilités . . . . .	248
13.4 Résumé . . . . .	253
13.5 Entraînement . . . . .	255
<b>14 Opérations bit-à-bit . . . . .</b>	<b>267</b>
14.1 Décalages . . . . .	267
14.1.1 À gauche . . . . .	267
14.1.2 À droite . . . . .	268
14.2 Négation . . . . .	268
14.3 Opérateurs booléens . . . . .	269
14.3.1 et . . . . .	269
14.3.2 ou inclusif . . . . .	270
14.3.3 ou exclusif : xor . . . . .	271
14.4 Résumé . . . . .	273
14.5 Entraînement . . . . .	275

---

## TABLE DES MATIÈRES

---

<b>15 Types génériques</b>	<b>281</b>
15.1 Pointeurs de fonctions	282
15.1.1 Définition	282
15.1.2 Appel	283
15.1.3 Constructions plus complexes	284
15.1.4 Typedef	286
15.2 Pointeurs génériques	288
15.2.1 Intérêt	289
15.2.2 Exemple avec qsort	289
15.2.3 Manipulation	291
15.3 Fonctions variadiques	294
15.4 Résumé	296
15.5 Entraînement	298
<b>16 Projet final</b>	<b>303</b>
16.1 Étapes	303
16.2 Possibilités de sujet	303
16.3 Livrables	304
16.4 Évaluation	304
16.4.1 (... / 9 points) Code	305
16.4.2 (... / 7 points) Fonctionnalités	305
16.4.3 (... / 5 points) Soutenance et rapport	306
<b>V Bonus : jouons rapidement avec SDL 1.2</b>	<b>307</b>
<b>17 Initialisation</b>	<b>311</b>
17.1 Préambule	311
17.2 Mise en place	311
17.2.1 Linux	311
17.2.2 Windows	313
17.3 Fenêtre	314
17.3.1 SDL Init	314
17.3.2 SDL SetVideoMode	315
<b>18 Affichage</b>	<b>317</b>
18.1 Dessiner un rectangle	318
18.1.1 SDL FillRect	318
18.1.2 SDL Rect	319
18.2 Créer une surface	320
18.2.1 SDL CreateRGBSurface	320
18.2.2 SDL BlitSurface	322

---

## TABLE DES MATIÈRES

---

18.3 Images en BMP . . . . .	324
18.3.1 SDL Image . . . . .	327
18.3.2 Dessiner une sous-image . . . . .	328
18.4 Résumé . . . . .	331
<b>19 Événements</b>	<b>333</b>
19.1 Récupération d'événements . . . . .	333
19.1.1 SDL WaitEvent . . . . .	333
19.1.2 SDL PollEvent . . . . .	334
19.2 Analyse d'événements . . . . .	334
19.2.1 Types d'événements . . . . .	334
19.2.2 Clavier . . . . .	335
19.2.3 Boutons de souris . . . . .	337
19.2.4 Coordonnées de souris . . . . .	337
19.3 Résumé . . . . .	340
 <b>VI Examens de l'année précédente</b>	 <b>343</b>
 <b>A Données statistiques partiels 2021 - 2022</b>	 <b>347</b>
A.1 Semestre 1 . . . . .	348
A.1.1 Alternance . . . . .	348
A.1.2 Janvier . . . . .	348
A.2 Semestre 2 . . . . .	349
A.2.1 Alternance . . . . .	349
A.2.2 Janvier . . . . .	349
 <b>B Sujets partiels 2021 - 2022</b>	 <b>351</b>
B.1 Semestre 1 - Alternance . . . . .	352
B.2 Semestre 1 - Janvier . . . . .	358
B.3 Semestre 2 - Sujet zéro . . . . .	364
B.4 Semestre 2 - Alternance . . . . .	370
B.5 Semestre 2 - Janvier . . . . .	376

## TABLE DES MATIÈRES

---

---

# Exercices

---

## Chapitre 1

1	Activité – Formaliser une logique . . . . .	23
2	Activité – ★ Compiler Hello ESGI! . . . . .	28
3	Activité – ★ Votre Hello ESGI . . . . .	35

## Chapitre 2

4	Activité – ★ Déclarer et définir des variables . . . . .	42
5	Activité – ★★ Afficher des variables . . . . .	49
6	Activité – ★★ Lire et afficher une variable . . . . .	52
1	Exercice noté – ★ Affectations . . . . .	56
2	Exercice noté – ★ Traduction en Langage C . . . . .	57
3	Exercice noté – ★★ Dépassement capacité d'un int . . . . .	58
4	Exercice noté – ★★ Imprécision flottante . . . . .	59
5	Exercice noté – ★★ ★ Hexadécimal ? . . . . .	60

## Chapitre 3

7	Activité – ★ Calculer pour l'utilisateur . . . . .	63
6	Exercice noté – ★★ Opérations . . . . .	74
7	Exercice noté – ★★ Division par zéro . . . . .	75
8	Exercice noté – ★★ ★ Affection d'une addition ? . . . . .	76
9	Exercice noté – ★★ Imprécision et opérations . . . . .	77
10	Exercice noté – ★★ ★ Message codé . . . . .	78

## Chapitre 4

8	Activité – ★ Déterminer une catégorie d'âge . . . . .	84
9	Activité – ★ Intersection d'événements . . . . .	85
10	Activité – ★★ Calculer l'amende d'un excès de vitesse . . . . .	88
11	Exercice noté – ★★ ★ Acheter un article . . . . .	95
12	Exercice noté – ★★ ★ Calculatrice . . . . .	96
13	Exercice noté – ★★ Racines d'un polynôme du second degré . . . . .	97
14	Exercice noté – ★★ ★ Code obscurantiste . . . . .	98
15	Exercice noté – ★★ ★ Validation d'un mot de passe . . . . .	99

## Chapitre 5

16	Exercice noté – ★★ ★ Liste des diviseurs . . . . .	109
17	Exercice noté – ★★ ★ Force brute . . . . .	109

---

## EXERCICES

---

18	Exercice noté – ★★★ Affichage en binaire . . . . .	109
19	Exercice noté – ★★★ PGCD . . . . .	109
20	Exercice noté – ★★★ Jeu du plus ou moins . . . . .	110

### Chapitre 6

11	Activité – ★ Définir une fonction . . . . .	119
21	Exercice noté – ★★★ Fonction de menu . . . . .	129
22	Exercice noté – ★★★ Calcul moyenne . . . . .	130
23	Exercice noté – ★★★ Fonction puissance . . . . .	131
24	Exercice noté – ★★★ Suite de Fibonacci . . . . .	132
25	Exercice noté – ★★★ Réagencer un code . . . . .	135

### Chapitre 7

26	Exercice noté – ★★★ Statistiques sur un tableau . . . . .	147
27	Exercice noté – ★ Propriétés dans un labyrinthe . . . . .	148
28	Exercice noté – ★★★ Coder les fonction de string.h . . . . .	150
29	Exercice noté – ★★★ Recherche dans un tableau . . . . .	151
30	Exercice noté – ★★★ Décodage Vigenère . . . . .	151

### Chapitre 8

12	Activité – ★★★ Fonction pour échanger des valeurs . . . . .	154
31	Exercice noté – ★ Pointeurs sur des variables . . . . .	163
32	Exercice noté – ★ Pointinception . . . . .	164
33	Exercice noté – ★★★ Concaténation de chaînes de caractères . . . . .	165
34	Exercice noté – ★★★ Construction liste extensible . . . . .	166
35	Exercice noté – ★★★ Trier des valeurs . . . . .	168
36	Exercice noté – ★★★ Allocation d'une grille de labyrinthe . . . . .	171

### Chapitre 10

37	Exercice noté – ★★★ Statistiques sur liste d'entiers . . . . .	193
38	Exercice noté – ★★★ Extraire information d'une chaîne de caractères	193
39	Exercice noté – ★★★ Mini-interpréteur sur liste d'entiers . . . . .	194
40	Exercice noté – ★★★ Jeu du Morpion . . . . .	195
41	Exercice noté – ★★★ Enregistrement et recherche de numéros . . . . .	196

### Chapitre 11

42	Exercice noté – ★ Compteur de lancements . . . . .	209
43	Exercice noté – ★★★ Codage Vigenère depuis fichier . . . . .	209
44	Exercice noté – ★★★ Sauvegarde et chargement en binaire . . . . .	210
45	Exercice noté – ★★★ Enregistrement et recherche de numéros avec sauvegarde . . . . .	211

**Chapitre 12**

46	Exercice noté – ★★★ Définir une structure Vecteur2d . . . . .	226
47	Exercice noté – ★★★ Structure pour gérer une grille . . . . .	227
48	Exercice noté – ★★★ Gérer un combat de personnages . . . . .	229
49	Exercice noté – ★★★ Implémenter une liste chaînée . . . . .	230

**Chapitre 13**

50	Exercice noté – ★★ Mise en place d'un Makefile . . . . .	255
51	Exercice noté – ★★★ Profilage d'un code . . . . .	257
52	Exercice noté – ★★★ Implémentation table de hachage . . . . .	259
53	Exercice noté – ★★★ Représentation d'un lexique . . . . .	264

**Chapitre 14**

54	Exercice noté – ★★ Application opérations bit-à-bit . . . . .	275
55	Exercice noté – ★★★ Gérer une grille sur un entier . . . . .	277
56	Exercice noté – ★★★ Buffer pour lecture et écriture bit-à-bit . . .	278

**Chapitre 15**

57	Exercice noté – ★★★ Trier une liste de points . . . . .	298
58	Exercice noté – ★★★ Benchmark de qsort . . . . .	298
59	Exercice noté – ★★★ Trier des pointeurs de fonctions . . . . .	299
60	Exercice noté – ★★★★★ HashMap et ArrayList génériques . . . . .	301

## EXERCICES

---



Première partie

Modalités de ce cours



# Table des matières

<b>1</b>	<b>Préambule</b>	<b>5</b>
1.1	Comment s'organise un cours ?	5
1.1.1	Séances de cours	5
1.1.2	Support de cours	7
1.2	Quelques règles pour travailler ensemble	7
1.2.1	Politesse	8
1.2.2	Respect	8
1.2.3	Rigueur	8
1.2.4	Authenticité	8
<b>2</b>	<b>Se préparer aux évaluations</b>	<b>9</b>
2.1	Des conseils pour réussir l'UE Langage C ?	9
2.1.1	Première partie : découverte	9
2.1.2	Seconde partie : acquisition	9
2.1.3	Troisième partie : perfectionnement	9
2.1.4	L'assiduité	10
2.1.5	S'entraîner	10
2.2	Comment s'organisent les évaluations ?	10
2.2.1	Rendu des travaux pratiques en autonomie	10
2.2.2	Examen blanc	12
2.3	Planification cours et évaluations	12
2.3.1	Semestre 1	13
2.3.2	Semestre 2	14
2.3.3	Semestre 3	15



---

# 1 Préambule

---

Bienvenue au cours de langage C à l'ESGI ! Pour accompagner les cours dispensés en classe par votre professeur, vous avez le présent support de cours. Celui-ci essaiera de répondre à vos appréhensions et vous guider dans votre apprentissage du langage C cette année :

- Modalités du cours partie **I** : règles pour un bon déroulé de séance, conseils pour appréhender son apprentissage cette année et planning des évaluations.
- Cours de la première année (Semestre 1 partie **II** et 2 partie **III**) : bases du langage.
- Cours de la seconde année (Semestre 3 partie **IV**) : notions plus avancées.
- Sujets de l'année précédente (Semestre 1 et 2) partie **VI** pour se préparer à l'examen papier (pour le Semestre 3 ce sera un projet avec soutenance chapitre **16**).

Bon courage !

## 1.1 Comment s'organise un cours ?

### 1.1.1 Séances de cours

Les séances de cours s'articulent en général sur le schéma suivant :

- Questions / Feedback sur le cours précédent.
- Correction exercices à rendre (partie entraînement des chapitres).
- Début d'une nouvelle notion.
- Présentation (slides fournies sur MyGES) de la notion.

- Exercices d'application pendant la présentation (listés comme activité dans le support) :
  - Temps donné pour essayer et présenter sa solution au professeur.
  - Proposition d'une solution par le professeur.
- Questions / Feedback sur la notion.
- Temps donné pour travailler les exercices en autonomie (fournis dans le support de cours partie entraînement, version découpée fournie sur MyGES) à rendre pour la séance suivante.
- Le professeur circule pendant le temps en autonomie au besoin pour guider les étudiants vers une solution et répondre à leurs problématiques individuelles.
- Les étudiants qui terminent le travail attendu en avance peuvent étudier la suite du cours sur ce support ou travailler sur un projet pertinent en langage C.

### Exercices d'application : activité

Les exercices d'applications seront donnés, traités et corrigés dans la foulée. Leur idée est de permettre à l'étudiant de se familiariser et découvrir la notion.

### Exercices notés : entraînement

Les exercices notés ont pour but de vérifier l'assimilation de la notion par l'étudiant et d'attribuer une note pour s'en donner une idée chiffrée. Les exercices notés sont donnés en groupes à la fin d'une notion. L'ensemble de la notation des exercices d'une notion fait une note qui compte pour le Contrôle Continu. Ces exercices seront à rendre sur MyGES dans les travaux pratiques à l'endroit qui correspond à la notion étudiée.

### Difficulté des exercices

La résolution des exercices ne demandent pas en général de dépasser les notions vues en classe. Si vous pensez à une solution qui semble inabordable pour la difficulté attendue, peut-être que poser à nouveau le problème donnée aiderait à une résolution plus simple.

Les niveaux de **difficulté** relatifs à la notion abordée :

1. ★ - application triviale du cours.
2. ★★ - application du cours.
3. ★★★ - problème simple.
4. ★★★★ - problème plus difficile.
5. ★★★★★ - challenge.

### 1.1.2 Support de cours

Ce présent support de cours vous accompagnera durant cette année de langage C. Ouch, mais il est énorme ce truc ! Pas de panique, on va y aller pas à pas. Il peut paraître long et bourratif au premier abord mais vous l'apprécierez probablement au fil de l'année lorsque vous aurez besoin de revenir sur une notion précédente ou vous avancerez sur la suivante.

À noter que pour chaque séance votre enseignant fera l'effort de découper ce support pour que vous n'ayez que le contenu pertinent pour la séance qu'il aborde. Individuellement chaque partie est moins indigeste.

Le support de cours propose en général :

- Une **partie expliquée et rédigée** de la notion abordée (peut aider à la compréhension).
- Un **résumé** des éléments vus pour cette notion (peut aider à la révision ou faire les exercices).
- Des **exercices** d'entraînement sur cette notion (permet de pratiquer la notion pour l'assimiler).

## 1.2 Quelques règles pour travailler ensemble

Bienvenue dans l'enseignement supérieur, cependant il ne faut pas oublier certains principes élémentaires pour travailler avec votre professeur et vis-à-vis de vos camarades. Nous pouvons travailler dans une bonne ambiance, mais elle doit viser l'objectif qui est de vous permettre d'assimiler le contenu proposé en vue de l'obtention de votre diplôme et par la suite de votre réussite professionnelle. Pour ceci, nous nous efforcerons de respecter au mieux quelques règles pour travailler ensemble.

### 1.2.1 Politesse

Arriveriez-vous en retard devant un film au cinéma ? À moins d'aimer le challenge de comprendre celui-ci en plein milieu et interrompre avec agacement les spectateurs déjà présents, j'en doute. Organisez vous au mieux et en cas d'imprévu, ne dérangez pas la séance en cours, présentez vous pour la séance suivante. Si vous avez besoin de sortir, ne partez pas vous promener l'air de rien. Prévenez votre professeur, c'est la moindre des politesses.

### 1.2.2 Respect

Que penseriez-vous d'être client et que le professionnel en face vous fasse attendre la fin de son combat sur Clash Royale, ne pas casser son coup Tinder ou s'engouffrer dans une longue partie de League of Legends sur son temps de travail ? Ce serait probablement l'individu le plus professionnel que vous rencontreriez, non ? En langage C, le téléphone portable, jeux, vidéos et autres sont sources de distractions impertinentes. Notez qu'il s'agit aussi ici de respecter vos camarades et votre propre apprentissage. Vous avez choisi l'ESGI pour progresser dans d'autres aptitudes, respectez vous et attendez la pause.

### 1.2.3 Rigueur

L'employé décrit précédemment vous a rendu les clés de votre voiture, mais impossible de la trouver. Rageant, non ? Lorsque vous rendez un livrable à votre enseignant et à l'avenir à un client ou votre patron, faites attention à ce que vous rendez. Pour vous évaluer votre enseignant a besoin des sources viables de votre programme et ne pas trouver votre contenu avec pénibilité. Votre professeur a fait l'effort de vous concocter ce support de cours pour vous offrir plus de sérénité dans votre apprentissage, faites que ce soit réciproque.

### 1.2.4 Authenticité

Rendez le travail auquel vous avez contribué, le professeur se doute que vous êtes en capacité d'effectuer un copier-coller d'internet, d'un camarade ou de sa propre correction et de le twister si besoin. Ce n'est malheureusement pas là-dessus qu'il vous évalue et qu'il veut vous élever par ce cours. Imaginez le soupir d'un professeur qui vient de corriger 10 devoirs identiques à la suite et qui se demande auquel il doit attribuer les points ou si chacun ayant probablement contribué équitablement, il vaudrait mieux diviser la note par le nombre de participants. Essayez honnêtement, vous êtes là pour apprendre et gagner en capacités : un travail de votre part même imparfait ou incomplet sera d'autant plus apprécié et profitable pour votre enrichissement en langage C.



---

## 2 Se préparer aux évaluations

---

### 2.1 Des conseils pour réussir l’UE Langage C ?

L’apprentissage du langage C est distribué sur deux ans en trois parties. Chaque partie correspond à un **niveau d’exigence croissant** et un investissement de l’étudiant tout aussi grandissant pour réussir cette Unité d’Enseignement. Accrochez vous !

#### 2.1.1 Première partie : découverte

L’examen de la première partie du cours (partie **II**) aborde des notions relativement élémentaires. Celles-ci seront facilement accessibles avec une bonne assiduité en classe. L’objectif de cette première partie est d’évaluer la compréhension de la syntaxe de base du langage C et son application sur des demandes simples ne dépassant pas la notion de boucles.

#### 2.1.2 Seconde partie : acquisition

Cependant, cette bonne assimilation des notions de la première partie ne garantissent pas un succès sans travail pour l’assimilation de la seconde partie (partie **III**). Un maintien du sérieux reste primordial. En effet, les notions vont avancer jusqu’aux pointeurs. À ce moment, nous commençons à effleurer les concepts du langage C. Cette partie commencera à demander une logique dans la conception du code : choix des notions à utiliser pour répondre à un problème simple.

#### 2.1.3 Troisième partie : perfectionnement

La dernière partie (partie **IV**) aborde des notions plus avancées du langage C. Celle-ci a plus un but de raffinement des concepts jusque là connus. En effet, bien que les notions vues précédemment permettent de mener à bien un projet, les notions vues ici permettent de le mener avec plus de maintenabilité, d’efficacité et approfondir la connaissance des paradigmes du langage C. On attend ici d’être capable choisir ses outils dans le langage comme des briques de base pour répondre à ses besoins le plus pertinemment possible : découper son code en modules, choisir une structure de données pertinente, produire un code générique et maintenable.

### 2.1.4 L'assiduité

Ceci paraît trivial et pourtant ce probablement ce qui vous ferait gagner le plus de temps. Un étudiant qui écoute en classe et pose des questions permettant la précision d'une incompréhension lors du cours fera gagner à ses camarades et lui-même un temps précieux à ne pas se torturer l'esprit à lire et comprendre le cours en dehors de la séance.

Un temps est souvent donné pour essayer les exercices en classe, c'est le moment d'identifier ce qui peut poser problème et profiter de la présence du professeur pour lui demander conseil. Le temps investi en classe est du temps gagné à la maison, en particulier s'il permet de terminer les exercices avant la fin de la séance.

### 2.1.5 S'entraîner

En première approche les exercices vus en classe permettent une première appréhension de la notion. Puis viennent les exercices à réaliser en autonomie pour approfondir celle-ci.

Pour se tester et s'entraîner, les sujets de l'année précédente sont disponible en annexes [VI](#).

Il est aussi possible de s'entraîner d'avantage sur des plateformes en ligne. Par exemple sur [CodinGame](#).

Pour réviser le cours du langage C, le plus pertinent est probablement de pratiquer. Une bonne idée peut-être de se donner un projet avec l'objectif de votre choix qui permettrait une mise en pratique des différentes notions du cours. À noter qu'un apprentissage par cœur ne permet pas l'acquisition des automatismes et de la logique attendus lorsque vous codez. Un minimum reste tout de même requis pour connaître vos possibilités en langage C.

## 2.2 Comment s'organisent les évaluations ?

### 2.2.1 Rendu des travaux pratiques en autonomie

Le contrôle continu est essentiellement représenté par vos rendus des travaux demandés en autonomie et à rendre sur **MyGES**. Notez que ces travaux sont réalisés de votre part sous des environnements de développement très différents les uns des autres (Linux, Windows, Mac, ChromeBook, Online GDB ou autre).

Votre professeur évaluera vos travaux sous un Linux 64 bits et n'a besoin de rien d'autre que vos **sources** (fichiers `.c` et `.h`). Tout autre fichier ne sera pas considéré comme évaluable et peut être pénalisé sauf si demandé explicitement dans l'exercice. N'organisez les fichiers dans un dossier que lorsqu'un exercice attend potentiellement plusieurs fichiers sources. Vous pouvez noter vos réponses aux questions d'un exercice en commentaires `/* ... */` et proposer plusieurs version d'un même fichier `exercice_X_1`, `exercice_X_2`, ..., `exercice_X_N` lorsque c'est pertinent.

Souvent les exercices seront attendus pour la séance suivante jusqu'à leur correction. En effet, ils seront corrigés en début d'une séance pour aborder la notion suivante. Notez que vous pouvez rendre vos exercices tant que votre professeur ne les a pas notés. Une fois notés c'est terminé, pensez donc à les rendre dans les temps.

Pour rendre les exercices, vous pouvez les rendre par groupes, cependant votre note sera échelonnée en fonction du nombre de participants annoncé ou découvert en évaluant les travaux. Cette notation suit la formule suivante :

$$\frac{NoteRenduFinale}{20} = \max \left( \frac{\frac{NoteRendu}{20} \times \lfloor 22 - 2^{NombreParticipants} \rfloor}{20}, 0 \right)$$

Nombre de participants	Note maximale
1	20
2	18
3	14
4	6
5+	0

Notez donc que la coopération est autorisée, mais évitez de nombreuses duplications de code (exemple : partage sur Discord à quelqu'un qui envoie le code de votre groupe, récupération sur internet ou autre) car ça se voit et les participants rendant le même travail que votre groupe se cumuleront comme participants dans votre notation.

De même, notez que rendre sa proposition de correction à son professeur se voit et vous vaudra la note de 0, même lorsque les noms des variables sont modifiés. Rendez votre travail, c'est là dessus que vous êtes évalués.

### 2.2.2 Examen blanc

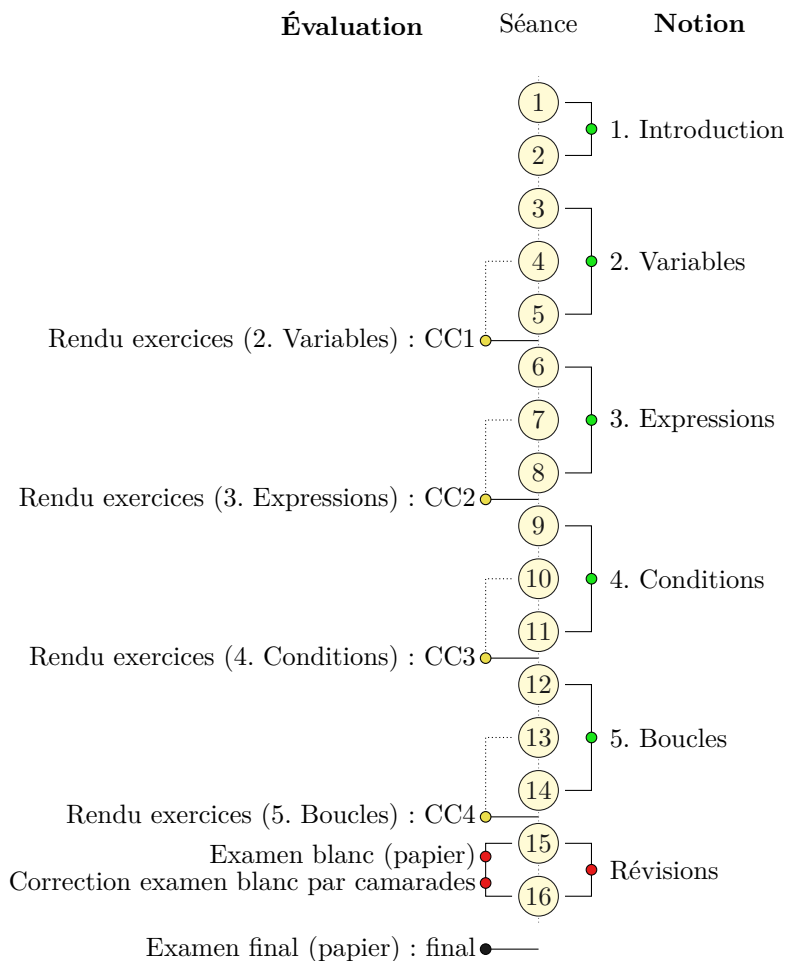
La **séance de fin de Semestre (1 et 2)**, nous organiserons une évaluation papier d'1h30 surveillée par le professeur pendant la première partie de la séance. Puis le temps d'1h30 après la pause sera dédié à la correction magistrale de cette évaluation avec correction de votre copie par un camarade en vue de préparer l'examen final.

## 2.3 Planification cours et évaluations

Vous trouverez ici la planification du cours en amont : notions abordées et moment des évaluations en fonction de la séance traitée. Chaque numéro correspond à une séance de 1h30.

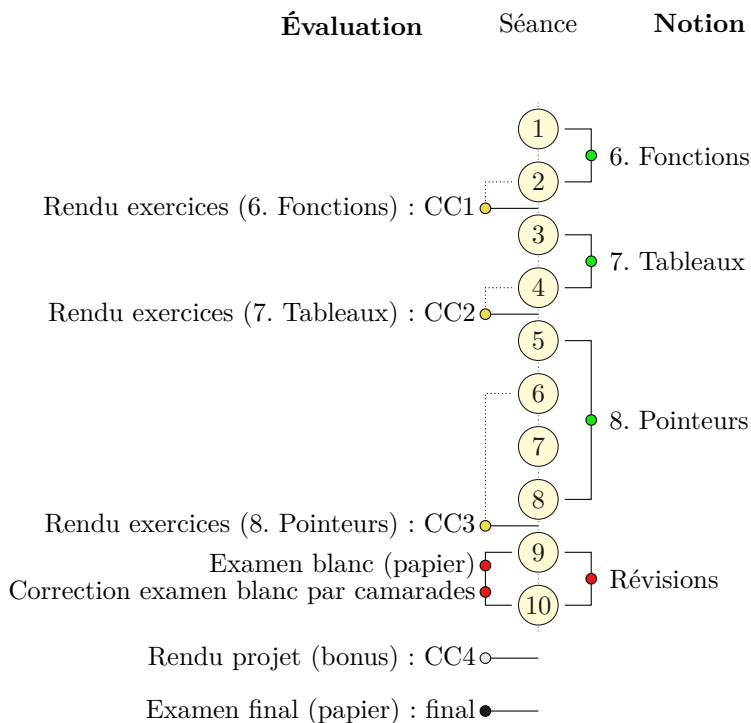
## 2.3.1 Semestre 1

## Partie II



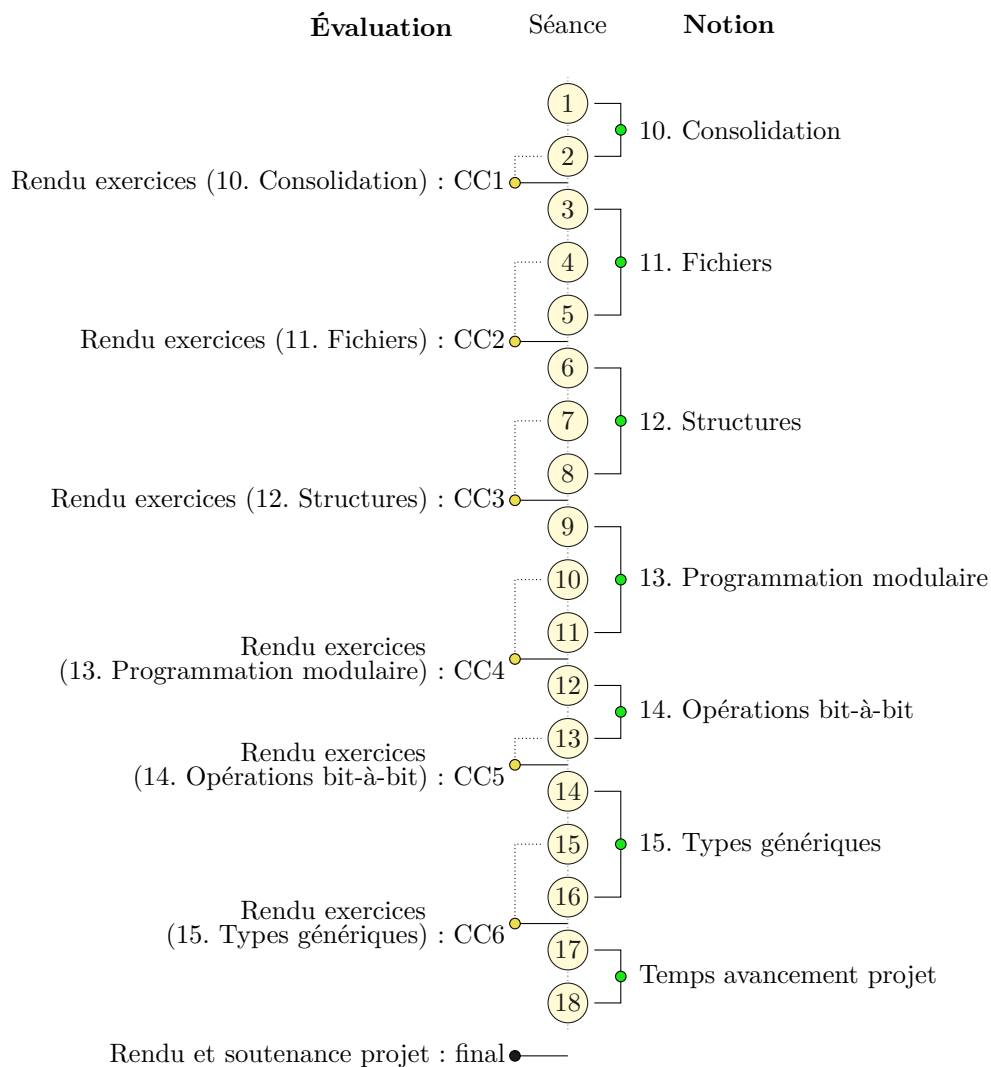
## 2.3.2 Semestre 2

## Partie III



### 2.3.3 Semestre 3

#### Partie IV







Deuxième partie

Découverte



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>23</b>
1.1	Algorithmique . . . . .	23
1.1.1	Activité introductive à la programmation . . . . .	23
1.2	Langage C . . . . .	24
1.2.1	Motivation . . . . .	24
1.2.2	Compilation . . . . .	25
1.2.3	Édition des liens . . . . .	26
1.3	Première application console . . . . .	26
1.3.1	Code source .c . . . . .	26
1.4	Préparer ses outils . . . . .	28
1.4.1	Installation gcc . . . . .	29
1.4.2	Compilation avec gcc . . . . .	31
1.4.3	Mac alternative : Xcode . . . . .	32
1.4.4	Online GDB : compilateur en ligne . . . . .	32
1.5	Résumé . . . . .	33
1.6	Entraînement . . . . .	35
<b>2</b>	<b>Variables</b>	<b>37</b>
2.1	Introduction et motivation . . . . .	37
2.2	Variables typées . . . . .	39
2.2.1	Déclaration . . . . .	39
2.2.2	Entiers signés . . . . .	40
2.2.3	Entiers non-signés . . . . .	41
2.2.4	Nombres à virgule flottante . . . . .	41
2.2.5	Constantes . . . . .	42
2.3	Format d’affichage . . . . .	43
2.3.1	Printf . . . . .	43

---

2.3.2	Formats pour entiers . . . . .	43
2.3.3	Formats pour flottants . . . . .	47
2.4	Adresse d'une variable . . . . .	49
2.5	Scanf . . . . .	50
2.6	Résumé . . . . .	53
2.7	Entraînement . . . . .	56
<b>3</b>	<b>Expressions</b>	<b>61</b>
3.1	Opérations classiques . . . . .	61
3.1.1	Addition . . . . .	61
3.1.2	Soustraction et multiplication . . . . .	63
3.1.3	Compatibilité entre entiers et flottants . . . . .	63
3.2	Division . . . . .	64
3.2.1	Division entière . . . . .	64
3.2.2	Modulo : reste de la division euclidienne . . . . .	64
3.2.3	Division fractionnaire . . . . .	65
3.3	Coercition : un changement de type . . . . .	66
3.4	Réaffectation d'une variable avec opérateur . . . . .	69
3.5	Résumé . . . . .	72
3.6	Entraînement . . . . .	74
<b>4</b>	<b>Conditions</b>	<b>79</b>
4.1	Sélection d'instructions avec if . . . . .	79
4.2	Comparaisons . . . . .	82
4.3	Opérateurs booléens . . . . .	84
4.3.1	Intersection . . . . .	84
4.3.2	Union . . . . .	86
4.3.3	Négation . . . . .	87
4.4	Ternaire : expression sous condition . . . . .	89
4.5	Switch : se brancher un à résultat . . . . .	90
4.6	Résumé . . . . .	94
4.7	Entraînement . . . . .	95
<b>5</b>	<b>Boucles</b>	<b>101</b>
5.1	While : répétition sous condition . . . . .	101
5.2	Do while : répétition si besoin . . . . .	102
5.3	For : un pas après l'autre . . . . .	103
5.4	Contrôle de la boucle : break or continue . . . . .	105

---

---

5.5	Résumé . . . . .	108
5.6	Entraînement . . . . .	109



---

# 1 Introduction

---

## 1.1 Algorithmique

Aujourd'hui entourés d'outils numériques en tout genre : téléphone portable, ordinateur et autres. Leur utilisation est devenue si triviale, mais en est-il autant de la conception de toutes leurs réponses automatiques à nos demandes ?

### 1.1.1 Activité introductive à la programmation

Si nous vous demandions de trier une liste de nombres du plus petit au plus grand, vous y arriveriez aisément, non ? Est-ce aussi le cas si vous deviez l'expliquer à un camarade ou pire encore le formaliser pour une machine ? Testons l'exercice avec le français, votre langage de programmation :

Activité 1 (Formaliser une logique).

1. Découper une feuille de papier en au moins 8 morceaux et y écrire des nombres.
2. Mélanger ces nombres.
3. Trier ces nombres.
4. Expliquer oralement à un camarade (voisin) la méthode choisie pour trier ces nombres et lui demander de la mettre en œuvre.
5. Écrire sur papier les instructions à suivre pour que tout camarade puisse trier une liste de nombres selon la méthode choisie.
6. Faire tester la méthode à votre professeur qui ne connaît que le jeu d'instructions suivant : déplacer des papiers avec ses mains, comparer des papiers deux à deux.

Cet exercice a conduit à l'écriture d'un **algorithme** en langage compréhensible par un humain. En effet, c'est comme une recette de cuisine : à partir d'éléments donnés, il suffit de suivre une suite d'**instructions** pour arriver automatiquement à un résultat souhaité.

Des exemples d'algorithmes ont déjà été rencontré durant la scolarité comme l'addition, la soustraction, la multiplication et la division posées. En effet, ces méthodes prennent au moins deux nombres et donnent automatiquement le résultat attendu. Il en est de même pour le plus explicite algorithme d'Euclide qui calcule le PGCD de deux nombres à l'aide de divisions successives.

Effectuer ces calculs est souvent humainement fastidieux d'autant plus pour des grands nombres. Ceci motive l'usage de la calculatrice par exemple. Mais pour ceci il faut avoir été capable d'indiquer à une machine comment réaliser ces tâches de manière automatique. Cette communication avec la machine peut se faire à l'aide d'un **langage de programmation**. Pour notre cours, nous avons choisi le langage C.

## 1.2 Langage C

### 1.2.1 Motivation

De nombreux langages de programmation existent que ce soit pour permettre de jouer aux derniers jeux vidéos, aller sur le réseau social à la mode, sécuriser nos transactions bancaires et tant d'autres choses.

Il existe une multitude de langages de programmation qui répondent à plusieurs besoins, certains vont préférer une écriture de code rapide au détriment du contrôle sur la machine et potentiellement de l'efficacité, d'autres vont se concentrer sur les performances que l'on peut obtenir mais ceci demandera souvent de garder un langage qui offre beaucoup de contrôle sur la machine et demande un plus long temps d'écriture.

L'initiation commencera avec le langage C, pourquoi ce choix ?

Nous avons décidé que vous mettriez les "mains dans le cambouis" ! En effet, en langage C vous allez pouvoir opérer sur la mémoire en manipulant des adresses et ainsi avoir une idée de comment on peut la gérer. Ceci est en général fait de manière cachée dans d'autres langages de programmation.

Le langage C, c'est un assemblage de concepts de bases logiques et relativement proches des actions d'un ordinateur moderne pour commencer à appréhender son fonctionnement. De plus le langage C reste un langage qui offre de bonnes performances du fait d'être compilé en langage machine (ordres donnés au processeur par la lecture d'une bande de 0 et de 1). Ceci vous fait ensuite une application directement lisible de la machine, ce qui offre des performances qui dépendent de



vous et non d'un interpréteur.

Le langage C est à la base d'une grande majorité de langages de programmation impérative (instructions exécutées successivement) aujourd'hui. Suite à votre apprentissage du langage C et ses paradigmes vous pourrez prendre du recul sur celui-ci pour mieux appréhender d'autres langages de programmation.

Historiquement, le langage C a été inventé par Brian Kernighan, Dennis Ritchie et Ken Thompson dans l'idée de réécrire le système d'exploitation UNIX en 1970 après une version en assembleur en 1969. Aujourd'hui, ce langage est encore utilisé pour des applications demandant de hautes performances comme les systèmes d'exploitation classiques et embarqués (Linux depuis 1991), les jeux vidéos et moteurs de rendu 3D (comme Blender) par exemple.

Nous noterons quelques versions importantes :

- 1970 : Version pour écrire UNIX (Dennis Ritchie et Kenneth Thompson).
- 1972 : Version distribuée par les laboratoires Bell (Brian Kernighan, Dennis Ritchie et Kenneth Thompson).
- 1989 : Norme ANSI (C89), normalisation pour compatibilité sur toutes les machines.
- 1999 : Norme C99, tableaux de taille variable, commentaires à la C++.
- 2011 : Norme C11, ajout de fonctionnalités pour de la programmation multi-thread.

Dans le cadre de notre cours, nous resterons sur la norme **ANSI**, suffisante pour les paradigmes étudiés, bien que vous soyez invités à explorer d'avantage les fonctionnalités du langage.

### 1.2.2 Compilation

Le langage C reste un langage qui reste entre la compréhension humaine et la possibilité d'interprétation facile par la machine mais ne peut pas être exécuté directement. Pour que la machine puisse effectuer les actions souhaitées, il est nécessaire de transformer ce code en un ensemble de directives que la machine peut exécuter, plus concrètement en langage machine (assimilable au langage assembleur). On dira que ce code directement lisible par la machine est **compilé**.

### 1.2.3 Édition des liens

La dernière étape pour obtenir un **exécutable** fonctionnel est l'édition des liens / **linkage**. Le linkage c'est le référencement vers des fonctionnalités externes au code écrit comme par exemple la bibliothèque standard `stdio.h`. C'est comme avoir à disposition des outils accessibles potentiellement fabriqués par quelqu'un d'autre et des recettes de cuisines ou parties de recettes à disposition. Ceci est une étape d'édition de liens qui indique à l'avance au programme où il faudra aller pour les utiliser.

## 1.3 Première application console

### 1.3.1 Code source .c

Pour indiquer à la machine le comportement à adopter, nous devons écrire du **code source** en langage C. C'est dans un fichier `.c` que nous écrivons les différentes **instructions** en langage C comme illustré dans `hello_esgi.c` ci-dessous. Un tel fichier doit être ouvert et édité avec un environnement de développement C (Integrated development environment : IDE conseillé sous Windows) ou avec le bloc note par défaut et non un logiciel de traitement de texte. Il en existe un grand nombre et le choix s'orientera vers celui qui vous convient le mieux : [Atom](#), [CLion](#), [CodeBlocks](#), [Emacs](#), [Notepad++](#), [Qt Creator](#), [Sublime Text](#), [Vim](#), [Visual Studio Code](#) ou autre.

00\_introduction/hello\_esgi.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      printf("Hello ESGI !\n");
6      exit(EXIT_SUCCESS);
7  }
```

Ici, les deux premières lignes indiquent les bibliothèques où trouver les procédures pour l'affichage et la sortie du programme. C'est comme indiquer à l'avance que l'on va avoir besoin d'ouvrages pour utiliser des recettes de cuisine sans avoir à en écrire leur contenu car ces recettes seront attachées au programme au linkage.

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

La bibliothèque `stdio.h` correspond à la bibliothèque **standard** d'entrées et sorties (*input* and *output*), elle fournit notamment la procédure de `printf` (ligne 5).

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

La bibliothèque `stdlib.h` correspond à la bibliothèque **standard** générale, elle fournit notamment la procédure de `exit` (ligne 6).

```
int main() {  
    ...  
}
```

La procédure `main` est l'entrée du programme : lorsque l'application est lancée, on exécute les instructions qui remplacent les `...` donnés entre les accolades qui suivent `main`.

Ceci fait que lorsque l'application se lance, on exécute la ligne suivante :

```
printf("Hello ESGI !\n");
```

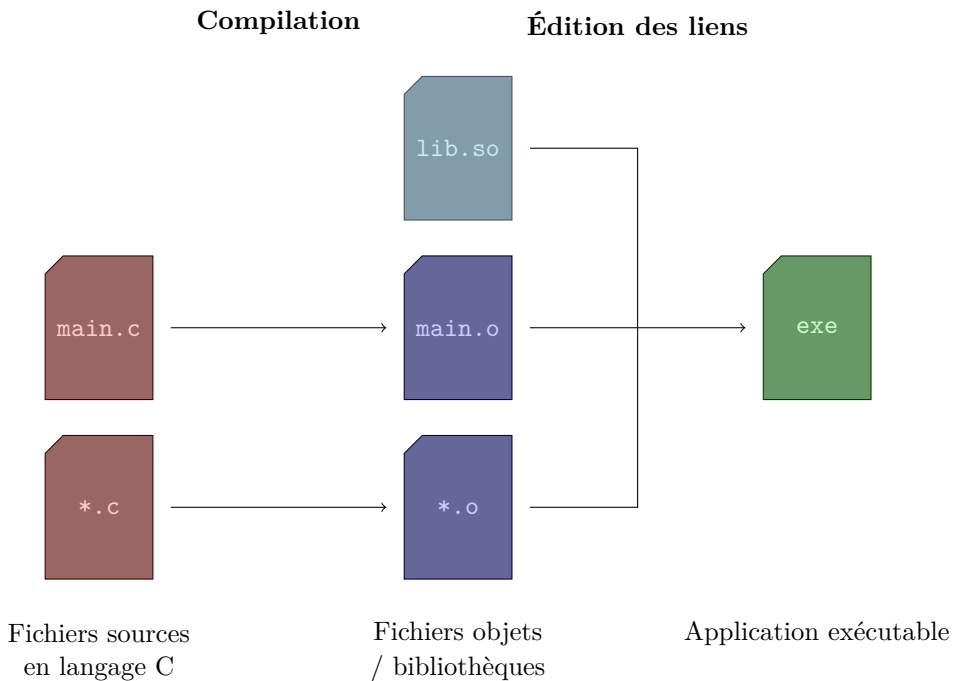
ce qui imprimera à l'écran "Hello ESGI!" suivi d'un retour à la ligne. Puis la ligne :

```
exit(EXIT_SUCCESS);
```

qui termine le programme indiquant que c'est un arrêt normal de celui-ci.

Notons qu'en langage C, lorsqu'on a terminé d'écrire notre instruction, on l'indique à l'aide d'un point virgule `;`. Ceci sera probablement un oubli fréquent au début de votre formation et vous vaudra d'en être informé par le compilateur par `error: expected ';'.`

Cependant ce code en tant que tel ne peut pas être lu directement par la machine. Il nécessite d'être transformé en langage machine à l'aide d'un **compilateur** et d'un **linkeur**. Pas de panique, ceci se fait généralement sans s'en soucier (une commande ou une pression de bouton) à l'aide des outils dédiés.



## 1.4 Préparer ses outils

Bon, c'est bien beau cette théorie, mais ce serait plus pratique que la machine lance le programme histoire de la voir, non ?

Pour ceci vous devrez installer les outils énoncés plus haut : compilateur et éditeur des liens. Notez qu'ils sont en général regroupés comme une même application. Pour suivre au mieux le cours, l'installation de `gcc` vous est recommandée. En cas de difficultés, pas de panique, [Online GDB](#) est un compilateur en ligne qui fera l'affaire.

Notez que vous pourriez apprécier l'auto-complétion de certains outils. Cependant si vous débutez et vu que vous serez évalué sur papier la première année, je vous conseille d'avoir un éditeur avec seulement l'indentation automatique pour vous forcer à prendre en main la syntaxe et ce qui est nécessaire pour faire du code C compilable.

## Activité 2 (★ Compiler Hello ESGI!).

La suite de ce chapitre propose différentes solutions pour installer un compilateur sur votre ordinateur en fonction de votre système d'exploitation.

Choisissez la solution proposée ou extérieure qui vous convient le mieux pour vous permettre de suivre ce cours sans encombre.

En cas de problème bloquant sans solution, notez que vous avez la possibilité d'utiliser un compilateur en ligne : [Online GDB](#).

### 1.4.1 Installation gcc

#### Linux

Sous Linux, nous proposons d'installer `gcc` pour compiler et linker un programme écrit en C.

Ceci à par la saisie des instructions suivantes dans un terminal :

```
sudo apt update
sudo apt install build-essential
sudo apt-get install manpages-dev
```

Il est ensuite possible de vérifier que l'installation est réussie en lançant `gcc` :

```
gcc --version
```

ce qui doit renvoyer un retour similaire au suivant :

```
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
↪ There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
↪ PURPOSE.
```

## Windows : MSYS2

Pour vous permettre de suivre au mieux le cours de la même manière que si vous utilisiez un Linux, je vous conseille d'installer vos outils à l'aide de **MSYS2**. Ici, vous avez à suivre les instructions d'installation puis de lancer MSYS2. Ensuite, vous pourrez saisir les commandes suivantes pour installer vos outils :

```
pacman -Syu
pacman -S --needed base-devel mingw-w64-x86_64-toolchain
pacman -S --needed base-devel mingw-w64-x86_64-toolchain
pacman -S base-devel gcc vim cmake clang
```

Il est ensuite possible de vérifier que l'installation est réussie en lançant `gcc` dans MSYS2 :

```
gcc --version
```

ce qui doit renvoyer un retour similaire au suivant :

```
gcc.exe (Rev3, Built by MSYS2 project) 12.1.0
```

Vous avez maintenant accès à un environnement viable pour travailler en langage C comme sous Linux.

## Mac

Mac est basé sur un noyau Linux, vous devriez pouvoir installer `gcc` de la même manière s'il n'est pas déjà présent :

```
brew update
brew upgrade
brew info gcc
brew install gcc
brew cleanup
```

Vérifier ensuite que l'installation est réussie en lançant `gcc` :

```
gcc --version
```

ce qui doit renvoyer un retour similaire au suivant :

```
Using built-in specs.
Target: i686-apple-darwin11
Configured with: {ignore long text...}
Thread model: posix
gcc version 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
↳ 2336.9.00)
```

### 1.4.2 Compilation avec gcc

Il est possible d'obtenir un exécutable avec `gcc` par la commande

```
gcc -o [APPLICATION] [FICHIERS SOURCES]
```

où `[APPLICATION]` est remplacé par le nom que l'on souhaite donner à l'application et `[FICHIERS SOURCES]` est la liste des fichiers source séparés deux à deux par un espace. Pour compiler et linker notre fichier `hello_esgi.c` en `SuperApp`, dans le répertoire où se trouve le fichier, il faut saisir :

```
gcc -o SuperApp hello_esgi.c
```

`SuperApp` pourra ensuite se lancer par la commande suivante :

```
./SuperApp
```

Ceci donne la réponse suivante dans le terminale :

```
Hello ESGI !
```

le `./` est nécessaire pour dire que l'on commence de l'emplacement courant dans le système de fichier, sinon le système d'exploitation cherchera dans ses commandes et applications installées.

Dans la suite du cours, votre professeur supposera que vous travaillez avec `gcc` sous MSYS2 64 bits ou sous un environnement Linux 64 bits.

### 1.4.3 Mac alternative : Xcode

Vous pouvez aussi installer Xcode pour développer en langage C sous Mac. Un tutoriel est disponible au lien suivant :

<https://www.cs.auckland.ac.nz/~paul/C/Mac/xcode/>.

Notez que le téléchargement de Xcode peut prendre un certain temps.

### 1.4.4 Online GDB : compilateur en ligne

Si aucune des solutions proposées précédemment ne vous convient, l'installation d'outils sur votre ordinateur personnel vous dérange, vous avez un Chrome-Book ou que vous n'avez pas les droits pour installer les outils nécessaires, vous avez toujours la possibilité de travailler en langage C à l'aide d'un compilateur en ligne : [Online GDB](#).

Notez que cet environnement fonctionne sous une machine Linux 64 bits et convient pour suivre ce cours.



## 1.5 Résumé

Quelques mots-clés :

**Algorithme** : Procédé automatique qui depuis une entrée opère dessus pour obtenir une sortie souhaitée.

**Instruction** : Ordre donné à un ordinateur (en langage C, c'est l'ensemble de caractères précédant un point-virgule).

**Programme** : Idée conceptuelle d'instructions ordonnées qui aboutissent à un comportement souhaité.

**Langage de programmation** : Ensemble de mots clés qui permettent l'écriture d'instructions.

**Code source** : Suite d'instructions données dans un langage de programmation qui définissent un programme.

**Binaire** : Représentation d'information comme une suite de 0 et de 1.

**Langage machine** : Suite de valeurs binaires qui indiquent à la machine les actions à effectuer sur les composants de l'ordinateur.

**Exécutable / Application** : fichier donné en langage machine qui réalise un programme.

**Compilation** : Étape de transformation en langage machine d'un code source donné.

**Linkage** : Édition des liens entre différentes parties d'un code source compilé menant à la création d'un exécutable.

Compiler un code source en langage C donné par des fichiers [FICHIERS SOURCES] en une application [APPLICATION] :

```
gcc -o [APPLICATION] [FICHIERS SOURCES]
```

**main** indique au programme où commencer à lire les instructions lors du lancement de l'application.

Ajouter les fonctionnalités d'une bibliothèque C [BIBLIOTHÈQUE] pour les utiliser dans le code source :

```
#include <[BIBLIOTHÈQUE]>
```

Commande issue de `stdio.h` (standard d'entrées et sorties) pour envoyer du texte pour impression dans le terminal :

```
printf("[TEXTE] ");
```

Terminer l'exécution de l'application :

```
exit(EXIT_SUCCESS);
```

## 1.6 Entraînement

Activité 3 (★ Votre Hello ESGI).

1. Créez un fichier `ESGI_1_TP_1_NOM_Prenom.c` en remplaçant `NOM` et `Prenom` par les vôtres.
2. Écrivez un programme qui affiche la sortie suivante (en remplaçant `NOM` et `Prenom` par les vôtres) :

```
ESGI : Bachelor premiere annee  
Seance 1  
NOM Prenom
```

3. Compilez et linkez le programme pour obtenir un exécutable (peu importe le système d'exploitation).
4. Lancez votre programme pour admirer le résultat !



---

## 2 Variables

---

### 2.1 Introduction et motivation

Nous avons vu précédemment un programme qui affiche du texte pour communiquer avec l'utilisateur via une console. Il est possible de donner plus de vie à un programme pour prendre en compte et traiter des données communiquées par l'utilisateur. Pour ceci, je propose de tester le code suivant :

ask\_age.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int age; /* Déclare une variable */
6      printf("Quel est votre age ? "); /* Affiche du texte
   ↪ */
7      scanf("%d", &age); /* Lit une valeur et la met dans
   ↪ 'age' */
8      printf("Vous avez donc %d ans !\n", age); /* Affiche
   ↪ la valeur de 'age' */
9      exit(EXIT_SUCCESS);
10 }
```

Après compilation et exécution, ceci donne par exemple la sortie suivante :

```
# gcc -ansi -Wall -o ask_age ask_age.c
# ./ask_age
Quel est votre age ? 42
Vous avez donc 42 ans !
```

Notons qu'un code en C permet de donner des instructions à la machine, mais aussi de communiquer avec des humains. N'importe qui peut être amené à relire votre code, y compris vous-même (aujourd'hui ou dans quelques temps), pensez donc à laisser des commentaires pour faciliter cette lecture. Un commentaire est une suite de caractères ignorés de la machine et se déclare entre les symboles `/*` et `*/` :

#### Exemple de commentaire

```
1  /* Un commentaire qui ne semble pas utile mais qui */  
2  /* permettra une lecture avec moins de douleur */  
3  /* pour mon professeur et moi dans quelques années.*/
```

Nous avons dans un premier temps demandé à avoir un **variable** `age` dans laquelle ranger un nombre entier. C'est comme avoir un assistant qui traite un ensemble de boîte où ranger des informations. On lui demande au préalable une boîte à laquelle on donne un nom pour pouvoir la redemander plus tard et l'utiliser.

```
int age;
```

À l'aide de `printf` nous imprimons les caractères à envoyer à l'utilisateur pour lui demander son âge. C'est ensuite `scanf` qui permet de scanner la saisie de l'utilisateur pour la traiter.

```
printf("Quel est votre age ? ");  
scanf("%d", &age);
```

Une fois l'information lue et traitée, nous pouvons répondre en partageant l'information récupérée. Donc si l'utilisateur saisit '42', nous enregistrons cette information dans `age` puis nous pouvons communiquer la valeur de `age` qui est '42'.

```
printf("Vous avez donc %d ans !\n", age);
```

À ce point, certaines subtilités sont à remarquer et vont être explicitées ensuite :

1. `age` est précédée d'un mot clé `int` à la ligne 5 (explication en section 2.2).

2. la suite de caractères `%d` est présente pour `printf` et `scanf` aux lignes 7 et 8 (explication en section 2.3).
3. `age` est précédée du symbole `&` dans `scanf` et non dans `printf` (explication en section 2.4).

Une autre subtilité intéressante à remarquer est que les options `-ansi` `-Wall` ont été ajoutées lors de la compilation avec `gcc`. Ceci indique que l'on souhaite respecter la norme ANSI et `-Wall` donne des avertissements pour aider à produire un meilleur code. Ce code peut encore être optimisé en ajoutant l'option `-O2` voir `-O3` pour améliorer les performances du programme dès la compilation.

## 2.2 Variables typées

### 2.2.1 Déclaration

En langage C, une variable possède un type qu'il faut indiquer à l'avance à la machine pour qu'elle sache l'espace mémoire à réserver pour la variable. Ce type permet principalement de paramétrer la taille en mémoire de la variable et choisir de gérer des nombres entiers (une boîte avec un nombre de billes) ou des nombres à virgule flottante (un bidon avec une quantité d'eau).

Un nom de variable ne doit pas commencer par un chiffre et peut être composé de lettres, chiffres et de tiret-bas. Par convention un nom de variable commencera par une lettre minuscule. Veillez à bien nommer vos variables, ça remplace des commentaires et augmente la lisibilité de votre code.

Une telle déclaration se fait sous la forme suivante :

```
type nom_de_variable;  
type nom_de_variable = VALEUR;  
type nom_de_variable1 = VALEUR1, nom_de_variable2 = VALEUR2;
```

D'où l'exemple, vu précédemment nous déclarions une variable avec pour `nom_de_variable` :  
`age` de type `int`.

```
int age;
```

À noter que par défaut une telle déclaration ne permet pas de connaître la valeur de la variable (ceci dépend de ce qui a été écrit dans la mémoire précédemment). Il est donc possible et souvent préférable d'indiquer une valeur par défaut

pour cette variable avant de l'utiliser pour avoir plus de contrôle sur son code. Par exemple :

```
int reponse = 42;
```

Il est aussi possible de déclarer plusieurs variables de même type sur une même ligne en séparant leur déclaration par une virgule :

```
int first = 1, second = 2, third = 3;
```

Si plus tard la variable devait être amenée à changer de valeur, il est possible de lui affecter une nouvelle valeur en utilisant le symbole '='. À bien noter que ce symbole comme précédemment dit 'mettre la valeur de droite à l'emplacement donné à gauche'.

```
reponse = 1337;
```

## 2.2.2 Entiers signés

Le type `int` indique que l'on souhaite manipuler cette variable comme un nombre entier. Il existe cependant d'autres types d'entiers signés avec plus ou moins de contenance :

Type	Espace mémoire	Borne inférieure	Borne supérieure
<code>char</code>	1 octet	-128	127
<code>short</code>	2 octets	-32 768	32 767
<code>int</code>	4 octets	-2 147 483 648	2 147 483 647
<code>long</code>	8 octets	-9 223 372 036 854 775 808	9 223 372 036 854 775 807

À noter que le type `char` est généralement utilisé pour sauvegarder un caractère ASCII. En C, il existe une représentation de caractères en ASCII donnés entre simple apostrophes qui correspondent à la valeur numérique qui les représente :

```
char zero = '0'; /* vaut 48 */  
char lettre_majuscule = 'A'; /* vaut 65 */  
char lettre_minuscule = 'a'; /* vaut 97 */
```

À noter que sur la machine, la mémoire est une longue bande de 0 et de 1 que l'on nomme de bits (binary digits). On considère que nous pouvons représenter cette mémoire par groupes de 8 bits qui correspondent chacun à un octet. Donc 2 octets c'est 16 bits, 4 octets c'est 32 bits et ainsi de suite.



### 2.2.3 Entiers non-signés

Les types `char`, `short`, `int`, `long` sont des types **signés** ceci veut dire que ce sont des entiers qui admettent un signe et donc qui peuvent être négatifs. Il existe aussi la possibilité de déclarer des entiers non signés en précisant le mot-clé `unsigned` devant le type d'entier :

Type	Espace mémoire	Borne inférieure	Borne supérieure
<code>unsigned char</code>	1 octet	0	255
<code>unsigned short</code>	2 octets	0	65 535
<code>unsigned int</code>	4 octets	0	4 294 967 295
<code>unsigned long</code>	8 octets	0	18 446 744 073 709 551 615

### 2.2.4 Nombres à virgule flottante

Cependant les entiers ne permettent pas de représenter directement des nombres à virgule, pour ceci, nous avons des nombres flottants donnés par le type `float`. Un nombre flottant est en réalité une approximation (la mémoire de la machine n'étant pas infinie). Pour améliorer la plage de valeurs que peut prendre cette approximation et la précision de l'approximation, il existe aussi le type `double` qui occupe deux fois la taille d'un `float` comme donné dans la table suivante :

Type	Espace mémoire	Borne inférieure	Borne supérieure
<code>float</code>	4 octets	$-3,402\,823 \cdot 10^{38}$	$3,402\,823 \cdot 10^{38}$
<code>double</code>	8 octets	$-1,797\,693 \cdot 10^{308}$	$1,797\,693 \cdot 10^{308}$
<code>long double</code>	16 octets	$-1,189\,731 \cdot 10^{4\,932}$	$1,189\,731 \cdot 10^{4\,932}$

Pour affecter une valeur flottante à une variable, la notation à virgule est correspond à la notation anglaise : la virgule devient un point et il faut au moins un chiffre de l'un des côtés du point (le reste est considéré rempli de zéros). Le point n'est pas obligatoire si la valeur est entière. Ceci donne par exemple l'affectation suivante :

```
float valeur_flotante = 0.42;
valeur_flotante = .42;
valeur_flotante = 1337.;
valeur_flotante = 1235;
```

Il est aussi possible de donner une valeur en notation scientifique à l'aide d'une syntaxe `[MATISSE]e[EXPOSANT]`. Ceci permet par exemple d'écrire  $42\,000 = 42 \cdot 10^3$  comme `42.e3` :

```
float valeur_flotante = 42.e3;
```

Une subtilité est que dans le cas de `long double`, pour des valeurs qui ne tiennent pas sur un `double`, il faudra préciser un `L` derrière la valeur pour indiquer que l'on souhaite construire un `long double` (par défaut les valeurs flottantes sont des `double`).

```
long double giant = 1e20481L;
```

### 2.2.5 Constantes

Il est possible dans un programme de devoir réutiliser des valeurs constantes que l'on définit une unique fois. Pour éviter de devoir changer cette valeur arbitraire à chaque fois qu'on la trouve dans un programme, il est possible de définir une valeur constante à l'aide du mot-clé `const` pour ne la changer qu'en haut du programme. Cependant le programme ne pourra pas changer cette valeur pendant l'exécution.

```
const int reponse_inalienable = 42;
```

Activité 4 (★ Déclarer et définir des variables).

Compléter le programme suivant en fonction de ce qui est indiqué en commentaire :

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

int main() {

    /* TODO : déclarer un entier ent1 */
    /* TODO : déclarer un nombre à virgule vir1 */
    /* TODO : définir un entier ent2 à valeur 42 */
    /* TODO : définir un entier chr1 à valeur '@' */
    /* TODO : faire prendre à vir1 la valeur 13.37 */
    /* TODO : faire prendre à ent1 la valeur de ent2 */

    exit(EXIT_SUCCESS);
}
```

## 2.3 Format d’affichage

### 2.3.1 Printf

Nous avons vu précédemment que la procédure `printf` de la bibliothèque `stdio.h` permet d’imprimer des caractères dans la console pour communiquer avec l’utilisateur. Cette procédure permet aussi d’imprimer les valeurs de nos variables à l’écran.

`printf` demande dans un premier temps une chaîne de caractère (texte donnée entre guillemets). Dans cette chaîne de caractères, il est possible de préciser des parties que l’on voudra remplacer par des valeurs données par le programme. Ces précisions se font par le caractère `%` et ensuite on donne le format qui indique comme lire et écrire la valeur. Pour imprimer le caractère `'%'`, il faut le doubler : `%%`. Pour chaque format, `printf` attendra une valeur dans la liste qui suit la chaîne de caractères.

Dans notre exemple, nous avons utilisé `%d`, ceci permet d’afficher des entiers `char`, `short`, `int`. Dans le même esprit, il est possible de proposer l’exemple suivant :

```
printf("%d + %d = %d\n", 1, 1, 2);
```

Ce qui affichera dans la sortie de la console la ligne suivante :

```
1 + 1 = 2
```

### 2.3.2 Formats pour entiers

Nous avons aussi vu qu’un caractère est représenté par un entier. Il est naturellement possible d’afficher le nombre associé au caractère par `%d` mais aussi d’afficher le caractère représenté par `%c` :

```
char chiffre = '2';  
int lettre = 'x';  
printf("chiffre = %d : '%c'\n", chiffre, chiffre);  
printf("lettre = %d : '%c'\n", lettre, lettre);
```

En sortie :

```
chiffre = 50 : '2'
lettre = 120 : 'x'
```

Notons que `%d` ne peut afficher que des entiers `int` et de plus petite contenance. Pour afficher des `long`, il sera nécessaire d'ajouter un `l` dans le format pour obtenir le format `%ld` et préciser cette différence.

```
long big = 5000000000000;
printf("big est un entier : %d\n", big);
printf("oups, big porte bien son nom : %ld\n", big);
```

En sortie :

```
big est un entier : 658067456
oups, big porte bien son nom : 5000000000000
```

À noter que le compilateur peut nous avertir à l'aide d'un warning précisant que l'on affiche un 'long int' (`long`) comme un `int` :

```
warning: format '%d' expects argument of type 'int', but argument
↪ 2 has type 'long int' [-Wformat=]
    printf("big est un entier : %d\n", big);
                                ~^
                                %ld
```

Nous avons aussi vu qu'en réalité les nombres sont représentés sous forme binaire en machine. Un format souvent présenté pour représenter le binaire sur des octets est la notation hexadécimale. Pour comprendre cette subtilité, il faut noter que nous représentons nos nombres en base 10, ce qui veut dire que nous avons des chiffres allant de 0 à 9 que nous regroupons devant une unité, dizaine, centaine, puissance de 10 pour former notre nombre :

$$42 = 4 \times 10 + 2 \times 1 = 4 \times 10^1 + 2 \times 10^0$$

Pour comprendre la notation binaire, c'est considérer qu'au lieu de prendre une base 10, nous prenons une base 2 et donc les chiffres sont 0 et 1 en notation binaire.

D'où :

$$\begin{aligned}
 42 &= 32 + 8 + 2 \\
 &= \mathbf{1} \times 32 + \mathbf{0} \times 16 + \mathbf{1} \times 8 + \mathbf{0} \times 4 + \mathbf{1} \times 2 + \mathbf{0} \times 1 \\
 &= \mathbf{1} \times 2^5 + \mathbf{0} \times 2^4 + \mathbf{1} \times 2^3 + \mathbf{0} \times 2^2 + \mathbf{1} \times 2^1 + \mathbf{0} \times 2^0 \\
 42 &= (101010)_2
 \end{aligned}$$

Ceci peut aussi se calculer par des division euclidiennes successives par 2 :

$$\begin{array}{r|l}
 \mathbf{42} & 2 \\
 \hline
 -2 \times 21 & \mathbf{21} \\
 \hline
 0 & -2 \times 10 \\
 \hline
 & \mathbf{10} \\
 & \hline
 & -2 \times 5 \\
 & \hline
 & \mathbf{5} \\
 & \hline
 & -2 \times 2 \\
 & \hline
 & \mathbf{2} \\
 & \hline
 & -2 \times 1 \\
 & \hline
 & \mathbf{1} \\
 & \hline
 & -2 \times 0 \\
 & \hline
 & \mathbf{0} \\
 & \hline
 & \mathbf{1}
 \end{array}$$

→ **101010**

La notation hexadécimale, c'est une notation en base 16 où on étend les chiffres par les lettres : 'a' = 'A' = 10, 'b' = 'B' = 11, 'c' = 'C' = 12, 'd' = 'D' = 13, 'e' = 'E' = 14, 'f' = 'F' = 15. Par exemple :

$$\begin{aligned}
 42 &= 32 + 10 \\
 &= \mathbf{2} \times 16 + \mathbf{10} \times 1 \\
 &= \mathbf{2} \times 16^1 + \mathbf{'a'} \times 16^0 \\
 42 &= (2a)_{16}
 \end{aligned}$$

Ceci peut aussi se calculer par des division euclidiennes successives par 16 :

$$\begin{array}{r|l}
 \mathbf{42} & 16 \\
 \hline
 -16 \times 2 & \mathbf{2} \\
 \hline
 10 & -16 \times 0 \\
 \hline
 & \mathbf{0} \\
 & \hline
 & \mathbf{2} \\
 & \hline
 & \mathbf{2}
 \end{array}$$

→ **2a**

Le passage de la notation binaire à hexadécimale se fait en regroupant par 4 les bits ce qui est équivalent à un chiffre en hexadécimale comme illustré dans la table de correspondances suivante :

Décimal :	0	1	2	3	4	5	6	7
Hexadécimal :	0	1	2	3	4	5	6	7
Binaire :	0000	0001	0010	0011	0100	0101	0110	0111
Décimal :	8	9	10	11	12	13	14	15
Hexadécimal :	8	9	a	b	c	d	e	f
Binaire :	1000	1001	1010	1011	1100	1101	1110	1111

Pour utiliser cette représentation en langage C, on fait précéder le nombre écrit en hexadécimal par 0x pour définir un entier et on utilise le format %x (minuscules) ou %X (majuscules) pour lire le nombre en hexadécimal :

```
int reponse = 0x2a;
printf("La reponse a la vie est %x !\n", reponse);
printf("Ah, pour les humains ? C'est %d !\n", reponse);
```

En sortie :

```
La reponse a la vie est 2a !
Ah, pour les humains ? C'est 42 !
```

Si l'entier n'est pas assez long, il est possible de combler d'espaces les caractères où il manque des chiffres en précisant soit le nombre d'espaces à ajouter après le % pour avoir un nombre de chiffre attendus au minimum :

```
printf("%7d\n", 1);
printf("%7d\n", 1000);
printf("%7d\n", 1000000);
```

En sortie :

```
    1
  1000
1000000
```

Le même, si on ajoute un 0 avant ce nombre, ceci comblera de zéros au lieu d'espaces. Illustrons cet exemple pour visualiser le plus grand entier que l'on peut représenter (en non-signé avec le format %u) :

```
unsigned long giant = 0xffffffffffffffff;
printf("int   : 0x%016x (%u)\n", giant, giant);
printf("long  : 0x%016lx (%lu)\n", giant, giant);
```

En sortie :

```
int   : 0x00000000ffffffff (4294967295)
long  : 0xffffffffffffffff (18446744073709551615)
```

### 2.3.3 Formats pour flottants

Le format d'affichage des nombres flottants est différent de celui des entiers. En effet le format d'affichage se base sur la représentation mémoire de ce qu'on lui donne, celle est flottants est différente de celle des entiers, ce qui demande l'utilisation d'autres formats :

- %f affiche un flottant sous forme numérique.
- %e affiche un flottant en notation scientifique.
- %g propose un affichage automatique (choix de la précision à afficher et passage au besoin en notation scientifique).

```
float v = 12345.6789;
printf("Avec %f : %f\n", v);
printf("Avec %e : %e\n", v);
printf("Avec %g : %g\n", v);
```

En sortie :

```
Avec %f : 12345.678711
Avec %e : 1.234568e+04
Avec %g : 12345.7
```

Avec un nombre plus grand, ceci donnerait :





```
pi vaut environ 3.14
Pas assez precis ? C'est aussi 3.1415926535897932
```

Activité 5 (★ Afficher des variables).

Compléter et modifier si besoin le programme suivant en fonction de ce qui est indiqué en commentaire :

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

int main() {

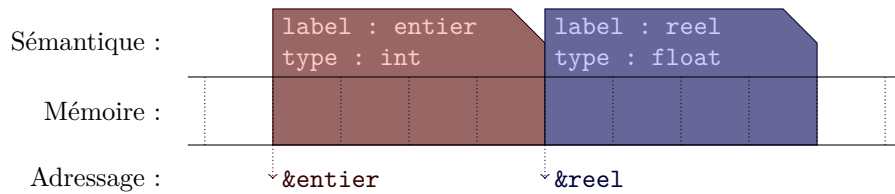
    float reel = 13.37;
    char caractere = 'Z';
    int entier = 42000000000;

    /* TODO : afficher reel */
    /* TODO : afficher caractere */
    /* TODO : afficher caractere en décimal */
    /* TODO : afficher entier : 42000000000 */
    /* TODO : afficher entier en hexadécimal */

    exit(EXIT_SUCCESS);
}
```

## 2.4 Adresse d'une variable

Nous avons dit que la mémoire est une longue bande valeurs binaires. Pour certaines utilisations, il sera nécessaire de savoir où une variable commence sur cette bande. Pour ceci chaque variable a une adresse que la machine lui a attribué à la création de la variable et que l'on peut lui demander. Un peu comme quand on vous demande l'adresse de votre résidence. Cette adresse s'obtient en préfixant le nom de la variable par `&`.



Il est possible d'afficher l'adresse d'une variable avec `%p` :

```
int variable = 42;
printf("Valeur de variable : %d\n", variable);
printf("Adresse de variable : %p\n", &variable);
```

En sortie :

```
Valeur de variable : 42
Adresse de variable : 0x7ffdafbec408
```

## 2.5 Scanf

Cette adresse est utile pour la procédure `scanf` qui lit une valeur saisie par l'utilisateur puis la range à un endroit demandé : d'où l'adresse de la variable pour lui livrer la valeur lue.

`scanf` fonctionne dans la même idée que `printf` : on fournit un format puis la liste des emplacements qui recevront les valeurs lues.

```
int variable = 42;
printf("variable vaut %d\n", variable);
printf("Entrez une nouvelle valeur pour variable : ");
scanf("%d", &variable);
printf("variable vaut maintenant %d\n", variable);
```

En sortie :

```
variable vaut 42
Entrez une nouvelle valeur pour variable : 1337
variable vaut maintenant 1337
```

Nous retrouvons globalement le même principe de formatage qu'avec `printf` :

```
char caractere;
printf("Entrez un caractere : ");
scanf("%c", &caractere);
printf("Voici votre caractere : '%c'\n", caractere);
```

En sortie :

```
Entrez un caractere : #
Voici votre caractere : '#'
```

Pour les flottants, le seul format à garder en tête est `%f`, mais une subtilité apparaît pour gérer les `double` qui réclament un `%lf` et `long double` reste valable avec `%Lf`.

```
float petit_flottant;
double moyen_flottant;
long double grand_flottant;
printf("Entrez trois valeurs flottantes :\n");
scanf("%f %lf %Lf", &petit_flottant, &moyen_flottant,
    ↪ &grand_flottant);
printf("Voici vos valeurs :\n * %g\n * %g\n * %Lg\n",
    ↪ petit_flottant, moyen_flottant, grand_flottant);
```

En sortie :

```
Entrez trois valeurs flottantes :
3.1415 1.e-6 42.e1337
Voici vos valeurs :
* 3.1415
* 1e-06
* 4.2e+1338
```

Activité 6 (★★ Lire et afficher une variable).

Compléter le programme suivant en fonction de ce qui est indiqué en commentaire :

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

int main() {

    double reel;

    /* TODO : demander lecture de reel */
    /* TODO : lire reel */
    /* TODO : afficher reel */

    exit(EXIT_SUCCESS);
}
```

## 2.6 Résumé

Écrire un commentaire :

```
/* [COMMENTAIRE] */
```

Les entiers signés :

```
char caractere; /*  
  de -128  
  à 127  
*/  
short entier_court; /*  
  de -32 768  
  à 32 767  
*/  
int entier; /*  
  de -2 147 483 648  
  à 2 147 483 647  
*/  
long entier_long; /*  
  de -9 223 372 036 854 775 808  
  à 9 223 372 036 854 775 807  
*/
```

Les entiers non-signés :

```
unsigned char octet; /*  
  de 0  
  à 255  
*/  
unsigned short entier_naturel_court; /*  
  de 0  
  à 65 535  
*/  
unsigned int entier_naturel; /*  
  de 0  
  à 4 294 967 295  
*/  
unsigned long entier_naturel_long; /*  
  de 0
```

```
â 18 446 744 073 709 551 615
*/
```

Les flottants :

```
float flottant; /*
de -3.402 823 e 38
â 3.402 823 e 38
*/
double grand_flottant; /*
de -1.797 693 e 308
â 1.797 693 e 308
*/
long double enorme_flottant; /*
de -1.189 731 e 4 932
â 1.189 731 e 4 932
*/
```

Déclarations plus évoluées :

```
type variable = [VALEUR_PAR DEFAULT];
type variable_1, variable_2;
```

Changer la valeur d'une variable :

```
variable = [VALEUR];
```

Constante (même valeur pour toute durée du programme) :

```
const type variable = [VALEUR];
```

Exemples de saisie de valeurs par défaut :

```
char caractere = 'A';
int entier = 42;
float pi = 3.141;
float scientifique = 1.414e20;
long double enorme_flottant = 1.1414e201;
```

Adresse d'une variable :

```
&variable;
```

Formats pour `printf` (impression terminal) :

```
/* char */
printf("%c %d\n", 'A', 'A');
/* short, int */
printf("%d %x %04d '%4d'\n", 42, 0x2a, 42, 42);
/* long */
printf("%ld %lx\n", 42, 0x2a);
/* entier : unsigned */
printf("%u %x\n", 42, 0x2a);
/* float, double */
printf("%f %g %e %.3f\n", 42., 42., 42., 42.);
/* long double */
printf("%Lf %Lg %Le\n", 42.1, 42.1, 42.1);
/* adresse */
printf("%p\n", &variable);
```

Formats pour `scanf` (lecture terminal) :

```
/* char */
scanf("%c %d", &caractere, &caractere);
/* short, int */
scanf("%d %x", &entier, &entier);
/* long */
scanf("%ld %lx", &entier_long, &entier_long);
/* entier : unsigned */
scanf("%u %x", &entier_naturel, &entier_naturel);
/* float */
scanf("%f", &flottant);
/* double */
scanf("%lf", &grand_flottant);
/* long double */
scanf("%Lf", &enorme_flottant);
```

## 2.7 Entraînement

Exercice noté 1 (★ Affectations).

L'ordinateur de votre binôme de projet vient de tomber en panne, mais il a heureusement laissé des instructions pour terminer sa partie, à vous de jouer.

01\_binome.c : Code de votre binôme

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

int main() {

    char car;
    /* TODO : créer un entier ent */
    /* TODO : mettre '4' dans car */
    /* TODO : mettre 2 dans ent */
    /* TODO : afficher le caractère car et ent */
    /* TODO : afficher l'entier correspondant à car */

    exit(EXIT_SUCCESS);
}
```



Des camarades d'une autre classe : Alice, Bob et Charlie ont chacun commencé leur projet de jeu vidéo révolutionnaire en langage C, mais leur code ne fonctionne pas, aider chacun de vos camarades en leur laissant des commentaires pour comprendre leurs erreurs et répondez aux améliorations demandées.

Exercice noté 2 (★ Traduction en Langage C).

02\_alice.c : Code d'Alice

```
import studio

def main :
pi <- 3,14
print(pi)
# Pourquoi ça n'affiche pas pi ???
exit()
```

Alice aimerait :

1. Régler le nombre de décimales après la virgule à deux décimales.
2. Finalement une manière automatique d'afficher le résultat proprement.
3. Et puis en notation scientifique, car c'est une vraie scientifique.

Exercice noté 3 (★★ Dépassement capacité d'un int).

03\_bob.c : Code de Bob

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

int main() {
    int gros_nombre;
    printf("Entrez un gros nombre : ");
    scanf("%d", &gros_nombre);
    printf("%d, un gros nombre ?\n", gros_nombre);
    gros_nombre = 999999999999999999;
    printf("%d, un gros nombre !\n", gros_nombre);
    exit(EXIT_SUCCESS);
}
```

Bob aimerait :

1. Juste des gros nombres entier, les nombres négatifs ne l'intéressent pas, quel type lui permet d'avoir le plus gros nombre entier ?
2. Afficher le plus gros nombre possible avec ce type.
3. Prouver à Bob que c'est le plus gros nombre en l'affichant en hexadécimal.
4. Expliquer à Bob ce qui limite ses gros nombres avec le type `int`.

Exercice noté 4 (★★ Imprécision flottante).

04\_charlie.c : Code de Charlie

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

int main() {
    float racine = 1.414213562373095048;
    printf("racine de 2 vaut %g\n", racine);
    printf("variable : %.18f\n", racine);
    printf("texte      : 1.414213562373095048\n");
    /* C'est différent, étrange ... */
    float clavier;
    printf("Recopiez :\n>1.414213562373095048\n>");
    scanf("%f", &clavier);
    printf("copie : %.18f\n", clavier);
    printf("texte : 1.414213562373095048\n");
    /* C'est la machine qui bug ! */
    exit(EXIT_SUCCESS);
}
```

Charlie aimerait :

1. Pouvoir écrire un nombre aussi précis que son texte dans son programme.
2. Que l'utilisateur puisse saisir un nombre aussi précis au clavier.

**Exercice noté 5 (★★★ Hexadécimal?).**

Oscar est passé en coup de vent nous donner du travail pour vous. Nous avons besoin de votre expertise pour écrire le programme suivant :

1. Sauvegarder fb40 8be9, mais pas sur un long, il dit que ça prend trop de mémoire pour si peu et l'afficher comme un entier positif.
2. Afficher l'hexadécimal (en majuscules) de 42 et '42' à côté.
3. Puis il a terminé sur une blague disant que vous qui êtes informaticien comprendriez, 'if 212 063 991 488 173 then 223 196 547 513 038', que veut-il dire par ces nombres ?

**05\_final.c : Code final**

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

int main() {
    /* À vous de jouer : */

    exit(EXIT_SUCCESS);
}
```

Pensez à créer une archive ESGI\_1\_TP\_2\_NOM\_Prenom.zip avec vos propositions de réponses dans les fichiers 01\_binome.c, 02\_alice.c, 03\_bob.c, 04\_charlie.c et 05\_final.c puis de le déposer à votre professeur sur MyGES.

---

## 3 Expressions

---

Souvent une variable sera une valeur qui sera amenée à changer : c'est-à-dire qu'elle peut être utile pour sauvegarder un résultat ou pour faire évoluer une donnée. Ceci peut être donné par un calcul et l'évaluation d'une expression. Une expression correspond par exemple à l'ensemble de nombres et d'opérations que l'on peut écrire sur une calculatrice pour obtenir un résultat.

### 3.1 Opérations classiques

#### 3.1.1 Addition

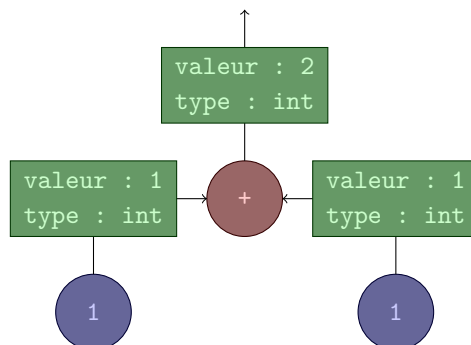
Une opération élémentaire est l'addition.

```
printf("1 + 1 = %d\n", 1 + 1);
```

En sortie :

```
1 + 1 = 2
```

Dans le code suivant,  $1 + 1$  est une expression que la machine évalue pour construire une valeur connue : 2 et peut ensuite l'afficher.

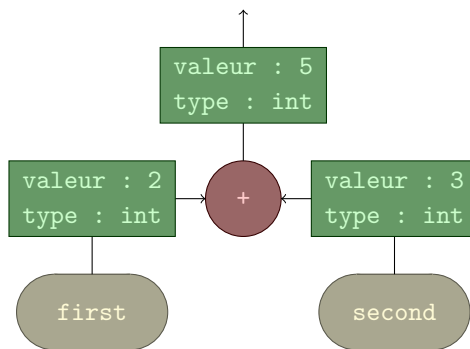


Dans une expression, lorsque le nom d'une variable est présent, celui-ci est remplacé par la valeur de cette variable puis l'expression est évaluée.

```
int first = 2;
int second = 3;
printf("%d + %d = %d\n", first, second, first + second);
```

En sortie :

```
2 + 3 = 5
```



À noter que le résultat de cette expression peut aussi être sauvegardé dans une variable en affectant à une variable le résultat de l'expression.

```
int first = 2;
int second = 3;
int resultat;
resultat = 2 + 3;
printf("%d + %d = %d\n", first, second, resultat);
```

En sortie :

```
2 + 3 = 5
```

### 3.1.2 Soustraction et multiplication

D'autres opérations classiques sont possibles sur les entiers comme la soustraction avec le tiret du milieu '-' ou la multiplication avec l'étoile '\*' :

```
int first = 2;
int second = 3;
printf("%d - %d = %d\n", first, second, first - second);
printf("%d * %d = %d\n", first, second, first * second);
```

En sortie :

```
2 - 3 = -1
2 * 3 = 6
```

À noter que le langage C respecte la règle de priorité de la multiplication sur l'addition. Pour forcer une priorité, ceci se fait à l'aide de parenthèses.

```
printf("2 * 3 + 5 = %d\n", 2 * 3 + 5);
printf("2 * (3 + 5) = %d\n", 2 * (3 + 5));
```

En sortie :

```
2 * 3 + 5 = 11
2 * (3 + 5) = 16
```

Activité 7 (★ Calculer pour l'utilisateur).

Écrire un code qui demande deux nombres réels à l'utilisateur, puis lui afficher :

1. L'addition de ces nombres.
2. La soustraction de ces nombres.
3. La multiplication de ces nombres.

### 3.1.3 Compatibilité entre entiers et flottants

À noter que ces opérations marchent aussi avec les nombres flottants et sont compatibles entre entiers et flottants. La subtilité à prendre en compte avec les nombres flottants est que du fait qu'ils représentent une approximation des nombres à virgule, ils peuvent introduire des imprécisions numériques :

```
float first = 4.9f;
float second = 4.3f;
int third = 1;
printf("%f + %f + %d = %f\n", first, second, third, first + second
↪ + third);
```

En sortie :

```
4.900000 + 4.300000 + 1 = 10.200001
```

## 3.2 Division

### 3.2.1 Division entière

La division peut s'appliquer à l'aide du symbole slash '/'. Lorsqu'elle est appliquée à deux entiers, son résultat est celui de la division entière (le quotient de la division euclidienne) :

```
printf("4 / 2 = %d\n", 4 / 2);
printf("3 / 2 = %d\n", 3 / 2);
```

En sortie :

```
4 / 2 = 2
3 / 2 = 1
```

### 3.2.2 Modulo : reste de la division euclidienne

Comme pour la division euclidienne posée, il est possible de récupérer le reste à l'aide du symbole '%' : opération qui se nommera 'modulo' :

```
printf("7 = %d * 2 + %d\n", 7 / 2, 7 % 2);
```

En sortie :

```
7 = 3 * 2 + 1
```



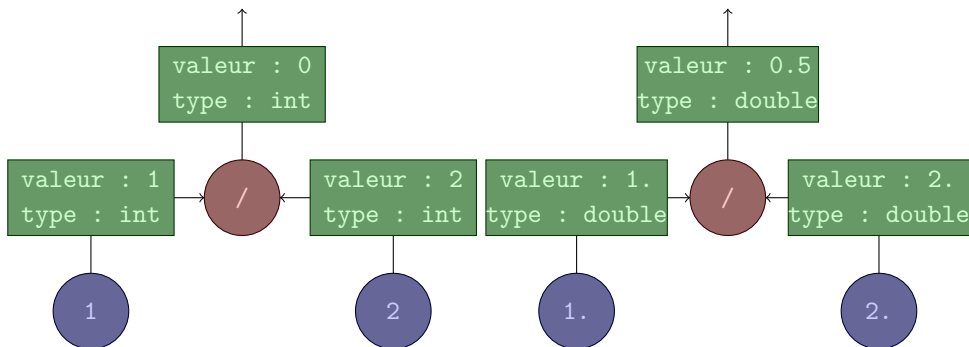
### 3.2.3 Division fractionnaire

Une remarque que l'on peut faire est que si on sauvegarde le résultat d'une division entière dans un flottant, ceci ne permet pas d'avoir un nombre à virgule. Cependant si l'un des deux nombres est flottant nous aurons l'évaluation de la division fractionnaire :

```
float resultat = 1 / 2;
printf("1 / 2 = %g\n", resultat);
printf("1. / 2 = %g\n", 1. / 2);
printf("1 / 2. = %g\n", 1 / 2.);
printf("1. / 2. = %g\n", 1. / 2.);
```

En sortie :

```
1 / 2 = 0
1. / 2 = 0.5
1 / 2. = 0.5
1. / 2. = 0.5
```



À noter qu'en l'absence de parenthèses, le choix de priorité d'évaluations de la division se fait de la gauche vers la droite.

```
printf("2. / 3. / 5. = %g\n", 2. / 3. / 5.);
printf("(2. / 3.) / 5. = %g\n", (2. / 3.) / 5.);
printf("2. / (3. / 5.) = %g\n", 2. / (3. / 5.));
```

En sortie :

```
2. / 3. / 5. = 0.133333
(2. / 3.) / 5. = 0.133333
2. / (3. / 5.) = 3.33333
```

En cas de doute sur l'ordre de priorité dans les évaluation, optez pour un parenthésage explicite.

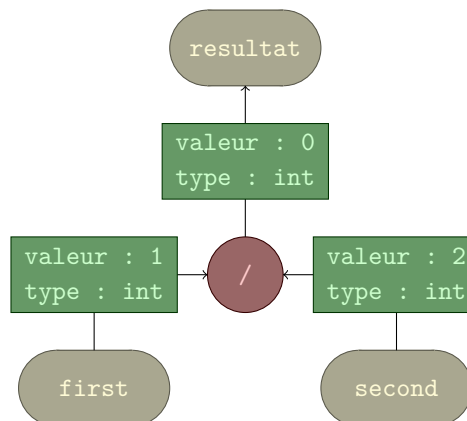
### 3.3 Coercition : un changement de type

Cependant, dans certains cas où l'on souhaite une division fractionnaire, il se peut que les entiers soient issus de variables :

```
int first = 1;
int second = 2;
float resultat;
resultat = first / second;
printf("%d / %d = %g\n", first, second, resultat);
```

En sortie :

```
1 / 2 = 0
```



Pour ceci, il est nécessaire de transformer au moins un entier en nombre flottant. Deux manières en utilisant les outils vus précédemment sont possibles :

1. (À éviter) Sauvegarder la valeur au préalable dans un flottant.
2. (Acceptable) Multiplier par 1. :

```
int first = 1;
int second = 2;
float resultat;
resultat = (first * 1.) / second;
printf("%d / %d = %g\n", first, second, resultat);
```

En sortie :

```
1 / 2 = 0.5
```

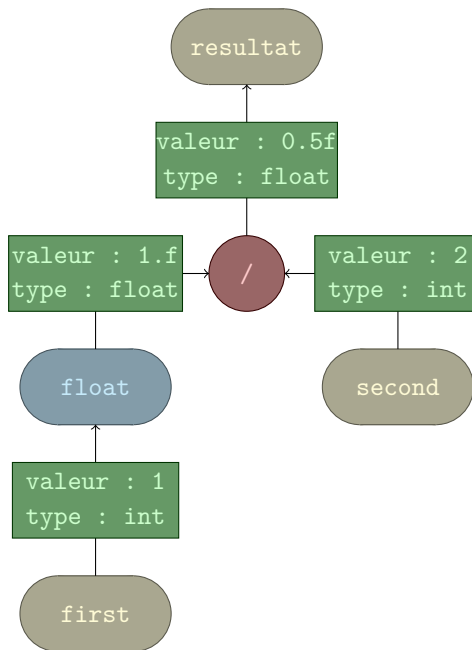
Une autre méthode est de forcer le changement de signe : ceci se fait en précisant le type à obtenir entre parenthèse avant l'expression lors de la conversion.

```
1 int first = 1;
2 int second = 2;
3 float resultat;
4 resultat = (float)first / second;
5 printf("%d / %d = %g\n", first, second, resultat);
```

En sortie :

```
1 / 2 = 0.5
```

Ceci, permet à la ligne 4 de passer de `int` à `float` pour `first` (la conversion de type est prioritaire). Ce concept est la coercion de type, souvent on l'entendra et on en parlera comme le fait de 'caster' un type.



Ce type d'opération est aussi utile pour utiliser un plus grand contenant dans le cas où l'opération dépasse ce que peut supporter le type, par exemple de `int` à `long` :

```

int first = 1000000;
int second = 1000000;
long resultat = first * second;
printf("(sans cast) %d * %d = %ld\n", first, second, resultat);
printf("(avec cast) %d * %d = %ld\n", first, second, (long)first *
↪ second);

```

En sortie :

```

(sans cast) 1000000 * 1000000 = -727379968
(avec cast) 1000000 * 1000000 = 1000000000000

```

### 3.4 Réaffectation d'une variable avec opérateur

Nous avons vu que nous pouvons affecter une valeur à une variable depuis une expression. Ceci se fait souvent pour ajouter 1 à une variable.

```
int nombre = 1;
printf("nombre = %d\n", nombre);
nombre = nombre + 1;
printf("nombre = %d\n", nombre);
```

En sortie :

```
nombre = 1
nombre = 2
```

En réalité ce type d'écriture peut être abrégée : si on applique une opération à une variable, il est possible de le noter comme une affectation avec l'opérateur collé au symbole '=' :

```
int nombre = 1;
printf("nombre = %d\n", nombre);
nombre += 1;
printf("(+ 1) nombre = %d\n", nombre);
nombre *= 5;
printf("( * 5) nombre = %d\n", nombre);
nombre -= 3;
printf("(- 3) nombre = %d\n", nombre);
nombre /= 2;
printf("( / 2) nombre = %d\n", nombre);
```

En sortie :

```
nombre = 1
(+ 1) nombre = 2
(* 5) nombre = 10
(- 3) nombre = 7
(/ 2) nombre = 1
```

Dans le cas où on ajoute 1 ou où on soustrait 1, ceci peut se simplifier par simplement l'opérateur unaire ++ ou -- (avant ou après le nom de la variable).

```
int i = 0;
i++; printf("i = %d\n", i);
++i; printf("i = %d\n", i);
i--; printf("i = %d\n", i);
--i; printf("i = %d\n", i);
```

En sortie :

```
i = 1
i = 2
i = 1
i = 0
```

Il faut noter que ces opérations qui changent la valeur de la mémoire sont dites avec 'effets de bord'. Mais tout comme les opérations classiques, elles renvoient une valeur :

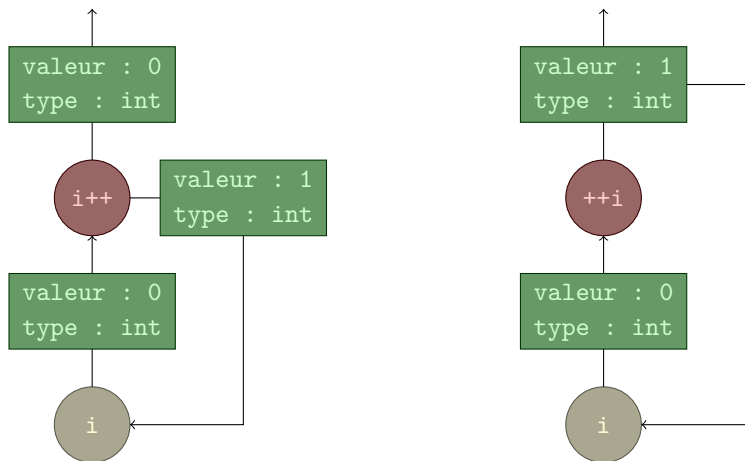
```
int a = 21;
int b = 3;
int c = 4;
printf("%d, %d, %d\n", a, b, c);
c = b = a *= 2; /*c = (b = (a *= 2))*/
printf("%d, %d, %d\n", a, b, c);
```

En sortie :

```
21, 3, 4
42, 42, 42
```

Dans le cas particulier de l'incrémentation ++ et la décrémentation --, la valeur de l'expression dépend du placement de l'opérateur :

- `i++` renvoie la valeur de `i` avant modification.
- `++i` renvoie la valeur de `i` après modification (comme les autres affectations et réaffectations).



```
int i = 1;
printf("i++ = %d\n", i++);
printf("++i = %d\n", ++i);
printf("i = %d\n", i);
```

En sortie :

```
i++ = 1
++i = 3
i = 3
```

### 3.5 Résumé

Opérateurs arithmétiques (entiers) :

```
int a, b;
a + b; // addition
a - b; // soustraction
a * b; // multiplication
a / b; // division entière
a % b; // modulo : reste de la division euclidienne
```

Opérateurs flottants :

```
float a, b;
a + b; // addition
a - b; // soustraction
a * b; // multiplication
a / b; // division fractionnaire
```

Affectation :

```
a = [VALEUR]; /* expression renvoie a */
b = a = [VALEUR];
```

Réaffectations avec opérateur :

```
a = a [OP] b; /* <=> */ a [OP]= b;
a = a + b; /* <=> */ a += b;
a = a - b; /* <=> */ a -= b;
a = a * b; /* <=> */ a *= b;
a = a / b; /* <=> */ a /= b;
a = a % b; /* <=> */ a %= b;
a = a + 1;
/* <=> */ a++; /* renvoie a avant modification */
/* <=> */ ++a; /* renvoie a après modification */
a = a - 1;
/* <=> */ a--; /* renvoie a avant modification */
/* <=> */ --a; /* renvoie a après modification */
```

Conversion de type :



```
(type)variable;
```

Résultat flottant pour division d'entiers :

```
int a, b;  
(a * 1.) / b;  
(float)a / b;
```

Dépassement de capacité d'un int :

```
int a, b;  
long res;  
res = (long)a * b;
```

## 3.6 Entraînement

Exercice noté 6 (\*\* Opérations).

Le chat de votre binôme a renversé son café sur son ordinateur. Vous lui sauveriez la mise en terminant sa partie.

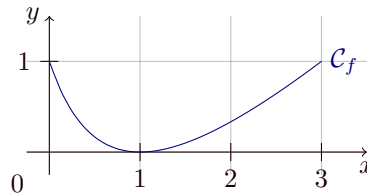
01\_binome.c : Code de votre binôme

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

int main() {
    int a, b;
    /* TODO : demander des valeurs pour a et b */
    /* TODO : afficher l'addition de a et b */
    /* TODO : échanger les valeurs de a et de b */
    /* TODO : afficher la soustraction de a et b */
    long c;
    /* TODO : affecter à c le résultat de la
     ↪ multiplication de a et b */
    float d;
    /* TODO : affecter à d le résultat de la division
     ↪ fractionnaire de a et b */
    exit(EXIT_SUCCESS);
}
```

## Exercice noté 7 (★★ Division par zéro).

Alice est totalement perdue et a besoin de votre aide. Son programme lui dit 'Exception en point flottant (core dumped)'. Elle veut juste calculer des valeurs de la fonction  $x \mapsto \frac{(x-1)^2}{x+1}$ . Elle a réussi à tracer la fonction sur Geogebra pour vérifier et vous donne son graphique. Aidez Alice.



## 02\_alice.c : Code d'Alice

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

int main() {
    int x = 0;
    int y = x - 1 * x - 1 / x + 1;
    printf("f(%d) = %d\n", x, y);
    x = 1;
    y = x - 1 * x - 1 / x + 1;
    printf("f(%d) = %d\n", x, y);
    x = 3;
    y = x - 1 * x - 1 / x + 1;
    printf("f(%d) = %d\n", x, y);
    exit(EXIT_SUCCESS);
}
```

Exercice noté 8 (★★ Affection d'une addition?).

Bob est fier car son programme compile et fonctionne. Que pensez-vous de son code? Pourriez-vous proposer à Bob une autre manière d'écrire son code pour plus de lisibilité et éviter des problèmes inattendus ensuite?

#### 03\_bob.c : Code de Bob

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

int main() {
    long big = 0;
    int ajout;
    printf("big vaut %ld, faites le grossir : ", big);
    scanf("%d", &ajout);
    big = ajout+++big;
    printf("big vaut %ld !\n", big);
    exit(EXIT_SUCCESS);
}
```

## Exercice noté 9 (★★ Imprécision et opérations).

Charlie a un problème, lorsqu'il ajoute un flottant pour avancer de pas en pas, ça ne change pas sa variable. Proposez à Charlie un changement pour faire fonctionner son code.

## 04\_charlie.c : Code de Charlie

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

int main() {
    float resultat = 1.;
    float ajout = 1e-9;
    printf("resultat = %.15f\n", resultat);
    printf("on ajoute %.15f\n", ajout);
    resultat += ajout;
    printf("resultat = %.15f\n", resultat);
    /* Pourquoi resultat ne change pas ? */
    exit(EXIT_SUCCESS);
}
```

**Exercice noté 10 (\*\*\* Message codé).**

Oscar vous indique que  $p = 4\,285\,404\,239$  est un nombre sûr pour échanger des messages secrets sans être lu par vos camarades de classes. Il vous dit que vous pouvez lui envoyer et recevoir des messages de sa part avec la clé  $k = 2\,015\,201\,261$ . Pour ça il faut juste multiplier le nombre à envoyer ou celui reçu par  $k$  puis calculer le reste de la division euclidienne par  $p$ . Pour tester, Oscar vous envoie le message '0x5c003212'. Écrivez un programme qui permet de lire ce message.

**05\_final.c : Code final**

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

int main() {
    /* À vous de jouer : */

    exit(EXIT_SUCCESS);
}
```

Pensez à créer une archive `ESGI_1_TP_3_NOM_Prenom.zip` avec vos propositions de réponses dans les fichiers `01_binome.c`, `02_alice.c`, `03_bob.c`, `04_charlie.c` et `05_final.c` puis de le déposer à votre professeur via MyGES.

---

## 4 Conditions

---

### 4.1 Sélection d'instructions avec if

Selon la configuration des données, il est possible de vouloir choisir des traitements différents. Il est possible de se dire que si un test est vérifié, alors on applique un certain traitement.

Par exemple, pour éviter d'être à découvert, nous pouvons vérifier que nous avons l'argent pour acheter le nouveau jeu vidéo à la mode :

```
const int prix_jeu = 60;
int argent = 0;
printf("Combien avez-vous d'argent ? ");
scanf("%d", &argent);

if(prix_jeu < argent) {
    printf("J'achète le jeu !\n");
} else {
    printf("Il faudra encore economiser ...\n");
}
```

Exemple de sortie 1 :

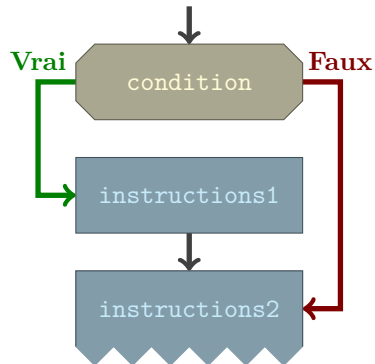
```
Combien avez-vous d'argent ? 42
Il faudra encore economiser ...
```

Exemple de sortie 2 :

```
Combien avez-vous d'argent ? 67
J'achète le jeu !
```

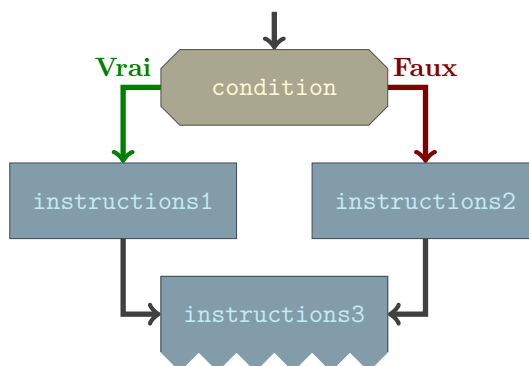
Ici, la syntaxe minimale attendue est celle du `if` auquel il faut fournir une condition de validation. Si cette condition est validée, alors ce qui suit le `if` sera exécuté. Pour cela on placera des accolades `{}` après le `if` dont le contenu sera lu si la condition est vérifiée :

```
if(condition) {  
    /* instructions1 à lire si la condition est vraie. */  
}  
/* instructions2 */
```



Il est aussi possible de préciser un comportement dans le cas où le test de la condition échoue. Pour cela on placera le mot clé `else` après le bloc d'instruction du `if` :

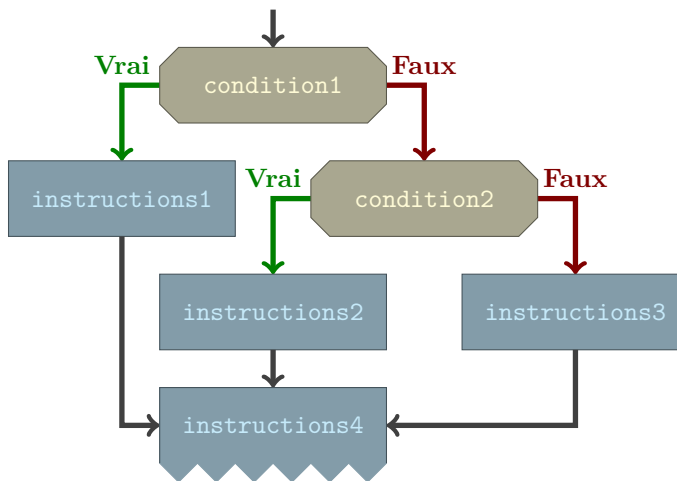
```
if(condition) {  
    /* instructions1 à lire si la condition est vraie. */  
} else {  
    /* instructions2 à lire si la condition est fausse. */  
}  
/* instructions3 */
```





Il est aussi possible de chaîner des tests à réaliser au cas le précédent échoue en choisissant de vérifier une nouvelle condition par un `else if` :

```
if(condition1) {  
    /* instructions1 à lire si la condition 1 est vraie. */  
} else if(condition2) {  
    /* instructions2 à lire si la condition 2 est vraie. */  
} else {  
    /* instructions3 à lire les conditions 1 et 2 sont fausses. */  
}  
/* instructions4 */
```



Ceci permet par exemple d'avoir une alternative dans notre exemple d'achat de jeu :

```
const int prix_jeu = 60;  
const int prix_minecraft = 20;  
  
int argent = 0;  
printf("Combien avez-vous d'argent ? ");  
scanf("%d", &argent);  
  
if(prix_jeu < argent) {  
    printf("J'achète le jeu !\n");  
} else if(prix_minecraft < argent) {
```

```
printf("J'achète minecraft !\n");
} else {
printf("Il faudra encore economiser ...\n");
}
```

En sortie :

```
Combien avez-vous d'argent ? 42
J'achète minecraft !
```

## 4.2 Comparaisons

Les comparaisons sont un autre type d'opérateur qui regarde deux valeurs et renvoie une valeur de vérité si la condition est vérifiée : la valeur retournée sera 0 si c'est faux et une autre valeur si c'est vrai. Cette valeur de vérité peut ensuite être utilisée par notre instruction de contrôle `if` comme une condition.

Un premier opérateur de comparaison est par exemple `<` qui permet de vérifier que le nombre de gauche est strictement plus petit que le nombre de droite :

```
const int petit = 1;
const int grand = 42;
printf("petit < grand = %d\n", petit < grand);
printf("grand < petit = %d\n", grand < petit);
if(petit < grand) {
printf("petit < grand !\n");
}
if(grand < petit) {
printf("grand < petit ???\n");
}
```

En sortie :

```
petit < grand = 1
grand < petit = 0
petit < grand !
```

Notons que l'opérateur `=` effectue l'affectation d'une valeur dans une variable. Pour tester l'égalité, l'opérateur de comparaison est `==`

```
const int deux = 2;
printf("2 == deux ? valeur de vérité : %d\n", 2 == deux);
printf("1 == deux ? valeur de vérité : %d\n", 1 == deux);
```

En sortie :

```
2 == deux ? valeur de vérité : 1
1 == deux ? valeur de vérité : 0
```

Une liste plus exhaustive d'opérateurs de comparaison est la suivante :

Comparaison	Correspondance
<code>a == b</code>	Égalité : vrai si <code>a</code> est la même valeur que <code>b</code>
<code>a != b</code>	Différence : vrai si <code>a</code> est une valeur différente de <code>b</code>
<code>a &lt; b</code>	vrai si <code>a</code> est strictement plus petit que <code>b</code>
<code>a &lt;= b</code>	vrai si <code>a</code> est plus petit ou égal à <code>b</code>
<code>a &gt; b</code>	vrai si <code>a</code> est strictement plus grand que <code>b</code>
<code>a &gt;= b</code>	vrai si <code>a</code> est plus grand ou égal à <code>b</code>

À noter qu'une valeur de vérité peut être sauvegardée dans un variable et passée à une condition :

```
const int deux = 2;
int verification;
verification = deux == 2;
printf("verification = %d\n", verification);
if(verification) {
    printf("deux == 2\n");
} else {
    printf("deux != 2\n");
}
```

En sortie :

```
verification = 1
deux == 2
```

### Activité 8 (★ Déterminer une catégorie d'âge).

Écrire un code qui demande son âge à l'utilisateur puis lui affiche la catégorie des groupes établis selon le cycle de vie correspondante :

- **Enfant** : moins de 14 ans.
- **Adolescent** : de 15 à 24 ans.
- **Adulte** : de 25 à 64 ans.
- **Aîné** : plus de 65 ans.

## 4.3 Opérateurs booléens

Il est possible de construire des expressions booléennes en assemblant des valeurs de vérités par des opérateurs booléens. Ceci est utile par exemple pour vérifier deux conditions en même temps.

### 4.3.1 Intersection

On peut vouloir vérifier que plusieurs conditions sont vraies en même temps. Par exemple si l'on souhaite vérifier qu'un point  $(x, y)$  appartient à une boîte de sommets  $(0, 0)$  et  $(1, 1)$ . Pour faire ceci nous allons utiliser l'opérateur d'intersection 'et' donné en langage C par `&&` :

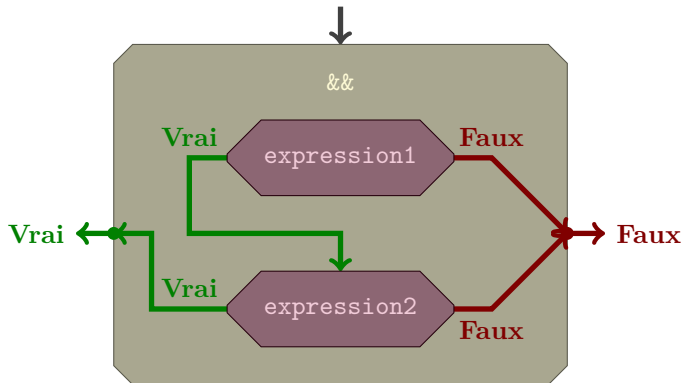
```
float x, y;
printf("Entrez des coordonnées x y : ");
scanf("%f %f", &x, &y);
if((x >= 0) && (x <= 1) && (y >= 0) && (y <= 1)) {
    printf("Bien joué : (%g, %g) est dans la boîte ((0, 0), (1,
        ↪ 1))\n", x, y);
} else {
    printf("Raté : (%g, %g) n'est pas dans la boîte ((0, 0), (1,
        ↪ 1))\n", x, y);
}
```

En sortie :

```
Entrez des coordonnées x y : 0.42 0.37
Bien joué : (0.42, 0.37) est dans la boîte ((0, 0), (1, 1))
```

En sortie :

```
Entrez des coordonnées x y : 1.01 0.5
Raté : (1.01, 0.5) n'est pas dans la boîte ((0, 0), (1, 1))
```



En effet, si une des conditions n'est pas vérifiée, alors le résultat est faux. Il est possible de l'observer en construisant la table de vérité l'opérateur d'intersection :

```
printf("&& | 0 | 1 |\n");
printf("----+----+\n");
printf(" 0 | %d : %d |\n", 0 && 0, 0 && 1);
printf("----+ - + - +\n");
printf(" 1 | %d : %d |\n", 1 && 0, 1 && 1);
printf("----+----+\n");
```

En sortie :

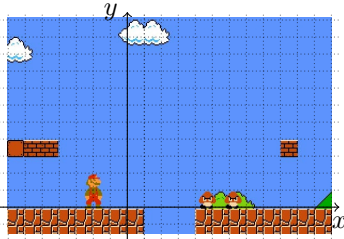
```
&& | 0 | 1 |
----+----+
 0 | 0 : 0 |
----+ - + - +
 1 | 0 : 1 |
----+----+
```

Activité 9 (★ Intersection d'événements).

Écrire un code qui demande son âge à l'utilisateur puis lui indiquer s'il est dans la catégorie des jeunes de 18 à 25 ans.

### 4.3.2 Union

Un autre opérateur est l'opérateur de réunion 'ou inclusif' donné en langage C par `||`. Cet opérateur permet de valider un test si au moins une des valeurs est vraies.

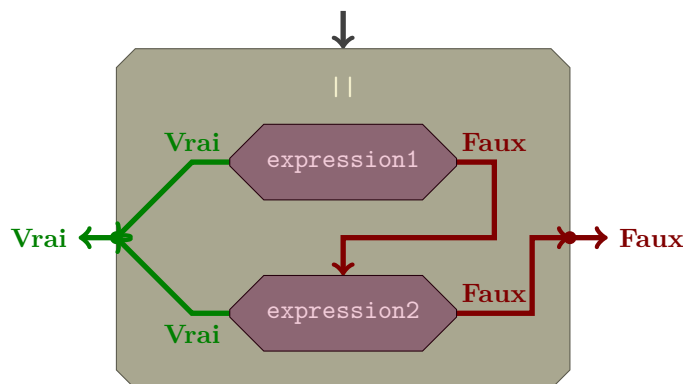


Par exemple, sur l'illustration ci-contre du jeu Super Mario Bros., la plateforme est séparée en deux morceaux. Nous pourrions donc vouloir vérifier que le personnage est sur l'une des plateformes à l'aide de son abscisse  $x$ . En effet, si  $x \leq 1$  le personnage est sur le sol, mais si  $x \geq 4$  est aussi sur le sol. Cette situation peut se traduire par le code suivant :

```
float x_personnage;
printf("Quelle est l'abscisse du personnage ? ");
scanf("%f", &x_personnage);
if((x_personnage <= 1) || (x_personnage >= 4)) {
    printf("Le personnage est sur le sol.\n");
} else {
    printf("Oups, regardez sous vos pieds !\n");
}
```

En sortie :

```
Quelle est l'abscisse du personnage ? -2
Le personnage est sur le sol.
```



De même que pour l'opérateur d'intersection, nous pouvons regarder la table de vérité de l'opérateur de réunion :

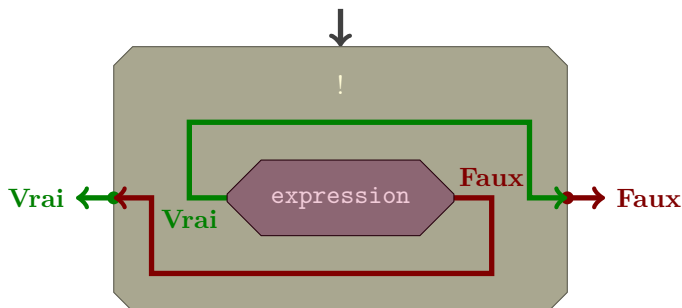
```
printf("|| | 0 | 1 |\n");
printf("---+---+---+\n");
printf(" 0 | %d : %d |\n", 0 || 0, 0 || 1);
printf("---+ - + - +\n");
printf(" 1 | %d : %d |\n", 1 || 0, 1 || 1);
printf("---+---+---+\n");
```

En sortie :

```
|| | 0 | 1 |
---+---+---+
 0 | 0 : 1 |
---+ - + - +
 1 | 1 : 1 |
---+---+---+
```

### 4.3.3 Négation

Un autre opérateur booléen unaire est celui de la négation. Il permet de changer la valeur de vérité d'un résultat et se place en C comme un `!` avant l'expression à évaluer.



Par exemple, on peut avoir une variable qui sauvegarde si un personnage peut lancer un sort et si le sort n'est plus disponible alors il ne peut pas être lancé :

```
int sort_disponible = 1;
if(! sort_disponible) {
```

```
printf("Ce sort ne peut pas être lancé.\n");
} else {
printf("C'est abracadabran !\n");
sort_disponible = 0;
printf("Le sort est maintenant épuisé.\n");
}
```

En sortie :

```
C'est abracadabran !
Le sort est maintenant épuisé.
```

De même une utilisation de l'opérateur de négation peut être utilisée comme expression pour maintenir l'état d'activité d'une case cochée, puis décochée, puis cochée :

```
int case_cochee = 1;
printf("case_cochée ? %d\n", case_cochee);
case_cochee = ! case_cochee;
printf("case_cochée ? %d\n", case_cochee);
case_cochee = ! case_cochee;
printf("case_cochée ? %d\n", case_cochee);
```

En sortie :

```
case_cochée ? 1
case_cochée ? 0
case_cochée ? 1
```

#### Activité 10 (★ Calculer l'amende d'un excès de vitesse).

Un étudiant révise le code de la route et s'interroge sur les excès de vitesse. Proposer un code qui demande vitesse et limitation puis calcule si l'utilisateur risque une amende et un retrait de points :

- Excès de vitesse inférieur à 20 km/h (avec vitesse maximale autorisée supérieure à 50 km/h) :
  - Amende forfaitaire de 68 euros;
  - Retrait d'1 point sur permis de conduire.



- Excès de vitesse inférieur à 20 km/h (avec vitesse maximale autorisée inférieure ou égale à 50 km/h) :
  - Amende forfaitaire de 135 euros ;
  - Retrait d'1 point sur permis de conduire.
- Excès de vitesse égal ou supérieur à 20 km/h et inférieur à 30 km/h :
  - Amende forfaitaire de 135 euros ;
  - Retrait de 2 points sur permis de conduire.
- Excès de vitesse égal ou supérieur à 30 km/h et inférieur à 40 km/h :
  - Amende forfaitaire de 135 euros ;
  - Retrait de 3 points sur permis de conduire.
- Excès de vitesse égal ou supérieur à 40 km/h et inférieur à 50 km/h :
  - Amende forfaitaire de 135 euros ;
  - Retrait de 4 points sur permis de conduire.
- Excès de vitesse supérieur ou égal à 50 km/h :
  - Amende de 1 500 euros ;
  - Retrait de 6 points sur permis de conduire.

## 4.4 Ternaire : expression sous condition

Si nous devons naturellement donner le minimum de deux valeurs, nous utiliserions une comparaison pour tester l'état des variables, puis la structure `if-else` pour sauvegarder la plus petite valeur comme il suit :

```
int a, b;
int minimum;
printf("Entrez deux entiers : ");
scanf("%d %d", &a, &b);
if(a < b) {
    minimum = a;
} else {
    minimum = b;
}
printf("Le minimum de %d et %d est %d\n", a, b, minimum);
```

En sortie :

```
Entrez deux entiers : 42 1337
Le minimum de 42 et 1337 est 42
```

En langage C, il existe une syntaxe nommée ternaire qui reproduit ce schéma de if-else et se comporte comme une expression :

```
[CONDITION] ? [VALEUR SI VRAI] : [VALEUR SI FAUX]
```

Ceci permet d'écrire le code suivant qui se comporte comme le précédent :

```
int a, b;
int minimum;
printf("Entrez deux entiers : ");
scanf("%d %d", &a, &b);
minimum = (a < b) ? a : b;
printf("Le minimum de %d et %d est %d\n", a, b, minimum);
```

En sortie :

```
Entrez deux entiers : 42 1337
Le minimum de 42 et 1337 est 42
```

## 4.5 Switch : se brancher un à résultat

Un autre outil intéressant est le `switch`. Son intérêt est de pouvoir remplacer la if-else dans le cas où on regarde une succession d'égalités comme pour un menu.

```
printf("1 - Jouer\n2 - Options\n3 - Ragequit\n---\nVotre choix ?
↪ ");
scanf("%d", &choix);
if(choix == 1) {
    printf("Que le jeu commence !\n");
} else if(choix == 2) {
    printf("Paramétrons ça.\n");
} else if(choix == 3) {
    printf("Bien, au revoir !\n");
} else {
```

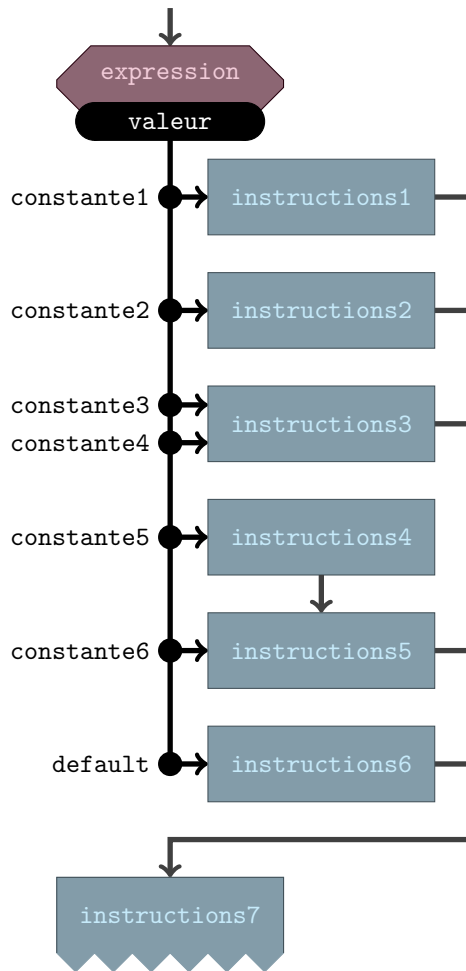
```
printf("Hum, le choix %d, je ne comprends pas ce choix...\n",  
    ↪ choix);  
}
```

En sortie :

```
1 - Jouer  
2 - Options  
3 - Ragequit  
---  
Votre choix ? 1  
Que le jeu commence !
```

La syntaxe du `switch` peut se substituer à ce code : elle regarde une expression et saute à le label de la constante correspondant au résultat. À noter que pour sortir du `switch`, nous utiliserons l'instruction `break`; sinon le programme lit le code au label suivant :

```
switch(expression) {  
    case [CONSTANTE 1] :  
        /* si expression == [CONSTANTE 1] */  
        break;  
    case [CONSTANTE 2] :  
        /* si expression == [CONSTANTE 2] */  
        break;  
    case [CONSTANTE 3] :  
    case [CONSTANTE 4] :  
        /* si expression == [CONSTANTE 3] ou expression == [CONSTANTE  
    ↪ 4] */  
        break;  
    case [CONSTANTE 5] :  
        /* si expression == [CONSTANTE 5] */  
    case [CONSTANTE 6] :  
        /* si expression == [CONSTANTE 5] ou [EXPRESSION] ==  
    ↪ [CONSTANTE 6] */  
        break;  
    /* ... */  
    default :  
        /* équivalent du else */  
}
```



Dans le cas de notre exemple de menu, ceci donnerait le code suivant :

```

int choix;
printf("1 - Jouer\n2 - Options\n3 - Ragequit\n---\nVotre choix ?
↪ ");
scanf("%d", &choix);
switch(choix) {
    case 1 :
        printf("Que le jeu commence !\n");
        break;
    case 2 :

```

```
    printf("Paramétrons ça.\n");  
    break;  
case 3 :  
    printf("Bien, au revoir !\n");  
    break;  
default :  
    printf("Hum, le choix %d, je ne comprends pas ce choix...\n",  
        ↪ choix);  
}
```

En sortie :

```
1 - Jouer  
2 - Options  
3 - Ragequit  
---  
Votre choix ? 3  
Bien, au revoir !
```

## 4.6 Résumé

Structure de contrôle sous condition :

```
if(condition1) {  
    /* Instructions si condition1 vraie. */  
} else if(condition2) {  
    /* Instructions si condition1 fausse mais condition2 vraie. */  
} else {  
    /* Instructions si aucune des conditions n'est vraie. */  
}
```

Tests de comparaison de nombres :

```
a == b; /* a est égal à b ? */  
a != b; /* a est différent de b ? */  
a < b; /* a est plus petit ou égal b ? */  
a <= b; /* a est strictement plus petit que b ? */  
a > b; /* a est plus grand ou égal b ? */  
a >= b; /* a est strictement plus grand que b ? */
```

Opérateurs booléens :

```
a && b; /* ET : a et b sont vraies ? */  
a || b; /* OU : a ou b est vraie ? */  
!a; /* NON : a n'est pas vraie ? */
```

Expression ternaire :

```
[CONDITION] ? [VALEUR SI VRAIE] : [VALEUR SI FAUSSE];
```

Switch :

```
switch([EXPRESSION]) {  
    case [CONSTANTE 1] :  
        /* Instructions si [EXPRESSION] == [CONSTANTE 1] */  
        break;  
    /* ... */  
    default :  
        /* Instructions par défaut */  
}
```

## 4.7 Entraînement

Exercice noté 11 (\*\*\* Acheter un article).

Alice fait les soldes mais les vendeurs ont oublié d'afficher le prix soldé. Elle aimerait faire un programme pour calculer automatiquement si elle peut acheter un article. Pour ceci, elle fournit, pour chaque article, les informations suivantes :

1. Son argent.
2. Le prix de l'article hors soldes.
3. Le taux de remise en pourcentage.

Ceci pourrait donner la sortie suivante :

```
Votre argent : 42
Le prix de l'article (hors soldes) : 50
Remise en % : -30
L'article en solde vaut 35
J'achète !
```

**Exercice noté 12 (★★★ Calculatrice).**

Votre binôme vous laisse faire le menu, il complétera le reste après votre passage.

Le menu doit proposer les options suivantes :

1. Calculer.
2. Quitter.

Finalement, votre binôme n'a pas le temps de faire la partie calculer, il a piscine. Si l'utilisateur choisi 'Calculer', vous devez lui proposer d'entrer des entiers et une opération sous la forme '[NOMBRE 1] [OPERATEUR] [NOMBRE 2]' et calculer. [OPERATEUR] peut être l'addition, la soustraction, la multiplication, la division ou le modulo. Ce qui donne quelque chose comme la sortie suivante :

```
1 - Calculer
2 - Quitter
---
Votre choix : 1
>>> 6 * 7
6 * 7 = 42
```



---

**Exercice noté 13 (★★ Racines d'un polynôme du second degré).**

Bob aimerait automatiser la récupération des racines réelles d'un polynôme du second degré  $ax^2 + bx + c$  pour faire un jeu avec des lancers de projectiles depuis son cours de mathématiques du lycée. On rappelle que :

- Si  $a \neq 0$ , on regarde le discriminant  $\Delta = b^2 - 4ac$ .
  - Si  $\Delta < 0$ , il n'y a pas de solution réelle.
  - Si  $\Delta = 0$ ,  $x_0 = \frac{-b}{2a}$  est l'unique solution.
  - Sinon,  $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$  et  $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$  sont les deux solutions.
- Sinon si  $b \neq 0$ , il y a une solution  $x_0 = \frac{-c}{b}$ .
- Sinon c'est une fonction constante.

Proposez un code à Bob pour répondre à sa problématique.

Notez que  $\sqrt{\phantom{x}}$  se trouve comme `sqrt` dans la bibliothèque `math.h`. Dans la commande de compilation `gcc`, il faudra ajouter `-lm` pour l'utiliser.

Exercice noté 14 (\*\*\* Code obscurantiste).

Charlie se prépare pour l'**IOCCC**, une grande douleur pour son binôme qui essaie de le lire et ne comprend pas son code. Étudiez le concept proposé par Charlie, puis proposez un code tel que vous l'auriez écrit pour qu'il soit lisible d'un camarade.

04\_charlie.c : Code de Charlie

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

int main() {
    int a, b;
    printf("Entrez deux nombres : ");
    scanf("%d %d", &a, &b);
    if(!(a - b))
        printf("%d et %d sont égaux\n", a, b);
    else
        printf("Entre %d et %d, le plus petit est %d\n",
            ↪ a, b, ((long)(unsigned int)(a - b) == a - b) ?
            ↪ b : a);
    exit(EXIT_SUCCESS);
}
```

**Exercice noté 15 (★★ Validation d'un mot de passe).**

Robert veut faire un système avec deux authentifications. L'utilisateur doit fournir les codes secrets '42' et '1337' pour avoir accès aux fonctionnalités de son logiciel. Il se réserve aussi un mot de passe administrateur '1235' qu'il faut écrire comme les deux codes secrets. Les combinaisons suivantes donnent accès au programme :

- 42 1337
- 1337 42
- 1235 1235

Robert ne souhaite pas que vous touchiez au code où il n'a pas placé de commentaires. Complétez son code pour l'aider à mettre en place ce système.

**05\_robert.c : Code de Robert**

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

int main() {
    int first_pass = 0, second_pass = 0;

    if(first_pass > second_pass) {
        /* Echanger first_pass et second_pass */

    }
    if(/* Condition : si mot de passe erroné */

    ) {
        printf("Accès refusé.\n");
        exit(EXIT_SUCCESS);
    }
    printf("Bienvenue !\n");
    exit(EXIT_SUCCESS);
}
```



---

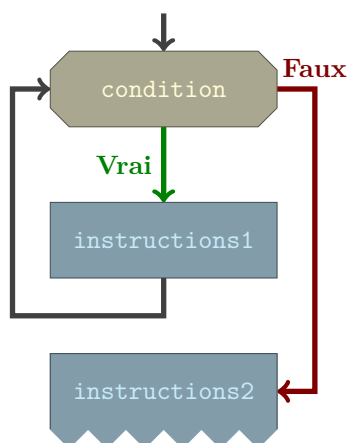
## 5 Boucles

---

Dans votre code, vous pouvez être amené à répéter un procédé jusqu'à arriver au résultat attendu. Pour ceci nous utiliserons des boucles. Les boucles permettent de lire des instructions et de relancer la lecture de celles-ci si besoin.

### 5.1 While : répétition sous condition

Une première boucle est la boucle `while` celle-ci fonctionne dans le même esprit que le `if` : si une condition est valide, on lit les instructions. La différence est que lorsque les instructions ont été lues, le `if` continue sur le code qui suit alors que la boucle `while` retourne vérifier si la condition est toujours vraie. Dans le cas où la condition est toujours vraie, elle exécute à nouveau les instructions qui la suivent. Elle se donne par la syntaxe suivante :



```
while(condition) {  
    /* instructions1 tant que condition est vraie */  
}  
/* instructions2 */
```

Par exemple, si on souhaite demander un nombre positif à un utilisateur, il est possible de lui redemander si sa saisie ne satisfait pas les attentes de notre programme :

```
int nombre = 0;  
printf("Entrez un nombre positif s'il-vous-plaît : ");  
scanf("%d", &nombre);  
while(nombre < 0) {  
    printf("Je répète, entrez un nombre positif s'il-vous-plaît :  
    ↪ ");
```

```
scanf("%d", &nombre);  
}  
printf("Oh, un nombre positif : %d\n", nombre);
```

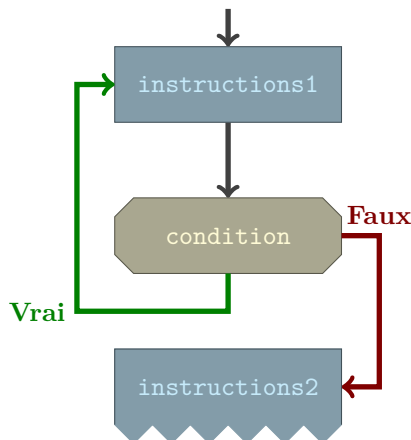
En sortie :

```
Entrez un nombre positif s'il-vous-plaît : -1  
Je répète, entrez un nombre positif s'il-vous-plaît : -2  
Je répète, entrez un nombre positif s'il-vous-plaît : 42  
Oh, un nombre positif : 42
```

## 5.2 Do while : répétition si besoin

Pour le type d'utilisation vue précédemment, un autre type de boucle peut-être judicieux : la syntaxe `do while`, celle-ci permet de placer la condition à la fin : ceci veut dire que le code sera lu peu importe la condition puis répété tant la condition reste vraie. Sa syntaxe est la suivante :

```
do {  
    /* instructions1 répétées tant que condition est vraie */  
} while(condition);  
/* instructions2 */
```



Dans l'exemple vu précédemment, ceci donnerait :

```
int nombre = 0;
do {
    printf("Entrez un nombre positif s'il-vous-plaît : ");
    scanf("%d", &nombre);
} while(nombre < 0);
printf("Oh, un nombre positif : %d\n", nombre);
```

En sortie :

```
Entrez un nombre positif s'il-vous-plaît : -1
Entrez un nombre positif s'il-vous-plaît : 42
Oh, un nombre positif : 42
```

### 5.3 For : un pas après l'autre

Souvent, nous utiliserons les boucles pour itérer sur des procédés et encore plus souvent pour compter de 0 à une valeur donnée. Ceci peut se faire avec une boucle `while` :

```
int compteur = 0;
while(compteur < 5) {
    printf("Le compteur vaut %d\n", compteur);
    ++compteur;
}
printf("Et voilà !\n");
```

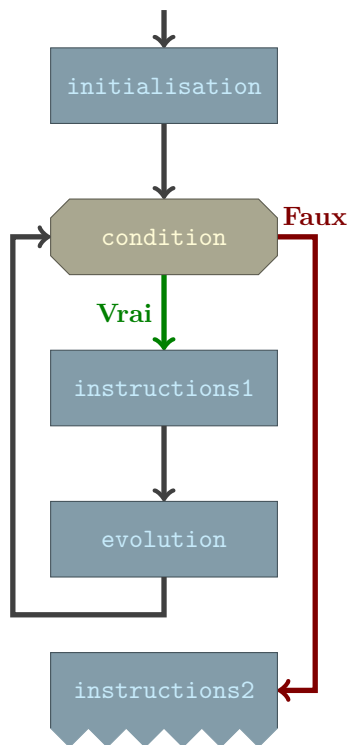
En sortie :

```
Le compteur vaut 0
Le compteur vaut 1
Le compteur vaut 2
Le compteur vaut 3
Le compteur vaut 4
Et voilà !
```

La réalité est que souvent la valeur donnée au compteur ou l'incréméntation du compteur seront oubliés ce qui conduira à des résultats étranges ou des boucles infinies. Une syntaxe qui convient à ce type de procédé est la boucle `for` : elle

embarque une initialisation, une condition comme la boucle `while` et le procédé d'itération. Sa syntaxe est donnée par :

```
for(initialisation; condition; evolution) {  
    /* instructions1 tant que condition est vraie */  
}  
/* instructions2 */
```



Ceci fait que l'on peut utiliser la boucle `for` pour produire le même résultat que précédemment comme il suit :

```
int compteur;  
for(compteur = 0; compteur < 5; ++compteur) {  
    printf("Le compteur vaut %d\n", compteur);  
}  
printf("Et voilà !\n");
```



En sortie :

```
Le compteur vaut 0
Le compteur vaut 1
Le compteur vaut 2
Le compteur vaut 3
Le compteur vaut 4
Et voilà !
```

Une convention souvent utilisée sera de commencer à compter à partir de 0. Nous pouvons imbriquer des boucles par exemple pour dessiner un triangle :

```
int ligne, colonne;
for(ligne = 1; ligne <= 5; ++ligne) {
    for(colonne = 1; colonne <= ligne; ++colonne) {
        printf("*");
    }
    printf("\n");
}
```

En sortie :

```
*
**
***
****
*****
```

## 5.4 Contrôle de la boucle : break or continue

Il est possible de forcer l'interruption d'une boucle à l'aide du mot clé **break**. Par exemple si on demande un nombre positif, mais que l'utilisateur refuse de coopérer :

```
int nombre = 0;
int repetitions = 0;
do {
    printf("Entrez un nombre positif s'il-vous-plaît : ");
    scanf("%d", &nombre);
}
```

```
++repetitions;
if(repetitions > 3) {
    printf("Ça suffit !\n");
    break;
}
} while(nombre < 0);

if(nombre < 0) {
    printf("Ragequit.\n");
    exit(EXIT_FAILURE);
}
printf("Oh, un nombre positif : %d\n", nombre);
exit(EXIT_SUCCESS);
```

En sortie :

```
Entrez un nombre positif s'il-vous-plaît : -4
Entrez un nombre positif s'il-vous-plaît : -42
Entrez un nombre positif s'il-vous-plaît : -421
Entrez un nombre positif s'il-vous-plaît : -4210
Ça suffit !
Ragequit.
```

Il est aussi possible de forcer la boucle à se relancer depuis la vérification de la condition à l'aide du mot clé `continue`. Ceci est utile par exemple pour ne pas faire une partie des instructions de la boucle si une condition est vérifiée :

```
int nombre = 0;
int repetitions = 0;
do {
    printf("Entrez un nombre positif : ");
    scanf("%d", &nombre);
    ++repetitions;
    if(repetitions < 2) {
        continue;
    }
    printf("Entrez un nombre négatif : ");
    scanf("%d", &nombre);
    nombre *= -1;
} while(nombre < 0);
```

```
printf("Oh, un nombre positif : %d\n", nombre);
```

En sortie :

```
Entrez un nombre positif : -1
Entrez un nombre positif : -4
Entrez un nombre négatif : -42
Oh, un nombre positif : 42
```

## 5.5 Résumé

Boucle while : répète les instructions tant qu'une condition est valide :

```
while([CONDITION]) {  
    [INSTRUCTIONS]  
}
```

Boucle do while : exécute les instructions puis répète ces instructions tant qu'une condition est valide :

```
do {  
    [INSTRUCTIONS]  
} while([CONDITION]);
```

Boucle for : permet une initialisation, répète les instructions tant qu'une condition est valide et effectue une mise à jour à chaque fin d'exécution des instructions :

```
for([INITIALISATION]; [CONDITION]; [EVOLUTION]) {  
    [INSTRUCTIONS]  
}
```

Interruption d'une boucle :

```
[BOUCLE] {  
    break; /* sort de la boucle */  
}
```

Relance d'une boucle :

```
[BOUCLE] {  
    continue; /* retourne à la vérification de la condition */  
}
```

## 5.6 Entraînement

Exercice noté 16 (★★ Liste des diviseurs).

Écrire un programme qui affiche la liste des diviseurs d'un nombre comme dans la sortie suivante :

```
Entrez un entier : 42
Liste des diviseurs de 42 :
1, 2, 3, 6, 7, 14, 21, 42
```

Exercice noté 17 (★★★ Force brute).

Alice trouve que Oscar fait trop le malin avec son programme de messages codés. Pour rappel, Oscar envoie des messages qu'il faut multiplier par  $k = 2\,015\,201\,261$  modulo  $p = 4\,285\,404\,239$ . Robert vous indique que la clé secrète de Oscar multipliée avec  $k$  modulo  $p$  doit faire 1 et que sa méthode brute force lui trouve cette clé en environ 30 secondes. Proposez un programme qui trouve la clé secrète d'Oscar.

Exercice noté 18 (★★★ Affichage en binaire).

Bob aimerait demander un nombre entier à l'utilisateur et afficher sa représentation en binaire. Robert lui dit que soustraire les puissances de 2 depuis la plus grande possible permet de le faire. Voici un extrait du résultat du programme de Robert :

```
Entrez un nombre : 42
42 = (101010)_2
```

Exercice noté 19 (★★ PGCD).

Charlie aimerait implémenter l'algorithme d'Euclide pour calculer le Plus Grand Diviseur Commun et en afficher les différentes étapes comme dans la sortie ci-dessous. Écrire le programme qui réalise cette sortie.

```
Entrez deux entiers : 1337 42
1337 = 42 * 31 + 35
42 = 35 * 1 + 7
35 = 7 * 5 + 0
pgcd(1337, 42) = 7
```

**Exercice noté 20 (\*\*\* Jeu du plus ou moins).**

Oscar vous propose de l'affronter à un jeu : il va choisir un nombre aléatoirement entre 0 et 1000 et vous devrez le deviner. Les seules indications qu'il pourra vous donner est de savoir si le nombre que vous donnez est plus petit ou plus grand que le sien. Préparez un programme contre lequel jouer. Ceci peut par exemple donner la sortie suivante :

```
Nous avons choisi un nombre entre 0 et 1000
A quel nombre pensez-vous ? 500
Trop petit.
A quel nombre pensez-vous ? 750
Trop grand.
A quel nombre pensez-vous ? 625
Trop grand.
A quel nombre pensez-vous ? 562
Trop petit.
A quel nombre pensez-vous ? 593
Trop petit.
A quel nombre pensez-vous ? 609
Bien joué, le nombre était en effet 609
```

Robert vous propose le code suivant pour simuler de l'aléatoire :

**05\_robert.c : Code de Robert**

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

int main() {
    srand(time(NULL));
    const int max = 1001;
    int nombre = rand() % max;
    /* À vous de jouer */
    exit(EXIT_SUCCESS);
}
```

# Troisième partie

## Notions de base





# Table des matières

<b>6</b>	<b>Fonctions</b>	<b>115</b>
6.1	La fonction main . . . . .	115
6.2	Définition de fonctions . . . . .	117
6.3	Portée des variables . . . . .	120
6.3.1	Variables locales à une fonction . . . . .	120
6.3.2	Variables globales . . . . .	124
6.3.3	Portée des variables . . . . .	124
6.4	Déplacer sa définition de fonction . . . . .	126
6.4.1	Déclaration d'une fonction . . . . .	126
6.4.2	Appels mutuels entre fonctions interdépendantes . . . . .	127
6.5	Résumé . . . . .	128
6.6	Entraînement . . . . .	129
<b>7</b>	<b>Tableaux</b>	<b>137</b>
7.1	Tableau à une dimension . . . . .	137
7.2	Chaînes de caractères . . . . .	141
7.3	Tableau à plusieurs dimensions . . . . .	142
7.4	Résumé . . . . .	146
7.5	Entraînement . . . . .	147
<b>8</b>	<b>Pointeurs</b>	<b>153</b>
8.1	Un type d'adresse . . . . .	153
8.2	Arithmétique des pointeurs . . . . .	155
8.3	Allocation dynamique . . . . .	158
8.4	Résumé . . . . .	161
8.5	Entraînement . . . . .	163

<b>9</b>	<b>Projet : labyrinthe</b>	<b>173</b>
9.1	Sujet . . . . .	173
9.2	Évaluation . . . . .	174
9.2.1	(... / 6 points) Aspects code . . . . .	174
9.2.2	(... / 6 points) Aspects fonctionnalités . . . . .	174
9.2.3	(... / 15 points) Améliorations . . . . .	175

---

## 6 Fonctions

---

Dans ce que nous avons présenté, nous laissons supposer que tout le code pouvait s'écrire dans les accolades suivant le `main()`. Cependant, plus le code grandit plus il devient difficile à lire, à maintenir et certains procédés se répètent. Un vrai casse-tête. En réalité, nous pouvons créer des procédés que nous pouvons utiliser sans avoir à les recoder à chaque utilisation. C'est le cas de `printf` et `scanf` que nous avons emprunté à la bibliothèque `stdio.h`. Ces procédures se nomment **fonctions**.

### 6.1 La fonction main

Pour définir une fonction, il faut lui donner un nom, un type de retour et des arguments par la syntaxe suivante :

```
typeDeRetour nomDeLaFonction (type1 argument1, type2 argument2,  
↪ ..., typeN argumentN) {  
    /* instructions */  
}
```

Tout comme nous l'avions fait sans le savoir en définissant la fonction de `main`, de type de retour `int` et sans arguments :

```
int main() {  
    /* Nos instructions. */  
}
```

En réalité, la fonction `main` peut renvoyer une valeur à la fin à l'aide du mot clé `return` qui en précise la valeur au lieu d'utiliser `exit`. Par convention 0 est utilisé pour dire que tout s'est bien passé et une autre valeur est considérée comme un code d'erreur :

```
int main() {  
    /* Nos instructions. */  
    return 0;  
}
```

Cette fonction `main` peut en réalité aussi être définie avec des arguments :

```
int main(int argc, char * argv[]) {  
    /* Nos instructions. */  
}
```

Les arguments que nous pouvons donner à la fonction `main` permettent de récupérer données ajoutées lors du lancement du programme en ligne de commande ou si des fichiers sont glissés sur le programme pour l'ouvrir. `main_arguments.c` propose de visualiser les arguments reçus par la fonction.

main\_arguments.c

```
1  #include <stdio.h>  
2  #include <stdlib.h>  
3  
4  int main(int nombre_arguments, char * valeurs_arguments[])  
    ↪ {  
5      int i;  
6      printf("%d arguments :\n", nombre_arguments);  
7      for(i = 0; i < nombre_arguments; ++i) {  
8          printf(" - \"%s\"\n", valeurs_arguments[i]);  
9      }  
10     exit(EXIT_SUCCESS);  
11 }
```

En sortie :

```
# gcc -ansi -Wall -O2 -o main_arguments main_arguments.c  
# ./main_arguments argument 1 "argument 2"  
4 arguments :  
- "./main_arguments"  
- "argument"  
- "1"  
- "argument 2"
```

Note : Pour le moment, ne pas se focaliser sur le type `valeurs_arguments` ceci sera étudié dans les deux chapitres qui suivent.

## 6.2 Définition de fonctions

Nous pouvons aussi définir nos propres fonctions. Par exemple, une fonction `saluer` qui imprimera "Bonjour" dans la sortie de la console lorsqu'elle est appelée :

```
void saluer() {
    printf("Bonjour !\n");
}

int main() {
    saluer();
    saluer();
    exit(EXIT_SUCCESS);
}
```

En sortie :

```
Bonjour !
Bonjour !
```

Ici, son type de retour est `void` ce qui signifie que la fonction ne renvoie aucune valeur. Cependant une fonction peut renvoyer une valeur lorsqu'elle est appelée. C'est par exemple le cas de `printf` qui renvoie le nombre de caractères imprimés :

```
int retour_printf;
retour_printf = printf("123456789\n");
printf("printf avait imprimé %d caractères\n", retour_printf);
```

En sortie :

```
123456789
printf avait imprimé 10 caractères
```

De même `scanf` renvoie le nombre de formats pour lesquels la lecture a été réussie. À noter que `scanf` peut imposer un format que l'utilisateur doit respecter :

```
int first, second;
int count;
printf(">>> ");
```

```
while(scanf("%d + %d", &first, &second) == 2) {
    printf("%d\n", first + second);
    printf(">>> ");
}
printf("Terminé\n");
```

En sortie :

```
>>> 5 + 7
12
>>> 5 * 7
Terminé
```

Avec `scanf` nous avons pu rencontrer des problèmes lorsqu'un utilisateur peu coopératif ne respecte pas le format attendu. Avec les outils que nous avons étudié ensemble, nous pouvons maintenant gérer le code problématique suivant :

```
int age = -1;
while(age < 0) {
    printf("Entrez votre age : ");
    scanf("%d", &age);
}
printf("Vous avez donc %d ans\n", age);
```

En sortie :

```
Entrez votre age : NON
Entrez votre age : Entrez votre age : Entrez votre age : Entrez
↪ votre age : Entrez votre age : Entrez votre age : Entrez votre
↪ age : Entrez votre age : Entrez votre age : ...
```

Une possibilité est de vider les caractères présents jusqu'au retour à la ligne envoyé par l'utilisateur pour envoyer son entrée :

```
int age = -1;
while(age < 0) {
    printf("Entrez votre age : ");
    if(scanf("%d", &age) != 1) {
        while(getchar() != '\n');
    }
}
```

```
}  
printf("Vous avez donc %d ans\n", age);
```

En sortie :

```
Entrez votre age : NON  
Entrez votre age : Euh, tu ne veux plus crasher ?  
Entrez votre age : Vraiment 1111111111 ????  
Entrez votre age : Bon, OK  
Entrez votre age : 42  
Vous avez donc 42 ans
```

Nous pouvons nous-mêmes construire des fonctions avec un type de retour comme par exemple pour effectuer des opérations :

```
int addition(int first, int second) {  
    return first + second;  
}  
  
int main() {  
    printf("addition de 1 et 2 = %d\n", addition(1, 2));  
    exit(EXIT_SUCCESS);  
}
```

En sortie :

```
addition de 1 et 2 = 3
```

### Activité 11 (★ Définir une fonction).

Définir une fonction carré telle que permette au code suivant de fonctionner :

```
#include <stdio.h>  
#include <stdlib.h>  
  
/* TODO : fonction carre qui prend un int en entrée et renvoie un  
↪ int en sortie */  
  
int main() {  
    int a = 41;
```

```
printf("(%d + 1) * (%d + 1) = %d\n", a, a,  
      (a + 1) * (a + 1));  
printf("carre(%d + 1) = %d\n", a, carre(a + 1));  
if((a + 1) * (a + 1) == carre(a + 1))  
    printf("Bravo !\n");  
return 0;  
}
```

## 6.3 Portée des variables

Une variable est déclarée et utilisable dans la section correspondante à cette déclaration. Jusqu'ici nous avons déclaré des variables dans la fonction main. Nous ne nous sommes donc pas soucié de la portée : espace sur lequel elle reste définie.

### 6.3.1 Variables locales à une fonction

#### Fonction main

Il est possible de déclarer une variable à tout moment dans la fonction main et de l'utiliser autant que l'on peut le souhaiter dans la fonction main à la suite de cette déclaration.

```
int main() {  
    int variableDeMain = 42;  
    printf("%d\n", variableDeMain);  
    float autreVariableDeMain = 13.37;  
    printf("%g\n", autreVariableDeMain);  
    variableDeMain = 1;  
    printf("%d\n", variableDeMain);  
    exit(EXIT_SUCCESS);  
}
```

Cependant, cette variable ne peut pas être utilisée en dehors de la fonction main :

```
void test() {  
    variableDeMain = 10;  
}  
  
int main() {  
    int variableDeMain = 42;
```



```
printf("%d\n", variableDeMain);  
test();  
printf("%d\n", variableDeMain);  
exit(EXIT_SUCCESS);  
}
```

Le code suivant provoquera en effet une erreur de compilation :

```
# gcc -o prog main.c  
main.c: In function 'test':  
main.c:5:2: error: 'variableDeMain' undeclared (first use in this  
↪ function)  
    variableDeMain = 10;  
    ^~~~~~  
main.c:5:2: note: each undeclared identifier is reported only once  
↪ for each function it appears in
```

### Paramètres d'une fonction

Pour passer une valeur à une fonction, il est donc possible de lui envoyer par argument :

```
void test(int variableDeMain) {  
    variableDeMain = 10;  
    printf("%d\n", variableDeMain);  
}  
  
int main() {  
    int variableDeMain = 42;  
    printf("%d\n", variableDeMain);  
    test(variableDeMain);  
    printf("%d\n", variableDeMain);  
    exit(EXIT_SUCCESS);  
}
```

Ce qui donnerait la sortie suivante :

```
42  
10  
42
```

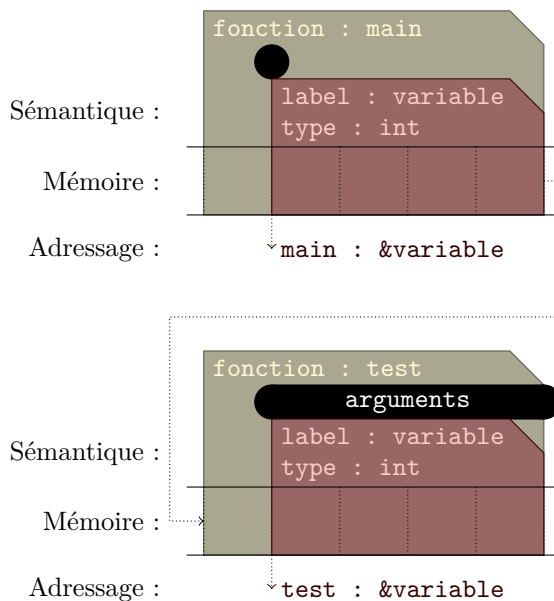
Notons ici que bien que les deux variables aient le même nom, ceci ne correspond pas au même emplacement mémoire :

```
void test(int variable) {
    variable = 10;
    printf("%p : %d\n", &variable, variable);
}

int main() {
    int variable = 42;
    printf("%p : %d\n", &variable, variable);
    test(variable);
    printf("%p : %d\n", &variable, variable);
    exit(EXIT_SUCCESS);
}
```

Ce qui donnerait la sortie suivante :

```
0x7ffc773564b4 : 42
0x7ffc7735649c : 10
0x7ffc773564b4 : 42
```



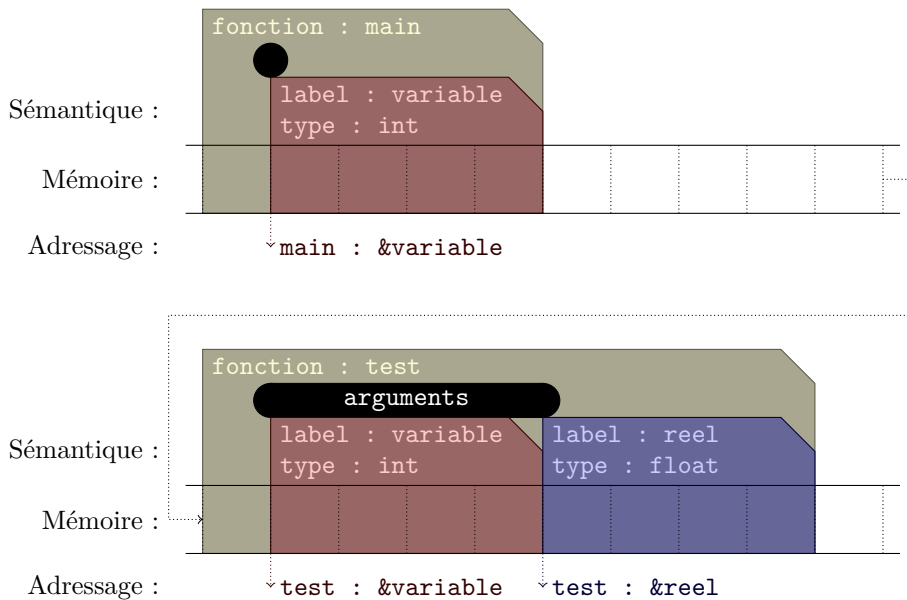
En effet, la fonction `test` déclare une variable `variableDeMain` en paramètre et lorsque `test` est appelée, c'est la valeur passée en argument qui est copiée comme lors d'une affectation.

## Variables locales

Tout comme la fonction `main`, tout fonction peut définir des variables qui lui sont locales autres que ses paramètres.

```
void test(int variable) {
    float reel = 13.37;
    variable = 10;
    printf("%p : %d\n", &variable, variable);
    printf("%p : %g\n", &reel, reel);
}

int main() {
    int variable = 42;
    printf("%p : %d\n", &variable, variable);
    test(variable);
    printf("%p : %d\n", &variable, variable);
    exit(EXIT_SUCCESS);
}
```



### 6.3.2 Variables globales

Cependant de rares élément peuvent avoir l'intérêt d'être partagés dans tout le code. Pour ceci, il est possible de déclarer une variable globale : avant le champs de fonctions. Le code suivant définit une variable globale que chaque fonction peut utiliser :

```
int variable;

void test() {
    variable = 10;
    printf("%p : %d\n", &variable, variable);
}

int main() {
    variable = 42;
    printf("%p : %d\n", &variable, variable);
    test();
    printf("%p : %d\n", &variable, variable);
    exit(EXIT_SUCCESS);
}
```

Ce qui donnerait la sortie suivante :

```
0x55f64c9e2014 : 42
0x55f64c9e2014 : 10
0x55f64c9e2014 : 10
```

Ceci reste à utiliser de manière raisonnée. Par exemple, partager la fenêtre de son interface graphique dans tout le code sans avoir à le passer à chaque fonction en argument est pertinent. Cependant, partager avec tout le code une opérande d'un calcul particulier l'est moins et vous demandera de produire plus de code. Utilisez les variables globales que lorsque nécessaire.

### 6.3.3 Portée des variables

Nous avons vu que les variables définies en argument d'une fonction ou dans son bloc de définition lui sont propres et ne sont visibles que pour la fonction. Ceci vient du fait qu'à l'exécution, lors de l'appel à la fonction, on empile ces variables (on leur fait une place en RAM dans la pile) et lorsque le bloc associé à la fonction se termine, on dépile ces variables (on libère la place qu'elles occupaient dans la

RAM). La portée des variables locales aux fonctions ayant le même nom peut se représenter comme il suit :

```
void test() {  
    int variable = 10;  
    printf("%p : %d\n", &variable, variable);  
}  
  
int main() {  
    int variable = 42;  
    printf("%p : %d\n", &variable, variable);  
    test();  
    printf("%p : %d\n", &variable, variable);  
    exit(EXIT_SUCCESS);  
}
```

Dans le cas où une variable globale existe avec le même nom, c'est la variable locale qui prend la priorité :

```
int variable;  
  
void test() {  
    int variable = 10;  
    printf("%p : %d\n", &variable, variable);  
}  
  
int main() {  
    int variable = 42;  
    printf("%p : %d\n", &variable, variable);  
    test();  
    printf("%p : %d\n", &variable, variable);  
    exit(EXIT_SUCCESS);  
}
```

Il est aussi possible de déclarer des variables directement dans des blocs. Dans ce cas, elles existent dans le code jusqu'à la fin du bloc. Ces variables prennent la priorité si elles sont contenues dans des blocs pour lesquels une variable du même nom est définie :

```
void test() {  
    int variable = 10;  
    printf("%p : %d\n", &variable, variable);  
    if(variable == 10) {  
        int variable = 42;  
        printf("%p : %d\n", &variable, variable);  
    }  
    printf("%p : %d\n", &variable, variable);  
    {  
        int variable = 1337;  
        printf("%p : %d\n", &variable, variable);  
    }  
    printf("%p : %d\n", &variable, variable);  
}
```

## 6.4 Déplacer sa définition de fonction

### 6.4.1 Déclaration d'une fonction

Une fonction peut être déclarée à l'avance, avant même d'être définie. Ceci permet de placer la déclaration de la fonction avant son appel et sa définition après et où souhaité dans le code :

```
int addition(int, int);  
  
int main() {  
    printf("addition de 1 et 2 = %d\n", addition(1, 2));  
    exit(EXIT_SUCCESS);  
}  
  
int addition(int first, int second) {  
    return first + second;  
}
```

En sortie :

```
addition de 1 et 2 = 3
```

À noter que le nommage des variables en argument n'est pas obligatoire dans la déclaration, mais la déclaration et la définition doivent avoir la même signature : le même nom et les mêmes types de retour et d'arguments.

### 6.4.2 Appels mutuels entre fonctions interdépendantes

Il se peut que dans le code, on souhaite faire que deux fonctions s'appellent mutuellement comme une fonction `ping` qui appelle `pong` et réciproquement. Ceci ne sera possible qu'en déclarant les fonctions avant de les définir :

```
void tic(int);
void tac(int);

void tic(int nombre) {
    if(nombre > 0) {
        printf("> tic !\n");
        tac(nombre - 1);
    }
}

void tac(int nombre) {
    if(nombre > 0) {
        printf("< tac !\n");
        tic(nombre - 1);
    }
}

int main() {
    tic(4);
    exit(EXIT_SUCCESS);
}
```

En sortie :

```
> tic !
< tac !
> tic !
< tac !
```

## 6.5 Résumé

Définition d'une fonction (définit la procédure à exécuter lors de son appel) :

```
[TYPE DE RETOUR] [NOM DE LA FONCTION] ([ARGUMENT 1], [ARGUMENT 2],  
↪ ..., [ARGUMENT N]) {  
    [INSTRUCTIONS]  
}
```

Déclaration d'une fonction (autorise son appel dans le code qui suit) :

```
[TYPE DE RETOUR] [NOM DE LA FONCTION] ([ARGUMENT 1], [ARGUMENT 2],  
↪ ..., [ARGUMENT N]);
```

Appel de la fonction :

```
[NOM DE LA FONCTION] ([VALEUR 1], [VALEUR 2], ..., [VALEUR N]);
```

Le mot-clé **return** termine les instructions dans le corps d'une fonction et renvoie la valeur associée (si le type de retour n'est pas **void**).

Exemple :

```
/* Déclaration : */  
int addition(int, int);  
  
/* Définition : */  
int addition(int first, int second) {  
    int resultat = first + second;  
    return resultat;  
}  
  
int main() {  
    /* Appel : */  
    printf("addition de 1 et 1 : %d\n", addition(1, 1));  
    exit(EXIT_SUCCESS);  
}
```



## 6.6 Entraînement

Exercice noté 21 (\*\*\* Fonction de menu).

Écrire une fonction `menu` qui continue tant que l'utilisateur n'a pas entré un nombre qui soit 1, 2 ou 3 et renvoie le choix de l'utilisateur de sorte de l'intégrer au code suivant :

01\_menu.c

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

int menu() {
    /* À vous de jouer */
}

int main() {
    printf("%d, bien reçu.\n", menu());
    exit(EXIT_SUCCESS);
}
```

**Exercice noté 22 (\*\*\* Calcul moyenne).**

Compléter le code c-dessous en écrivant la fonction `moyenne` qui lit des flottants positifs depuis le clavier et en calcule la moyenne comme dans la sortie qui suit :

**02\_moyenne.c**

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

float moyenne();

int main() {
    printf("La moyenne de ");
    printf("= %g\n", moyenne());
    exit(EXIT_SUCCESS);
}
```

```
La moyenne de 12 16 15 -1
= 14.3333
```

Exercice noté 23 (★★★ Fonction puissance).

1. Écrire une fonction `puissance` qui calcule  $a^n$  pour  $a$  et  $n$  deux entiers en respectant la déclaration suivante :

```
long puissance(long a, long n);
```

2. Écrire une fonction `puissance_rapide` qui calcule  $a^n$  à l'aide de cet algorithme :

---

**Algorithm 1:** Exponentiation rapide

---

```
input  : Entier valeur
input  : Entier exposant
output: Entier resultat

1 resultat ← 1 ;
2 while exposant > 0 do
3   if exposant modulo 2 = 1 then
4     resultat ← resultat × valeur ;
5   end
6   valeur ← valeur × valeur ;
7   exposant ← exposant / 2;
8 end
```

---

3. Écrire une fonction `puissance_modulo` et `puissance_rapide_modulo` qui calculent  $a^n \bmod p$  pour  $a$ ,  $n$  et  $p$  trois entiers. Ceci permettra de comparer l'efficacité de ces deux méthodes. On propose de transformer le calcul de puissance en puissance modulaire. À noter que  $a^n \bmod p$  reste plus petit que  $p$  mais que  $a^n$  peut dépasser  $p$  et la capacité de votre entier. Ceci vous demandera donc de transformer les multiplications pour rester modulo  $p$ . À vous de jouer en respectant les déclarations suivantes :

```
unsigned long puissance_modulo(unsigned long a, unsigned long
↪ n, unsigned long p);
unsigned long puissance_rapide_modulo(unsigned long a,
↪ unsigned long n, unsigned long p);
```

4. Comparez les temps qu'il faut à ces deux fonctions pour calculer le nombre suivant :

$$42^{2\ 460\ 320\ 538} \bmod 4\ 285\ 404\ 239$$

## Exercice noté 24 (\*\*\* Suite de Fibonacci).

Alice, Bob et Charlie ont entendu parler de la suite de Fibonacci en cours d'algorithmique. Cette suite est donnée pour tout entier naturel  $n$  par la définition suivante :

$$\mathcal{F}_n = \begin{cases} 0 & \text{Si } n = 0 \\ 1 & \text{Si } n = 1 \\ \mathcal{F}_{n-1} + \mathcal{F}_{n-2} & \text{Sinon} \end{cases}$$

Alice, Bob et Charlie ont chacun une méthode bien à eux pour calculer cette suite. Vous devrez implémenter les fonctions relatives à la méthode de chacun. Le code suivant n'est pas à modifier et lorsque `fibonacci_Alice`, `fibonacci_Bob` et `fibonacci_Charlie` fonctionneront vous donnera la sortie ci-dessous :

04\_fibonacci.c

```
#include <stdio.h>
#include <stdlib.h>
unsigned long fibonacci_Alice(int n);
unsigned long fibonacci_Bob(int n);
unsigned long fibonacci_Charlie(int n);
void afficher_fibonacci(
    int borne,
    const char * nom,
    unsigned long (*fibonacci)(int)) {
    int i;
    printf("%s : ", nom);
    for(i = 0; i < borne; ++i) {
        if(i) { printf(", "); }
        printf("%lu", fibonacci(i));
    }
    printf("\n");
}

int main() {
    afficher_fibonacci(16, "Alice ", fibonacci_Alice);
    afficher_fibonacci(16, "Bob   ", fibonacci_Bob);
    afficher_fibonacci(16, "Charlie", fibonacci_Charlie);
    exit(EXIT_SUCCESS);
}
```

```
Alice   : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,  
↪ 610  
Bob     : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,  
↪ 610  
Charlie : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,  
↪ 610
```

1. Alice a choisi d'implémenter la formule donnée par la définition mathématique de la suite et d'appeler deux fois `fibonacci_Alice` dans sa fonction `fibonacci_Alice`. Écrivez cette fonction.
2. Bob dit qu'il a trouvé un algorithme bien mieux qui fait le travail avec une boucle :

---

**Algorithm 2:** Fibonacci linéaire

---

```
input  : Entier indice  
output: Entier last  
1 Entier current  $\leftarrow$  1 ;  
2 Entier last  $\leftarrow$  0 ;  
3 while indice  $>$  0 do  
4   | temporaire  $\leftarrow$  current ;  
5   | current  $\leftarrow$  current + last ;  
6   | last  $\leftarrow$  temporaire ;  
7   | indice  $\leftarrow$  indice- 1;  
8 end
```

---

3. Charlie lui a trouvé l'algorithme d'un gourou qui prétend jouer avec les matrices et pouvoir calculer efficacement chaque élément de la suite de Fibonacci. Implémentez l'algorithme 'Fibonacci logarithmique' suivant pour attester de son fonctionnement :
4. Demandez maintenant les 42 premières valeurs de la suite de Fibonacci à l'aide des trois méthodes, puis les 50 premières. Votre programme semble grandement ralenti, avez-vous une idée de la méthode qui pose problème et la raison de cette perte d'efficacité ?

---

**Algorithm 3:** Fibonacci logarithmique

---

```
input : Entier indice
output: Entier last

1 Entier current  $\leftarrow$  1 ;
2 Entier last  $\leftarrow$  0 ;
3 Entier matrix00  $\leftarrow$  1 ; Entier matrix01  $\leftarrow$  1 ;
4 Entier matrix10  $\leftarrow$  1 ; Entier matrix11  $\leftarrow$  0 ;
5 while indice  $>$  0 do
6   if indice modulo 2 = 1 then
7     new_current  $\leftarrow$  matrix00  $\times$  current + matrix01  $\times$  last ;
8     new_last  $\leftarrow$  matrix10  $\times$  current + matrix11  $\times$  last ;
9     current  $\leftarrow$  new_current, last  $\leftarrow$  new_last ;
10  end
11  new_matrix00  $\leftarrow$  matrix00  $\times$  matrix00 + matrix01  $\times$  matrix10 ;
12  new_matrix01  $\leftarrow$  matrix00  $\times$  matrix01 + matrix01  $\times$  matrix11 ;
13  new_matrix10  $\leftarrow$  matrix10  $\times$  matrix00 + matrix11  $\times$  matrix10 ;
14  new_matrix11  $\leftarrow$  matrix10  $\times$  matrix01 + matrix11  $\times$  matrix11 ;
15  matrix00  $\leftarrow$  new_matrix00 ; matrix01  $\leftarrow$  new_matrix01 ;
16  matrix10  $\leftarrow$  new_matrix10 ; matrix11  $\leftarrow$  new_matrix11 ;
17  indice  $\leftarrow$  indice / 2 ;
18 end
```

---

## Exercice noté 25 (★★ Réagencer un code).

Libérez la fonction `main` et remaniez le code suivant pour plus de lisibilité et avec une meilleure découpe en fonctions. On souhaite ici :

1. Factoriser le calcul deux opérands et d'un opérateur en une fonction.
2. Avoir une fonction pour le menu.
3. Avoir une fonction qui affiche les informations du menu.
4. Avoir une fonction par fonctionnalité du menu.
5. Avoir une fonction de lancement de la calculatrice appelée par la fonction `main`.
6. Avoir quelques commentaires associés aux fonctions (documentation du code).

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

int main() {
    int first, second;
    int variable = 0;
    char op;
    int choix;
    int resultat;
    do {
        printf("1 - Calculer\n2 - Modifier variable\n3 - Voir
        ↪ variable\n0 - Quitter\n---\nVotre choix : ");
        scanf("%d", &choix);
        switch(choix) {
            case 1 :
                printf(">>> ");
                scanf("%d %c %d", &first, &op, &second);
                switch(op) {
                    case '+' : resultat = first + second; break;
                    case '-' : resultat = first - second; break;
                    case '*' : resultat = first * second; break;
                    case '/' : resultat = first / second; break;
                    case '%' : resultat = first % second; break;
                }
                printf(">>> %d %c %d = %d\n", first, op, second,
                ↪ resultat);
                break;
        }
    } while (choix != 0);
}
```

```
        case 2 :
            printf(">>> variable = %d ", variable);
            scanf(" %c %d", &op, &second);
            switch(op) {
                case '+' : variable += second; break;
                case '-' : variable -= second; break;
                case '*' : variable *= second; break;
                case '/' : variable /= second; break;
                case '%' : variable %= second; break;
                case '=' : variable = second; break;
            }
            printf(">>> variable = %d\n", variable);
            break;
        case 3 :
            printf(">>> variable = %d\n", variable);
            break;
        default : break;
    }
} while(choix != 0);
printf("Au revoir.\n");
exit(EXIT_SUCCESS);
}
```



---

## 7 Tableaux

---

### 7.1 Tableau à une dimension

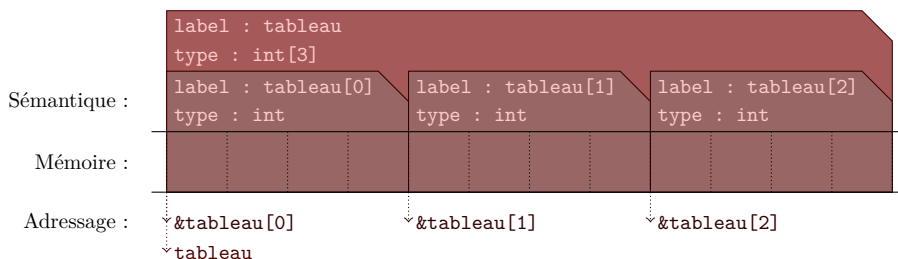
Nous avons vu comment gérer des variables, mais on peut vouloir gérer une liste de mêmes valeurs, comme les températures des derniers mois ou encore une table des plus hauts scores obtenus par des joueurs. Il ne semble pas acceptable de déclarer une variable à la main surtout si on maintient un tableau de score des 100 derniers joueurs par exemple. Pour faire ceci, nous pouvons construire des tableaux.

Un tableau se déclare comme une variable avec des crochets []. on peut lors de la déclaration du tableau indiquer les valeurs qui le composent par défaut et aussi indiquer à l'avance une taille souhaitée si nous n'avons pas ces informations à la déclaration. L'accès à un élément du tableau par son indice se fait ensuite par la syntaxe `tableau[indice]` et peut être géré comme une variable.

```
type nomTableau[] = {valeur1, valeur2, ..., valeurN};
type nomTableau[TAILLE];
nomTableau[indice]; /* Appel élément d'indice 'indice' */
```

Ceci se transposerait par exemple comme il suit pour construire un tableau de trois entiers.

```
int tableau[] = {1, 2, 3};
int tableau[3];
tableau[0] = 1; /* affectation première valeur */
```



À bien noter que les indices des tableaux commencent à 0.

```
int liste[] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, -1};
int i;
for(i = 0; liste[i] >= 0; ++i) {
    if(i) {
        printf(", ");
    }
    printf("%d", liste[i]);
}
printf("\n");
```

En sortie :

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

Plutôt que de saisir ces valeurs à la main, il est possible de souhaiter construire les valeurs de la liste vue précédemment (Cette séquence est la suite de Fibonacci) :

```
const int taille_liste = 16;
int liste[taille_liste];
int i;
for(i = 0; i < taille_liste; ++i) {
    if(i < 2) {
        liste[i] = i;
    } else {
        liste[i] = liste[i - 1] + liste[i - 2];
    }
}
for(i = 0; i < taille_liste; ++i) {
    if(i) {
        printf(", ");
    }
    printf("%d", liste[i]);
}
printf("\n");
```

En sortie :

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610
```

À noter qu'en langage C, lorsqu'on fabrique un tableau, il correspond uniquement au fait de demander un nombre d'éléments successif en mémoire et l'appel au nom du tableau donnera l'adresse du premier élément :

```
int tableau[] = {1, 2, 3};
printf("adresse de tableau : %p\n", tableau);
printf("adresse de tableau[0] : %p\n", &tableau[0]);
printf("adresse de tableau[1] : %p\n", &tableau[1]);
printf("adresse de tableau[2] : %p\n", &tableau[2]);
```

En sortie :

```
adresse de tableau : 0x7ffcff40ff4c
adresse de tableau[0] : 0x7ffcff40ff4c
adresse de tableau[1] : 0x7ffcff40ff50
adresse de tableau[2] : 0x7ffcff40ff54
```

En effet, l'adresse du premier élément correspond à au nom du tableau et ici, nos éléments ont des adresses espacées de la taille d'un `int` soit 4 octets. Un tableau est un peu comme une rue de maisons où les maisons se trouvent à adresses successives et sont séparées par l'espace qu'occupe une maison.

À noter que l'opérateur `sizeof` permet de déterminer la taille en octets d'un type ou d'un élément.

```
printf("sizeof(char) : %lu\n", sizeof(char));
printf("sizeof(short) : %lu\n", sizeof(short));
printf("sizeof(int) : %lu\n", sizeof(int));
printf("sizeof(long) : %lu\n", sizeof(long));
```

En sortie :

```
sizeof(char) : 1
sizeof(short) : 2
sizeof(int) : 4
sizeof(long) : 8
```

Il est possible de passer un tableau à une fonction sans préciser sa taille :

```
void afficher_liste(int liste[]) {
    int i;
    for(i = 0; liste[i] >= 0; ++i) {
        if(i) {
            printf(", ");
        }
        printf("%d", liste[i]);
    }
    printf("\n");
}

int main() {
    int liste[12] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, -1};
    afficher_liste(liste);
    exit(EXIT_SUCCESS);
}
```

En sortie :

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

Cependant, bien qu'utiliser `sizeof` sur un tableau ou une variable pour connaître sa taille semble intéressant, il reste à utiliser avec parcimonie car ceci pourra être source d'erreur si utilisé dans une fonction. Préférez une connaissance du type et de ce que fait votre code. En effet, dans une fonction un tableau est considéré comme son adresse, c'est la taille de l'adresse qui sera considérée :

```
void afficher_taille_tableau(float tableau[]) {
    printf("depuis une fonction : %lu\n", sizeof(tableau));
}

int main() {
    float tableau[16];
    printf("depuis main : %lu\n", sizeof(tableau));
    afficher_taille_tableau(tableau);
    exit(EXIT_SUCCESS);
}
```

En sortie :

```
depuis main : 64
depuis une fonction : 8
```

## 7.2 Chaînes de caractères

Les tableaux ont aussi un intérêt particulier puisqu'ils permettent de construire des chaînes de caractères. En réalité ce sont des tableaux de caractères. À noter qu'une chaîne de caractères se termine par le symbole `'\0'` :

```
char tableau_de_char[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', ' ',
    ↪ '!', '\0'};
int i;
for(i = 0; tableau_de_char[i] != '\0'; ++i) {
    putchar(tableau_de_char[i]);
}
putchar('\n');
```

En sortie :

```
Bonjour !
```

En réalité, le langage C permet de déclarer une chaîne de caractères depuis une chaîne constante donnée entre double guillemets et afficher une chaîne de caractères de manière moins fastidieuse depuis le format `%s` :

```
char texte[] = "Bonjour !";
printf("%s\n", texte);
```

En sortie :

```
Bonjour !
```

Une chaîne de caractère peut aussi être lue depuis l'entrée standard de la console avec `scanf` et le format `%s` :

```
char nom[64];
printf("Quel est votre nom ? ");
```

```
scanf("%s", nom);
printf("Bonjour %s !\n", nom);
```

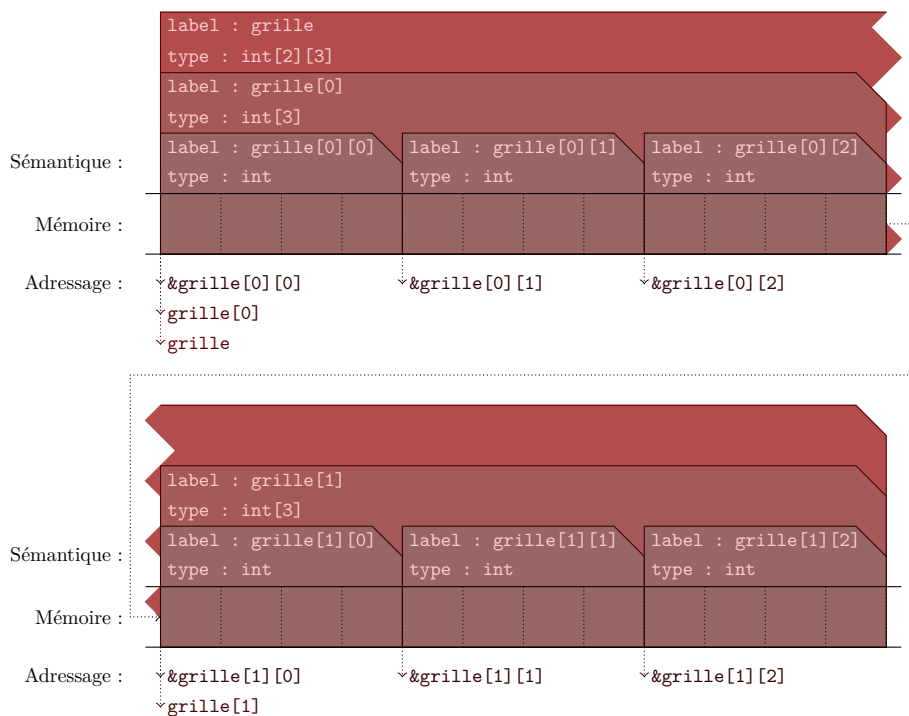
En sortie :

```
Quel est votre nom ? M.Trancho
Bonjour M.Trancho !
```

À noter qu’une bibliothèque standard, `string.h`, est dédiée à la manipulation de chaînes de caractères. Recoder ses fonctions et découvrir ses fonctionnalités fera partie des exercices.

## 7.3 Tableau à plusieurs dimensions

Imaginons maintenant que nous aimerions définir un terrain où vivrait notre personnage, ceci peut se faire avec un tableau, mais il existe une alternative : les tableaux à plusieurs dimensions.



Par exemple un tableau à deux dimensions nous permettrait d'avoir la possibilité de renseigner deux indices au lieu de un ce qui convient à la définition d'une grille. Ceci se déclare comme un tableau, mais il faudra ajouter des crochets pour chaque dimension supplémentaire :

```
int grille[2][3];  
/* 2 lignes */  
/* 3 colonnes */
```

```
char terrain[5][6] = {  
    "#.###",  
    "#.#.#",  
    "#...#",  
    "#.#.#",  
    "###.#"  
};  
int ligne, colonne;  
for(ligne = 0; ligne < 5; ++ligne) {  
    for(colonne = 0; colonne < 6; ++colonne) {  
        putchar(terrain[ligne][colonne]);  
    }  
    putchar('\n');  
}
```

En sortie :

```
#.###  
#.#.#  
#...#  
#.#.#  
###.#
```

Cependant, lorsqu'un tableau à plusieurs dimensions est passé à une fonction, seule sa première dimension peut être laissée sans valeur fixe, les autres doivent être renseignées.

```
void afficher_terrain(char terrain[][6], int hauteur) {  
    int ligne, colonne;  
    for(ligne = 0; ligne < hauteur; ++ligne) {  
        for(colonne = 0; colonne < 6; ++colonne) {
```

```
        putchar(terrain[ligne][colonne]);
    }
    putchar('\n');
}

int main() {
    char terrain[5][6] = {
        "#.###",
        "#.#.#",
        "#...#",
        "#.#.#",
        "###.#"
    };
    afficher_terrain(terrain, 5);
    exit(EXIT_SUCCESS);
}
```

En sortie :

```
#.###
#.#.#
#...#
#.#.#
###.#
```

L'astuce pour passer un tableau à plusieurs dimensions à une fonction sans dimensions figées peut être de placer la récupération des dimensions avant le tableau dans la liste d'arguments :

```
void afficher_terrain(int hauteur, int largeur, char
↪ terrain[hauteur][largeur]) {
    int ligne, colonne;
    for(ligne = 0; ligne < hauteur; ++ligne) {
        for(colonne = 0; colonne < largeur - 1; ++colonne) {
            putchar(terrain[ligne][colonne]);
        }
        putchar('\n');
    }
}
```



```
int main() {  
    char terrain[5][7] = {  
        "#.####",  
        "#.#..#",  
        "#...##",  
        "#.#..#",  
        "####.#"  
    };  
    afficher_terrain(5, 7, terrain);  
    exit(EXIT_SUCCESS);  
}
```

En sortie :

```
#.####  
#.#..#  
#...##  
#.#..#  
####.#
```

## 7.4 Résumé

Un tableau à une dimension peut se déclarer de la manière suivante :

```
[TYPE] [NOM_TABLEAU] [] = {[VALEUR 1], ..., [VALEUR N]};  
[TYPE] [NOM_TABLEAU] [[NOMBRE VALEURS]];
```

Chaque élément d'un tableau est une variable et l'accès à l'élément d'indice `indice` se fait comme il suit :

```
tableau[indice];
```

La valeur que représente le nom du tableau est l'adresse de son premier élément. Les éléments d'un tableau sont alignés en mémoire et leurs adresses sont séparées par la taille du type du tableau.

L'opérateur `sizeof` permet de déterminer l'espace qu'occupe un type en mémoire.

Une chaîne de caractères est un tableau de `char` et peut être définie par une chaîne constante :

```
char texte[] = "Texte";
```

Une chaîne de caractères a pour dernier élément le caractère nul `'\0'`.

Une chaîne de caractères se gère en entrée-sortie par le format `%s` :

```
printf("%s\n", texte);  
scanf("%s", texte);
```

Un tableau à plusieurs dimensions peut se déclarer de la manière suivante :

```
[TYPE] [NOM_TABLEAU] [[DIMENSION 1]] ... [[DIMENSION N]];
```

Son accès se fait de la manière suivante :

```
tableau[indice_1][indice_2]...[indice_n];
```

## 7.5 Entraînement

Exercice noté 26 (\*\*\* Statistiques sur un tableau).

Complétez le code ci-dessous pour qu'il donne la sortie suivante :

```
minimum : valeurs[6] = 0
maximum : valeurs[1] = 9
moyenne : 4.5
```

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

#define TAILLE 10
/* min_index renvoie l'indice du plus petit élément de valeurs */
int min_index(int valeurs[], int taille);
/* min_value renvoie le plus petit élément de valeurs */
int min_value(int valeurs[], int taille);
/* max_index renvoie l'indice du plus grand élément de valeurs */
int max_index(int valeurs[], int taille);
/* max_value renvoie le plus grand élément de valeurs */
int max_value(int valeurs[], int taille);
/* moyenne renvoie la moyenne des éléments de valeurs */
float moyenne(int valeurs[], int taille);

int main() {
    int valeurs[TAILLE] = {5, 9, 1, 4, 8, 3, 0, 6, 2, 7};
    printf("minimum : valeurs[%d] = %d\n", min_index(valeurs,
        ↪ TAILLE), min_value(valeurs, TAILLE));
    printf("maximum : valeurs[%d] = %d\n", max_index(valeurs,
        ↪ TAILLE), max_value(valeurs, TAILLE));
    printf("moyenne : %g\n", moyenne(valeurs, TAILLE));
    exit(EXIT_SUCCESS);
}
```

## Exercice noté 27 (★★ Propriétés dans un labyrinthe).

Alice a commencé un jeu de labyrinthe mais rencontre quelques difficultés pour coder les fonctions `is_valid` et `is_finish`. Pourriez-vous faire en sorte que les '#' et les sorties de la carte bloquent le '@' lorsqu'on le déplace avec les touches 'zqsd' et que le jeu se termine lorsque le '@' arrive sur le '.' ? Alice vous indique qu'il faudra ajouter `-lncurses` pour compiler sous Linux et vous propose [ce lien](#) pour Windows.

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

#define HAUTEUR 9
#define LARGEUR 33

/* is_valid vérifie que les coordonnées (x, y) sont valides pour
   ↳ un déplacement */
int is_valid(int x, int y, int hauteur, int largeur, char
   ↳ grille[hauteur][largeur]) {
    return 1;
}

/* is_finish vérifie que l'emplacement sur lequel se trouve les
   ↳ coordonnées (x, y) est une sortie */
int is_finish(int x, int y, int hauteur, int largeur, char
   ↳ grille[hauteur][largeur]) {
    return 0;
}

int main() {
    char grille[HAUTEUR][LARGEUR] = {
        "# #####",
        "#      # # #      #",
        "### # ## #      # # ## #####",
        "#  # # #####      #      #",
        "# ##### # # ##### #####",
        "# # # # #      #      #",
        "# ## # ##### #####",
        "#      #      #",
        "#####.#"};
    int x = 1, y = 0;
    int move_x, move_y;
```

```
int i;
initscr();
noecho();
cbreak();
do {
    clear();
    for(i = 0; i < HAUTEUR; ++i) {
        mvprintw(i, 0, "%s", grille[i]);
    }
    mvprintw(y, x, "@");
    mvprintw(y, x, "");
    refresh();
    move_x = x; move_y = y;
    switch(getch()) {
        case 'z': move_y = y - 1; break;
        case 's': move_y = y + 1; break;
        case 'q': move_x = x - 1; break;
        case 'd': move_x = x + 1; break;
    }
    if(is_valid(move_x, move_y, HAUTEUR, LARGEUR, grille)) {
        x = move_x; y = move_y;
    }
} while(! is_finish(x, y, HAUTEUR, LARGEUR, grille));
refresh();
clrtoeol();
refresh();
endwin();
exit(EXIT_SUCCESS);
}
```

## Exercice noté 28 (★★★ Coder les fonction de string.h).

Bob aimerait voir si vous savez recoder les fonctions classiques de la bibliothèque `string.h` et vous met au défi d'en recoder quelques unes juste pour voir. Il vous indique rapidement ce qu'elles font :

1. `strlen` donne la taille de la chaîne de caractères.
2. `strcpy` copie une chaîne de caractères dans une autre.
3. `strcat` ajoute une chaîne de caractères à une autre.
4. `strcmp` renvoie 0 si les chaînes sont identiques, une valeur négative si la première précède la seconde dans l'ordre lexicographique et une valeur positive si c'est l'inverse.

À vous de les coder en version `esgi_` et vérifier que le comportement est semblable à celles de la bibliothèque standard :

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

int esgi_strlen(const char texte[]);
void esgi_strcpy(char destination[], const char source[]);
void esgi_strcat(char destination[], const char source[]);
int esgi_strcmp(const char first[], const char second[]);

int main() {
    char texte[] = "Welcome to ESGI !";
    char hello[] = "Hello";
    char copie[50];
    printf("esgi_strlen(\"%s\") = %d\n", texte, esgi_strlen(texte));
    esgi_strcpy(copie, "Eleve, ");
    printf("copie = \"%s\"\n", copie);
    esgi_strcat(copie, texte);
    printf("copie = \"%s\"\n", copie);
    printf("esgi_strcmp(\"Hello\", \"Hello\") = %d = 0\n",
        ↪ esgi_strcmp(hello, "Hello"));
    printf("esgi_strcmp(\"Hello\", \"Bonjour\") = %d > 0\n",
        ↪ esgi_strcmp(hello, "Bonjour"));
    printf("esgi_strcmp(\"Hello\", \"Hell\") = %d > 0\n",
        ↪ esgi_strcmp(hello, "Hell"));
    printf("esgi_strcmp(\"Bonjour\", \"Hello\") = %d < 0\n",
        ↪ esgi_strcmp("Bonjour", hello));
    exit(EXIT_SUCCESS);
}
```

## Exercice noté 29 (★★ Recherche dans un tableau).

Charlie souhaite faire la recherche dans un tableau.

1. Codez une fonction `recherche` qui recherche si une valeur se trouve dans un tableau et renvoie son indice si on l'y trouve et -1 sinon :

```
int recherche(int valeur, int tableau[], int taille);
```

2. Charlie vous informe que sur certains tableaux des méthodes plus efficaces existent comme la dichotomie. Charlie vous donne un algorithme qu'il a trouvé sur la dichotomie.

Codez une fonction `recherche_dichotomique` à partir de celui-ci.

---

**Algorithm 4:** Recherche dichotomique dans une liste
 

---

```

input  : Entier valeur
input  : Tableau liste
input  : Entier taille
output: Entier

1 start ← 0;
2 end ← taille;
3 while start < end do
4   | indice ← (start + end) / 2;
5   | if valeur < liste[indice] then
6   |   | end ← indice;
7   | else if valeur > liste[indice] then
8   |   | start ← indice + 1;
9   | else
10  |   | return indice;
11  |   | end
12 end
13 return -1;
```

---

3. Proposez des exemples pour vérifier que votre code fonctionne.

## Exercice noté 30 (★★★ Décodage Vigenère).

Oscar a trouvé sur internet une méthode connue pour envoyer des messages secrets : le Chiffre de **Vigenère**. Il a vanté les mérites de l'**ESGI**, a gribouillé "Fgtrsmx mxmjgefz li d'KAKA, swm u'kax Gykej, zc tjknijka pw rirygoi U uc Tqzpsf?" sur un tableau et a disparu sans donner plus d'explications. Pourriez-vous mettre au point un programme pour décoder ce message et lui répondre?





---

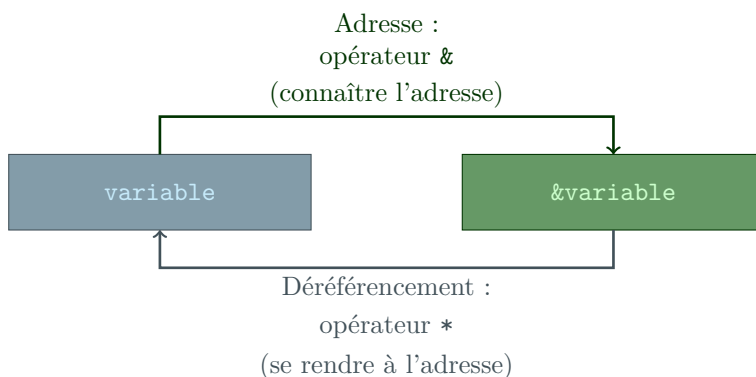
## 8 Pointeurs

---

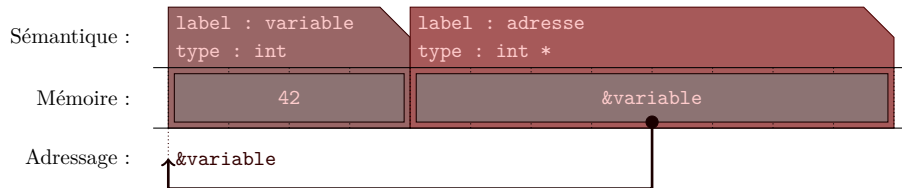
### 8.1 Un type d'adresse

Nous savons que chaque variable a une adresse, mais si nous voulons sauvegarder une adresse ou tenir un carnet d'adresse, il serait possible de le sauvegarder dans un `long` bien qu'à la fin nous ne saurions plus quel est le type de la variable représentée. Pour ceci nous avons un outil nommé **pointeur**. Un pointeur c'est une variable typée qui permet de représenter l'adresse d'une autre variable typée ou d'un emplacement mémoire.

Par exemple, si nous avons un type `Maison`, nous pourrions vouloir référencer l'adresse de cette `Maison` par un type `Adresse` d'une `Maison`. Ce type `Adresse` d'une `Maison` a pour syntaxe le type référencé suivi d'une étoile `*`. Autrement dit, l'adresse d'une `Maison` peut se sauvegarder dans une variable de type `Maison *`. Le type `Maison` est une illustration qui peut être remplacée par le type connu.



Cependant, si nous n'avons que l'adresse `Maison * adresse` d'une variable `Maison house`, nous ne pourrons pas en faire grand chose car `adresse` correspond à `&house`, ce qui nous intéresse c'est la variable elle-même (la `Maison house`). Pour aller à la `Maison house`, nous devons déréférencer la adresse, ce qui se fait en ajoutant une étoile `*` devant l'adresse : `*adresse` correspond à `house`.



```
int variable = 42;
int * adresse = &variable;
printf("Maison :   variable : %d\n", variable);
printf("Adresse : &variable : %p\n", &variable);
printf("Adresse :   adresse : %p\n", adresse);
printf("Maison :   *adresse : %d\n", *adresse);
printf("Maison : *&variable : %d\n", *&variable);
```

En sortie :

```
Maison :   variable : 42
Adresse : &variable : 0x7fff76c890b4
Adresse :   adresse : 0x7fff76c890b4
Maison :   *adresse : 42
Maison : *&variable : 42
```

### Activité 12 (\*\*\* Fonction pour échanger des valeurs).

Un camarade fictif est lassé de déclarer une variable temporaire pour chaque échange de valeurs et veut coder une fonction qui le fasse. Cependant son code ne fonctionne pas. Sans changer la logique du code, proposer une solution pour le faire fonctionner :

```
void echanger(int first, int second) {
    int temporaire;
    temporaire = first;
    first = second;
    second = temporaire;
}

int main() {
```

```
int first = 42, second = 1337;
printf("first = %d, second = %d\n", first, second);
echanger(first, second);
printf("first = %d, second = %d\n", first, second);
exit(EXIT_SUCCESS);
}
```

Un pointeur est intéressant pour deux cas de figures :

1. Modifier la valeur d'une variable dans une fonction depuis son adresse.
2. Ne pas recopier tout un élément en le passant à une fonction par exemple, mais uniquement son adresse (tableaux).

## 8.2 Arithmétique des pointeurs

Nous avons vu précédemment qu'un tableau correspond en réalité à l'adresse de sa première valeur. En réalité, la manipulation par le nom du tableau par un pointeur sur le tableau est équivalente :

```
1 void afficher_tableau(int * tableau) {
2     int i;
3     for(i = 0; tableau[i] > 0; ++i) {
4         if(i) {
5             printf(", ");
6         }
7         printf("%d", tableau[i]);
8     }
9     printf("\n");
10 }
11
12 int main() {
13     int tableau[] = {1, 2, 3, -1};
14     int * pointeur = tableau;
15     printf("tableau : %p\n", tableau);
16     printf("pointeur : %p\n", pointeur);
17     afficher_tableau(tableau);
18     pointeur[1] = 42;
19     afficher_tableau(tableau);
20     exit(EXIT_SUCCESS);
21 }
```

En sortie :

```
tableau : 0x7ffd298aee40
pointeur : 0x7ffd298aee40
1, 2, 3
1, 42, 3
```

Notons que dans ce code, nous récupérons le tableau comme un pointeur sur un tableau et que nous pouvons l'utiliser comme un tableau pour imprimer ses éléments. De même, à la ligne 18, nous utilisons le pointeur comme un tableau pour modifier l'élément d'indice 1.

Nous avons vu que pour un tableau, les adresses de chaque élément se suivaient en mémoire séparées par la taille qu'occupe le type en mémoire. En réalité, il est possible de jouer avec les adresse avec ce que l'on nomme **l'arithmétique des pointeur** : c'est-à-dire que si on ajoute 1 à l'adresse d'un élément d'un tableau ou un pointeur, on a l'adresse de l'élément suivant :

```
int tableau[] = {1, 2, 3, -1};
int * pointeur = tableau;
printf("tableau : %p\n", tableau);
printf("&tableau[1] : %p\n", &tableau[1]);
printf("&pointeur[1] : %p\n", &pointeur[1]);
printf("(tableau + 1) : %p\n", tableau + 1);
printf("(pointeur + 1) : %p\n", pointeur + 1);
```

En sortie :

```
tableau : 0x7fff25428330
&tableau[1] : 0x7fff25428334
&pointeur[1] : 0x7fff25428334
(tableau + 1) : 0x7fff25428334
(pointeur + 1) : 0x7fff25428334
```

Pour aller plus loin avec cette mécanique, nous pouvons déréréferencer les adresses des différents éléments et obtenir leurs valeurs ce qui est équivalent à l'accès entre crochets (c'est ce que fait la machine) :

```
int tableau[] = {1, 2, 3, -1};
int * pointeur = tableau;
```

```
printf("&pointeur[1] :    %p\n", &pointeur[1]);
printf("(pointeur + 1) :  %p\n", pointeur + 1);
printf("pointeur[1] :    %d\n", pointeur[1]);
printf("*(pointeur + 1) : %d\n", *(pointeur + 1));
printf("*(1 + pointeur) : %d\n", *(1 + pointeur));
printf("1[pointeur] :    %d\n", 1[pointeur]);
```

En sortie :

```
&pointeur[1] :    0x7ffdab94b194
(pointeur + 1) :  0x7ffdab94b194
pointeur[1] :    2
*(pointeur + 1) : 2
*(1 + pointeur) : 2
1[pointeur] :    2
```

À noter que les deux dernière lignes sont plus une démonstration qu'un concept à utiliser dans votre code : pour montrer le pointeur et l'indice sont interchangeables et que la machine applique exactement cette opération de déréférencement pour la notation entre crochets.

À noter que ceci est comptable avec l'idée de modifier la valeur d'un pointeur pour itérer sur ses éléments comme pour une chaîne de caractères :

```
void afficher_chaine(char * chaine) {
    for(; *chaine != '\0'; ++chaine) {
        putchar(*chaine);
    }
    putchar('\n');
}

int main() {
    char texte[] = "Hello ESGI !";
    afficher_chaine(texte);
    exit(EXIT_SUCCESS);
}
```

```
Hello ESGI !
```

Par exemple, ceci permet d'utiliser l'opérateur `++` sur le pointeur pour passer d'une adresse à la suivante (`pointeur = pointeur + 1`) et obtenir chaque caractère, on déréférence l'adresse courante.

## 8.3 Allocation dynamique

Nous avons vu comment obtenir des tableaux, mais le problème principal est que leur taille reste en général figée à la compilation. Il est possible de réclamer un espace mémoire dont la taille est donnée à l'exécution par l'allocation dynamique.

De la même manière qu'un tableau est donné par l'adresse de son premier élément, il est possible de récupérer l'adresse du premier élément d'une plage réclamée par appel à la fonction `malloc`. on demande à `malloc` une plage mémoire d'une taille donnée en octets et `malloc` renvoie une plage valide si réussi et `NULL` sinon. Il faudra penser à libérer la mémoire en fin de programme ou lorsqu'il n'est plus intéressant de la garder en mémoire avec la fonction `free`. Une manière simple de faire appel à `malloc` est la suivante :

```
type * tableau = malloc(sizeof(type) * taille);  
/* équivalent à type tableau[taille] avec taille une variable */  
/* utiliser tableau */  
free(tableau);
```

Une manière propre de gérer la mémoire peut se faire de la manière suivante :

```
type * tableau = NULL;  
if((tableau = (type *)malloc(sizeof(type) * taille)) == NULL) {  
    /* traitement à appliquer, sauvegarde si possible */  
    exit(EXIT_FAILURE);  
}  
/* utiliser tableau */  
free(tableau);  
tableau = NULL;
```

Nous maintenons un pointeur qui ne pointe vers aucune adresse et aucun emplacement mémoire à `NULL`. Dans le cas où l'allocation par `malloc` échoue, c'est en réalité souvent critique et difficilement rattrapable, donc en général, il faudrait sauvegarder si possible le travail de l'utilisateur et terminer le programme. Une fois que la plage mémoire est libérée par `free`, le pointeur regarde une adresse devenue invalide, on repasse sa valeur à `NULL`.

Pour vérifier que le programme libère bien tout la mémoire qu'il a demandé, il est possible d'utiliser la commande `valgrind` [LANCEMENT DU PROGRAMME] pour le vérifier. Si toute la mémoire réclamée a bien été libérée, alors la commande doit terminer par une sortie semblable à la suivante :

```
==3396== HEAP SUMMARY:
==3396==       in use at exit: 0 bytes in 0 blocks
==3396==    total heap usage: 1 allocs, 1 frees, 168 bytes
↳ allocated
==3396==
==3396== All heap blocks were freed -- no leaks are possible
```

Il existe aussi d'autres fonctions d'allocation qui effectuent des actions supplémentaires :

- `calloc` qui alloue une plage mémoire et assure tous les éléments sont initialisés à 0.
- `realloc` qui permet de changer la taille d'une plage mémoire déjà allouée.

Par exemple les instructions suivantes allouent une plage mémoire, puis agrandissent la plage mémoire en gardant les éléments précédemment calculés :

```
int i;
int taille = 5;
int * tableau = NULL;
if((tableau = (int *)calloc(taille, sizeof(int))) == NULL) {
    exit(EXIT_FAILURE);
}
for(i = 0; i < taille; ++i) {
    tableau[i] = i;
}
for(i = 0; i < taille; ++i) {
    if(i) { printf(", "); }
    printf("%d", tableau[i]);
}
printf("\n");

taille = 10;
if((tableau = (int *)realloc(tableau, sizeof(int) * taille)) ==
↳ NULL) {
    exit(EXIT_FAILURE);
}
```

```
}  
for(i = 5; i < taille; ++i) {  
    tableau[i] = 10 - i;  
}  
for(i = 0; i < taille; ++i) {  
    if(i) { printf(", "); }  
    printf("%d", tableau[i]);  
}  
printf("\n");  
free(tableau);  
tableau = NULL;
```

En sortie avec valgrind :

```
0, 1, 2, 3, 4  
0, 1, 2, 3, 4, 5, 4, 3, 2, 1  
==3610==  
==3610== HEAP SUMMARY:  
==3610==      in use at exit: 0 bytes in 0 blocks  
==3610==    total heap usage: 3 allocs, 3 frees, 1,084 bytes  
↳ allocated  
==3610==  
==3610== All heap blocks were freed -- no leaks are possible
```



## 8.4 Résumé

Un pointeur est une type d'adresse : `type*` permet de sauvegarder l'adresse d'une variable `type`.

```
type variable;
&variable; /* adresse de variable */
type * adresse;
adresse = &variable; /* adresse prend comme valeur l'adresse de
↳ variable */
*adresse; /* déréréférence pour agir comme 'variable' */
```

Un tableau est l'adresse de son premier élément, manipuler un pointeur est équivalent à manipuler un tableau :

```
type tableau[];
type * pointeur = tableau;
pointeur[indice]; /* équivalent à tableau[indice] */
```

Un pointeur peut subir des opérations arithmétiques :

```
type * pointeur;
pointeur + 1; /* adresse de l'emplacement voisin suivant (au
↳ sizeof(type) près) */

pointeur - 1; /* adresse de l'emplacement voisin précédent (au
↳ sizeof(type) près) */

pointeur++; /* pointeur regarde l'élément suivant */
```

Réclamer de la mémoire :

```
type * pointeur = NULL;
pointeur = malloc(sizeof(type) * taille); /* alloue un tableau de
↳ taille 'taille' */

pointeur = calloc(taille, sizeof(type)); /* alloue un tableau de
↳ taille 'taille' et initialise les valeurs à 0 */

pointeur = realloc(pointeur, sizeof(type) * taille); /* change la
↳ taille de la plage mémoire allouée */
```

Gestion d'allocation dynamique propre :

```
type * tableau = NULL;
if((tableau = (type *)malloc(sizeof(type) * taille)) == NULL) {
    /* traitement à appliquer, sauvegarde si possible */
    exit(EXIT_FAILURE);
}
/* utiliser tableau */
free(tableau);
tableau = NULL;
```

## 8.5 Entraînement

Exercice noté 31 (★ Pointeurs sur des variables).

Vous êtes chargé d'effectuer un déménagement. Pour ceci, vous devrez modifier le code suivant mais seulement en ajoutant les caractères '\*', '&' et des parenthèses :

01\_demenagement.c : Code d'un déménagement

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

int main() {
    int ma_maison = 42;
    int nouvelle_maison = 0;
    int mon_adresse = ma_maison;
    int nouvelle_adresse = nouvelle_maison;
    printf("[%c] Je sais où est ma maison ?\n", (mon_adresse
    ↪ == &ma_maison) ? 'V' : 'X');
    printf("[%c] Je sais où déménager ?\n",
    ↪ (nouvelle_adresse == &nouvelle_maison) ? 'V' : 'X');
    nouvelle_adresse = mon_adresse;
    mon_adresse = 0;
    printf("[%c] J'ai tout déménagé dans ma nouvelle maison
    ↪ ?\n", (ma_maison == 0) ? 'V' : 'X');
    printf("[%c] J'ai oublié des choses dans ma maison ?\n",
    ↪ (nouvelle_maison == 42) ? 'V' : 'X');
    mon_adresse = nouvelle_adresse;
    mon_adresse++;
    printf("[%c] J'habite à ma nouvelle adresse ?\n",
    ↪ (mon_adresse == &nouvelle_maison) ? 'V' : 'X');
    printf("[%c] J'ai ajouté des choses dans ma maison ?\n",
    ↪ (nouvelle_maison == 43) ? 'V' : 'X');
    exit(EXIT_SUCCESS);
}
```

Ceci devrait transformer toutes les X en V de sorte d'obtenir la sortie suivante :

```
[V] Je sais où est ma maison ?  
[V] Je sais où déménager ?  
[V] J'ai tout déménagé dans ma nouvelle maison ?  
[V] J'ai oublié des choses dans ma maison ?  
[V] J'habite à ma nouvelle adresse ?  
[V] J'ai ajouté des choses dans ma maison ?
```

Exercice noté 32 (★★ Pointinception).

Vérifiez que vous avez compris le système d'adressage et déréférencement en complétant le code suivant.

```
int valeur = 42;  
int * ptr = ...; /* TODO : pointe sur valeur */  
int ** pptr = ...; /* TODO : pointe sur ptr */  
int *** ppptr = ...; /* TODO : pointe sur pptr */  
printf("valeur via valeur : %d\n", valeur);  
printf("valeur via ptr : %d\n", ...);  
printf("valeur via pptr : %d\n", ...);  
printf("valeur via ppptr : %d\n", ...);
```

```
valeur via valeur : 42  
valeur via ptr : 42  
valeur via pptr : 42  
valeur via ppptr : 42
```

## Exercice noté 33 (★★ Concaténation de chaînes de caractères).

Alice une habituée du langage python essaie de concaténer des chaînes de caractères avec l'opérateur '+' mais le compilateur refuse de faire ce qu'elle veut. Proposez à Alice une manière de faire ce qu'elle souhaite en langage C à partir de son code :

02\_alice.c

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

int main() {
    char nom[100], prenom[50];
    char * full_name = NULL;
    printf("Bonjour, entrez votre nom et prénom : ");
    scanf("%s %s", nom, prenom);
    full_name = prenom + ' ' + nom;
    printf("Vous êtes donc %s !\n", full_name);
    exit(EXIT_SUCCESS);
}
```

## Exercice noté 34 (★★★ Construction liste extensible).

Bob aimerait écrire un programme qui gère une liste aussi grande qu'il le souhaite. Pour vous aider, Bob vous propose de commencer avec son code :

03\_bob.c

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

int main() {
    const int ajouts = 10000;
    int * liste = malloc(sizeof(int));
    int * nouvelle_liste = NULL;
    int taille = 0;
    int i;
    int j;
    for(i = 0; i < ajouts; ++i) {
        liste[taille++] = i;
        nouvelle_liste = malloc(sizeof(int) * (taille + 1));
        for(j = 0; j < taille; ++j) {
            nouvelle_liste[j] = liste[j];
        }
        liste = nouvelle_liste;
    }
    printf("liste : [");
    for(i = 0; i < taille; ++i) {
        if(i) printf(", ");
        printf("%d", liste[i]);
    }
    printf("]\n");
    free(liste);
    exit(EXIT_SUCCESS);
}
```

1. Lorsque Bob lance `valgrind`, il semble qu'il y ait des fuites mémoires dans son programme :

```
==6552== HEAP SUMMARY:
==6552==      in use at exit: 200,020,000 bytes in 10,000
↳ blocks
==6552==    total heap usage: 10,002 allocs, 2 frees,
↳ 200,061,028 bytes allocated
```

Pourriez-vous y remédier ?

2. Modifiez le code pour supprimer les variables `j` et `nouvelle_liste`.
3. Bob aimerait maintenant que sa liste puisse gérer 1 000 000 000 ajouts. Mais il semble que ré-allouer à chaque ajout semble ralentir le programme. Essayez d'allouer par blocs de 10, observez-vous une amélioration des performances ?

## Exercice noté 35 (★★★ Trier des valeurs).

Nous voudrions comparer les performances de votre méthode pour trier un tableau à celle d'un algorithme connu pour être efficace : vous pouvez reprendre la méthode proposée lors du premier exercice ou en choisir une autre. Voici l'algorithme d'une méthode de tri efficace que vous devrez implémenter :

**Algorithm 5:** Tri par tas

---

```

input : Tableau valeurs
input : Entier taille
output: Tableau valeurs

1 for indice  $\leftarrow$  1 to taille - 1 do
2   current  $\leftarrow$  indice;
3   parent  $\leftarrow$  (current - 1) / 2;
4   while (current  $\neq$  0)  $\wedge$  (valeurs[current] > valeurs[parent]) do
5     echanger (valeurs[current], valeurs[parent]);
6     current  $\leftarrow$  parent;
7     parent  $\leftarrow$  (current - 1) / 2;
8   end
9 end
10 for indice  $\leftarrow$  taille - 1 to 1 do
11   echanger (valeurs[indice], valeurs[0]);
12   parent  $\leftarrow$  0;
13   while parent  $\times$  2 + 1 < indice do
14     if (parent  $\times$  2 + 2  $\geq$  indice)
15        $\vee$  (valeurs[parent  $\times$  2 + 1] > valeurs[parent  $\times$  2 + 2]) then
16       current  $\leftarrow$  parent  $\times$  2 + 1;
17     else
18       current  $\leftarrow$  parent  $\times$  2 + 2;
19     end
20     if valeurs[current] < valeurs[parent] then
21       break;
22     end
23     echanger (valeurs[current], valeurs[parent]);
24     parent  $\leftarrow$  current;
25   end
26 end

```

---



Pour comparer les performances des deux méthodes, on vous demande de compléter le code suivant :

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

#define TAILLE_DEMO 100
#define TAILLE_PERF 100000

/* afficher_tableau affiche la liste de ses éléments au format
↳ [e1, e2, ..., eN] */
void afficher_tableau(int * valeurs, int taille);
/* copie_tableau recopie les éléments de source dans destination
↳ */
void copier_tableau(int * destination, int * source, int taille);
/* aleatoire remplit un tableau par des valeurs aléatoires */
void aleatoire(int * valeurs, int taille);
/* swap échange les valeurs référencées par deux adresses */
void swap(int * a, int * b);
/* trier_algo est une implémentation du tri par tas */
void trier_algo(int * valeurs, int taille);
/* trier_eleve est votre méthode pour trier un tableau */
void trier_eleve(int * valeurs, int taille);
/* verif_trier renvoie 1 si la tableau est trié et 0 sinon */
int verif_trier(int * valeurs, int taille);

int main() {
    srand(time(NULL));
    int valeurs[TAILLE_PERF];
    int methode_eleve[TAILLE_PERF];
    int methode_algo[TAILLE_PERF];
    clock_t start, stop;

    aleatoire(valeurs, TAILLE_DEMO);
    copier_tableau(methode_eleve, valeurs, TAILLE_DEMO);
    copier_tableau(methode_algo, valeurs, TAILLE_DEMO);
    printf("non trié : ");
    afficher_tableau(valeurs, TAILLE_DEMO);
    trier_algo(methode_algo, TAILLE_DEMO);
    printf("methode_algo : ");
    afficher_tableau(methode_algo, TAILLE_DEMO);
```

```
trier_eleve(methode_eleve, TAILLE_DEMO);
printf("methode_eleve : ");
afficher_tableau(methode_eleve, TAILLE_DEMO);

aleatoire(valeurs, TAILLE_PERF);
copier_tableau(methode_eleve, valeurs, TAILLE_PERF);
copier_tableau(methode_algo, valeurs, TAILLE_PERF);
printf("tableau de taille %d :\n", TAILLE_PERF);
start = clock();
trier_algo(methode_algo, TAILLE_PERF);
stop = clock();
double temps_algo = (stop - start);
printf("methode_algo %s\n", verif_trier(methode_algo,
    ↪ TAILLE_PERF) ? "est trié" : "n'est pas trié");
printf("Temps écoulé : %g s\n", temps_algo / 1000000.);
start = clock();
trier_eleve(methode_eleve, TAILLE_PERF);
stop = clock();
double temps_eleve = (stop - start);
printf("methode_eleve %s\n", verif_trier(methode_eleve,
    ↪ TAILLE_PERF) ? "est trié" : "n'est pas trié");
printf("Temps écoulé : %g s\n", temps_eleve / 1000000.);
printf("Soit une différence de l'ordre d'un facteur %g\n",
    ↪ temps_eleve / temps_algo);
exit(EXIT_SUCCESS);
}
```

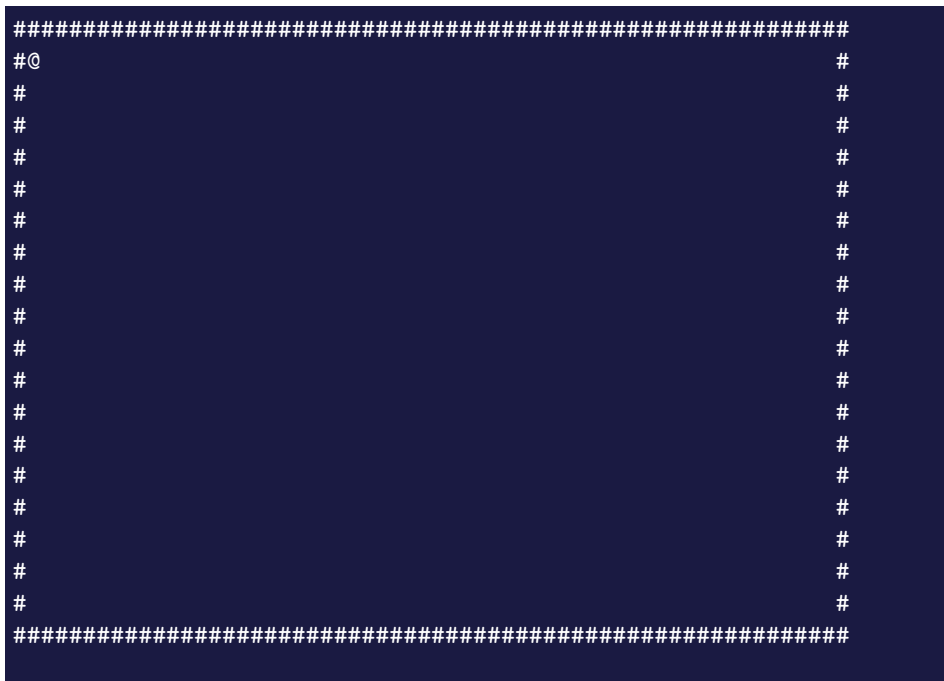
Vérifiez que le programme trie correctement le tableau avec les deux méthodes, puis comparez les temps écoulés pour trier le tableau.

## Exercice noté 36 (★★★ Allocation d'une grille de labyrinthe).

Charlie a vu qu'Alice s'était lancé dans un jeu de labyrinthe et aimerait aussi créer un jeu de labyrinthe mais avec des niveaux qui n'auraient pas nécessairement la même taille. Il a besoin de votre aide pour compléter son code et obtenir la sortie ci-dessous :

```
#include <stdio.h>
#include <stdlib.h>
#include <ncurses.h>
/* donne un pointeur sur une case de la grille */
char * grille_case(char * grille, int x, int y, int largeur, int
↳ hauteur);
/* crée une grille de taille donnée */
char * creer_grille(int largeur, int hauteur);
/* affiche une grille à l'écran */
void afficher_grille(char * grille, int largeur, int hauteur);

int main() {
    int largeur = 60, hauteur = 20;
    char * grille = creer_grille(largeur, hauteur);
    int x = 1, y = 1;
    initscr();
    noecho();
    cbreak();
    do {
        clear();
        afficher_grille(grille, largeur, hauteur);
        mvprintw(y, x, "@");
        mvprintw(y, x, "");
        refresh();
        getch();
        /* gestion des événements */
    } while(1);
    refresh();
    clrtoeol();
    refresh();
    endwin();
    free(grille);
    exit(EXIT_SUCCESS);
}
```



Maintenant vous êtes prêt à commencer le projet !

---

## 9 Projet : labyrinthe

---

### 9.1 Sujet

Étudiant de l'ESGI, nous avons une mission importante pour vous : vous devez créer un jeu de labyrinthe sur la base de ce que vous avez vu en cours. Pour ceci vous pourrez vous armer d'un coéquipier ou deux. Ceci peut par exemple ressembler à la sortie suivante :

```
#####
#@# #      ## # # ##      # # # #      # #
# # ### # # # # ## # # ## # # ## ## # # #
# # # # # # # # # ## # # # # ## # # ## # #
## # # ## # # ## # # # ### # # # ## # # # #
# # # # # # # # # # # # # ## # ## ## # # #
# # # ### # ## ## ## # ### ## # ## ### # ##
# ### # # # # ##### # # # # # # ## ## # #
## # ## # # ## # ## # # # # # # ##### # # ##
# ##### # ## # # ## # ### # # # # # # # #
# # # # # ##### # # # # ##### # # # # #
# # ## # ### # ## # ## # ##### # # # # #
# # # ### # # # # # # # # ## ## # # ## ##
# # # # ## # # # # # # # # ## ##### # # ##
# ## # ### # # # # # # # ##### # ## ##### #
# # # # # # # # # # # # # ##### # ## ##### #
# # # # # # # # # # # # # ## # # ## # # #
# # # # # # # # # # # # # ##### # # # #####
## # # # ## ## # # # # # # # ## ## x#
#####
```

Le joueur symbolisé par le symbole @ doit atteindre la sortie symbolisée par le symbole x. Cependant des murs # le bloquent et l'empêchent d'y arriver directement. Depuis un menu, permettez au joueur de choisir un niveau de difficulté (taille de carte) et proposez lui de rejouer. Vous pouvez ajouter des améliorations par choix dans celles proposées ou en en proposant jusqu'à 3 qui pourront être valorisées.

## 9.2 Évaluation

Les bibliothèques autorisées sont les bibliothèques standards et celles vues en cours et éventuellement la bibliothèque SDL. L'enseignant évaluera les projets sous Linux. Vérifiez que votre projet est viable avant de l'envoyer.

L'aspect technique du projet sera évalué selon le barème ci-dessous. Sans améliorations, la note maximale est de 12 / 20. Toutes les améliorations ne sont pas obligatoires pour obtenir la note de 20 / 20 : la notation peut atteindre au maximum 27 / 20.

### 9.2.1 (... / 6 points) Aspects code

Le code compile	0.5 point
Aucun warning	0.5 point
Code indenté	1 point
Conventions de nommage pertinentes	1 point
Découpe en fonctions pertinentes	1 point
Documentation (commentaires) pertinente	1 point
Rapport ou README.md	1 point

### 9.2.2 (... / 6 points) Aspects fonctionnalités

Pas de crash pendant exécution	1 point
Le code libère la mémoire	0.5 point
Menu	0.5 point
Affiche une grille	0.5 point
Déplace le personnage	0.5 point
Le personnage est bloqué par les murs	0.5 point
Le personnage ne sort pas de la carte	0.5 point
Termine le niveau	0.5 point
Permet de rejouer	0.5 point
Différentes tailles de carte	0.5 point
Niveau de labyrinthe (à la main)	0.5 point

### 9.2.3 (... / 15 points) Améliorations

Aucun warning avec <code>-Wall</code>	0.5 point
Séparation modulaire du code ( <code>.h</code> )	0.5 point
Un makefile est donné pour compiler	0.5 point
Couleurs pertinentes	0.5 point
Niveaux supplémentaires	1 point
Éditeur de niveaux	1 point
Avec caméra fixée sur le joueur	1 point
Adversaires (qui se déplacent vers le joueur pour le capturer)	1 point
La machine indique un chemin à suivre pour sortir	1 point
Sous interface graphique (SDL) : vue de haut ou minicarte	1 point
En vue style 3D (SDL)	1 point
Gestion de versions avec git	1 point
Labyrinthe généré automatiquement	2 points
Amélioration personnelle justifiée dans le rapport (Jusqu'à 3)	+1 point (chacune)

Le projet devra être déposé sur MyGES à votre enseignant dans une archive :

— `ESGI_1_PROJET_NOM1_Prenom1_NOM2_Prenom2.zip`

À vous de jouer !





Quatrième partie

Notions avancées



# Table des matières

<b>10 Consolidation des bases</b>	<b>183</b>
10.1 Remise dans le bain rapide . . . . .	183
10.1.1 Un programme en C . . . . .	183
10.1.2 Variables . . . . .	184
10.1.3 Opérations . . . . .	185
10.1.4 Tests et disjonctions . . . . .	186
10.1.5 Boucles . . . . .	187
10.1.6 Fonctions . . . . .	188
10.1.7 Tableaux . . . . .	189
10.1.8 Pointeurs . . . . .	190
10.2 Entraînement . . . . .	193
<b>11 Fichiers</b>	<b>197</b>
11.1 Ouverture et création de fichiers . . . . .	197
11.2 Lecture et écriture . . . . .	198
11.2.1 Avec fonctions formatées . . . . .	198
11.2.2 Par caractère . . . . .	200
11.2.3 Par bloc mémoire . . . . .	201
11.3 Se déplacer dans un fichier . . . . .	203
11.4 Flux standards d'entrées et sorties . . . . .	205
11.4.1 stdout . . . . .	205
11.4.2 stdin . . . . .	205
11.4.3 stderr . . . . .	206
11.5 Résumé . . . . .	207
11.6 Entraînement . . . . .	209

---

<b>12 Structures</b>	<b>213</b>
12.1 Typedef	213
12.2 Structures	214
12.3 Définition	215
12.4 Avec des pointeurs	216
12.4.1 Champs de bits	219
12.5 Unions	221
12.6 Énumérations	222
12.7 Résumé	225
12.8 Entraînement	226
<b>13 Programmation modulaire</b>	<b>233</b>
13.1 Retour sur la compilation	233
13.1.1 Externaliser une fonction	233
13.1.2 Compilation séparée	235
13.2 Directives préprocesseur	237
13.2.1 include	238
13.2.2 define	239
13.2.3 conditionnement	244
13.2.4 Module	245
13.3 Makefiles	248
13.3.1 Concept	248
13.3.2 Possibilités	248
13.4 Résumé	253
13.5 Entraînement	255
<b>14 Opérations bit-à-bit</b>	<b>267</b>
14.1 Décalages	267
14.1.1 À gauche	267
14.1.2 À droite	268
14.2 Négation	268
14.3 Opérateurs booléens	269
14.3.1 et	269
14.3.2 ou inclusif	270
14.3.3 ou exclusif : xor	271
14.4 Résumé	273
14.5 Entraînement	275

---

---

<b>15 Types génériques</b>	<b>281</b>
15.1 Pointeurs de fonctions . . . . .	282
15.1.1 Définition . . . . .	282
15.1.2 Appel . . . . .	283
15.1.3 Constructions plus complexes . . . . .	284
15.1.4 Typedef . . . . .	286
15.2 Pointeurs génériques . . . . .	288
15.2.1 Intérêt . . . . .	289
15.2.2 Exemple avec qsort . . . . .	289
15.2.3 Manipulation . . . . .	291
15.3 Fonctions variadiques . . . . .	294
15.4 Résumé . . . . .	296
15.5 Entraînement . . . . .	298
 <b>16 Projet final</b>	 <b>303</b>
16.1 Étapes . . . . .	303
16.2 Possibilités de sujet . . . . .	303
16.3 Livrables . . . . .	304
16.4 Évaluation . . . . .	304
16.4.1 (... / 9 points) Code . . . . .	305
16.4.2 (... / 7 points) Fonctionnalités . . . . .	305
16.4.3 (... / 5 points) Soutenance et rapport . . . . .	306



---

## 10 Consolidation des bases

---

Les vacances sont passées, mais nous revoilà à étudier le langage C ! On peut faire quelques rappels rapides avant d'avancer.

### 10.1 Remise dans le bain rapide

#### 10.1.1 Un programme en C

Dans un programme en C :

1. Vous commencez par importer les bibliothèques dont les fonctionnalités vous seront utiles avec `#include`.
2. Vous listez les déclarations des fonctions que vous pourriez vouloir définir.
3. Vous écrivez les instructions à suivre au lancement de votre application dans la fonction `main`.
4. Vous listez les déclarations de vos fonctions.

Rappelez vous, ça ressemble à ça :

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

/* Déclaration des fonctions */

int main() {

    /* Instructions de l'application */

    exit(EXIT_SUCCESS);
}

/* Définition des fonctions */
```

Pour compiler un tel programme depuis un fichier source `main.c`. On peut utiliser `gcc` et préciser le nom du programme avec l'option `-o` :

```
gcc -o monProgramme main.c
./monProgramme
```

### 10.1.2 Variables

En langage C, les variables sont typées. Les types atomiques sont :

**Entiers :**

```
char /* 1 octet */
short /* 2 octets */
int /* 4 octets */
long /* 8 octets */
unsigned type /* positif */
```

**Flottants :**

```
float /* 4 octets */
double /* 8 octets */
long double /* 16 octets */
```

Vous devez les utiliser lorsque vous déclarez une variable ou pour forcer un changement de type :

```
unsigned int valeurPositive = 42; /* Affectation de 42 */
valeurPositive = 0x2a; /* Réaffectation en hexadécimal */
```

Pour imprimer des caractères dans le terminal et afficher les valeurs de vos variables vous avez la fonction d'affichage formaté `printf` :

```
printf("caractere : \'%c\'\\n", '@');
printf("entier : %d\\n", 42);
printf("hexadécimal : %x\\n", 0x2a);
printf("entier positif : %u\\n", 3000000000u);
printf("entier long : %ld\\n", 42000000000);
printf("flottant : %g\\n", 3.14);
printf("adresse : %p\\n", NULL);
```

Pour lire les caractères saisis par l'utilisateur dans son terminal, vous avez `scanf`. Notez que vous devrez passer une adresse à `scanf` pour réussir votre affectation :



```
int entier;
scanf("%d", &entier);
float flottant;
scanf("%f", &flottant);
char caractere;
scanf(" %c", &caractere);
```

### 10.1.3 Opérations

Souvent il va être intéressant de combiner les valeurs en utilisant des opérateurs. Les opérateurs classiques sont les suivants :

```
/*first et second deux variables de type entier ou flottant*/
first + second /* addition */
first - second /* soustraction */
first * second /* multiplication */
first / second /* division */
first % second /* modulo : entiers */
```

Notez que dans le cas d'entiers, vous seriez amenés à utiliser une coercition pour gérer un dépassement de capacité d'un `int` ou pour réaliser une division avec résultat flottant :

```
int first, second;
/* ... */
long multiplication = (long)first * second;
float division = (float)first / second;
```

En langage C, il existe des raccourcis pour incrémenter une valeur ou utiliser un opérateur :

```
int valeur = 1;
valeur = valeur + 1; /* ajoute 1 */
valeur += 1;         /* ajoute 1 */
valeur++;            /* ajoute 1 */
++valeur;            /* ajoute 1 */
```

### 10.1.4 Tests et disjonctions

Il est possible de réaliser une disjonction de cas sous condition à l'aide d'un `if-else` :

```
int first, second;
scanf("%d %d", &first, &second);
if(first > second) {
    printf("%d est plus grand.\n", first);
} else if(first < second) {
    printf("%d est plus grand.\n", second);
} else {
    printf("les deux sont égaux.\n");
}
```

On différencie le test d'égalité `==` de l'affectation `=`. Les opérateurs de comparaison sont les suivants :

<i>/* égalité : */</i>	<i>/* différence : */</i>
<code>a == b</code>	<code>a != b</code>
<i>/* plus petit strict : */</i>	<i>/* plus grand strict : */</i>
<code>a &lt; b</code>	<code>a &gt; b</code>
<i>/* plus petit ou égal : */</i>	<i>/* plus grand ou égal : */</i>
<code>a &lt;= b</code>	<code>a &gt;= b</code>

Pour assembler les valeurs de vérité renvoyées par ces opérateurs, on utilise les opérateurs suivants :

```
a && b /* vrai lorsque les deux le sont */
a || b /* vrai lorsque d'un l'est */
! a /* vrai lorsque faux et réciproquement */
```

Il est possible de synthétiser une affectation dans un `if-else` à l'aide d'une opération ternaire :

```
if(a < b) {  
    res = a;  
} else {  
    res = b;  
}
```

 $\Leftrightarrow$ 

```
res = (a < b) ? a : b;
```

Pour des conditions simples d'égalité à une constante, il est possible d'utiliser un `switch` pour se brancher à un bloc correspondant :

```
switch(expression) {  
    case 1 : {  
        /* instructions */  
    } break;  
    case 2 : {  
        /* instructions */  
    } break;  
    default : {  
        /* instructions */  
    }  
}
```

### 10.1.5 Boucles

Il est possible d'automatiser la répétition d'instructions à l'aide de boucles.

On utilise :

- `while` pour répéter les instructions tant qu'une condition est vraie.

```
while(condition) {  
    /* instructions */  
}
```

- `do-while` pour répéter à nouveau les instructions lorsqu'une condition est vérifiée.

```
do {  
    /* instructions */  
} while(condition);
```

- for lorsque la boucle while peut s'écrire avec une initialisation et une évolution des données utilisées pour la condition.

```
for(initialisation; condition; evolution) {  
    /* instructions */  
}
```

Il est aussi possible de relancer la boucle prématurément à l'aide du mot-clé `continue` ou de la stopper avant vérification de la condition par le mot-clé `break`.

### 10.1.6 Fonctions

Une fonction se définit par un type de retour, un nom d'appel et des paramètres :

```
typeRetour nomFonction(/* paramètres */) {  
    /* instructions */  
}  
  
/* Exemple de fonction d'addition d'entiers : */  
int addition(int first, int second) {  
    int res; /* variable locale à la fonction */  
    res = first + second;  
    return res; /* renvoi lors de l'appel */  
}
```

Cette fonction peut être déclarée en amont par sa signature (type de retour, nom et type des paramètres) et appelée par son nom :

```
/* Exemple de fonction d'addition d'entiers : */  
/* déclaration */  
int addition(int, int);
```

```
int main() {  
    /* appel */  
    int deux = addition(1, 1);  
    exit(EXIT_SUCCESS);  
}  
  
/* définition */  
int addition(int first, int second) {  
    return first + second;  
}
```

### 10.1.7 Tableaux

Un tableau est l'outil en langage C qui permet de créer une liste de taille donnée pour une type de variable que l'on souhaite répéter :

```
type tableau[TAILLE_CONSTANTE];  
type tableau[] = {valeur1, valeur2, ..., valeurN};  
  
int liste[] = {1, 2, 3};  
liste[1]; /* accès au second élément : d'indice 1 */
```

Les tableaux sont utilisés pour gérer les chaînes de caractères (se terminant par un marque de fin '\0') :

```
char chaine[] = "Hello ESGI !";  
printf("%s\n", chaine); /* affichage de chaine */  
scanf("%s", chaine); /* lecture d'un mot au clavier */
```

Il est possible de gérer des tableaux à plusieurs dimensions :

```
/* tableau à deux dimensions */  
int grille[HAUTEUR][LARGEUR] = {  
    {0, 0, 0, 0},  
    {0, 1, 2, 0},  
    {0, 0, 0, 0}
```

```
};

grille[ligne][colonne]; /* accès au tableau */

/* passage d'un tableau à deux dimensions */
void afficherGrille(int largeur, int hauteur,
    grille[hauteur][largeur]) {
    /* instructions */
}
```

### 10.1.8 Pointeurs

Toute donnée en mémoire possède une adresse, les pointeurs sont l'outil qui permet d'utiliser ces adresses dans votre programme. Nous avons vu que nous pouvons récupérer l'adresse d'une variable à l'aide de l'opération unaire `&`. Pour sauvegarder cette adresse dans une variable, on construira une pointeur sur cette variable dont le type prendra une étoile par rapport à celui de la variable. L'accès à la donnée pointée se fera ensuite à l'aide d'un déréférencement par l'opérateur unaire `*` :

```
int variable = 42;
int * pointeur = &variable;
*pointeur = 1337;
printf("%d\n", variable); /* affiche 1337 */
```

Pour rappel, un tableau correspond à l'adresse de son premier élément (d'indice 0), les autres étant alignés en mémoire après celui-ci. La manipulation d'un tableau peut aussi s'appliquer avec un pointeur :

```
int tableau[] = {1, 2, 3, -1};
int * pointeur = tableau;
int i;
for(i = 0; pointeur[i] >= 0; ++i) {
    printf("%d\n", pointeur[i]);
}
```

À noter qu'un pointeur peut changer de valeur (pointer vers une autre adresse), ce qui permet par exemple de jouer avec l'arithmétique des pointeurs :

```
char texte[] = "Hello !";
char * pointeur = NULL;
for(pointeur = texte; *pointeur != '\0'; ++pointeur) {
    if(*pointeur >= 'a' && *pointeur <= 'z')
        *pointeur += 'A' - 'a';
}
printf("%s\n", texte); /* affiche "HELLO !" */
```

L'un des grands intérêts des pointeurs est de pouvoir manipuler des plages mémoire allouées dynamiquement : de taille décidée à l'exécution du programme.

```
float * notes = NULL;
float somme = 0;
int nombre;
int i;
printf("Combien de CC ? ");
scanf("%d", &nombre);
if(nombre <= 0) {
    printf("Pas de notes pas de moyenne.\n");
    exit(EXIT_FAILURE);
}
/* allocation dynamique depuis le nombre donné par l'utilisateur
↳ */
if((notes = (float *)malloc(sizeof(float) * nombre)) == NULL) {
    printf("Erreur d'allocation.\n");
    exit(EXIT_FAILURE);
}
for(i = 0; i < nombre; ++i) {
    scanf("%f", notes + i);
    somme += notes[i];
}
printf("La moyenne de ");
for(i = 0; i < nombre; ++i) {
    if(i && i == nombre - 1) printf(" et ");
    else if(i > 0) printf(", ");
}
```

```
    printf("%g", notes[i]);  
}  
printf(" est %g\n", somme / nombre);  
  
free(notes);  
notes = NULL;  
exit(EXIT_SUCCESS);
```

Les principales fonctions d'allocation sont les suivantes :

```
/* allouer une plage mémoire */  
malloc(/*taille mémoire en octets*/)  
  
/* allouer un tableau avec chaque élément à 0 */  
calloc(/*taille tableau*/, /*taille élément*/)  
  
/* modifier la taille d'une plage allouée */  
realloc(/*plage à modifier*/, /*nouvelle taille en octets*/)
```

À jour et prêt pour la suite ? Pas de panique, on fait des exercices là dessus pour se remettre dans le bain.



## 10.2 Entraînement

Exercice noté 37 (★★★ Statistiques sur liste d'entiers).

Écrire un programme qui lit une liste d'entiers et affiche les statistiques suivantes :

- Valeur minimale.
- Valeur maximale.
- Moyenne des valeurs.

```
Entrez des entiers positifs : 14 12 10 8 7 -1
min : 7
max : 14
moyenne : 10.2
```

Exercice noté 38 (★★★ Extraire information d'une chaîne de caractères).

Compléter le code suivant de manière à extraire les informations de la chaîne de caractère. Puis afficher ces informations :

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

int main() {
    const char * infos = "Linus Torvalds 52 ans C";
    /* TODO : extraire les informations */
    exit(EXIT_SUCCESS);
}
```

```
Prenom : Linus
Nom : Torvalds
Age : 52
Parle couramment la langue C
```

**Exercice noté 39 (★ ★ ★ Mini-interpréteur sur liste d'entiers).**

Un ami a entendu parler du langage Brainfuck. C'est un langage où vous pouvez vous déplacer dans la mémoire et y changer des valeurs. Il aimerait que vous en codiez un mini-interpréteur dans une plage de taille donnée par l'utilisateur en suivant les règles suivantes :

- '+' ajoute 1 à la case mémoire regardée par le curseur.
- '-' retire 1 à la case mémoire regardée par le curseur.
- '=' égalise toutes les cases mémoire à la valeur de la case mémoire regardée.
- '>' déplace le curseur vers la droite (retour au début si dépasse de la plage mémoire).
- '<' déplace le curseur vers la gauche (retour à la fin si dépassé de la plage mémoire).
- '.' affiche le contenu de la plage mémoire.

```
taille de la mémoire : 4
.
[0, 0, 0, 0]
+.
[1, 0, 0, 0]
=.
[1, 1, 1, 1]
>++.
[1, 3, 1, 1]
=.
[3, 3, 3, 3]
<--.
[1, 3, 3, 3]
<++++.
[1, 3, 3, 7]
-->>-.
[1, 2, 3, 5]
```

**Exercice noté 40 (★★★ Jeu du Morpion).**

Coder sous console un jeu de Morpion :

- Jeu à deux joueurs dans une grille de  $3 \times 3$ .
- Tour par tour chacun place un pion.
- Lorsque 3 pions d'un même joueur sont alignés, il gagne.

```
+--+--+  
|X| |O|   Le joueur X gagne !  
+--+--+  
| |X| |  
+--+--+  
|O| |X|  
+--+--+
```

**Exercice noté 41 (★★★ Enregistrement et recherche de numéros).**

Écrire un programme qui demande une liste de noms et associe à chaque nom un numéro. Une fois que l'utilisateur a validé sa liste, lui proposer de rechercher un nom. Si le nom a été renseigné, on lui affiche le numéro associé. Une sortie de ce programme pourrait ressembler à la suivante :

```
Nom (None pour arrêter) : Personne
Numéro : 42
Nom (None pour arrêter) : Moi
Numéro : 1337
Nom (None pour arrêter) : Lui
Numéro : 1234
Nom (None pour arrêter) : None
Nom à rechercher (None pour arrêter) :
>>> Personne
Le numéro de "Personne" est 42
Nom à rechercher (None pour arrêter) :
>>> Toi
"Toi" non trouvé.
Nom à rechercher (None pour arrêter) :
>>> Moi
Le numéro de "Moi" est 1337
Nom à rechercher (None pour arrêter) :
>>> None
```

---

# 11 Fichiers

---

Nous avons vu comment communiquer avec l'utilisateur, organiser nos instructions pour réaliser un programme. Cependant, à la fin de l'exécution de notre programme, nous reviendrions à zéro en le relançant. Il serait donc intéressant d'avoir un moyen de sauvegarder cet état. De même nous nous sommes limités à des entrées courtes et simples données par un utilisateur lors de l'exécution de son programme.

Pour sauvegarder et récupérer de l'information, il est possible de passer par la gestion des fichiers sur l'espace disque.

## 11.1 Ouverture et création de fichiers

Les fichiers vont se gérer dans le programme à l'aide du type `FILE *`. Celui-ci permettra de garder un pointeur sur un fichier ouvert par le programme. L'ouverture d'un fichier depuis l'espace disque se fait par la commande `fopen` de `stdio.h` :

```
fopen(/* chemin du fichier */, /* mode d'ouverture */) 
```

- Le premier argument est le chemin du fichier sur l'espace disque.
- Le second argument est le mode d'ouverture du fichier, ceci dépend si l'on souhaite lire, écrire, s'y déplacer et autre :

Mode	Lecture	Écriture	Création	Effacement	Ajout en fin
"r"	✓				
"w"		✓	✓	✓	
"a"		✓	✓		✓
"r+"	✓	✓			
"w+"	✓	✓	✓	✓	
"a+"	✓	✓	✓		✓

Une fois les opérations dans le fichier terminées, on arrête son utilisation par `fclose` sur celui-ci :

```
FILE * fichier = NULL;
/* Tentative d'ouverture / création d'un fichier */
if((fichier = fopen(/* chemin */, /* mode */)) == NULL) {
    /* Gestion de l'impossibilité d'ouverture */
}
/* Opérations avec le fichier */
/* ... */
/* Fermeture du fichier */
fclose(fichier);
fichier = NULL;
```

L'exemple suivant permet la création d'un fichier `MonFichier.txt` vide :

```
FILE * fichier = NULL;
if((fichier = fopen("MonFichier.txt", "w")) == NULL) {
    printf("Erreur de création de mon fichier.\n");
    exit(EXIT_FAILURE);
}
printf("Fichier créé avec succès.\n");
fclose(fichier);
fichier = NULL;
exit(EXIT_SUCCESS);
```

Pour alléger la lecture des codes suivants, la vérification de l'existence ou création du fichier peut être omise. Cependant, pensez à le gérer pour éviter des mauvaises surprises dans vos codes.

## 11.2 Lecture et écriture

### 11.2.1 Avec fonctions formatées

Pour communiquer avec l'utilisateur via le terminal nous avons vu des fonctions issues de `stdio` qui le permettent telles que `printf`, `scanf`. Ces fonctions existent aussi en version pour les fichiers.

## fprintf

`fprintf` permet l'impression formatée de caractères dans un fichier donné. Il fonctionne de la même manière que `printf`, à la différence qu'il faut fournir un premier argument supplémentaire : le pointeur sur le fichier :

```
FILE * fichier = fopen("test.txt", "w");
fprintf(fichier, "Hello fichier !\n");
fclose(fichier);
```

De même `fprintf` peut être utilisé pour sauvegarder des données fournies par à un utilisateur :

```
int renseignerInfos(const char * pseudo) {
    int age;
    printf("Quel est votre âge ? ");
    scanf("%d", &age);
    FILE * infos = fopen(pseudo, "w");
    /* sauvegarde la valeur de age */
    fprintf(infos, "%d\n", age);
    fclose(infos);
}
```

## fscanf

La fonction `fscanf` fonctionne de la même manière pour récupérer le contenu formaté d'un fichier :

```
int lireInfos(const char * pseudo, int * age) {
    FILE * infos = NULL;
    if((infos = fopen(pseudo, "r")) == NULL) {
        return 0; /* l'utilisateur est inconnu */
    }
    /* récupère la valeur de age */
    fscanf(infos, "%d", age);
    fclose(infos);
    return 1;
}
```

### 11.2.2 Par caractère

Dans des cas plus pratiques, nous pourrions vouloir récupérer ou écrire les informations caractère par caractère. Par exemple si vous souhaitez coder un message en conservant la mise en page. Pour ceci, nous avons à disposition les fonctions `fputc` pour écrire un caractère dans un fichier et `fgetc` pour lire un caractère depuis un fichier.

#### `fgetc`

`fgetc` va lire et renvoyer un par un les caractères d'un fichier donné. Lorsque `fgetc` est arrivé à la fin du fichier, la valeur renvoyée sera EOF (End Of File).

```
FILE * fichier = fopen("message.txt", "r");
int caractere;
/* tant qu'on lit un caractère dans le fichier */
while((caractere = fgetc(fichier)) != EOF) {
    putchar(caractere); /* on affiche le caractere */
}
fclose(fichier);
```

#### `fputc`

Il est tout aussi possible d'écrire dans un fichier caractère par caractère. Ceci peut se faire à l'aide de `fputc`.

```
FILE * input = fopen("message.txt", "r");
FILE * output = fopen("resultat.txt", "w");
int caractere;
const int cle = 5;
/* tant qu'on lit un caractère dans le fichier */
while((caractere = fgetc(input)) != EOF) {
    /* on le code s'il est alphabétique */
    if(caractere >= 'a' && caractere <= 'z')
        caractere = (caractere - 'a' + cle) % 26 + 'a';
    else if(caractere >= 'A' && caractere <= 'Z')
```



```
    caractere = (caractere - 'A' + cle) % 26 + 'A';  
    /* on l'écrit dans le fichier de sortie */  
    fputc(caractere, output);  
}  
fclose(input);  
fclose(output);
```

### 11.2.3 Par bloc mémoire

Pour effectuer des sauvegardes, nous pourrions souhaiter de ne pas avoir nécessité que le fichier soit humainement lisible et ne prenne pas une place plus importante que celle des données réelles. Pour ceci, il est possible d'écrire dans un fichier en binaire directement.

Cette écriture en binaire se fera par octets. On lui passera un tableau ou un pointeur contenant ou recevant les données qui nous intéressent. On précisera que le fichier manipulé n'est pas un fichier texte en ajoutant un "b" après le mode d'ouverture "r", "w" ou "a". Les fonctions utilisées ensuite pour la lecture et l'écriture seront `fread` et `fwrite`.

#### `fwrite`

La fonction `fwrite` prend en paramètres :

- Pointeur sur les données à écrire.
- Taille d'un élément.
- Nombre d'éléments.
- Pointeur sur le fichier où écrire.

`fwrite` renvoie le nombre d'éléments écrits dans le fichier.

```
int sauvegarderListe(const char * filepath, const int * liste, int  
↳ taille) {  
    FILE * output = fopen(filepath, "wb");  
    /* écriture de la taille : une variable */  
    if(fwrite(&taille, sizeof(int), 1, output) != 1) {  
        printf("Erreur écriture taille\n");  
        return 0;  
    }  
}
```

```
/* écriture de la liste : un tableau */
if(fwrite(liste, sizeof(int), taille, output) != taille) {
    printf("Erreur écriture liste\n");
    return 0;
}
fclose(output);
return 1;
}
```

### fread

Une liste ainsi écrite précédemment peut être chargée par le même type de procédé. `fread` permet la lecture de données et fonctionne similairement à `fwrite`. À noter que `fread` ne procède pas à l'allocation de la plage mémoire dans laquelle elle écrit. Ce sera votre rôle de vous assurer qu'elle existe et peut recevoir les données.

La fonction `fread` prend en paramètres :

- Pointeur sur les données à récupérer.
- Taille d'un élément.
- Nombre d'éléments.
- Pointeur sur le fichier où lire.

`fread` renvoie le nombre d'éléments lus depuis le fichier.

```
int chargerListe(const char * filepath, int ** liste, int *
↳ taille) {
    FILE * input = fopen(filepath, "rb");
    /* lecture de la taille : nécessaire à l'allocation */
    if(fread(taille, sizeof(int), 1, input) != 1) {
        printf("Erreur lecture taille\n");
        return 0;
    }
    /* allocation de la liste */
    if((*liste = (int *)malloc(sizeof(int) * *taille)) == NULL) {
        printf("Erreur allocation liste\n");
    }
}
```

```
    return 0;
}
/* lecture des éléments de la liste */
if(fread(*liste, sizeof(int), *taille, input) != *taille) {
    printf("Erreur lecture liste\n");
    return 0;
}
fclose(input);
return 1;
}
```

## 11.3 Se déplacer dans un fichier

Des fonctions sont proposées pour jouer sur la position du curseur lors de l'écriture et de la lecture dans un fichier.

`ftell` prend pour argument un fichier et indique la position actuelle du curseur dans ce fichier.

`rewind` prend pour argument un fichier et rembobine le fichier au début.

`fseek` permet de placer le curseur à un endroit souhaité dans un fichier. `fseek` prend en arguments :

- Le fichier dans lequel déplacer le curseur.
- La position relative où déplacer le curseur par rapport à l'information donnée au paramètre suivant.
- Point de repère depuis lequel appliquer le décalage :
  - `SEEK_SET` : début du fichier.
  - `SEEK_CUR` : position actuelle dans le fichier.
  - `SEEK_END` : fin du fichier.

En exemple d'application de ces fonctions, nous proposons un code qui lit le texte pour compter le nombre de phrases présentes puis lit chaque phrase :

```
int carInChaine(char car, const char * chaine) {
    for(; *chaine != '\0'; ++chaine) {
```

```
        if(car == *chaine)
            return 1;
    }
    return 0;
}

int lirePhrase(FILE * file, long * start, long * end) {
    int car;
    while((car = fgetc(file)) != EOF) {
        if(! carInChaine(car, "\t\n ")) {
            break;
        }
    }
    if(start) /* on récupère la position du début de la phrase */
        *start = ftell(file) - 1;
    do {
        if(car == '.') {
            break;
        }
    } while((car = fgetc(file)) != EOF);
    if(end) /* on récupère la position de la fin de la phrase */
        *end = ftell(file);
    return car != EOF;
}

void afficherPortionFichier(FILE * file, long start, long end) {
    int car;
    /* on se replace dans le fichier à la position indiquée */
    fseek(file, start, SEEK_SET);
    while(ftell(file) != end) {
        putchar(fgetc(file));
    }
}
```

```
int main() {
    FILE * file = fopen("message.txt", "r");
    int i;
    long start, end;
    for(i = 0; lirePhrase(file, NULL, NULL); ++i);
    printf("%d phrases.\n", i);
    /* on rembobine le fichier au début */
    rewind(file);
    for(i = 0; lirePhrase(file, &start, &end); ++i) {
        printf(" - Phrase %d (%ld caracteres): ", i + 1, end - start);
        afficherPortionFichier(file, start, end);
        printf("\n");
    }
    fclose(file);

    exit(EXIT_SUCCESS);
}
```

## 11.4 Flux standards d'entrées et sorties

En réalité, lorsque vous utilisez `printf` et `scanf`, vous travaillez également dans des `FILE *` standards qui correspondent aux entrées et sorties de votre terminal.

### 11.4.1 `stdout`

Le flux de sortie dans lequel on imprime nos caractères lors des appels à `printf` par exemple est `stdout`. On peut l'utiliser avec `fprintf` :

```
printf("Par printf\n");
fprintf(stdout, "Par fprintf\n");
```

### 11.4.2 `stdin`

De la même manière il existe le flux de sortie standard `stdin` peut être utilisé avec `fscanf` :

```
int nombreScanf, nombreFscanf;
scanf("%d", &nombreScanf);
fscanf(stdin, "%d", &nombreFscanf);
printf("%d %d\n", nombreScanf, nombreFscanf);
```

### 11.4.3 stderr

Le flux standard qui sera cependant le plus utile est `stderr`. C'est un flux standard de sortie dédié à l'écriture des erreurs ou des logs. À noter que `stdout` ne sera souvent réellement imprimé dans le terminal que lorsque son buffer est plein ou lorsqu'il imprime un retour à la ligne. `stderr` sera imprimé peu importe les caractères donnés.

```
FILE * fichier = NULL;
if((fichier = fopen("fichier_a_ne_pas_creer", "r")) == NULL) {
    fprintf(stderr, "Erreur main() : \"fichier_a_ne_pas_creer\" est
    ↪ introuvable\n");
    exit(EXIT_FAILURE);
}
fclose(fichier);
```

À noter que cette sortie peut être redirigée vers un fichier de logs, laissant dans le terminal uniquement `stdout` :

```
# gcc -o prog main.c
# ./prog
Erreur main() : "fichier_a_ne_pas_creer" est introuvable
# ./prog 2>log
# cat log
Erreur main() : "fichier_a_ne_pas_creer" est introuvable
```

## 11.5 Résumé

Il est possible de gérer un fichier avec le type `FILE *`. Ce fichier s'ouvre avec `fopen` et se ferme avec `fclose` :

```
FILE * fichier = NULL;
if((fichier = fopen(/* chemin */, /* mode */)) == NULL) {
    /* traitement erreur */
}
/* utilisation fichier */
fclose(fichier);
```

Les principaux modes sont :

- `r` : lecture.
- `w` : écriture (création et effacement du fichier).
- `a` : écriture (création et ajout en fin du fichier).
- `r+` : écriture et lecture.
- `w+` : écriture et lecture (création et effacement du fichier).
- `a+` : écriture et lecture (création et ajout en fin du fichier).

Il est possible d'imprimer une chaîne de caractères formatée dans un fichier avec `fprintf` et lire une chaîne de caractères formatée avec `fscanf` :

```
fprintf(fichier, /* format */, /* variables */);
scanf(fichier, /* format */, /* adresses */);
```

Il est aussi possible de procéder caractère par caractère avec `fgetc` et `fputc` :

```
char car;
while((car = fgetc(fichier)) != EOF) {
    fputc(car, fichier);
}
```

De manière plus avancée, il est possible de passer en binaire en ajoutant `b` au mode. Ceci permet d'écrire nos données avec `fwrite` et lire avec `fread` :

```
if(fwrite(/* pointeur */, /* taille élément */, /* nombre  
↪ éléments */, fichier) != /* nombre éléments */)  
    /* gestion erreur écriture */  
if(fread(/* pointeur */, /* taille élément */, /* nombre éléments  
↪ */, fichier) != /* nombre éléments */)  
    /* gestion erreur lecture */
```

Il est possible de jouer avec la position du curseur dans un fichier avec les fonctions suivantes :

```
ftell(fichier); /* donne la position du curseur */  
rewind(fichier); /* met le curseur au début du fichier */  
fseek(fichier, 1, SEEK_SET); /* se positionne après le premier  
↪ octet du fichier */  
fseek(fichier, 1, SEEK_CUR); /* déplace le curseur d'un caractère  
↪ vers l'avant */  
fseek(fichier, -1, SEEK_END); /* se positionne avant le dernier  
↪ octet du fichier */
```

La bibliothèque `stdio` définit des entrées en sorties standards :

```
stdout /* sortie standard, utilisée par printf */  
stdin /* entrée standard, utilisée par scanf */  
stderr /* sortie d'erreur standard */
```



## 11.6 Entraînement

Exercice noté 42 (★★ Compteur de lancements).

Écrire un programme qui compte le nombre de fois où il a été lancé. Ceci pourrait se faire à l'aide de la sauvegarde d'un entier dans un fichier.

```
# ./prog
Programme lancé 1 fois
# ./prog
Programme lancé 2 fois
# ./prog
Programme lancé 3 fois
# ./prog
Programme lancé 4 fois
```

Exercice noté 43 (★★★ Codage Vigenère depuis fichier).

Écrire un programme qui lit du texte depuis un fichier texte. Puis l'encode ou le décode avec la méthode du chiffre de Vigenère pour l'afficher sans la sortie standard.

```
# ./prog
Attendu : ./prog [FICHIER MESSAGE] [CLE]
Attendu : ./prog [FICHIER MESSAGE] [CLE] decode
# ./prog message.txt ESGI
Ipkutdk li lkfxw zzsh ipsmkbbxw !
Kvjat dsarà, çi hwbzeaz ti xgqv.
```

Exercice noté 44 (★★ Sauvegarde et chargement en binaire).

Écrire un programme qui sauvegarde et charge un personnage dans un fichier binaire. La manipulation du programme se fait en ligne de commande en passant des options au programme. L'écriture dans le fichier doit respecter la spécification suivante :

- (4 octets) entier : nombre de caractères constituant le nom du personnage.
- (N octets) chaîne de caractères : nom du personnage constitué de N caractères.
- (4 octets) entier : statistique de vie du personnage.
- (4 octets) entier : statistique d'attaque du personnage.
- (4 octets) entier : statistique de défense du personnage.
- (4 octets) entier : statistique de vitesse du personnage.

```
# ./prog
Attendu :
    ./prog -create [FICHIER] [NOM] [VIE] [ATK] [DEF] [VIT]
    ./prog -read [FICHIER]
# ./prog -create chouette.perso "Chouette Oiseau" 127 42 87 94
# ./prog -read chouette.perso
Personnage : {
    Nom : Chouette Oiseau
    Vie : 127
    Attaque : 42
    Défense : 87
    Vitesse : 94
}
```

```
# hexedit chouette.perso
00000000  0F 00 00 00 43 68 6F 75 65 74 74 65 ....Chouette
0000000C  20 4F 69 73 65 61 75 7F 00 00 00 2A Oiseau....*
00000018  00 00 00 57 00 00 00 5E 00 00 00 ...W...^...
00000024
--- chouette.perso --0x0/0x23-----
```

Exercice noté 45 (★★★ Enregistrement et recherche de numéros avec sauvegarde).

Écrire un programme qui :

- prend une option `-i [FICHIER]` pour ouvrir une liste de noms / numéros depuis un fichier existant.
- prend une option `-o [FICHIER]` pour enregistrer une liste de noms / numéros depuis la liste actuellement connue par le programme.
- Ouvre si elle existe une liste de noms indiquée.
- Propose à l'utilisateur l'ajout de nouvelles associations noms / numéros.
- Enregistre les associations dans une liste de noms indiquée.
- Permet la recherche d'un numéro depuis le nom associé.

```
# ./prog -o liste.txt
Nom (None pour arrêter) : Premier 1
Numéro : Nom (None pour arrêter) : Second 2
Numéro : Nom (None pour arrêter) : None
Nom à rechercher (None pour arrêter) :
>>> None
# ./prog -i liste.txt -o liste.txt
Nom (None pour arrêter) : Troisieme 3
Numéro : Nom (None pour arrêter) : None
Nom à rechercher (None pour arrêter) :
>>> Premier
Le numéro de "Premier" est 1
Nom à rechercher (None pour arrêter) :
>>> Second
Le numéro de "Second" est 2
Nom à rechercher (None pour arrêter) :
>>> Troisieme
Le numéro de "Troisieme" est 3
Nom à rechercher (None pour arrêter) :
>>> None
```



---

## 12 Structures

---

Dans notre code, nous embarquons souvent plusieurs informations qui correspondent en réalité à une même entité et qu'il faut passer à nos fonctions. Une alternative peu propre et qui peut vite poser problème lorsque nous souhaitons réutiliser une fonction est d'avoir ces données en variables globales. Nous nous intéresserons ici à organiser nos informations dans notre code pour plus de maintenabilité.

### 12.1 Typedef

Une manière de créer un synonyme d'un type est l'opérateur `typedef`. Il peut être pratique lorsqu'on fera appel régulièrement à un type long ou compliqué à écrire. Il est par exemple possible d'abrégier l'appel de `unsigned int` en `uint` :

```
typedef unsigned int uint;

int main() {

    uint entierPositif = 4000000000;
    printf("%u\n", entierPositif);

    exit(EXIT_SUCCESS);
}
```

Il est possible de complexifier le type sur lequel on veut faire un alias, comme un tableau ou un pointeur :

```
typedef int intListeStatique[];
/* intListeStatique sera équivalent au type d'un tableau de int
   ↪ */
typedef int * intListe;
/* intListe sera équivalent au type d'un pointeur sur un int */
```

```
/* équivalent à avoir "int * liste" en argument */
void afficherIntListe(intListe liste) {
    int i;
    for(i = 0; liste[i] >= 0; ++i) {
        if(i) printf(", ");
        printf("%d", liste[i]);
    }
    printf("\n");
}

int main() {
    /* équivalent à "int liste[] = {1, 2, 3, 4, -1};" */
    intListeStatique liste = {1, 2, 3, 4, -1};
    afficherIntListe(liste);

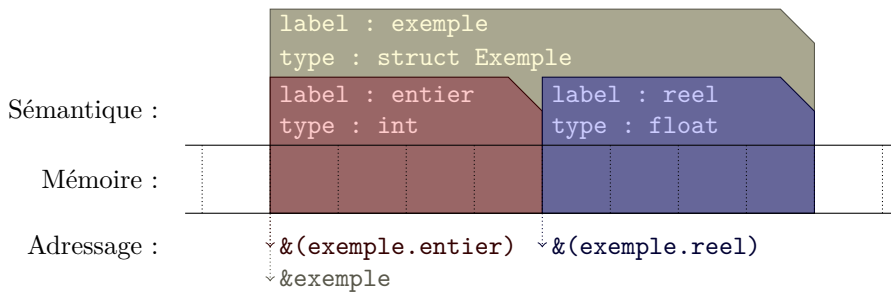
    exit(EXIT_SUCCESS);
}
```

## 12.2 Structures

En langage C, une structure correspondra à une entité qui permettra d'en regrouper plusieurs. Par exemple, lorsque vous avez une liste, il faudra potentiellement avoir à disposition la liste, sa taille, une capacité maximale et autres. Il est possible de modéliser cette liste par un élément qu'est la structure et ne passer que cet élément aux fonctions qui l'auraient en paramètres.

```
struct Exemple {
    int entier;
    float reel;
};

struct Exemple exemple;
```



## 12.3 Définition

Une structure se fabrique depuis le mot clé **struct**. On indique ensuite lors de sa définition les champs qui la composent. L'accès aux éléments de la structure se fera ensuite en séparant la variable fabriquée depuis la structure et le champs souhaité par un point ..

```
/* Schéma de construction d'une Liste */
struct Liste {
    int * elements; /* valeurs de la liste */
    int taille; /* taille de la liste */
};

int main() {
    /* Construction d'une liste */
    struct Liste liste;
    /* accès aux champs de la liste */
    liste.elements;
    liste.taille;

    exit(EXIT_SUCCESS);
}
```

Comme vu précédemment, il est possible de fabriquer un alias du même nom pour s'éviter l'écriture de **struct** à la construction d'une liste juste avant sa définition.

```
/* Alias */
typedef struct Liste Liste;
/* Schéma de construction d'une Liste */
struct Liste {
    int * elements; /* valeurs de la liste */
    int taille; /* taille de la liste */
};

int main() {
    /* Construction d'une liste */
    Liste liste;
    /* accès aux champs de la liste */
    liste.elements;
    liste.taille;

    exit(EXIT_SUCCESS);
}
```

## 12.4 Avec des pointeurs

À noter que souvent en langage C, les structures seront souvent gérées par pointeurs. L'accès aux champs d'un pointeur sur une structure peut se faire principalement de deux manière équivalentes :

```
Liste * liste;
/* déréférencement puis accès */
(*liste).elements;
/* version raccourcie */
liste->elements;
```

Il sera souvent plus pratique d'avoir des fonctions qui permettent l'allocation et la libération d'une structure. Ceci en particulier si elle est allouée dynamiquement ou que ses champs peuvent l'être :



```
Liste * Liste_alloc(int taille) {
    Liste * res = NULL;
    if((res = (Liste *)malloc(sizeof(Liste))) == NULL) {
        fprintf(stderr, "Liste_alloc : Erreur alloc liste\n");
        return NULL;
    }
    res->taille = taille;
    if((res->elements = (int *)calloc(taille, sizeof(int))) == NULL)
        ↪ {
        free(res);
        fprintf(stderr, "Liste_alloc : Erreur alloc éléments\n");
        return NULL;
    }
    return res;
}

void Liste_free(Liste ** liste) {
    free((*liste)->elements);
    free(*liste);
    *liste = NULL;
}

int main() {
    /* Construction d'un pointeur de liste */
    Liste * liste = Liste_alloc(4);
    /* accès aux champs de la liste référencée */
    liste->elements;
    liste->taille;

    Liste_free(&liste);

    exit(EXIT_SUCCESS);
}
```

Il est aussi possible de masquer l'aspect pointeur à l'utilisateur en le précisant

dans le typedef. À noter que si vous souhaitez aller plus loin et respecter les principes d'encapsulation que l'on peut voir dans d'autres langages, vous pouvez déclarer l'existence de la structure à l'avance. Notez que vous ne pourrez l'utiliser dans le code qu'après sa définition ou présenter un pointeur avant (nécessité pour le compilateur de connaître la taille de la structure lorsqu'il opère réellement dessus là où la taille d'une adresse est fixe).

```
/* Alias version pointeur */
typedef struct Liste * Liste;
/* Déclaration en amont */
struct Liste;

Liste Liste_alloc(int taille);

void Liste_free(Liste * liste);

int main() {

    /* Construction d'un pointeur de liste */
    Liste liste = Liste_alloc(4);
    /* La liste ne peut plus être manipulée directement */
    /* Il faut maintenant passer par des fonctions */
    Liste_free(&liste);

    exit(EXIT_SUCCESS);
}

/* Votre partie du programme : */

/* pourra être cachée à l'utilisateur de votre code */
/* Schéma de construction d'une Liste */
struct Liste {
    int * elements; /* valeurs de la liste */
    int taille; /* taille de la liste */
};
```

```

Liste Liste_alloc(int taille) {
    Liste res = NULL;
    if((res = (Liste)malloc(sizeof(struct Liste))) == NULL) {
        fprintf(stderr, "Liste_alloc : Erreur alloc liste\n");
        return NULL;
    }
    res->taille = taille;
    if((res->elements = (int *)calloc(taille, sizeof(int))) == NULL)
        ↪ {
        free(res);
        fprintf(stderr, "Liste_alloc : Erreur alloc éléments\n");
        return NULL;
    }
    return res;
}

void Liste_free(Liste * liste) {
    free((*liste)->elements);
    free(*liste);
    *liste = NULL;
}

```

### 12.4.1 Champs de bits

Une optimisation mémoire existe pour les structures. En effet, il est possible de préciser le nombre de bits sur lequel un champ doit être stocké. Ceci se fait en précisant après le champs : puis le nombre de bits nécessaires au maximum pour le champ. Ceci peut être intéressant si la structure contient beaucoup de booléens par exemple et qu'elle est gardée un nombre important de fois en mémoire.

```

struct Personnage {
    unsigned int pointsDeVie;
    unsigned int pointsDeVieMax;
    unsigned int niveau;
    unsigned long experience;
}

```

```
    unsigned char aStatutPoison;
    unsigned char aStatutParalyse;
    unsigned char aStatutEndormi;
    unsigned int  attaque;
    unsigned int  defense;
    unsigned int  attaqueSpe;
    unsigned int  defenseSpe;
    unsigned int  vitesse;
};

struct PersonnageCompress {
    unsigned int  pointsDeVie : 10;
    unsigned int  pointsDeVieMax : 10;
    unsigned int  niveau : 7;
    unsigned long experience : 40;
    unsigned char aStatutPoison : 1;
    unsigned char aStatutParalyse : 1;
    unsigned char aStatutEndormi : 1;
    unsigned int  attaque : 10;
    unsigned int  defense : 10;
    unsigned int  attaqueSpe : 10;
    unsigned int  defenseSpe : 10;
    unsigned int  vitesse : 10;
};

int main() {
    printf("%lu\n", sizeof(struct Personnage));
    printf("%lu\n", sizeof(struct PersonnageCompress));
    exit(EXIT_SUCCESS);
}
```

48

24

## 12.5 Unions

Dans une construction similaire à celle des structures, il existe les `union`. À noter qu'une `union` liste des champs qui sont des alias vers une même donnée. C'est-à-dire que si une `union` regroupe un `float` et un `int`, ces deux champs pointent vers le même emplacement mémoire, mais c'est l'accès au champ qui permet de déterminer comment retranscrire l'information en le type demandé.

```
union Scalaire {
    int entier;
    float flottant;
    double flottantPrecis;
};

int main() {
    union Scalaire nombre;
    nombre.entier = 42;
    printf("sizeof(Scalaire) : %lu\n", sizeof(union Scalaire));
    printf("entier : %d\n", nombre.entier);
    printf("flottant : %g\n", nombre.flottant);
    printf("flottantPrecis : %g\n", nombre.flottantPrecis);
    nombre.flottant = 42;
    printf("entier : %d\n", nombre.entier);
    printf("flottant : %g\n", nombre.flottant);
    printf("flottantPrecis : %g\n", nombre.flottantPrecis);
    nombre.flottantPrecis = 42;
    printf("entier : %d\n", nombre.entier);
    printf("flottant : %g\n", nombre.flottant);
    printf("flottantPrecis : %g\n", nombre.flottantPrecis);
    exit(EXIT_SUCCESS);
}
```

Il est possible de créer des structures à l'intérieur de l'union pour que l'information codée dans l'union correspondent à plusieurs champs :

```
union Scalaire {
    int entier;
```

```
float flottant;
double flottantPrecis;
struct {
    float x;
    float y;
};

int main() {
    union Scalaire nombre;
    nombre.x = 42;
    nombre.y = 1337;
    printf("sizeof(Scalaire) : %lu\n", sizeof(union Scalaire));
    printf("entier : %d\n", nombre.entier);
    printf("flottant : %g\n", nombre.flottant);
    printf("flottantPrecis : %g\n", nombre.flottantPrecis);
    printf("(x, y) : (%g, %g)\n", nombre.x, nombre.y);
    exit(EXIT_SUCCESS);
}
```

## 12.6 Énumérations

Souvent pour ajouter de la sémantique et de la lisibilité au code, il peut être utile de nommer des constantes. De même, on peut souhaiter considérer qu'une variable ne devrait prendre que certaines valeurs constantes prédéfinies. Il est possible de construire un type dont les valeurs attendues seront celles de noms prédéfinis à l'aide du mot clé `enum` :

```
enum MapItem {
    MAP_VIDE,
    MAP_JOUEUR,
    MAP_ADVERSAIRE,
    MAP_MUR,
    MAP_SORTIE
};
```

```
enum MapItem item;
```

Il est possible de se passer du mot clé `enum` lors de la déclaration d'une variable de ce type énuméré à l'aide d'un `typedef` :

```
typedef enum {  
    MAP_VIDE,  
    MAP_JOUEUR,  
    MAP_ADVERSAIRE,  
    MAP_MUR,  
    MAP_SORTIE  
} MapItem;  
  
MapItem item;
```

Ces types énumérés conviennent parfaitement à une utilisation dans un `switch`. À noter qu'il est recommandé d'ajouter l'option `-Wall` lors de la compilation. Si un élément de type énuméré n'est pas géré par le `switch` ceci sera précisé à la compilation.

```
typedef enum {  
    ITEM_MOB_DYNAMIC,  
    ITEM_MOB_STATIC,  
    ITEM_MOB_UNKNOWN  
} ItemMobility;  
  
ItemMobility getMapItemMobility(MapItem item) {  
    switch(item) {  
        case MAP_JOUEUR :  
        case MAP_ADVERSAIRE :  
            return ITEM_MOB_DYNAMIC;  
  
        case MAP_MUR :  
        case MAP_SORTIE :  
            return ITEM_MOB_STATIC;
```

```
    default :  
        return ITEM_MOB_UNKNOWN;  
    }  
}
```

À noter qu'en réalité un type énuméré est un entier dont certaines valeurs sont associées à un nom pour s'accorder avec l'utilisation qui doit en être faite. Il est possible dans l'énumération d'encoder certaines valeurs. Les valeurs successives suivront de un en un.

```
typedef enum {  
    MAP_VIDE = 0,  
    MAP_JOUEUR = 10,  
    MAP_ADVERSAIRE,  
    MAP_MUR = 20,  
    MAP_SORTIE  
} MapItem;
```

Puisque un type énuméré correspond à un entier et qu'un caractère aussi, il est possible de faire correspondre les valeurs du type énuméré à des caractères. Ceci peut apporter de la lisibilité dans le code et aussi avoir une conversion directe si cela devait être éditable dans un fichier par exemple.

```
typedef enum {  
    MAP_VIDE = ' ',  
    MAP_JOUEUR = '@',  
    MAP_MUR = '#',  
    MAP_ADVERSAIRE = 'x',  
    MAP_SORTIE = 'x'  
} MapItem;
```



## 12.7 Résumé

L'opérateur `typedef` permet de créer un synonyme d'un type :

```
typedef type synonyme;
```

Il est possible de créer des type structurés regroupant des champs sous un même nom :

```
struct Nom {  
    typeChamp nomChamp;  
};  
struct Nom variable;  
struct Nom * pointeur = &variable;  
variable.nomChamp; /* accès au champ */  
pointeur->nomChamp; /* accès au champ via pointeur */
```

De la même manière, on peut regrouper des champs sous un même nom mais que ceux-ci correspondent à un même emplacement mémoire pour réduire la taille de ce groupement en mémoire.

```
union Nom {  
    int entier;  
    float reel;  
};  
union Nom nombre;  
nombre.entier = 42; /* nombre.reel est aussi modifié */  
nombre.reel = 13.37; /* nombre.entier est aussi modifié */
```

Lorsqu'un entier prend des valeurs que l'on veut nommer, on peut construire un type énuméré :

```
typedef enum {  
    VALEUR1 = 0x1,  
    VALEUR2 = 0x2  
} Liste;  
Liste liste = VALEUR1;
```

## 12.8 Entraînement

Exercice noté 46 (★★ Définir une structure `Vecteur2d`).

Définir une structure `Vecteur2d` avec les champs suivants :

- `double x`.
- `double y`.

Puis définir des fonctions qui permettent la manipulation d'un élément  $V = (V_x, V_y)$  :

- **Translation** de vecteur  $T = (T_x, T_y)$  :

$$V = V + T$$

- **Agrandissement** de rapport  $\alpha$  et de centre  $C = (C_x, C_y)$  :

$$V = \alpha(V - C) + C$$

- **Rotation** d'angle  $\delta$  et de centre  $C = (C_x, C_y)$  :

$$V = \begin{pmatrix} \cos(\delta) & -\sin(\delta) \\ \sin(\delta) & \cos(\delta) \end{pmatrix} (V - C) + C$$

```
Vecteur2d : (0, 0)
Translation par un Vecteur2d : (1, 2)
(1, 2)
Agrandissement de rapport 0.5 et de centre un Vecteur2d : (1, 0)
(1, 1)
Rotation d'angle 135 deg et de centre un Vecteur2d : (0, 2)
(1.11022e-16, 3.41421)
```

Exercice noté 47 (★★ Structure pour gérer une grille).

Compléter le code suivant pour qu'il affiche une grille à l'écran :

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

typedef struct Grille Grille;
struct Grille {
    char * grille;
    int largeur;
    int hauteur;
};

/* donne un pointeur sur une case de la grille */
char * Grille_case(const Grille * grille, int x, int y);

/* crée une grille de taille donnée */
Grille Grille_creer(int largeur, int hauteur);

/* affiche une grille à l'écran */
void Grille_afficher(const Grille * grille);

/* libère une grille */
void Grille_free(Grille * grille);

int main() {
    int largeur = 60, hauteur = 20;
    Grille grille = Grille_creer(largeur, hauteur);
    int x = 1, y = 1;
    initscr();
    noecho();
    cbreak();
    do {
        clear();
        Grille_afficher(&grille);
        mvprintw(y, x, "@");
        mvprintw(y, x, "");
        refresh();
        getch();
    } while (1);
    /* gestion des événements */
}
```

```
} while(1);  
refresh();  
clrtoeol();  
refresh();  
endwin();  
Grille_free(&grille);  
exit(EXIT_SUCCESS);  
}
```

```
#####  
#@                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#                                           #  
#####
```



**Exercice noté 49 (★★★ Implémenter une liste chaînée).**

Un informaticien vous voit utiliser uniquement des tableaux pour vos liste. Il vous informe qu'il préfère utiliser des liste chaînées dans certains cas. Il vous propose d'implémenter les fonctionnalités d'une liste chaînée et d'une liste depuis un tableau pour vous en faire une idée vous-même. Vous préciserez pour chaque fonction sa complexité algorithmique en fonction de la taille de la liste ( $\mathcal{O}(1)$  pour constante lorsque ça ne dépend pas de la taille de la liste et  $\mathcal{O}(n)$  pour linéaire lorsqu'on pourrait au pire des cas passer sur chaque élément de la liste).

```
typedef struct LinkedList * LinkedList;
struct LinkedList {
    int value;
    LinkedList next;
};

/* Renvoie une liste vide */
LinkedList LL_empty();

/* Libère la liste */
void LL_free(LinkedList * liste);

/* Ajoute un élément en fin */
int LL_add_tail(LinkedList * liste, int value);

/* Ajoute un élément en tête */
int LL_add_head(LinkedList * liste, int value);

/* Ajoute un élément à une position donnée */
int LL_insert(LinkedList * liste, int id, int value);

/* Supprime l'élément en fin */
int LL_pop_tail(LinkedList * liste, int * value);

/* Supprime l'élément en tête */
int LL_pop_head(LinkedList * liste, int * value);

/* Supprime l'élément à une position donnée */
int LL_delete(LinkedList * liste, int id, int * value);

/* Affiche la liste */
void LL_print(FILE * flow, const LinkedList * liste);
```

```
typedef struct ArrayList ArrayList;
struct ArrayList {
    int * values;
    int size;
    int capacite;
};

/* Renvoie une liste vide */
ArrayList AL_empty();

/* Libère la liste */
void AL_free(ArrayList * liste);

/* Ajoute un élément en fin */
int AL_add_tail(ArrayList * liste, int value);

/* Ajoute un élément en tête */
int AL_add_head(ArrayList * liste, int value);

/* Ajoute un élément à une position donnée */
int AL_insert(ArrayList * liste, int id, int value);

/* Supprime l'élément en fin */
int AL_pop_tail(ArrayList * liste, int * value);

/* Supprime l'élément en tête */
int AL_pop_head(ArrayList * liste, int * value);

/* Supprime l'élément à une position donnée */
int AL_delete(ArrayList * liste, int id, int * value);

/* Affiche la liste */
void AL_print(FILE * flow, const ArrayList * liste);
```





---

# 13 Programmation modulaire

---

## 13.1 Retour sur la compilation

### 13.1.1 Externaliser une fonction

Jusqu'ici, nous avons essentiellement travaillé sur des travaux pratiques pour lesquels écrire le code dans un seul fichier semblait faire l'affaire. Cependant pour un projet ou quand le code commence à devenir plus conséquent et plus complexe, se limiter à un fichier peut vite devenir douloureux.

Prenons l'exemple du code suivant :

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

void maFonction() {
    printf("Ma Fonction\n");
}

int main() {
    maFonction();
    exit(EXIT_SUCCESS);
}
```

Ce code définit une fonction puis l'appelle. Imaginons que nous ne voudrions pas définir cette fonction dans ce fichier, mais dans un autre. Ceci est possible :

- Nous avons besoin de la déclaration de la fonction pour l'appeler dans notre code.
- Sa définition peut être sauvegardée dans un autre fichier.

**main.c**

```
#include <stdlib.h>

extern void maFonction();

int main() {
    maFonction();
    exit(EXIT_SUCCESS);
}
```

**mes\_fonctions.c**

```
#include <stdio.h>

void maFonction() {
    printf("Ma Fonction\n");
}
```

Pour compiler une telle configuration, il faut que les deux fichiers soient compilés et liés vers le même exécutable. Ceci se fait en ajoutant `mes_fonctions.c` dans la ligne de compilation :

```
gcc -o executable main.c mes_fonctions.c
```

`maFonction` a maintenant été externalisée dans un autre fichier. Le mot-clé `extern` est facultatif pour une fonction. Cependant, il est nécessaire si l'on souhaite utiliser une variable instanciée dans un autre fichier source :

main.c

```
#include <stdlib.h>

extern int variable;

extern void maFonction(int);

int main() {
    maFonction(variable);
    exit(EXIT_SUCCESS);
}
```

mes\_fonctions.c

```
#include <stdio.h>

int variable = 42;

void maFonction(int v) {
    printf("Ma Fonction %d\n", v);
}
```

### 13.1.2 Compilation séparée

Revenons sur la compilation. Rappelez vous, nous avons parlé de compilateur et d'éditeur des liens :

- **La compilation** c'est la transformation du code source en langage C en langage utilisable par la machine. Ceci produira des fichiers "Objets" avec pour extension `.o`.
- **L'édition des liens** c'est l'assemblage de ces fichiers `.o` en une application (exécutable) et le raccordement des définitions de chaque élément à sa déclaration et son appel.

S'il reste une déclaration non résolue (absence du fichier contenant la définition d'une fonction par exemple), le compilateur vous l'indiquera à l'édition des liens :

```
# gcc -o executable main.c
/tmp/cc5vxQgu.o : Dans la fonction « main » :
main.c:(.text+0xa) : référence indéfinie vers « maFonction »
collect2: error: ld returned 1 exit status
```

Nous avons vu ici une version simplifiée de la commande de compilation : tout est compilé puis lié. Cependant ceci veut dire que pour la modification d'un fichier, l'ensemble est recompilé. Il est donc possible de compiler les fichiers uns à uns avec l'option `-c`. Celle-ci fabrique un fichier `.o` associé à chaque fichier `.c` fourni. Ensuite on procède à l'édition des liens en fournissant les fichiers `.o` et bibliothèques utilisées.

```
# gcc -c mes_fonctions.c
# gcc -c main.c
# gcc -o executable main.o mes_fonctions.o
```

Si vous souhaitez visualiser la table des symboles d'un fichier `.o`, vous pouvez utiliser la commande `readelf` :

```
# readelf -s main.o

La table de symboles « .symtab » contient 13 entrées :

```

Num:	Valeur	Tail	Type	Lien	Vis	Ndx	Nom
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	main.c
...							
8:	0000000000000000	27	FUNC	GLOBAL	DEFAULT	1	main
9:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	
	↪ variable						
10:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	
	↪ _GLOBAL_OFFSET_TABLE_						
11:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	
	↪ maFonction						
12:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	exit

Nous utilisons aussi `printf`, une fonction définie ailleurs, sans l'avoir déclarée, non ? Vos fichiers ne sont en réalité pas si vides et vous l'avez fait avec votre

`#include <stdio.h>`. C'est une directive préprocesseur, voyons comment ceci fonctionne.

## 13.2 Directives préprocesseur

Un programme qui affiche "Hello ESGI!", ça tient en 10 lignes, regardez :

```
#include <stdio.h>

int main() {
    printf("Hello ESGI !\n");
    return 0;
}
```

Essayez maintenant la commande suivante :

```
gcc -o main.i -E -P main.c
```

Ceci ressemble plutôt à ça :

```
typedef long unsigned int size_t;
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
... /* environ 200 lignes */

int main() {
    printf("Hello ESGI !\n");
    return 0;
}
```

Cette commande permet de résoudre les directives préprocesseurs. En effet, avant d'être compilé, un code source en langage C est pré-traité pour qu'il ne reste réellement que des instructions en langage C. Ces directives préprocesseur permettent de gagner en lisibilité ou maintenabilité lorsque nous avons besoin que des informations soient inscrites en dur dans du code C. Dans notre cas, ceci nous permet de déclarer toutes les fonctionnalités que nous utiliserons de la bibliothèque `stdio.h` depuis son entête de déclaration. Notez que simple déclarer `printf` fonctionne sans erreur ni warning :

```
extern int printf(const char *, ...);

int main() {
    printf("Hello ESGI !\n");
    return 0;
}
```

### 13.2.1 include

La directive préprocesseur `include` permet de recopier le contenu d'un fichier à l'emplacement du fichier C courant où elle est appelée. C'est un outil puissant, mais à utiliser avec parcimonie. En général, nous les utiliserons pour inclure des fichiers d'entête dans notre code :

- `#include <entete.h>` : permet l'inclusion d'une bibliothèque présente ou installée sur la machine.
- `#include "entete.h"` : est réservé à l'inclusion relative de vos propres fichiers d'entête.

Ceci permet par exemple de définir sa propre bibliothèque de de fonctionnalités :

main.c

```
#include <stdlib.h>
#include "mes_fonctions.h"

int main() {
    maFonction(variable);
    exit(EXIT_SUCCESS);
}
```

**mes\_fonctions.h**

```
extern int variable;

extern void maFonction(int);
```

**mes\_fonctions.c**

```
#include <stdio.h>

int variable = 42;

void maFonction(int v) {
    printf("Ma Fonction %d\n", v);
}
```

Nous verrons dans la suite comment protéger cette bibliothèques contre des problèmes de doublons dans l'inclusion et en faire ainsi un module.

### 13.2.2 define

Nous avons déjà croisé une utilisation de la directive préprocesseur **define** pour la création de constantes. Cette directive fera que pour un nom donné, celui-ci sera remplacé par le code associé.

#### Expressions constantes

Ceci permet par exemple de déclarer des expression constantes :

```
#include <stdio.h>
#include <stdlib.h>
#define TAILLE 10

int main() {
    int i;
    for(i = 0; i < TAILLE; ++i) {
```

```
    printf("%d\n", i);
}
exit(EXIT_SUCCESS);
}
```

En général, on les introduit plutôt en début de fichier, mais ces directives peuvent être placées où souhaité dans le code. Leur définition peut être annulée avec la directive préprocesseur `undef` et elles peuvent être redéfinies :

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

int main() {
#define NOMBRE 42
    printf("define NOMBRE %d\n", NOMBRE);
#undef NOMBRE
#define NOMBRE 13.37
    printf("define NOMBRE %g\n", NOMBRE);
#undef NOMBRE
#define NOMBRE "1234"
    printf("define NOMBRE %s ?\n", NOMBRE);
    exit(EXIT_SUCCESS);
}
```

La directive `define` peut être utilisée pour remplacer un nom de fonction ou même du code :

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

#define print puts

int main() {
    print("Hello ESGI");
    /* Python de langage C ! */
    exit(EXIT_SUCCESS);
}
```



```
}
```

Pour revenir à la ligne dans le contenu associé à un `#define`, il faut utiliser un anti-slash :

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

#define INSTRUCTIONS printf("Hello ESGI\n"); \
                    exit(EXIT_SUCCESS);

int main() {
    INSTRUCTIONS
}
```

## Macros

Il est possible de définir des "Macros" à l'aide du `#define`. Ces macros vont écrire en dur du code en fonction de paramètres données. Ceci peut être utilisé par exemple pour remplacer des fonctions courtes garder une généricité sur les arguments fournis.

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

#define min(a, b) (a < b) ? a : b

int main() {
    int a = 42, b = 1337;
    printf("min(%d, %d) = %d\n", a, b, min(a, b));
    float c = 13.37, d = 42.;
    printf("min(%g, %g) = %g\n", c, d, min(c, d));
    exit(EXIT_SUCCESS);
}
```

Cependant, ceci reste à utiliser avec parcimonie. En effet, la macro va réécrire à l'identique la partie de code fournie en argument, là où une fonction travaillerait

avec la valeur de retour fournie. Il est donc conseillé de mettre les arguments entre parenthèses dans le code de la macro et d'être rigoureux avec les arguments dont la duplication de l'expression pourrait être problématique :

```
int fonction_min(int a, int b) {
    return (a < b) ? a : b;
}

#define macro_min(a, b) ((a) < (b)) ? (a) : (b)

int main() {
    int v, a, b;
    v = fonction_min(a = getchar() - '0', b = getchar() - '0');
    printf("fonction_min(%d, %d) = %d\n", a, b, v);
    v = macro_min(a = getchar() - '0', b = getchar() - '0');
    printf("macro_min(%d, %d) = %d\n", a, b, v);
    exit(EXIT_SUCCESS);
}
```

Ce code produit en réalité le code suivant :

```
int fonction_min(int a, int b) {
    return (a < b) ? a : b;
}

int main() {
    int v, a, b;
    v = fonction_min(a = getchar() - '0', b = getchar() - '0');
    printf("fonction_min(%d, %d) = %d\n", a, b, v);
    v = ((a = getchar() - '0') < (b = getchar() - '0')) ? (a =
    ↪  getchar() - '0') : (b = getchar() - '0');
    printf("macro_min(%d, %d) = %d\n", a, b, v);
    exit(0);
}
```

D'où une exécution potentiellement erronée :

```
24246
fonction_min(4, 2) = 2
macro_min(6, 4) = 6
```

Il est possible de transformer une expression fournie à une macro en une chaîne de caractère en précédant le nom de l'expression par un dièse # dans le code associé à la macro :

```
#define FAIRE_CALCUL(exp) printf("%s = %d\n", #exp, exp)

int main() {
    int valeur = 4;
    FAIRE_CALCUL(valeur * valeur + 2 * valeur + 1);
    exit(EXIT_SUCCESS);
}
```

Ce qui produit l'exécution suivante :

```
valeur * valeur + 2 * valeur + 1 = 25
```

Pour concaténer l'expression fournie à une macro avec du code : par exemple dans le nom d'une fonction. Il est possible d'appeler d'appeler l'opérateur ## dans le code associé à la macro :

```
#define macro_abs(a) ((a) < 0 ? -(a) : (a))
#define macro_carre(a) ((a) * (a))

#define call(f, x) printf("%s(%d) = %d\n", #f, x, macro_##f(x))

int main() {
    call(abs, -5);
    call(carre, -5);
    exit(EXIT_SUCCESS);
}
```

Ce qui produit l'exécution suivante :

```
abs(-5) = 5
carre(-5) = 25
```

### 13.2.3 conditionnement

Il est possible de conditionner du code sous condition. Par exemple, il peut être utile d’avoir un mode verbeux pour travailler sur votre code et un mode moins verbeux pour le fournir à un utilisateur.

Par exemple dans le code suivant, on peut adapter le code en fonction du fait que certaines `#define` existent ou non :

```
#define DEBUG

int main() {
    #if defined VERBOSE
        fprintf(stderr, "Entrée dans le main\n");
    #endif
    #if defined VERBOSE
        printf("42, la réponse à la vie !\n");
    #elif defined DEBUG
        printf("42, vous savez pourquoi.\n");
    #else
        printf("42 !\n");
    #endif
    #if defined VERBOSE
        fprintf(stderr, "Sortie du main\n");
    #endif
    exit(EXIT_SUCCESS);
}
```

Ceci produit le code suivant :

```
int main() {
    printf("42, vous savez pourquoi.\n");
}
```

```
    exit(0);  
}
```

Il est possible de réaliser un `#define` depuis la commande de compilation avec l'option `-D` :

```
gcc -o main.i -E -P main.c -DVERBOSE
```

Ce qui produit le code suivant :

```
int main() {  
    fprintf(stderr, "Entrée dans le main\n");  
    printf("42, la réponse à la vie !\n");  
    fprintf(stderr, "Sortie du main\n");  
    exit(0);  
}
```

Il est possible de raccourcir l'écriture des conditions vérifiant la définition d'un `#define` :

```
#if defined NOM  
#ifdef NOM  
  
#if !defined NOM  
#ifndef NOM
```

### 13.2.4 Module

Nous avons vu précédemment qu'il peut être intéressant de répartir ses fonctionnalités dans d'autres fichiers. Plus précisément nous découperons notre code en **modules**. Un module est un ensemble de fonctionnalités documentées pour lequel on fournit un fichier d'entête `.h` à l'utilisateur (un autre programmeur ou vous-même) et pour lequel vous avez implémenté les fonctionnalités associées dans un fichier `.c` du même nom.

Exemple d'organisation d'un module :

module.h

```
#ifndef DEF_HEADER_MODULE
#define DEF_HEADER_MODULE
/* Protection du module */

/**
 * Documentation du module et auteurs
 */

/* Macros publiques */
#define abs(x) ((x) < 0) ? -(x) : (x)

/* Types publiques du module */
typedef struct Point Point;
struct Point {
    float x;
    float y;
};

/* Variables publiques du module */
extern Point origine;

/* Fonctionnalités publiques du module */

/* Affiche un point dans la sortie standard */
extern void Point_afficher(const Point * point);

#endif
```

module.c

```
#include "module.h"
/* Inclusion des déclarations du module */

#include <stdio.h>
/* Autres inclusions */

/* Définition des variables globales relatives au module
↳ */
Point origine = {0, 0};

/* Fonctionnalité privée au module par le mot-clé static
↳ */
static void Point_print(FILE * flow, const Point * point)
↳ {
    if(! point) {
        fprintf(flow, "(nil)");
    }
    fprintf(flow, "(%g, %g)", point->x, point->y);
}

/* Définition des fonctionnalités annoncées par l'entête
↳ */
void Point_afficher(const Point * point) {
    Point_print(stdout, point);
}
```

## 13.3 Makefiles

### 13.3.1 Concept

Avec l'augmentation du nom de modules, l'ajout de bibliothèques et autres la compilation de votre projet va se complexifier. Pour ceci, il peut être intéressant d'automatiser sa compilation. À noter qu'un avantage offert par la programmation modulaire est que l'on peut travailler sur chaque module indépendamment des interdépendances entre les modules. Et donc avoir une compilation potentiellement plus rapide que recompiler l'intégralité du projet.

Le moyen proposé pour compiler un projet en langage C est un **Makefile**. Un Makefile est un fichier qui donne le schéma à suivre pour compiler le code. Celui-ci demande d'être créé à l'emplacement d'où vous souhaitez que l'utilisateur compile votre projet. Il n'aura ensuite besoin que d'utiliser la commande **make**.

#### Makefile

```
executable :  
    gcc -o executable *.c
```

```
make
```

### 13.3.2 Possibilités

Un Makefile s'organise comme une liste de cibles, dépendances associées et commandes à exécuter. Chaque commande est précédée d'une **tabulation**.

```
cible: dépendances  
    commande  
    ...  
    commande
```

Les dépendances vont permettre de relancer les commandes pour construire la cible si elles ont été modifiées.

La proposition de compilation est la suivante :



- Compilation séparée des fichiers `.c` en fichiers `.o` pour chaque module.
- Fichier source `.c` du module et fichiers `.h` inclus dans le module en dépendances.
- Linkage de tous les fichiers `.o` pour création de l'exécutable.
- Une commande pour nettoyer le répertoire de la compilation.

Ceci pourrait donner le Makefile suivant :

```
executable : main.o module.o
    gcc -o executable main.o module.o

main.o : main.c module.h
    gcc -c main.c

module.o : module.c module.h
    gcc -c module.c

clean :
    rm -rf *.o
```

La commande de nettoyage du dossier peut s'appeler de la manière suivante :

```
make clean
```

Cependant ce Makefile peut être assez redondant, il est possible d'introduire les symboles suivants :

Symbole	Correspondance
<code>\$@</code>	Cible
<code>\$^</code>	Toutes les dépendances
<code>\$&lt;</code>	Première dépendance

Ceci permet la transformation du Makefile en le suivant :

```
executable : main.o module.o
    gcc -o $@ $^
```

```
main.o : main.c module.h
        gcc -c $<

module.o : module.c module.h
        gcc -c $<
```

Pour aller plus loin, il est aussi possible d'utiliser une règle générique en utilisant un %. Celui-ci sera automatiquement remplacé de manière à construire les cibles manquantes correspondant au schéma donné. Cependant, il sera nécessaire d'ajouter les dépendances pour qu'elles soient prises en compte :

```
executable : main.o module.o
        gcc -o $@ $^

main.o : main.c module.h
module.o : module.c module.h

%.o : %.c
        gcc -c $<
```

Il est aussi possible de définir des variables pour votre Makefile. Leur définition prend la syntaxe `VARIABLE=VALEUR` et leur appel `$(VARIABLE)`. Elles sont utiles pour écrire à un endroit :

- le compilateur : `CC=gcc` (ou par exemple `CC=clang`).
- les drapeaux de compilation et optimisations dans `CFLAGS` (les optimisations peuvent être `-O1`, `-O2` et éventuellement `-O3`).
- les bibliothèques dans `CLIBS`.
- le nom de l'exécutable.

```
CC= gcc
CFLAGS= -O2 -Wall -Wextra -Werror -ansi
CLIBS= -lm
EXE= executable

$(EXE) : main.o module.o
        $(CC) $(CFLAGS) -o $@ $^ $(CLIBS)
```

```
main.o : main.c module.h
module.o : module.c module.h

%.o : %.c
    $(CC) $(CFLAGS) -c $<
```

En langage C, on structurera général un projet en séparant les différents fichiers dans des répertoires :

- `include` contient vos fichiers d'entêtes `.h`.
- `src` contient vos fichiers sources `.c`.
- `obj` permet de sauvegarder les fichiers compilés `.o`.

L'exécutable sera ainsi généré à la racine du projet avec les fichiers pertinents : `README.md`, `Makefile` et autres. Ceci donne le `Makefile` suivant :

```
CC= gcc
CFLAGS= -O2 -Wall -Wextra -Werror -ansi
CLIBS= -lm
EXE= executable
OBJ= obj/
SRC= src/
INCL= include/

$(EXE) : $(OBJ)main.o $(OBJ)module.o
    $(CC) $(CFLAGS) -o $@ $^ $(CLIBS)

$(OBJ)main.o : $(SRC)main.c $(INCL)module.h
$(OBJ)module.o : $(SRC)module.c $(INCL)module.h

$(OBJ)%.o : $(SRC)%.c
    $(CC) $(CFLAGS) -o $@ -c $<

clean :
    rm -rf $(OBJ)*.o
    rm -rf $(EXE)
```

Dans le cas où un temps de compilation potentiellement un peu plus important ne vous serait pas dérangeant, il est possible de gagner du temps dans la construction du Makefile en utilisant la fonction `wildcard`. Ceci permet d'obtenir une liste d'éléments suivant un schéma donné. En l'occurrence, dans notre cas obtenir les fichiers sources et entêtes. Et pour la génération de la liste des fichiers objets la fonction `patsubst` qui fera la substitution des `.c` en `.o` dans la listes des sources obtenue. Ceci fait que le Makefile suivant ne vous demandera pas de rajouter une dépendance à chaque ajout de module :

```
CC= gcc
CFLAGS= -O2 -Wall -Wextra -Werror -ansi
CLIBS= -lm
EXE= executable
OBJ= obj/
SRC= src/
INCL= include/
FILEC:= $(wildcard $(SRC)*.c)
FILEH:= $(wildcard $(INCL)*.h)
FILEO:= $(patsubst $(SRC)%.c,$(OBJ)%.o,$(FILEC))

$(EXE) : $(FILEO)
    $(CC) $(CFLAGS) -o $@ $^ $(CLIBS)

$(OBJ)main.o : $(SRC)main.c $(FILEH)
    $(CC) $(CFLAGS) -o $@ -c $<

$(OBJ)%.o : $(SRC)%.c $(INCL)%.h
    $(CC) $(CFLAGS) -o $@ -c $<

clean :
    rm -rf $(OBJ)*.o
    rm -rf $(EXE)
```

## 13.4 Résumé

Les directives préprocesseur sont des instructions traitées avant la compilation. Celles-ci permettent notamment :

```
#include <lib.h> /* copie l'entête d'une bibliothèque lib */
#include "module.h" /* copie l'entête de mon fichier module.h */
#define NOM EXPRESSION /* remplacera NOM dans le code par
↳ EXPRESSION */
#undef NOM /* supprime la définition de NOM dans le code qui suit
↳ */
```

La directive **define** peut prendre des paramètres dont les expression données lors de l'appel remplacera leur occurrence dans l'expression donnée dans la définition.

```
#define MACRO(exp) exp \ /* remplace exp par le code donné en
↳ argument et '\' permet de continuer la macro à la ligne */
#exp \ /* fabrique une chaîne de caractères de exp */
##exp## \ /* concatène exp avec le code adjacent dans la macro
↳ */
```

Il est possible d'utiliser des structures conditionnelles avec les directives préprocesseur :

```
#if CONDITION1
/* instructions1 */
#elif CONDITION2
/* instructions2 */
#else
/* instructions3 */
#endif
```

Le mot clé **defined** permet d'indiquer si une **define** a nommé ce nom. Celui-ci peut être abrégé dans les structures conditionnelles :

```
#if defined NOM
#ifdef NOM
```

```
#if ! defined NOM  
#ifndef NOM
```

On peut diviser son code en modules : un fichier `.c` (implémentation des fonctionnalités) et `.h` (déclarations à inclure pour l'utiliser dans un autre fichier source) du même nom. Pour compiler un code découpé en modules, on utilise un **Makefile** :

```
cible: dépendances  
      commandes
```

```
$@ # Cible  
$^ # Toutes les dépendances  
$< # Première dépendance  
VARIABLE= valeur # création d'une variable  
$(VARIABLE) # appel d'une variable
```

## 13.5 Entraînement

Exercice noté 50 (★★ Mise en place d'un Makefile).

Découper le code suivant en modules et écrire un Makefile permettant sa compilation modulaire :

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

typedef struct Point Point;
struct Point {
    double x;
    double y;
};

Point Point_creer(double x, double y) {
    Point p = {x, y};
    return p;
}

void Point_afficher(const Point * p) {
    printf("(%g, %g)", p->x, p->y);
}

double Point_distance(const Point * first, const Point * second) {
    return sqrt(pow(second->x - first->x, 2) + pow(second->y -
    ↪ first->y, 2));
}

typedef struct Triangle Triangle;
struct Triangle {
    Point first;
    Point second;
    Point third;
};

Triangle Triangle_creer(Point first, Point second, Point third) {
    Triangle triangle = {first, second, third};
    return triangle;
}
```

```
}

void Triangle_afficher(const Triangle * triangle) {
    printf("{Triangle : ");
    Point_afficher(&(triangle->first));
    printf(", ");
    Point_afficher(&(triangle->second));
    printf(", ");
    Point_afficher(&(triangle->third));
    printf("}");
}

double Triangle_perimetre(const Triangle * triangle) {
    return
        Point_distance(&(triangle->first), &(triangle->second))
        + Point_distance(&(triangle->second), &(triangle->third))
        + Point_distance(&(triangle->third), &(triangle->first));
}

int main() {
    Triangle triangle = Triangle_creer(
        Point_creer(0, 0),
        Point_creer(4, 0),
        Point_creer(0, 3)
    );
    Triangle_afficher(&triangle);
    printf("\nPerimetre : %g\n", Triangle_perimetre(&triangle));
    exit(EXIT_SUCCESS);
}
```



Exercice noté 51 (★★★ Profilage d'un code).

Écrire un fichier d'entête `profile.h` dont les macros permettent un profilage d'un code qui les utiliseraient :

- `DO_PROFILE` active l'aspect verbeux du profilage (inactif sinon).
- `START_FUNCTION` prend le type de retour, le nom et les arguments de la fonction.
- `END_FUNCTION` prend le retour de la fonction.

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

#define DO_PROFILE
#include "profile.h"

START_FUNCTION(int, f, int x)
END_FUNCTION(x * x)

START_FUNCTION(int, main)
    int a = 42;
    a = f(a);
    printf("a = %d\n", a);
    exit(EXIT_SUCCESS);
END_FUNCTION(0)
```

Le code ci-dessus se comporte comme le code ci-dessous :

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

int f(int x) {
    return x * x;
}

int main() {
    int a = 42;
    a = f(a);
    printf("a = %d\n", a);
    exit(EXIT_SUCCESS);
    return 0;
}
```

Si la définition préprocesseur `DO_PROFILE` n'est pas définie, le code donne la sortie suivante :

```
a = 1764
```

Si `DO_PROFILE` est définie, le code se montre plus verbeux et affiche les informations additionnelles suivantes dans la sortie standard d'erreur :

```
# Definition of int function main() in file "main.c" at line 10
< Starting function main :
# Definition of int function f(int x) in file "main.c" at line 7
< Starting function f :
> Ending function f with return x * x
a = 1764
> Exit in function main at line 14 with value EXIT_SUCCESS
```

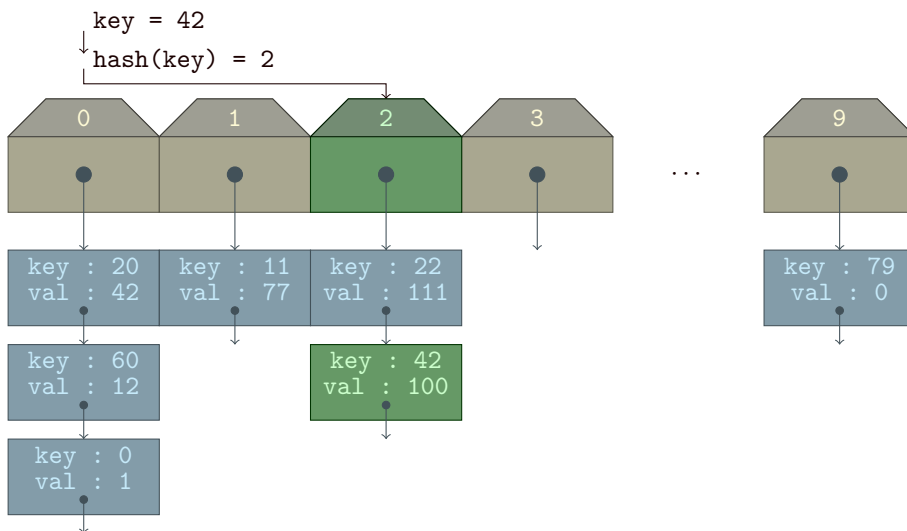
Renseignez vous sur quelques directives préprocesseur standard :

- `__FUNCTION__`
- `__FILE__`
- `__LINE__`
- `__VA_ARGS__`

## Exercice noté 52 (★★★ Implémentation table de hachage).

Nous vous proposons d'implémenter une table de hachage. Une table de hachage est une structure de données qui permet d'associer une **clé** à une **valeur**. Nous proposons ici de rester sur des clés entières et des valeurs entières.

Une table de hachage va réserver un tableau d'éléments d'une capacité donnée. Chaque élément sera accessible via une clé sur laquelle on applique une fonction de hachage. Pour les entiers, nous pouvons garder sa valeur et procéder modulo la capacité. Une fois l'emplacement donné par le hachage de la clé, nous allons à l'indice correspondant dans le tableau d'éléments. Ces éléments peuvent être au choix des tableaux ou des listes chaînées contenant les deux informations que sont la clé et la valeur associée.



La table ci-dessus correspond aux associations suivantes :

$\{0 \mapsto 1, 11 \mapsto 77, 20 \mapsto 42, 22 \mapsto 111, 42 \mapsto 100, 60 \mapsto 12, 77 \mapsto 0\}$ .

Nous aimerions comparer les performances de deux structures de données pour le comptage d'éléments et la recherche de ceux-ci. Un fichier `main.c` vous est donné pour effectuer la comparaison entre une table de hachage implémentée dans un module `hashmap` et un tableau dans un module `arraylist`. Les fichiers d'entête des modules vous sont donnés pour implémentation (à ne pas modifier) :

```
/* fichier hashmap.h */
#ifndef DEF_HEADER_HASHMAP
#define DEF_HEADER_HASHMAP

#include <stdio.h>

/* Table de hachage */
typedef struct HashMap HashMap;
struct HashMap;

/* création d'une table de hachage */
HashMap * HashMap_creer(int capacite);

/* libération d'une table de hachage */
void HashMap_free(HashMap ** hashmap);

/* ajout de 1 à la valeur associée à key, affectation à 1 si non
↪ trouvée */
int HashMap_ajouter(HashMap * hashmap, int key);

/* affiche la table de hachage dans un flux flow */
void HashMap_afficher(FILE * flow, const HashMap * hashmap);

/* renvoie le nombre d'ajouts de la clé key (valeur associée) */
int HashMap_compter(const HashMap * hashmap, int key);

#endif
```

```
/* fichier arraylist.h */
#ifndef DEF_HEADER_LISTE
#define DEF_HEADER_LISTE

#include <stdio.h>

/* Liste sous forme d'un tableau de valeurs */
typedef struct ArrayList ArrayList;
struct ArrayList;

/* création d'une liste */
ArrayList * ArrayList_creer(int capacite);
```

```
/* libération d'une liste */
void ArrayList_free(ArrayList ** arraylist);

/* ajout d'un élément dans la liste */
int ArrayList_ajouter(ArrayList * liste, int valeur);

/* affiche la liste dans un flux flow */
void ArrayList_afficher(FILE * flow, const ArrayList * liste);

/* compte le nombre d'occurrences d'une valeur dans la liste */
int ArrayList_compter(const ArrayList * liste, int valeur);

#endif
```

```
/* fichier main.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "arraylist.h"
#include "hashmap.h"

#define BENCHMARKS
#undef BENCHMARKS

#ifdef BENCHMARKS
#define CAPACITY 10000
#define ELEMENTS 1000000
#define RESEARCHES 10000
#else
#define CAPACITY 5
#define ELEMENTS 20
#endif

int main() {
    HashMap * map = HashMap_creer(CAPACITY);
    ArrayList * liste = ArrayList_creer(CAPACITY);
    long seed = time(NULL);
    int value;
```

```
    long i;
#ifdef BENCHMARKS
    int count;
    fprintf(stderr, "Filling ArrayList\n");
    clock_t start, stop;
    start = clock();
#endif
    srand(seed);
    for(i = 0; i < ELEMENTS; ++i) {
        value = rand() % ELEMENTS;
        ArrayList_ajouter(liste, value);
    }
#ifdef BENCHMARKS
    stop = clock();
    fprintf(stderr, "Filling completed in %g s\n", (double)(stop -
        ↪ start) / CLOCKS_PER_SEC);
    fprintf(stderr, "Filling HashMap\n");
    start = clock();
#endif
    srand(seed);
    for(i = 0; i < ELEMENTS; ++i) {
        value = rand() % ELEMENTS;
        HashMap_ajouter(map, value);
    }
#ifdef BENCHMARKS
    stop = clock();
    fprintf(stderr, "Filling completed in %g s\n", (double)(stop -
        ↪ start) / CLOCKS_PER_SEC);
#endif
#ifdef BENCHMARKS
    HashMap_afficher(stdout, map);
    ArrayList_afficher(stdout, liste);
#else
    seed *= 42;
    fprintf(stderr, "Research in ArrayList\n");
    start = clock();
    srand(seed);
    count = 0;
    for(i = 0; i < RESEARCHES; ++i) {
        value = rand() % ELEMENTS;
        count += ArrayList_compter(liste, value);
    }
#endif
```

```
}
stop = clock();
fprintf(stderr, "Research completed in %g s\n", (double)(stop -
↪ start) / CLOCKS_PER_SEC);
fprintf(stderr, "Counting %d elements\n", count);

fprintf(stderr, "Research in HashMap\n");
start = clock();
srand(seed);
count = 0;
for(i = 0; i < RESEARCHES; ++i) {
    value = rand() % ELEMENTS;
    count += HashMap_compter(map, value);
}
stop = clock();
fprintf(stderr, "Research completed in %g s\n", (double)(stop -
↪ start) / CLOCKS_PER_SEC);
fprintf(stderr, "Counting %d elements\n", count);
#endif
HashMap_free(&map);
ArrayList_free(&liste);
exit(EXIT_SUCCESS);
}
```

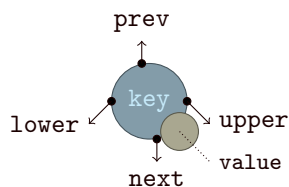
**Exercice noté 53 (\*\*\* Représentation d'un lexique).**

Nous vous proposons de représenter un lexique depuis la lecture de mots dans un fichier à l'aide du structure de nœuds chaînés. La structure est la suivante :

```
typedef struct Node Node;
struct Node {
    char key;
    int value;
    Node * lower;
    Node * upper;
    Node * next;
    Node * prev;
};
```

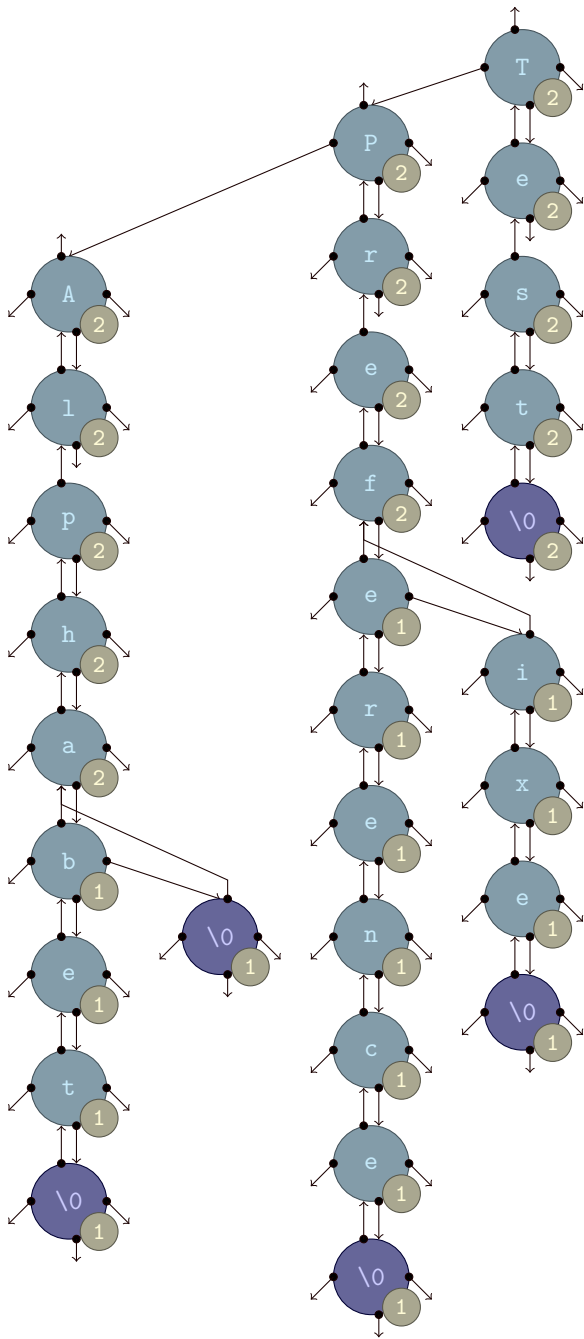
- **key** correspond à une lettre (la lettre regardée dans un mot).
- **value** correspond au nombre de fois où ce nœud a été visité avec la lettre donnée.
- **lower** correspond à un nœud vers une lettre plus petite mais au même indice dans une chaîne de caractères que le nœud courant.
- **upper** correspond à un nœud de même indice mais dont la lettre est plus grande.
- **next** correspond à un nœud dont la lettre suit la lettre courante dans un mot.
- **prev** correspond à un nœud vers la lettre qui précède celle du nœud courant.

Visuellement, on peut représenter un nœud comme suivant :



Sur les ajouts successifs des mots suivants, ceci donnerait le schéma associé ci-dessous : **Test, Preference, Prefixe, Test, Alpha et Alphabet.**





1. Depuis cette représentation, nous afficherons les mots en ordre lexicographique :

```
Alpha : 1
Alphabet : 1
Preference : 1
Prefixe : 1
Test : 2
```

2. Puis, nous afficherons le nombre d'occurrence de chaque préfixe :

```
2 : A
2 : Al
2 : Alp
2 : Alph
2 : Alpha
1 : Alphab
1 : Alphabe
1 : Alphabet
2 : P
2 : Pr
2 : Pre
2 : Pref
1 : Prefe
1 : Prefer
1 : Prefere
1 : Preferen
1 : Preferenc
1 : Preference
1 : Prefi
1 : Prefix
1 : Prefixe
2 : T
2 : Te
2 : Tes
2 : Test
```

---

## 14 Opérations bit-à-bit

---

Nous avons vu ensemble que la machine travaille sous une représentation binaire et que ceci pouvait être affiché en hexadécimal. Cependant, est-il possible de travailler directement avec cette représentation binaire sans bidouiller avec nos opérateurs arithmétiques ?

C'est possible avec des opérateurs dits bit-à-bit.

### 14.1 Décalages

Un premier opérateur est l'opérateur de décalage. Celui-ci permet de décaler la représentation binaire vers la droite ou vers la gauche dans la capacité de son type.

#### 14.1.1 À gauche

Le décalage à gauche de  $x$  par la valeur  $d$  décale les bits de  $d$  chiffres vers la gauche dans la représentation binaire de  $x$  et place des zéros aux emplacements vides. Utiliser  $\ll$  est équivalent à multiplier  $x$  par  $2^d$ . À noter que s'il y a dépassement de capacité, les bits sont perdus.

12	0	0	0	0	1	1	0	0
$12 \ll 2 = 48$	0	0	1	1	0	0	0	0

Il est par exemple possible de l'utiliser pour construire les puissances de 2 :

```
int i;
unsigned int valeur;
for(i = 0; i < 31; ++i) {
    valeur = ((unsigned int)1 << i);
    printf("%12u | 0x%08X | %2de bit à 1\n", valeur, valeur, i + 1);
}
```

Par défaut le 1 que l'on décale est en `int`. Pour s'éviter quelques désagrément, un conseil peut être de le caster en le type que l'on souhaite récupérer :

```
unsigned long x;
x = (1 << 42);
printf("%lu, le bit 42 n'existe pas ? On m'a menti !\n", x);
x = ((unsigned long)1 << 42);
printf("%lu, ouf, rassuré !\n", x);
```

### 14.1.2 À droite

L'opérateur de décalage à droite `>>` fonctionne de la même manière que l'opérateur de décalage à gauche.

24	0	0	0	1	1	0	0	0
24 >> 3 = 3	0	0	0	0	0	0	1	1

## 14.2 Négation

Il est possible de nier chaque bit à l'aide de l'opérateur de négation bit-à-bit `~`.

7	0	0	0	0	0	1	1	1
~7	1	1	1	1	1	0	0	0

À noter que la machine utilise cet opérateur de négation bit-à-bit notamment pour sa représentation des nombres négatifs. En effet, la machine utilise le complément à deux pour représenter des nombres négatifs : négation bit-à-bit puis ajout de 1. On peut le vérifier :

```
int i = 42;
printf("%d + %d = %d\n", i, ~i + 1, i + ~i + 1);
```

Cet opérateur peut être combiné avec les opérateurs de décalage pour obtenir un nombre pour lequel il manque une puissance de 2 (soit un 1 dans la représentation binaire).

```

int i;
unsigned int valeur;
for(i = 0; i < 31; ++i) {
    valeur = ~((unsigned int)1 << i);
    printf("%12u | 0x%08X | sans %2de bit à 1\n", valeur, valeur, i
        ↪ + 1);
}

```

À noter que l'opérateur de négatif fonctionne dans la plage de valeurs du type sur lequel il opère :

```

unsigned long x;
unsigned int nombre = 1;
x = ~(nombre << 2);
printf("%lu, Il y a un soucis ?\n", x);
x = ~((unsigned long)nombre << 2);
printf("%lu, Effectivement, c'est un peu plus long en fait !\n",
    ↪ x);

```

## 14.3 Opérateurs booléens

Nous avons vu précédemment les opérateurs booléens qui permettent d'opérer sur des valeurs de vérité. Cependant pour certaines applications, il peut être intéressant d'en avoir une version bit-à-bit.

### 14.3.1 et

L'opérateur d'intersection `&&` permet de vérifier que deux valeurs sont vraies. L'opérateur d'intersection bit-à-bit `&` permet d'allumer un bit correspondant à 1 si ceux des deux opérandes le sont et le laisser à 0 sinon.

3	0	0	1	1
6	0	1	1	0
&				
2	0	0	1	0

Pour gérer un système d'options, on pourrait définir une variable par option et vérifier si chacune est vraie. Cependant, une alternative proposée est de le faire dans une seule variable et appliquer un masque à l'aide de l'opérateur `&` pour vérifier que le bit correspondant est bien présent dans la valeur fournie. Pour lister ces options, on peut utiliser un type énuméré qui offrira plus de lisibilité ensuite dans le code.

```
typedef enum {
    OPTION1 = 0x0001,
    OPTION2 = 0x0002,
    OPTION3 = 0x0004
} Options;

int main() {
    Options valeur = OPTION2 + OPTION3;
    if(valeur & OPTION1)
        printf("option 1\n");
    if(valeur & OPTION2)
        printf("option 2\n");
    if(valeur & OPTION3)
        printf("option 3\n");
    exit(EXIT_SUCCESS);
}
```

### 14.3.2 ou inclusif

L'opérateur de réunion `||` permet de déterminer si au moins l'une des deux valeurs données en opérande est vraie. Son équivalent bit-à-bit `|` procède de même sur les bits des deux opérandes pour une nouvelle valeur entière.

3	0	0	1	1
6	0	1	1	0
7	0	1	1	1


L'opérateur de réunion bit-à-bit peut être utilisé pour gérer des options. Ceci permet d'assembler différentes options dans une même valeur sans chevauchements tels qu'il pourrait y avoir avec l'addition.

```
Options first = OPTION1 | OPTION2;
Options second = OPTION2 | OPTION3;
Options result = first | second;
if(result & OPTION1)
    printf("option 1\n");
if(result & OPTION2)
    printf("option 2\n");
if(result & OPTION3)
    printf("option 3\n");
```

### 14.3.3 ou exclusif : xor

L'opérateur de réunion exclusive  $\wedge$  aussi nommé "Xor" permet de vérifier qu'au moins l'une des opérande est vraie mais qu'il n'y a pas d'intersection : les deux ne sont pas vraies en même temps. Il est possible de l'utiliser sur des valeurs de vérité (ceci demandera potentiellement l'ajout de parenthèses pour les expressions) bien qu'il n'existe qu'en version bit-à-bit.

3	0	0	1	1
6	0	1	1	0
5	0	1	0	1



Il peut par exemple être utilisé pour récupérer les options qui ne sont pas communes entre deux opérandes :

```
Options first = OPTION1 | OPTION2;
Options second = OPTION2 | OPTION3;
Options result = first ^ second;
if(result & OPTION1)
    printf("option 1\n");
```

```
if(result & OPTION2)
    printf("option 2\n");
if(result & OPTION3)
    printf("option 3\n");
```

En algorithmique, nous avons vu l'échange de valeurs à l'aide d'une variable temporaire. Cependant, l'échange de deux entiers peut se faire sans variable temporaire avec un Xor.

```
int a = 42, b = 1337;
a ^= b; /* a ^ b */
b ^= a; /* b ^ a ^ b = a */
a ^= b; /* a ^ b ^ a = b */
```



## 14.4 Résumé

Il est possible d'opérer directement sur les bit à l'aide d'opérations bit-à-bit :

- Décalage bit-à-bit à gauche :

```
entier << decalage;  
/* renvoie la valeur de entier décalée de decalage vers la  
↪ gauche */  
/* En binaire : */  
/* 00000110 */  
/* << 2 */  
/* = 00011000 */
```

- Décalage bit-à-bit à droite :

```
entier >> decalage;  
/* renvoie la valeur de entier décalée de decalage vers la  
↪ droite */  
/* En binaire : */  
/* 00000110 */  
/* >> 2 */  
/* = 00000001 */
```

- Négation bit-à-bit :

```
~entier;  
/* renvoie la valeur de entier avec renversement des valeurs  
↪ des bits */  
/* En binaire : */  
/* ~ 00000110 */  
/* = 11111001 */
```

- **Intersection bit-à-bit :**

```
first & second;  
/* renvoie un entier où seuls les bits allumés dans first et  
↪ second sont conservés */  
/* En binaire : */  
/* 00000110 */  
/* & 00000011 */  
/* = 00000010 */
```

- **Union inclusive bit-à-bit :**

```
first | second;  
/* renvoie un entier où les bits allumés dans au moins first  
↪ ou second sont conservés */  
/* En binaire : */  
/* 00000110 */  
/* | 00000011 */  
/* = 00000111 */
```

- **Union exclusive bit-à-bit (Xor) :**

```
first ^ second;  
/* renvoie un entier où les bits allumés dans first ou second  
↪ sans l'être dans les deux sont conservés */  
/* En binaire : */  
/* 00000110 */  
/* ^ 00000011 */  
/* = 00000101 */
```

## 14.5 Entraînement

Exercice noté 54 (★★ Application opérations bit-à-bit).

Compléter le code suivant en utilisant les opérations bit-à-bit donnant la sortie ci-dessous :

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

int main() {
    unsigned char entier8;
    unsigned int  entier32;
    unsigned int  entier32bis;
    unsigned long entier64;
    int test;

    /* TODO : entier32 15e bit à 1, autres à 0 */
    printf("%08x (15e bit à 1)\n", entier32);
    /* TODO : entier32 14e bit à 0, autres à 1 */
    printf("%08x (14e bit à 0)\n", entier32);

    /* TODO : entier64 43e bit à 1, autres à 0 */
    printf("%016lx (43e bit à 1)\n", entier64);
    /* TODO : entier64 3e bit à 0, autres à 1 */
    printf("%016lx (3e bit à 0)\n", entier64);

    /* TODO : entier32 13e et 1e bit à 1, autres à 0 */
    printf("%08x (13e et 12e bit à 1)\n", entier32);

    /* TODO : entier32 mettre le 12e bit à 0 sans changer les autres
     → bits */
    printf("%08x (12e bit à 0)\n", entier32);

    /* TODO : entier32 changer le 13e bit sans changer les autres
     → bits */
    printf("%08x (changement 13e bit)\n", entier32);

    entier8 = 0x1;
    /* TODO : entier32 affecter au 11e bit la valeur de entier8 */
    printf("%08x (affectation 11e bit)\n", entier32);
```

```

/* TODO : mettre vrai dans test si le 11e bit de entier32 est 1
→ */
printf("%08x (test 11e bit : %d)\n", entier32, test);

entier32 = 0xFF;
/* TODO : mettre vrai dans test si les 3 bits de poids faible de
→ entier32 sont 1 */
printf("%08x (test des 3 bits de poids faible à 1 : %d)\n",
→ entier32, test);

entier32 = 0xF0;
/* TODO : mettre vrai dans test si les 4 bits de poids faible de
→ entier32 sont 0 */
printf("%08x (test des 4 bits de poids faible à 0 : %d)\n",
→ entier32, test);

entier32 = 0xFFE80F12;
entier32bis = 0x0017FOED;

/* TODO : mettre vrai dans test si les bits de entier32 et
→ entier32bis sont differents */
printf("%08x et %08x (test bits tous differents : %d)\n",
→ entier32, entier32bis, test);

exit(EXIT_SUCCESS);
}

```

```

00004000 (15e bit à 1)
ffffdfff (14e bit à 0)
0000040000000000 (43e bit à 1)
fffffffffffffb (3e bit à 0)
00001800 (13e et 12e bit à 1)
00001000 (12e bit à 0)
00000000 (changement 13e bit)
00000400 (affectation 11e bit)
00000400 (test 11e bit : 1)
000000ff (test des 3 bits de poids faible à 1 : 1)
000000f0 (test des 4 bits de poids faible à 0 : 0)
ffe80f12 et 0017f0ed (test bits tous differents : 1)

```

Exercice noté 55 (★★★ Gérer une grille sur un entier).

Implémentez les fonctionnalités qui permettent de gérer une grille d'éléments présents ou absents sur un entier long :

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

typedef unsigned long Bit8x8Grid;

Bit8x8Grid Bit8x8Grid_creer();

int Bit8x8Grid_getoffset(int x, int y);

void Bit8x8Grid_afficher(FILE * flow, Bit8x8Grid grille);

void Bit8x8Grid_set(Bit8x8Grid * grille, int x, int y, unsigned
↪ char valeur);

unsigned char Bit8x8Grid_get(Bit8x8Grid grille, int x, int y);

int main() {
    Bit8x8Grid grille = Bit8x8Grid_creer();
    Bit8x8Grid_set(&grille, 3, 1, 1);
    Bit8x8Grid_set(&grille, 7, 7, 1);
    Bit8x8Grid_afficher(stdout, grille);
    exit(EXIT_SUCCESS);
}
```

```
~~~~~
~~~#~~~
~~~~~
~~~~~
~~~~~
~~~~~
~~~~~
~~~~~
~~~~~#
```

Exercice noté 56 (★★★ Buffer pour lecture et écriture bit-à-bit).

Nous avons vu précédemment comment écrire octet par octet dans un fichier et effectuer des opérations bit-à-bit dans des variables. Cependant la taille d'une variable est limitée et nous aimerions être capable d'avoir une structure de données dans laquelle écrire et lire en bit-à-bit pour potentielle utilisation avec un fichier. Implémentez une telle structure. Celle-ci devrait fonctionner avec le code suivant :

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

#define VERBOSE
#undef VERBOSE

int main() {
    BitBuffer buffer = BitBuffer_creer();
    unsigned long infos[] = {
        1, 0x1,
        5, 0xf,
        16, 0xffff,
        32, 0x18181818,
        5, 0x7,
        16, 0xffff,
        32, 0x18181818,
        5, 0xf,
        1, 0x1,
        1, 0x1,
        1, 0x0,
        1, 0x1,
        1, 0x0,
        1, 0x0,
        1, 0x1,
        16, 0xffff,
        31, 0x8181818,
        5, 0x7,
        31, 0x8181818,
        5, 0xf,
        0
    };
    int i;
    for(i = 0; infos[2 * i] > 0; ++i) {
```

```
    BitBuffer_write(&buffer, infos[2 * i + 1], infos[2 * i]);  
#ifdef VERBOSE  
    BitBuffer_print(stderr, &buffer);  
#endif  
}  
    unsigned long data = 0;  
    for(i = 0; infos[2 * i] > 0; ++i) {  
        BitBuffer_read(&buffer, &data, infos[2 * i]);  
#ifdef VERBOSE  
        BitBuffer_print(stderr, &buffer);  
#endif  
        printf("(1) %d %lx\n", data == infos[2 * i + 1], data);  
    }  
    BitBuffer_free(&buffer);  
    exit(EXIT_SUCCESS);  
}
```

et afficher le résultat suivant :

```
(1) 1  
(1) f  
(1) ffff  
(1) 18181818  
(1) 7  
(1) ffff  
(1) 18181818  
(1) f  
(1) 1  
(1) 1  
(1) 0  
(1) 1  
(1) 0  
(1) 0  
(1) 1  
(1) ffff  
(1) 8181818  
(1) 7  
(1) 8181818  
(1) f
```





---

## 15 Types génériques

---

Souvent dans le code, pour gérer plusieurs types ou des appels de différentes fonctions, on peut être obligé de réécrire des algorithmes, devoir faire des disjonctions de cas à chaque utilisation. Ceci peut vite devenir pénible. Nous verrons donc ici comment réduire cette pénibilité en langage C.

Imaginons vouloir gérer une calculatrice à laquelle on ajouterait les opérations sous forme de fonctions :

```
#define OP(name, symb, a, b) \  
    printf("%s de %d et %d :\n", #name, a, b); \  
    printf("%d %s %d = %d\n", a, symb, b, name(a, b));  
  
int addition(int a, int b) { return a + b; }  
int soustraction(int a, int b) { return a - b; }  
int multiplication(int a, int b) { return a * b; }  
int division(int a, int b) { return a / b; }  
int modulo(int a, int b) { return a % b; }  
int decalageGauche(int a, int b) { return a << b; }  
int decalageDroite(int a, int b) { return a >> b; }
```

Une approche pour gérer la chose pourrait être par le code suivant :

```
int main() {  
    int first, second;  
    char op[5];  
    printf(">>> ");  
    scanf("%d %s %d", &first, op, &second);  
    if(strcmp("+", op) == 0) {  
        OP(addition, "+", first, second);  
    } else if(strcmp("-", op) == 0) {  
        OP(soustraction, "-", first, second);  
    } else if(strcmp("*", op) == 0) {
```

```
    OP(multiplication, "*", first, second);
} else if(strcmp("/", op) == 0) {
    OP(division, "/", first, second);
} else if(strcmp("%", op) == 0) {
    OP(modulo, "%", first, second);
} else if(strcmp("<<", op) == 0) {
    OP(decalageGauche, "<<", first, second);
} else if(strcmp(">>", op) == 0) {
    OP(decalageDroite, ">>", first, second);
}
exit(EXIT_SUCCESS);
}
```

Cependant, celui-ci va être difficilement maintenable. Il faudra le modifier à chaque fois que l'on ajoute une fonction, il y a de la duplication. Peut-être que nous pourrions faire mieux avec un tableau et une structure, mais comment sauvegarder une fonction ?

## 15.1 Pointeurs de fonctions

Un outil pour référencer une fonction est un pointeur sur une fonction.

### 15.1.1 Définition

Pour fabriquer un pointeur sur un type de fonction donné, nous allons le construire comment ayant même signature (type de retour, type et nombre des arguments identique). Pour que ce soit un pointeur et non une fonction, nous mettrons le nom entre parenthèses et ajouterons une étoile. Ceci prend la syntaxe suivante :

```
typeRetour (* nom)(typeArguments);
```

Nous avons maintenant un pointeur sur des fonctions :

```
int (* operateurBinaire)(int, int) = &addition;
operateurBinaire = &soustraction;
```

En réalité pour ne pas alourdir la syntaxe des pointeurs de fonctions, le compilateur accepte de passer directement la fonction et non son adresse :

```
int (* operateurBinaire)(int, int) = addition;
operateurBinaire = soustraction;
```

### 15.1.2 Appel

Une appel à la fonction référencée peut se faire par déréférencement de la fonction puis l'utiliser comme on le ferait avec la fonction référencée. Bien que l'absence de déréférencement ne soit pas pénalisée par le compilateur :

```
(*operateurBinaire)(1, 1); /* valide */
operateurBinaire(1, 1);    /* autorisée */
```

### Utilisation

Nous pouvons à présent maintenir une liste de nos opérations sans avoir à toucher les moments du code qui l'utilisent :

```
#define LISTOP(name, symb) {#name, #symb, name}
typedef struct OperationBinaire OperationBinaire;
struct OperationBinaire {
    char * nom;
    char * symbole;
    int (* operateur)(int, int);
};
OperationBinaire * getFromSymbole(const char * symbole,
    ↪ OperationBinaire * liste) {
    for(; liste->nom != NULL; ++liste) {
        if(strcmp(symbole, liste->symbole) == 0) {
            return liste;
        }
    }
    return NULL;
}
```

```
int main() {
    /* Ajoutez vos opérations ici : */
    OperationBinaire operations[] = {
        LISTOP(addition, +),
        LISTOP(soustraction, -),
        LISTOP(multiplication, *),
        LISTOP(division, /),
        LISTOP(modulo, %),
        LISTOP(decalageGauche, <<),
        LISTOP(decalageDroite, >>),
        {NULL}
    };

    int first, second;
    OperationBinaire *ops = NULL;
    char op[5];
    printf(">>> ");
    scanf("%d %s %d", &first, op, &second);
    if((ops = getFromSymbole(op, operations)) != NULL) {
        printf("%s de %d et %d :\n", ops->nom, first, second);
        printf("%d %s %d = %d\n", first, ops->symbole, second,
            ↪ ops->opérateur(first, second));
    }
    exit(EXIT_SUCCESS);
}
```

### 15.1.3 Constructions plus complexes

Il est possible de construire des types et réaliser des opérations plus complexes avec les pointeurs de fonctions. Une astuce peut être de considérer que pour travailler avec, on travaille à partir de son nom. Par exemple, si vous ajoutez des crochets derrière le nom vous pouvez déclarer un tableau statique, une étoile devant le nom donne un pointeur / tableau dynamique et ajouter des arguments derrière le nom permet de construire une fonction. Voyons comment ceci s'appliquerait pour un pointeur de fonction sur le schéma suivant :

```
typeRetour (* nom)(typeArguments);
```

### Tableaux statiques

On peut construire un tableau de ce type de pointeur de fonctions :

```
typeRetour (* nom[TAILLE])(typeArguments);
```

Ceci pourrait donner par exemple le tableau suivant :

```
int (* tableau[])(int, int) = {  
    addition,  
    soustraction,  
    NULL  
};  
  
tableau[1](2, 1);
```

### Tableaux dynamiques

On peut construire un pointeur sur ce type de pointeur de fonctions :

```
typeRetour (** nom)(typeArguments);
```

Une allocation dynamique d'un tableau de pointeurs vers les fonctions vues précédemment pourrait par exemple donner le code suivant :

```
int (** pointeur)(int, int) = NULL;  
if((pointeur = (int (**)(int, int))malloc(sizeof(int (*)(int,  
↪ int)) * 3)) == NULL) {  
    fprintf(stderr, "Erreur allocation.\n");  
    exit(EXIT_FAILURE);  
}  
  
*pointeur = addition;  
*(pointeur + 1) = soustraction;
```

```
*(pointeur + 2) = NULL;

(*(pointeur + 1))(2, 1);
pointeur[1](2, 1);
```

### Type de retour d'une fonction

On peut construire une fonction qui renvoie ce type de pointeur de fonctions :

```
typeRetour (* nom(ArgumentsFonction))(typeArguments);
```

Ceci pourrait par exemple être utilisé pour renvoyer une fonction selon un paramètre donné comme le symbole d'une opération :

```
int (* selectOperation(const char * symbole))(int, int) {
    if(strcmp(symbole, "+") == 0)
        return addition;
    else if(strcmp(symbole, "-") == 0)
        return soustraction;
    return NULL;
}
```

#### 15.1.4 Typedef

Il est possible de construire des types de plus en plus complexes avec les opérations vues précédemment. Cependant, ceci peut vite devenir abominable et difficile à lire. Bien que l'exemple suivant soit une allocation dynamique d'un pointeur sur une fonction qui renvoie l'adresse d'une fonction prenant en entrée un entier et renvoyant un entier. Trivial, non ?

```
int carre(int x) {
    return x * x;
}

int (* fc(int nb))(int) {
    switch(nb) {
```

```

    case 0 : return carre;
    default : return NULL;
}
}

int main() {
    int (** ppfc)(int)(int) = (int
    ↪  (**)(int)(int))malloc(sizeof(int (*)(int)(int)));
    *ppfc = fc;
    printf("%d\n", (**ppfc)(0)(4));
    exit(EXIT_SUCCESS);
}

```

Tout comme pour les types vus précédemment, il est possible de créer un synonyme sur un type de pointeur de fonctions avec typedef :

```

typedef int (* mapIntToInt)(int);

int carre(int x) {
    return x * x;
}

mapIntToInt fc(int nb) {
    switch(nb) {
        case 0 : return carre;
        default : return NULL;
    }
}

int main() {
    mapIntToInt (** ppfc)(int) = (mapIntToInt
    ↪  (**)(int))malloc(sizeof(mapIntToInt (*)(int)));
    *ppfc = fc;
    printf("%d\n", (*ppfc)(0)(4));
    exit(EXIT_SUCCESS);
}

```

```
}
```

Ceci est aussi applicable pour simplifier le code de manière à gérer le code de manière simplifiée. Attention à bien avoir conscience des types que vous manipulez à cause de l'abstraction que ceci peut créer.

```
typedef int (* mapIntToInt)(int);
typedef mapIntToInt (* selectMITIFromInt)(int);

int carre(int x) {
    return x * x;
}

mapIntToInt fc(int nb) {
    switch(nb) {
        case 0 : return carre;
        default : return NULL;
    }
}

int main() {
    selectMITIFromInt * ppfc = (selectMITIFromInt
    ↪ *)malloc(sizeof(selectMITIFromInt));
    *ppfc = fc;
    printf("%d\n", (*ppfc)(0)(4));
    exit(EXIT_SUCCESS);
}
```

## 15.2 Pointeurs génériques

Vous avez codé votre algorithme de tri ou une autre structure de données en C. Malheureusement cet outil ne convient que pour le type pour lequel c'est paramétré. Vous faites copié-collé et vous modifiez les types ?



### 15.2.1 Intérêt

Le typage en C est une contrainte qui limite la généricité du code. Par change, il est possible de créer des pointeurs génériques pour abstraire un type utilisé dans notre code. Ce type est le `void *`.

### 15.2.2 Exemple avec `qsort`

Prenons pour exemple la méthode de tri fournie par la bibliothèque standard. C'est une implémentation générique du tri rapide :

```
# man 3 qsort

QSORT(3)          Linux Programmer's Manual          QSORT(3)

NAME
    qsort, qsort_r - sort an array

SYNOPSIS
    #include <stdlib.h>

    void qsort(void *base, size_t nmemb, size_t size,
               int (*compar)(const void *, const void *));
```

Pour utiliser `qsort`, vous allez devoir fournir votre tableau, le nombre d'éléments, la taille de chacun puis une fonction de comparaison. Attention comme vous pouvez le voir, cette fonction de comparaison prend en paramètres des pointeur génériques. Ceci peut par exemple se faire pour trier une liste d'entiers en ordre croissant à l'aide de l'appel à `qsort` suivant et de la fonction de comparaison suivante :

```
int intcmp(const void * firstAddress, const void * secondAddress)
↪ {
    int first = *((int *)firstAddress);
    int second = *((int *)secondAddress);
    return first - second; /* first < second <=> first - second < 0
    ↪ */
}

void afficherListe(int * valeurs, int taille) {
```

```
int i;
for(i = 0; i < taille; ++i) {
    if(i) printf(", ");
    printf("%d", valeurs[i]);
}
printf("\n");
}

int main() {
    int valeurs[] = {5, 9, 2, 0, 6, 1, 3, 8, 7, 4};
    afficherListe(valeurs, 10);
    qsort(valeurs, 10, sizeof(int), &intcmp);
    afficherListe(valeurs, 10);
    exit(EXIT_SUCCESS);
}
```

Devoir construire des fonctions qui prennent obligatoirement des arguments en pointeur générique peut être fastidieux. Pour ceci, vous pouvez utiliser le type de pointeur correspondant à votre type et effectuer un cast de votre pointeur de fonction avec la signature générique attendue pour la passer à l'outil en nécessitant :

```
int intcmp(const int * first, const int * second) {
    return *first - *second;
}

/* ... */

int main() {

    int valeurs[] = {5, 9, 2, 0, 6, 1, 3, 8, 7, 4};
    afficherListe(valeurs, 10);
    qsort(valeurs, 10, sizeof(int), (int (*)(const void *, const
    ↪ void *))&intcmp);
    afficherListe(valeurs, 10);
    exit(EXIT_SUCCESS);
}
```

```
}
```

### 15.2.3 Manipulation

Vous pouvez aussi vous-même construire des pointeurs génériques pour sauvegarder une information dans vos structures de données et pour y associer des fonctionnalités. Ou comme dans le cas en `qsort` proposer des méthodes fonctionnant sur le type que vous donnera un utilisateur sous réserve qu'il vous fournisse les fonctionnalités nécessaires.

Ceci peut par exemple permettre la création d'une liste générique. Dans cette liste, l'aspect générique nous demande d'être plus vigilants sur les points suivants :

- Nous avons besoin de connaître la taille d'un élément.
- Lorsque nous avons besoin d'une fonctionnalité particulière sur un élément (affichage, allocation, libération et autres), il faut que celle-ci soit fournie lorsque nécessaire ou idéalement en amont.
- L'arithmétique des pointeurs ne se fera pas automatiquement. Une proposition connaissant le taille en octets et de travailler sur l'adresse avec des pas d'un octet. Ceci se fait par exemple avec des adresses en `unsigned char *`.

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

typedef struct Liste Liste;
struct Liste {
    void * data;
    long taille;
    long capacite;
    size_t elementTaille;
    void (* elementFree)(void *);
    void (* elementPrint)(void *);
};
```

```
int Liste_create(
    Liste * liste, long capacite, size_t elementTaille,
    void (* elementFree)(void *), void (* elementPrint)(void *)) {
    liste->data = NULL;
    liste->taille = 0;
    liste->capacite = capacite;
    liste->elementTaille = elementTaille;
    liste->elementFree = elementFree;
    liste->elementPrint = elementPrint;
    if((liste->data = malloc(elementTaille * capacite)) == NULL) {
        fprintf(stderr, "Liste_create : Erreur allocation liste.\n");
        return 1; /* code d'erreur allocation */
    }
    return 0;
}

void Liste_free(Liste * liste) {
    if(liste == NULL || liste->data == NULL) {
        return;
    }
    long i;
    if(liste->elementFree) {
        for(i = 0; i < liste->taille; ++i) {
            liste->elementFree((unsigned char *) (liste->data) + (i
            ↪ * liste->elementTaille));
        }
    }
    free(liste->data);
    liste->data = NULL;
}

void Liste_afficher(Liste * liste) {
    long i;
    printf("[");
```

```
for(i = 0; i < liste->taille; ++i) {
    if(i) printf(", ");
    liste->elementPrint((unsigned char *)(liste->data) + (i
↪ * liste->elementTaille));
}
printf("]\n");
}

int Liste_ajouter(Liste * liste, void * element) {
    if(liste->taille >= liste->capacite) {
        void * tmp = NULL;
        int new_capacite = liste->capacite * 2 + 10;
        if((tmp = realloc(liste->data, new_capacite *
↪ liste->elementTaille)) == NULL) {
            fprintf(stderr, "Liste_ajouter : Erreur
↪ reallocation.\n");
            return 1; /* code d'erreur allocation */
        }
        liste->data = tmp;
        liste->capacite = new_capacite;
    }
    memcpy((unsigned char *)(liste->data) + (liste->taille *
↪ liste->elementTaille), element, liste->elementTaille);
    ++(liste->taille);
    return 0;
}

void intPrint(int * value) {
    printf("%d", *value);
}

int main() {

    Liste intListe;
```

```
if(Liste_create(&intListe, 0, sizeof(int), NULL, (void (*)(void
↪ *))&intPrint)) {
    fprintf(stderr, "Erreur allocation intListe :
    ↪ arrêt.\n");
    exit(EXIT_FAILURE);
}
int valeur;
printf("Entrez des valeurs positives : ");
scanf("%d", &valeur);
while(valeur >= 0) {
    Liste_ajouter(&intListe, &valeur);
    scanf("%d", &valeur);
}
Liste_afficher(&intListe);
Liste_free(&intListe);

exit(EXIT_SUCCESS);
}
```

### 15.3 Fonctions variadiques

Nous avons jusqu'ici toujours défini des fonctions avec un nombre de paramètres connu. Cependant `printf` et `scanf` prennent bien un nombre d'argument variable, non ?

En effet, il est possible de fabriquer des fonctions avec un nombre d'arguments variables à l'aide de la bibliothèque `stdarg.h`. Celle-ci fournit un type `va_list` qui permet de gérer les arguments variables d'une fonction. Une fonction variadique s'organise de la manière suivante :

```
typeRetour fonctionVariadique(parametres, dernierParametre, ...) {
    va_list ap; /* argument pointer */
    va_start(ap, dernierParametre); /* initialisation */
    /* Traitement et récupération avec va_arg(ap, typeElement) */
    va_end(ap); /* arrêt d'utilisation */
}
```

```
}
```

Pour définir une fonction qui calcule une moyenne, ceci pourrait par exemple donner le code suivant :

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

double moyenne(int nombre, ...) {
    double somme = 0;
    int i;
    va_list ap;
    va_start(ap, nombre);
    for(i = 0; i < nombre; ++i) {
        somme += va_arg(ap, double);
    }
    va_end(ap);
    if(nombre)
        somme /= nombre;
    return somme;
}

int main() {
    printf("%g\n", moyenne(3, 10., 11., 13.));
    exit(EXIT_SUCCESS);
}
```

## 15.4 Résumé

Une variable qui peut référencer une fonction est modélisée par un pointeur de fonction :

```
typeRetour (* pointeur)(typeParametres); /* pointeur sur une
↳ fonction de signature typeRetour fonction(typeParametres) */
pointeur = fonction; /* affectation */
pointeur(arguments); /* appel à fonction via pointeur */
```

Il est possible de considérer un pointeur de fonction comme un type de variable pour y appliquer les opérations suivantes :

```
typeRetour (* tableau[])(typeParametres); /* tableau de pointeurs
↳ de fonctions */
typeRetour (** pointeur)(typeParametres); /* pointeur sur un
↳ pointeur de fonction */
typeRetour (* fonction(parametresDeFonction))(typeParametres); /*
↳ signature d'une fonction renvoyant un pointeur de fonction */
typedef typeRetour (* synonyme)(typeParametres); /* synonyme sur
↳ un pointeur de fonction */
```

Lorsque l'on souhaite manipuler des données de manière générique indépendamment de leur type, il est possible d'utiliser un pointeur générique `void *` :

```
void * pointeurGenerique; /* pointeur générique pour sauvegarde
↳ d'information */
(type *)pointeurGenerique; /* transformation du pointeur générique
↳ en le type réellement enregistré */
void * fonctionGenerique(void * pointeur); /* fonction utilisable
↳ dans un code générique */
```



Il est possible de définir une fonction prenant un nombre variable de paramètres à l'aide de la bibliothèque `stdarg` :

```
typeRetour fonctionVariadique(parametres, dernierParametre, ...) {  
    va_list ap; /* argument pointer */  
    va_start(ap, dernierParametre); /* initialisation */  
    /* Traitement et récupération avec va_arg(ap, typeElement) */  
    va_end(ap); /* arrêt d'utilisation */  
}
```

## 15.5 Entraînement

Exercice noté 57 (★★★ Trier une liste de points).

Proposer un programme qui trie une liste de points :

1. Par rapport à leur distance à l'origine.
2. Par rapport à un point donné.

```
Distance origine :  
(0, 1) 1  
(-3, 0) 3  
(0, 3) 3  
(2, 3) 3.60555  
(4, 0) 4  
Distance (3, 3) :  
(2, 3) 1  
(0, 1) 3.60555  
(0, 3) 3  
(4, 0) 3.16228  
(-3, 0) 6.7082
```

Exercice noté 58 (★★★★ Benchmark de `qsort`).

1. Implémenter un tri par tas générique.
2. Proposer un programme qui compare les performances du tri par tas avec celles de `qsort` de la bibliothèque standard `stdlib`.
3. Que penser des performances de `qsort` ?

Exercice noté 59 (★★★ Trier des pointeurs de fonctions).

Votre binôme est à deux doigts de ragequit le cours de langage C. Il essaie de trier des pointeurs de fonctions mais rien ne marche. Vous devez lui venir en aide ! Les pointeurs de fonctions seraient triés par ordre croissant de la valeur qu'ils renverraient pour une valeur donnée.

Code de votre binôme :

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

float carre(float x) {
    return x * x;
}

float cube(float x) {
    return x * x * x;
}

float inverse(float x) {
    return 1.f / x;
}

float oppose(float x) {
    return -x;
}

int fcmp(float first(float x), float second(float x)) {
    return first(x) - second(x);
}

int main() {
    float fonctions(float x)[] = {
        carre,
        cube,
        inverse,
        oppose
    };
    float targets[] = {
        -2.f,
        -0.5f,
```

```
    0.5f,  
    2.f  
};  
int i, j;  
for(i = 0; i < 4; ++i) {  
    qsort(fonctions(targets[i]), sizeof(float f(float x)), 4,  
        ↪ fcmp);  
    printf("for %g :\n", targets[i]);  
    for(j = 0; j < 4; ++j) {  
        printf(" - %s(%g) = %g\n", fonctions(targets[i])[j].name,  
            ↪ targets[i], fonctions(targets[i])[j]);  
    }  
}  
exit(EXIT_SUCCESS);  
}
```

Sortie attendue par le professeur :

```
for -2 :  
- cube(-2) = -8  
- inverse(-2) = -0.5  
- oppose(-2) = 2  
- carre(-2) = 4  
for -0.5 :  
- inverse(-0.5) = -2  
- cube(-0.5) = -0.125  
- carre(-0.5) = 0.25  
- oppose(-0.5) = 0.5  
for 0.5 :  
- oppose(0.5) = -0.5  
- cube(0.5) = 0.125  
- carre(0.5) = 0.25  
- inverse(0.5) = 2  
for 2 :  
- oppose(2) = -2  
- inverse(2) = 0.5  
- carre(2) = 4  
- cube(2) = 8
```

**Exercice noté 60 (★★★★ HashMap et ArrayList génériques).**

Oscar salue vos aptitudes en langage C pour être arrivé jusqu'ici, mais il a un dernier défi pour vous avant de passer au projet final. Il a pu observer des outils génériques dans d'autres langages de programmation. Il vous propose d'essayer de coder ce type de structures en langage C. L'idée serait de proposer une table de hachage qui puisse prendre des entrées génériques : comme des listes génériques. À vous de proposer une implémentation qui réponde à cette demande et soit en mesure de modéliser la sortie suivante :

```
HashMap<string, ArrayList> : {"grands" : ArrayList<long> :  
  ↳ [5000000000, 30000000000, -42000000000], "caracteres" :  
  ↳ ArrayList<char> : ['E', 'S', 'G', 'I'], "entiers" :  
  ↳ ArrayList<int> : [1, 17, 3, 42, 5], "a virgule" :  
  ↳ ArrayList<float> : [4.5, 13.37, 1.2, 8.1, 99.9, 4.2, 1.2],  
  ↳ "texte" : ArrayList<string> : ["Hello", "ESGI"]}
```



---

## 16 Projet final

---

### 16.1 Étapes

Nous allons procéder en quelques étapes pour ce projet :

1. **(Deadline : 03 Octobre 2022)** S'armer d'un ou deux camarades pour attaquer le projet.  
(Dans le cas où vous n'êtes pas affecté à un groupe de 2 à 3 personnes dans les temps sur MyGES, le professeur procédera à un matchmaking automatique).
2. **(Deadline : 31 Octobre 2022)** Choisir une thématique et un sujet puis les faire valider par le professeur.  
(Dans le cas où ceux-ci n'ont pas été donnés ou validés par le professeur dans les temps, le sujet sera imposé).
3. **(Deadline : 30 Novembre 2023)** Rendu d'une roadmap du projet : organiser le projet (répartition des tâches dans le groupe), listing des fonctionnalités, planification dans le temps.
4. **(Deadline : 01 Février 2023)** Rendu d'un prototype du projet (preuve de concept, base à peaufiner).
5. **(20 Février 2023)** Rendu final du projet.
6. **(24 Février 2023)** Soutenance.

### 16.2 Possibilités de sujet

Vous pouvez choisir un sujet qui met en avant des concepts algorithmiques ou relatifs à des structures de données. Ceci peut être du benchmarking, une utilisation et mise en évidence de celle-ci dans un contexte donné. Ceci peut s'intéresser à des structures chaînées (listes, arbres, graphes) et mettre en évidence leur manipulation dans un cas pratique ou comparer à d'autres structures sur des tailles

de données pertinentes.

Une possibilité est de réaliser un jeu vidéo à l'aide d'une bibliothèque graphique telle que SDL. À vous de proposer des mécaniques intéressantes qui permettent l'utilisation de type structurés et de types plus génériques. Ceci peut aussi permettre de s'intéresser à des méthodes algorithmiques et structures de données pour gérer un nombre d'entités important dans le contexte que vous proposerez.

Vous pouvez aussi vous intéresser à la programmation système. Ceci peut découler sur la reproduction de commandes Linux, la simulation d'un système de fichiers.

Il est aussi possible de s'intéresser à des fonctionnalités plus orientées web et base et données dont la récupération de pages web avec cURL ou des appels à une base de données via MySQL.

## 16.3 Livrables

Vous devrez rendre sur MyGES les documents suivants :

1. **(Deadline : 31 Octobre 2022)** Un document rapide donnant le sujet de votre projet (1 page) après validation du sujet par le professeur.
2. **(Deadline : 30 Novembre 2023)** Fichier roadmap du projet : organiser le projet (répartition des tâches dans le groupe), listing des fonctionnalités, planification dans le temps.
3. **(Deadline : 01 Février 2023)** Sources fonctionnelles et viables en langage C du **prototype** avec Readme et Makefile.
4. **(20 Février 2023)** Sources fonctionnelles et viables en langage C du **projet finalisé** avec Readme / rapport, Makefile et accès à un git.

## 16.4 Évaluation

Votre score final pourra varier entre 0 et 21 sur 20 points. L'obtention d'une note supérieure à 20 ne pourra pas être enregistrée sur MyGES mais vous donnera toute justification pour demander à votre professeur de mettre votre projet à l'honneur. À noter que votre score sera individuel : en effet vous devrez justifier dans le rapport et à l'aide d'un git de l'investissement de chacun pour une attribution correcte des points. En effet, 10 points sont communs au groupe (sous réserve d'avoir démontré une activité dans celui-ci) et 11 points sont individuels.



### 16.4.1 (... / 9 points) Code

(... / 7 points de groupe)

(... / 2 points individuels)

Cette année et l'année précédente nous avons étudié des concepts qui vous permettent l'écriture d'un code en langage C. La qualité du code et le niveau des concepts utilisés comme démonstration des capacités techniques du candidats seront valorisés. Veuillez à garder tout aussi lisible que pertinent.

Dans le code nous attendons :

- (... / 1 point : individuel) Un code propre, indenté, sans warnings et qui compile (avec `gcc` sous une machine Ubuntu 64 bits).
- (... / 1 point : individuel) De la documentation : un autre programmeur ou vous-même doivent pouvoir continuer votre projet après son rendu.
- (... / 1 point) Une découpe en modules pertinente.
- (... / 1 point) Structuration du code lorsque pertinent : types structurés, énumérations.
- (... / 1 point) Généricité du code lorsque pertinent : pointeurs de fonctions, types génériques.
- (... / 4 points) Relatif à la thématique :
  - (... / 2 points) Utilisation et mise en place de bibliothèques (graphique, système, base de données et autres).
  - (... / 2 points) Qualité de l'implémentation (structures de données, généricité, benchmarking, niveau conceptuel).

### 16.4.2 (... / 7 points) Fonctionnalités

(... / 1 point de groupe)

(... / 6 points individuels)

Votre pratique du langage C et de l'algorithmique vous a permis de développer une logique et capacité réussir avec succès le codage de fonctionnalités souhaitées. Vous démontrez ici votre capacité à faire vivre votre pensée dans une programme automatique et efficace.

Côté fonctionnalités, nous attendons :

- (... / 1 point) Un programme efficace (pas de latence inexpliquée) et qui ne crash pas (ou du moins précise la raison de l'arrêt du programme pour réussir à le faire fonctionner).

- (... / **2 points : individuel**) Appréciation de l'atteinte des objectifs fixés dans la roadmap.
- (... / **2 points : individuel**) Appréciation de la pertinence du prototype (qualité / niveau technique de la réalisation).
- (... / **2 points : individuel**) Appréciation de la pertinence du rendu final (qualité / niveau technique de la réalisation).

### 16.4.3 (... / 5 points) Soutenance et rapport

(... / **2 points de groupe**)

(... / **3 points individuels**)

La soutenance et rapport ont pour but d'aider à la prise en main de votre projet et comprendre votre cheminement dans sa conception. Le rapport peut être matérialisé par un fichier `README.md` et / ou un fichier `.pdf`. Montrez que votre projet a été réalisé en équipe avec professionnalisme.

Nous attendons du rapport et de la soutenance qu'ils nous informent sur :

- (... / **0.5 point**) Comment lancer votre projet ? Quelles sont ses dépendances ?
- (... / **1 point**) Le contexte et l'organisation de votre groupe pour répondre à la problématique : avez-vous établi un planning ? Qui a fait quoi et pourquoi ?
- (... / **1 point : individuel**) Pourquoi avez-vous organisé le code de cette manière ? Quels sont les éléments du cours ou du langage C que vous avez utilisé pour le mener à bien ? Ces choix ont-ils été pertinent pour avancer plus rapidement dans votre code, garder en maintenabilité ?
- (... / **1 point : individuel**) Quelles fonctionnalités proposez-vous dans ce projet ? Comment les avez-vous optimisées ? Valorisez l'apport de vos connaissances en algorithmique et leur intérêt pour ce projet en langage C.
- (... / **0.5 point**) Avez-vous rencontré des difficultés techniques, organisationnelles, relationnelles pour réaliser ce projet ? Comment les avez-vous dépassées ?
- (... / **1 point : individuel**) La concordance du projet avec votre formation. Quelles sont les aptitudes qui ont été requises de vous et comment votre formation à l'ESGI (ce cours ou d'autres) vous a aidé ou fait défaut ? Avez-vous aimé réaliser ce projet ? Pourquoi ?

Bon courage !

## Cinquième partie

# Bonus : jouons rapidement avec SDL 1.2



# Table des matières

<b>17 Initialisation</b>	<b>311</b>
17.1 Préambule . . . . .	311
17.2 Mise en place . . . . .	311
17.2.1 Linux . . . . .	311
17.2.2 Windows . . . . .	313
17.3 Fenêtre . . . . .	314
17.3.1 SDL Init . . . . .	314
17.3.2 SDL SetVideoMode . . . . .	315
<b>18 Affichage</b>	<b>317</b>
18.1 Dessiner un rectangle . . . . .	318
18.1.1 SDL FillRect . . . . .	318
18.1.2 SDL Rect . . . . .	319
18.2 Créer une surface . . . . .	320
18.2.1 SDL CreateRGBSurface . . . . .	320
18.2.2 SDL BlitSurface . . . . .	322
18.3 Images en BMP . . . . .	324
18.3.1 SDL Image . . . . .	327
18.3.2 Dessiner une sous-image . . . . .	328
18.4 Résumé . . . . .	331
<b>19 Événements</b>	<b>333</b>
19.1 Récupération d'événements . . . . .	333
19.1.1 SDL WaitEvent . . . . .	333
19.1.2 SDL PollEvent . . . . .	334
19.2 Analyse d'événements . . . . .	334
19.2.1 Types d'événements . . . . .	334
19.2.2 Clavier . . . . .	335

19.2.3 Boutons de souris . . . . .	337
19.2.4 Coordonnées de souris . . . . .	337
19.3 Résumé . . . . .	340

---

# 17 Initialisation

---

## 17.1 Préambule

Cette partie sur SDL fait suite des demandes d'étudiants, elle est donc facultative et à étudier en autonomie. Elle ne sera pas traitée explicitement en classe, mais sera valorisée si vous choisissez de l'exploiter dans votre projet. Amusez vous !

## 17.2 Mise en place

### 17.2.1 Linux

Sous Linux, vous pouvez installer SDL 1.2 à l'aide des commandes suivantes :

```
sudo apt-get install libsdl1.2-dev
sudo apt-get install libsdl-image1.2-dev
sudo apt-get install libsdl-gfx1.2-dev
sudo apt-get install libsdl-ttf2.0-dev
sudo apt-get install libsdl-mixer1.2-dev
sudo apt-get install libglib2.0-dev
```

La compilation avec SDL se fait en ajoutant `-lSDL` et les bibliothèques que vous utiliserez :

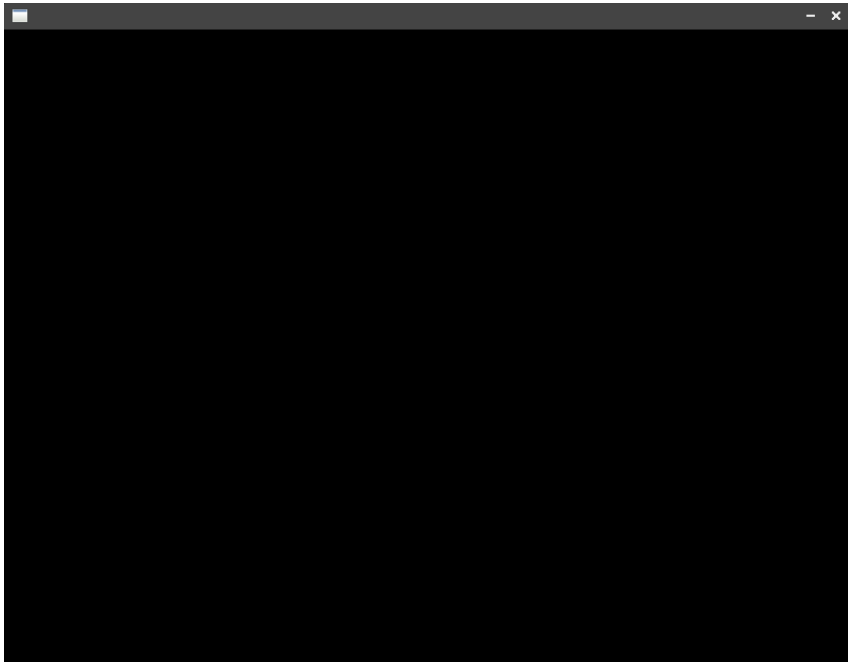
```
gcc main.c -lSDL
gcc main.c -lSDL -lSDL_image -lSDL_ttf -lSDL_gfx -lSDL_mixer
```

Vous pouvez vérifier la bonne installation de SDL en compilant le code suivant. Ce code lance une fenêtre sur fond noir et peut se fermer en cliquant sur la croix en haut :

```
#include <SDL/SDL.h>

int main() {
    SDL_Init(SDL_INIT_VIDEO);
    SDL_Surface * ecran = SDL_SetVideoMode(800, 600, 32,
    ↪   SDL_HWSURFACE);
    SDL_Event event;
    int active = 1;
    while(active) {
        SDL_WaitEvent(&event);
        switch(event.type) {
            case SDL_QUIT : active = 0; break;
        }
    }
    SDL_FreeSurface(ecran);
    SDL_Quit();
    exit(EXIT_SUCCESS);
}
```

L'exécution donne le résultat suivant :





### 17.2.2 Windows

Sous Windows, sous **MSYS2 MinGW x64**, vous pouvez installer SDL à l'aide des commandes suivantes :

```
pacman -S mingw-w64-x86_64-SDL
pacman -S mingw-w64-x86_64-SDL_image
pacman -S mingw-w64-x86_64-SDL_ttf
pacman -S mingw-w64-x86_64-SDL_gfx
pacman -S mingw-w64-x86_64-SDL_mixer
```

La compilation sous MSYS2 avec SDL se fait en ajoutant `-lmingw32 -lSDLmain -lSDL` et les bibliothèques que vous utiliseriez :

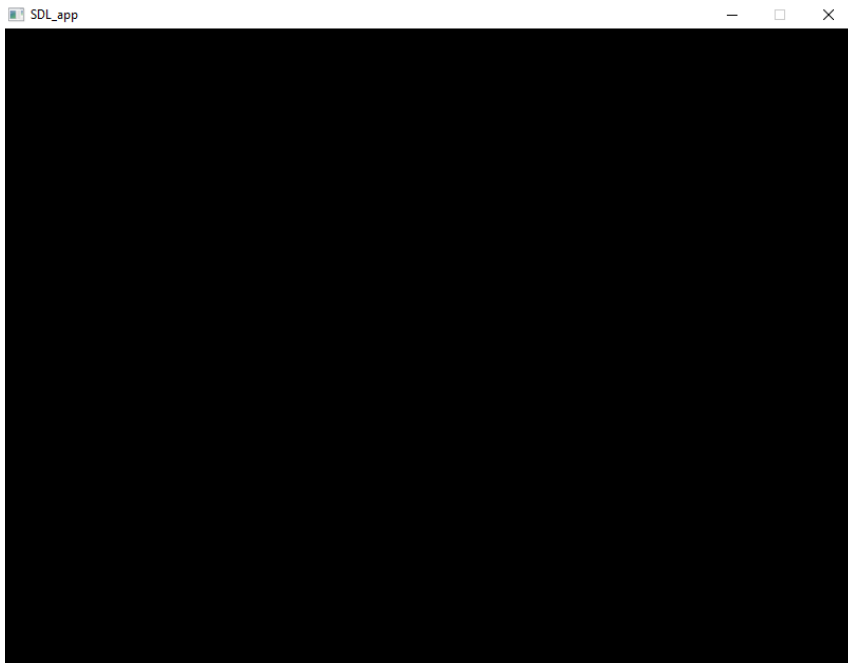
```
gcc main.c -lmingw32 -lSDLmain -lSDL
gcc main.c -lmingw32 -lSDLmain -lSDL -lSDL_image -lSDL_ttf
↪ -lSDL_gfx -lSDL_mixer
```

Compiler sous Windows avec MSYS2 vous demandera d'ajouter une définition en entête avant inclusion de SDL et d'avoir une signature complète pour la fonction `main` :

```
#define SDL_MAIN_HANDLED
#include <SDL/SDL.h>

int main(int argc, char * argv[]) {
    SDL_Init(SDL_INIT_VIDEO);
    SDL_Surface * ecran = SDL_SetVideoMode(800, 600, 32,
    ↪  SDL_HWSURFACE);
    SDL_Event event;
    int active = 1;
    while(active) {
        SDL_WaitEvent(&event);
        switch(event.type) {
            case SDL_QUIT : active = 0; break;
        }
    }
    SDL_FreeSurface(ecran);
    SDL_Quit();
    exit(EXIT_SUCCESS);
}
```

L'exécution donne le résultat suivant :



## 17.3 Fenêtre

Nous avons vu précédemment un code minimaliste permettant d'afficher et maintenir une fenêtre jusqu'à ce que l'utilisateur demande sa fermeture. SDL est une bibliothèque qui permet de travailler dans une fenêtre graphique.

### 17.3.1 SDL Init

Pour utiliser SDL dans une application, il faut inclure l'entête de la bibliothèque `SDL.h` qui donne accès à tous les modules par défaut de SDL. Dans le code, on annonce l'utilisation de SDL, nous appellerons `SDL_Init` avec les options choisies. Les principales options que nous pourrions utiliser sont les suivantes :

- `SDL_INIT_VIDEO` : initialise le sous-système graphique.
- `SDL_INIT_AUDIO` : initialise le sous-système audio.
- `SDL_INIT_TIMER` : initialise le sous-système de gestion du temps.
- `SDL_INIT EVERYTHING` : initialise tous les sous-systèmes.

Nous vérifierons que l'initialisation de SDL a fonctionné à l'aide de sa valeur de retour. En cas d'erreur le code d'erreur sera -1 et 0 en cas de succès. Notez que SDL peut allouer des ressources et peut nécessiter de mettre à l'arrêt les sous-systèmes utilisés, pensez donc à terminer le programme proprement avec `SDL_Quit` :

```
#include <SDL/SDL.h>

int main() {
    if(SDL_Init(SDL_INIT EVERYTHING) != 0) {
        fprintf(stderr, "Error in SDL_Init : %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
    }
    printf("Hello SDL\n");
    SDL_Quit();
    exit(EXIT_SUCCESS);
}
```

### 17.3.2 SDL SetVideoMode

Il est cependant bien plus intéressant d'utiliser SDL avec une fenêtre graphique. Dans SDL 1.2, la gestion de l'affichage, des images, de l'écran de la fenêtre se fait avec des surfaces `SDL_Surface *`. Ces surfaces doivent être libérées à l'aide de `SDL_FreeSurface`. La définition de l'écran de la fenêtre se fait par `SDL_SetVideoMode` et prend les paramètres suivants :

- **Largeur** de la fenêtre.
- **Hauteur** de la fenêtre.
- **Bits par pixel**. À noter que 32 bits par pixel, c'est 8 bits par canal RGBA et donc des valeurs entières entre 0 et 255.
- **Mode de création** de la fenêtre.

Les principaux modes de création de fenêtre que nous utiliserons sont les suivants :

- `SDL_HWSURFACE` : crée une surface vidéo dans la mémoire vidéo.
- `SDL_DOUBLEBUF` : double la surface vidéo (une surface est affichée et la seconde est utilisée pour travailler dessus, la surface affichée est actualisée par celle de travail par l'appel de `SDL_Flip`). nécessite `SDL_HWSURFACE`.
- `SDL_FULLSCREEN` : lance la fenêtre en plein écran.
- `SDL_RESIZABLE` : autorise l'utilisateur à changer les dimensions de la fenêtre à la volée.
- `SDL_NOFRAME` : retire le cadre de la fenêtre, ne laissant visible que la surface.

Exemple de lancement d'une fenêtre en plein écran :

```
#include <SDL/SDL.h>

int main() {
    if(SDL_Init(SDL_INIT_EVERYTHING) != 0) {
        fprintf(stderr, "Error in SDL_Init : %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
    }
    SDL_Surface * ecran = NULL;
    if((ecran = SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE |
    ↪ SDL_DOUBLEBUF | SDL_FULLSCREEN)) == NULL) {
        fprintf(stderr, "Error in SDL_SetVideoMode : %s\n",
        ↪ SDL_GetError());
        SDL_Quit();
        exit(EXIT_FAILURE);
    }
    printf("Hello FullScreen\n");
    SDL_Quit();
    exit(EXIT_SUCCESS);
}
```

Pour garder votre fenêtre active le temps d'en visualiser le résultat, vous pouvez ajouter les lignes suivantes (nous les étudierons plus en détails au moment de la gestion des événements) :

```
int active = 1;
SDL_Event event;
while(active) {
    SDL_WaitEvent(&event);
    switch(event.type) {
        case SDL_QUIT : active = 0; break;
        case SDL_KEYUP : active = 0; break;
        default : break;
    }
}
```

Ceci permet à l'utilisateur de terminer le programme en appuyant sur une touche ou en cliquant sur la croix en haut.

---

## 18 Affichage

---

Pour la suite, afin de ne pas alourdir le code proposé, vous pouvez considérer que nous utiliserons le code suivant pour envelopper notre travail :

```
#include <SDL/SDL.h>

int main() {
    if(SDL_Init(SDL_INIT EVERYTHING) != 0) {
        fprintf(stderr, "Error in SDL_Init : %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
    }
    SDL_Surface * ecran = NULL;
    if((ecran = SDL_SetVideoMode(800, 600, 32, SDL_HWSURFACE |
    ↪ SDL_DOUBLEBUF)) == NULL) {
        fprintf(stderr, "Error in SDL_SetVideoMode : %s\n",
        ↪ SDL_GetError());
        SDL_Quit();
        exit(EXIT_FAILURE);
    }
    SDL_WM_SetCaption("Initiation à la SDL !", NULL);

    /* Votre travail ici */

    int active = 1;
    SDL_Event event;
    while(active) {
        SDL_WaitEvent(&event);
        switch(event.type) {
            case SDL_QUIT : active = 0; break;
            case SDL_KEYUP : active = 0; break;
            default : break;
        }
    }
    SDL_Quit();
    exit(EXIT_SUCCESS);
}
```

## 18.1 Dessiner un rectangle

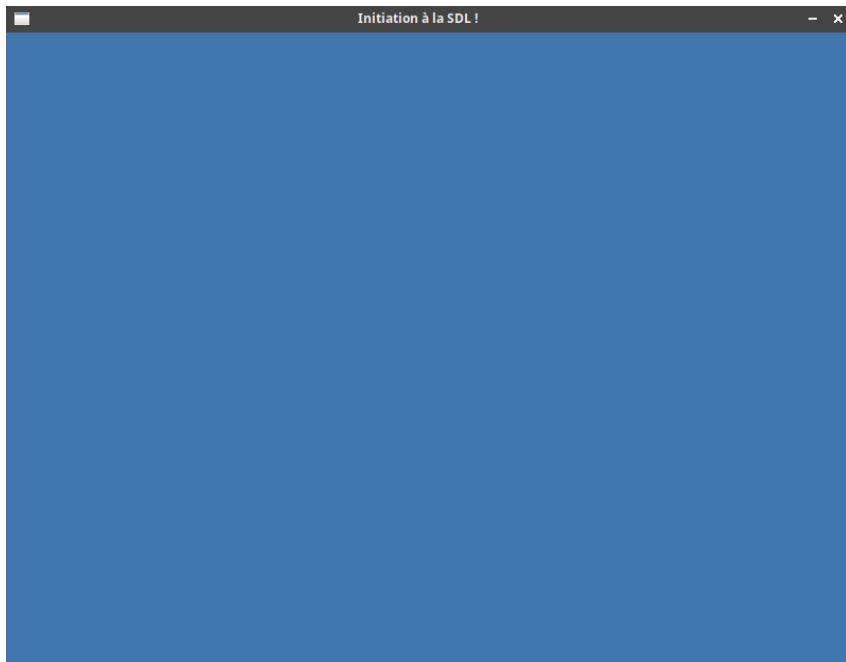
### 18.1.1 SDL FillRect

Pour changer la couleur de l'écran, nous pouvons appeler `SDL_FillRect`. Cette fonction colorise un rectangle de couleur unie sur une surface et prend les paramètres suivants :

- **Surface** sur laquelle le rectangle sera dessiné.
- **Rectangle** : positionnement et dimensions.
- **Couleur** : entier modélisant la couleur sur 32 bits.

Cette fonction peut par exemple être utilisée pour changer la couleur de l'écran :

```
SDL_FillRect(ecran, NULL, 0x4076ad);  
SDL_Flip(ecran);
```



4076ad est de l'hexadécimal qui peut se décomposer comme :

- 0x40 = 64 : taux de rouge sur 255.
- 0x76 = 118 : taux de vert sur 255.
- 0xad = 173 : taux de bleu sur 255.

Un logiciel avec édition de couleurs permet d'avoir ces informations. Notons que selon l'endianess de la machine, le codage de ce code couleur peut varier. Nous utiliserons donc en général la fonction `SDL_MapRGB`. Cette fonction prend en paramètres le format de pixel de la surface utilisée (informations relatives à la couleur) puis les codes couleurs des différents canaux. Le code précédent revient donc à la transformation suivante :

```
SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 64, 118, 173));
SDL_Flip(ecran);
```

### 18.1.2 SDL Rect

Par défaut le paramètre du rectangle utilisé précédemment est `NULL`. Ceci indique ici que le rectangle est l'intégralité de la surface. Cependant, il est possible de le paramétrer avec une structure `SDL_Rect` dont les champs sont les suivants :

- `w` : largeur du rectangle.
- `h` : hauteur du rectangle.
- `x` : abscisse du rectangle.
- `y` : ordonnée du rectangle.

Exemple d'utilisation de `SDL_Rect` pour afficher un rectangle :

```
SDL_Rect rectangle;
rectangle.x = 100;
rectangle.y = 50;
rectangle.w = 600;
rectangle.h = 500;
SDL_FillRect(ecran, &rectangle, SDL_MapRGB(ecran->format, 64, 118, 173));
SDL_Flip(ecran);
```



## 18.2 Créer une surface

### 18.2.1 SDL\_CreateRGBSurface

Jusqu'ici, la seule surface que nous ayons vue est l'écran. Cependant, il est possible de créer d'autres surfaces et les utiliser pour notre affichage. Pour créer une surface, nous utiliserons la fonction `SDL_CreateRGBSurface`. Cette fonction prend les paramètres suivants :

- **Mode de la texture :**
  - `SDL_HWSURFACE` pour définition de la surface dans la mémoire vidéo.
  - `SDL_SRCCOLORKEY` pour indiquer la présence du couleur devant être remplacée par de la transparence.
  - `SDL_SRCALPHA` pour créer un canal alpha et gérer la transparence générale.
- **Largeur.**
- **Hauteur.**
- **Bits par pixel.**
- **Masques des canaux RGBA :** rouge, vert, bleu et transparence.



Les masques des canaux de couleurs peuvent être initialisés par défaut à 0 ou gérés selon la position des bits de poids forts et poids faibles dans la machine :

```
#if SDL_BYTEORDER == SDL_BIG_ENDIAN
#define RMASK 0xff000000
#define GMASK 0x00ff0000
#define BMASK 0x0000ff00
#define AMASK 0x000000ff
#define RSHIFT 24
#define GSHIFT 16
#define BSHIFT 8
#define ASHIFT 0
#else
#define RMASK 0x000000ff
#define GMASK 0x0000ff00
#define BMASK 0x00ff0000
#define AMASK 0xff000000
#define RSHIFT 0
#define GSHIFT 8
#define BSHIFT 16
#define ASHIFT 24
#endif
```

Il est donc possible de créer une surface avec `SDL_CreateRGBSurface` par le code suivant :

```
/* creation de la surface */
SDL_Surface * surface = NULL;
if((surface = SDL_CreateRGBSurface(SDL_HWSURFACE, 400, 200, 32,
↪ RMASK, GMASK, BMASK, AMASK)) == NULL) {
    fprintf(stderr, "Error in SDL_CreateRGBSurface : %s\n",
↪ SDL_GetError());
    SDL_FreeSurface(ecran);
    SDL_Quit();
    exit(EXIT_FAILURE);
}
/* utilisation de la surface */

/* fin d'utilisation : */
SDL_FreeSurface(surface);
surface = NULL;
```

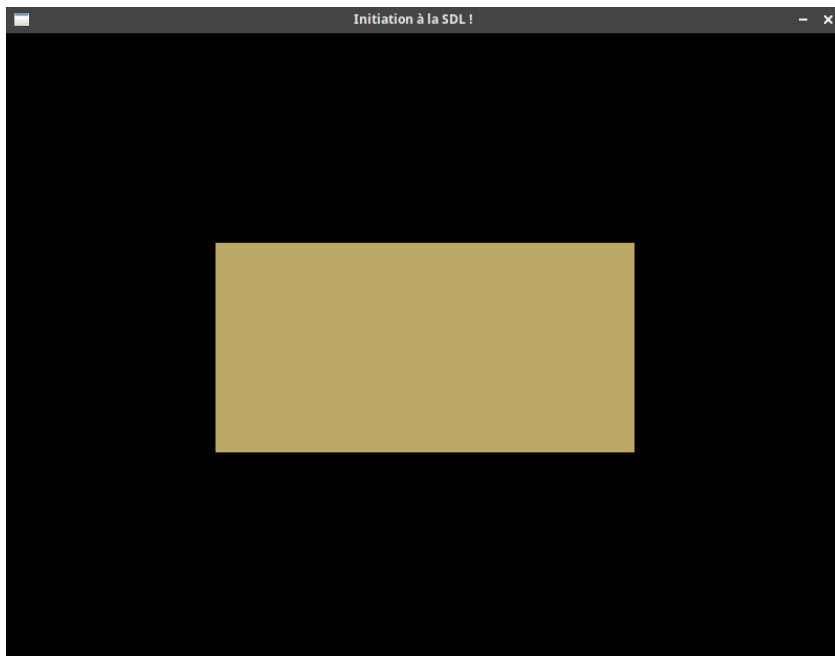
### 18.2.2 SDL BlitSurface

Lorsque l'on souhaite maintenant afficher une surface, nous pouvons utiliser `SDL_BlitSurface`. Cette fonction prend les paramètres suivants :

- **Surface source à afficher** : `SDL_Surface * source`.
- **Portion de la surface à afficher** : `SDL_Rect` (permet de sélectionner un morceau de l'image source et non toute la surface).
- **Surface de destination** : `SDL_Surface * destination`.
- **Position sur la surface de destination** : `SDL_Rect` (seuls les abscisses et ordonnées comptent).

Une utilisation de la surface précédente illustrant un appel à cette fonction pourrait être la suivante :

```
/* utilisation de la surface */
SDL_Rect position;
position.x = (ecran->w - surface->w) / 2;
position.y = (ecran->h - surface->h) / 2;
SDL_FillRect(surface, NULL, SDL_MapRGB(surface->format, 188, 169, 101));
SDL_BlitSurface(surface, NULL, ecran, &position);
SDL_Flip(ecran);
```



Ce type de surfaces peut aussi être utilisée pour pré-calculer une image créée procéduralement puis l'afficher (en plusieurs exemplaires) :

```

/* creation de la surface */
SDL_Surface * surface = NULL;
if((surface = SDL_CreateRGBSurface(SDL_HWSURFACE, 200, 200, 32,
↳ RMASK, GMASK, BMASK, AMASK)) == NULL) {
    fprintf(stderr, "Error in SDL_CreateRGBSurface : %s\n",
↳ SDL_GetError());
    SDL_FreeSurface(ecran);
    SDL_Quit();
    exit(EXIT_FAILURE);
}

/* generation de l'image procedurale */
SDL_Rect rectangle;
int i, j;
float shape = 0.5; /* 0.5 : etoile, 1 : carre, 2 : disque, 8 :
↳ carre arrondi */
float posNorm, radiusNorm;
radiusNorm = pow(abs(surface->w / 2), shape);
for(i = 0; i < surface->w; ++i) {
    for(j = 0; j < surface->h; ++j) {
        rectangle.x = i; rectangle.y = j;
        rectangle.w = 1; rectangle.h = 1;
        posNorm = pow(abs(i - surface->w / 2), shape) + pow(abs(j -
↳ surface->h / 2), shape);
        if(posNorm < radiusNorm) {
            int val = 255 * (1. - posNorm / radiusNorm);
            SDL_FillRect(surface, &rectangle,
↳ SDL_MapRGB(surface->format, val, val, val));
        }
    }
}

/* affichage de l'image */
SDL_Rect position;
position.x = (ecran->w - surface->w) / 2;
position.y = (ecran->h - surface->h) / 2;
SDL_BlitSurface(surface, NULL, ecran, &position);
position.x = (ecran->w - surface->w) - 10;
position.y = (ecran->h - surface->h) - 10;

```

```
SDL_BlendMode(SDL_BlendMode(SDL_BLENDMODE_NONE),  
position.x = 10;  
position.y = 10;  
SDL_BlendMode(SDL_BlendMode(SDL_BLENDMODE_NONE),  
SDL_Flip(ecran);  
  
/* liberation */  
SDL_FreeSurface(surface);  
surface = NULL;
```



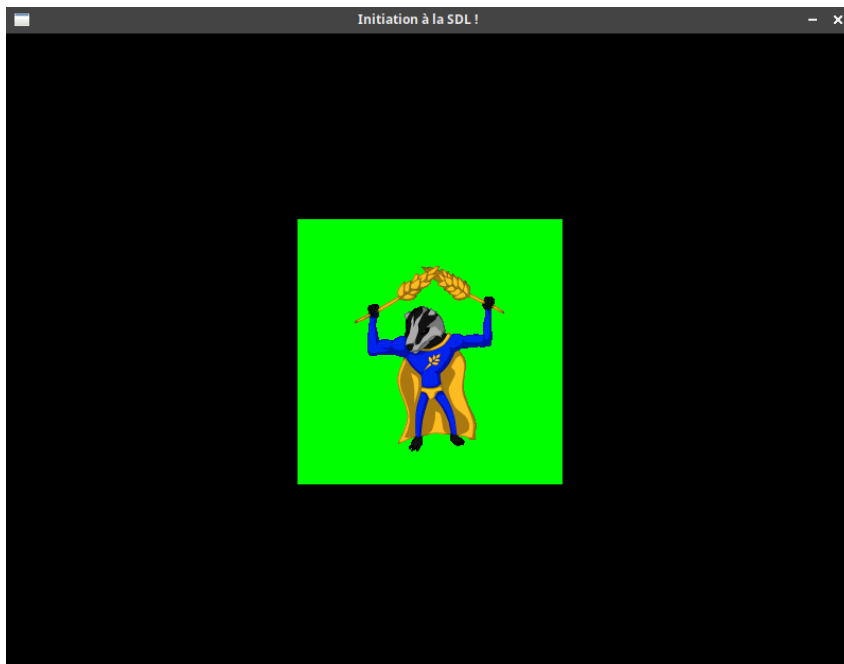
## 18.3 Images en BMP

Nous ne sommes pas obligés de générer des images procéduralement, il est aussi possible de charger des images matricielles depuis un fichier. Par défaut, SDL implémente le chargement d'images au format BMP. Celles-ci se chargent à l'aide de la fonction `SDL_LoadBMP` qui prend le chemin d'un fichier `.bmp` et renvoie une `SDL_Surface *` :

```
SDL_Surface * surface = NULL;
if((surface = SDL_LoadBMP("media/ble_heros.bmp")) == NULL) {
    fprintf(stderr, "Error in SDL_LoadBMP(\"media/ble_heros.bmp\") :
    ↪ %s\\n", SDL_GetError());
    SDL_FreeSurface(ecran);
    SDL_Quit();
    exit(EXIT_FAILURE);
}

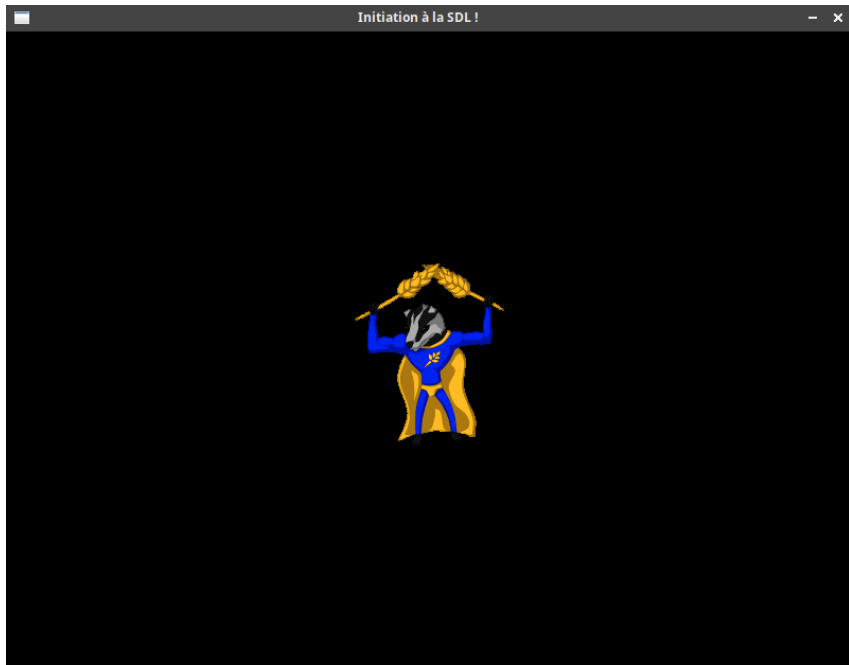
SDL_Rect position;
position.x = (ecran->w - surface->w) / 2;
position.y = (ecran->h - surface->h) / 2;
SDL_BlitSurface(surface, NULL, ecran, &position);
SDL_Flip(ecran);

SDL_FreeSurface(surface);
surface = NULL;
```



Pour ces images au format BMP, SDL propose de gérer la transparence en définissant une couleur qui peut devenir transparente à l'aide de la fonction `SDL_SetColorKey` :

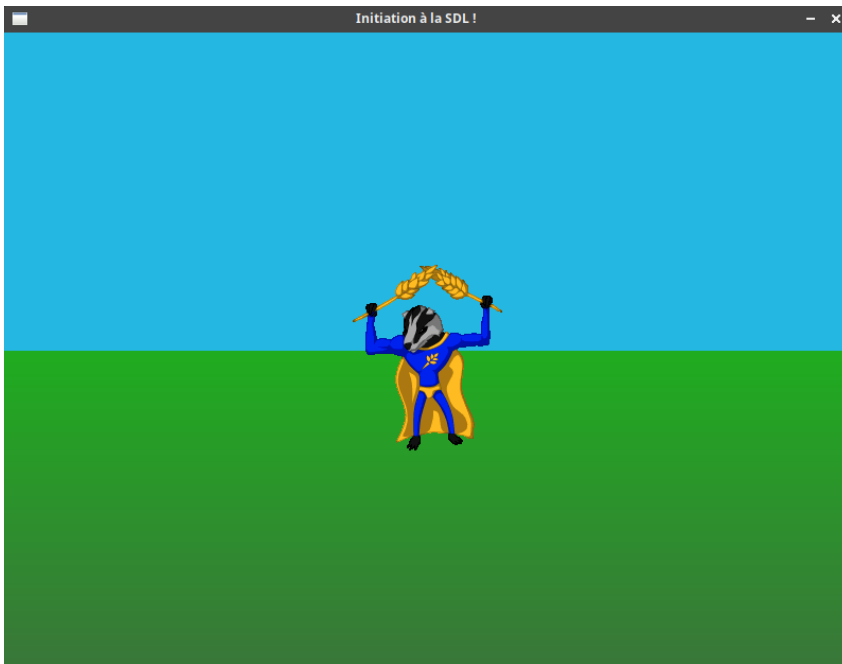
```
SDL_SetColorKey(surface, SDL_SRCCOLORKEY,  
↳ SDL_MapRGB(surface->format, 0, 255, 0));
```



Pour mieux visualiser la transparence de l'image, nous pouvons utiliser le fond suivant :

```
SDL_Rect rectangle;  
SDL_Color first = {32, 173, 32};  
SDL_Color second = {58, 118, 58};  
SDL_Color fond = {35, 183, 225};  
SDL_Color current;  
  
SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, fond.r,  
↳ fond.g, fond.b));  
int i;  
float t;  
for(i = ecran->h / 2; i < ecran->h; ++i) {  
    t = (float)(i - ecran->h / 2) / (ecran->h - ecran->h / 2);  
    current.r = (1 - t) * first.r + t * second.r;
```

```
current.g = (1 - t) * first.g + t * second.g;
current.b = (1 - t) * first.b + t * second.b;
rectangle.x = 0; rectangle.y = i;
rectangle.w = ecran->w; rectangle.h = 1;
SDL_FillRect(ecran, &rectangle, SDL_MapRGB(ecran->format,
→ current.r, current.g, current.b));
}
```



### 18.3.1 SDL Image

Il est possible d'enrichir les fonctionnalités de chargement de SDL avec le module `SDL_image`. Pour ceci, les modifications à apporter pour la compilation sont les suivantes :

- Ajout du fichier d'entête `SDL/SDL_image.h` :

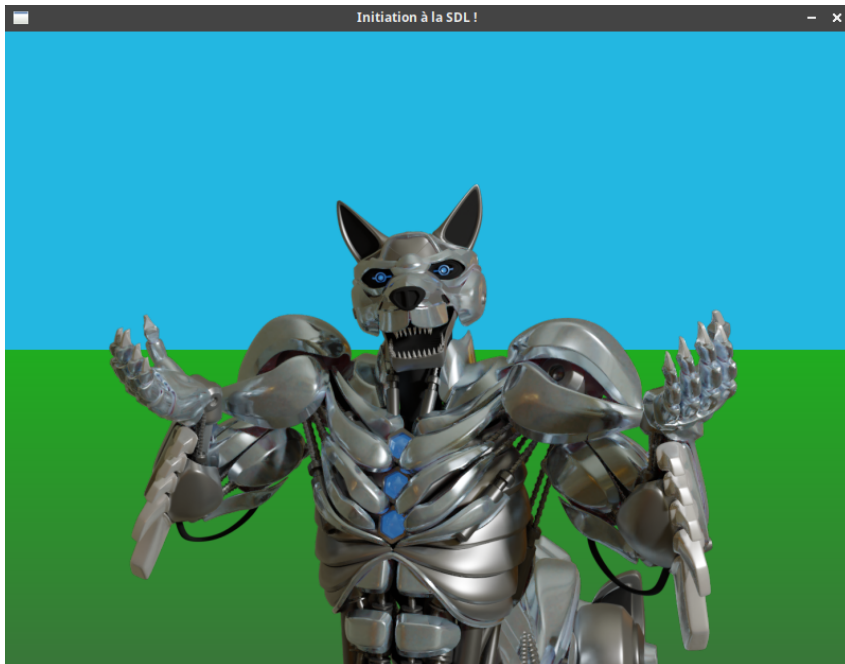
```
#include <SDL/SDL_image.h>
```

- Ajout de la bibliothèque `-lSDL_image` dans la commande de compilation :

```
gcc -o prog main.c -lSDL -lm -lSDL_image
```

Ceci permet par suite l'utilisation de `IMG_Load` prenant le chemin du fichier image en argument et renvoyant une surface :

```
SDL_Surface * surface = NULL;
if((surface = IMG_Load("media/TK.png")) == NULL) {
    fprintf(stderr, "Error in SDL_LoadBMP(\"media/TK.png\") : %s\n",
        ↪ SDL_GetError());
    SDL_FreeSurface(ecran);
    SDL_Quit();
    exit(EXIT_FAILURE);
}
```



### 18.3.2 Dessiner une sous-image

Une astuce utilisée pour des décors peut être le tilemapping. Ceci consiste à avoir les sous-images qui constitueront un décor en une image puis extraire les



différents morceaux de sous-images pour dessiner un décor dans un monde en pixel art par exemple.

Nous pouvons par exemple prendre le tileset suivant :



Pour gérer la découpe sur le tileset, nous pourrions utiliser le `SDL_Rect` proposé par `SDL_BlitSurface` pour la surface source :

```
char grille[6][8] = {
    { 1, 14, 13, 14, 13,  2,  0,  0},
    { 9, 22, 19, 20, 21, 10,  0,  0},
    { 9,  4,  1,  2,  3,  4,  0,  0},
    { 9, 16, 15,  4,  9, 16,  2,  0},
    { 7, 21,  5, 16, 15,  5, 10,  0},
    { 0,  7, 19, 20, 19, 20,  8,  0}
};

SDL_Rect tilePos;
SDL_Rect screenPos;
int i, j;
for(j = 0; j < 6; ++j) {
    for(i = 0; i < 8; ++i) {
        if(grille[j][i] == 0) continue;
        tilePos.x = 25 * ((grille[j][i] - 1) % 6);
        tilePos.y = 25 * ((grille[j][i] - 1) / 6);
        tilePos.w = 25;
        tilePos.h = 25;
        screenPos.x = i * 25;
        screenPos.y = j * 25;
        SDL_BlitSurface(surface, &tilePos, ecran, &screenPos);
    }
}
SDL_Flip(ecran);
```



## 18.4 Résumé

Inclusion de l'entête de la bibliothèque SDL :

```
#include <SDL/SDL.h>
```

Initialisation et arrêt de SDL :

```
SDL_Init(SDL_INIT_EVERYTHING);  
SDL_Quit();
```

Démarrage d'une fenêtre graphique :

```
/* Surface vidéo : */  
SDL_Surface * ecran = NULL;  
/* Lancement de la fenêtre graphique : */  
ecran = SDL_SetVideoMode(LARGEUR, HAUTEUR, 32, FLAGS);  
/* Rafraîchissement de l'écran */  
SDL_Flip(ecran);  
/* Libération de la fenêtre graphique : */  
SDL_FreeSurface(ecran);
```

Dessiner un rectangle / appliquer une couleur uniforme :

```
SDL_FillRect(SDL_Surface * surface, SDL_Rect * rectangle,  
↳ SDL_MapRGB(surface->format, couleur.rouge, couleur.vert,  
↳ couleur.bleu));
```

Création d'une surface :

```
SDL_Surface * SDL_CreateRGBSurface(FLAGS, LARGEUR, HAUTEUR, 32,  
↳ RMASK, GMASK, BMASK, AMASK);
```

Impression d'une surface sur une autre :

```
SDL_BlitSurface(SDL_Surface * source, SDL_Rect * sourceRectangle,  
↳ SDL_Surface * destination, SDL_Rect * destinationPosition);
```

Chargement d'une image BMP (par défaut avec SDL) et définition d'une couleur de transparence :

```
SDL_Surface * SDL_LoadBMP(const char * cheminFichier);  
SDL_SetColorKey(SDL_Surface * surface, SDL_SRCCOLORKEY,  
↳ SDL_MapRGB(surface->format, couleur.r, couleur.g, couleur.b));
```

Chargement d'une image avec SDL\_image :

```
#include <SDL/SDL_image.h>  
SDL_Surface * IMG_Load(const char * cheminFichier);
```

---

## 19 Événements

---

Nous avons vu précédemment quelques lignes pour maintenir une fenêtre active le temps que l'utilisateur appuie sur une touche ou ferme la fenêtre. Les événements avec SDL sont gérés par un type `SDL_Event`. Une proposition pour la gestion des événements est la suivante :

```
int active = 1; /* état du processus en cours */
SDL_Event event;
while(active) {
    /* récupération et traitement des événements */
    /* mise à jour de l'affichage */
}
```

### 19.1 Récupération d'événements

Nous avons principalement deux moyens de récupérer des événements :

- `SDL_WaitEvent` attente bloquante d'un événement.
- `SDL_PollEvent` récupération non bloquante d'événements.

#### 19.1.1 SDL WaitEvent

Avec `SDL_WaitEvent`, l'attente sera bloquant et nous ne savons pas combien de temps il faudra attendre l'événement suivant. Ceci fait que pour chaque événement, nous le traitons puis nous actualisons l'affichage en conséquence.

```
int active = 1; /* état du processus en cours */
SDL_Event event;
while(active) {
    /* récupération d'un événement : */
    SDL_WaitEvent(&event);
    /* traitement de l'événement */
    /* mise à jour de l'affichage */
}
```

### 19.1.2 SDL PollEvent

Dans le cas de `SDL_PollEvent` le traitement serait différent. Nous bouclerons sur les événements de la frame courante pour les traiter puis nous effectuerons l'actualisation de l'affichage. À noter qu'utiliser `SDL_PollEvent` est non-bloquant et donc ceci utiliserait votre CPU inutilement. Une proposition est d'utiliser `SDL_GetTicks` pour récupérer le temps en seconde écoulé depuis le lancement du programme et `SDL_Delay` pour mettre en attente la prochaine vérification des événements si le traitement de ceux-ci a été terminé plus vite qu'un temps réglé. On peut régler ce temps de manière à essayer de conserver un nombre d'image par secondes donné :

```
int active = 1; /* état du processus en cours */
int delay; /* gestion du temps de rafraîchissement */
const int FPS = 60; /* nombre d'images par seconde souhaité */
SDL_Event event;
while(active) {
    delay = SDL_GetTicks(); /* récupération du temps */
    /* récupération d'un événement : */
    while(SDL_PollEvent(&event)) {
        /* traitement de l'événement */
    }
    /* mise à jour de l'affichage */
    /* calcul d'un temps avec fin de boucle si besoin : */
    delay = (1000. / FPS) - (SDL_GetTicks() - delay);
    /* attente si nécessaire : */
    if(delay > 0) {
        SDL_Delay(delay);
    }
}
```

## 19.2 Analyse d'événements

### 19.2.1 Types d'événements

Lorsqu'un événement est récupéré, celui-ci possède un type. Les principaux différents types d'événements sont les suivants :

- `SDL_QUIT` : lorsque l'utilisateur clique sur la croix pour fermer la fenêtre.
- `SDL_KEYDOWN` : enfoncement d'une touche du clavier.
- `SDL_KEYUP` : relâchement d'une touche du clavier.

- `SDL_MOUSEBUTTONDOWN` : enfoncement d'un bouton de la souris.
- `SDL_MOUSEBUTTONUP` : relâchement d'un bouton de la souris.
- `SDL_MOUSEMOTION` : déplacement de la souris.

En général, un `switch` est une bonne idée pour gérer les événements :

```
switch(event.type) {
  case SDL_QUIT : {
    /* fin du processus */
    active = 0;
  } break;
  case SDL_KEYDOWN : {
    /* gestion événement touche enfoncée */
  } break;
  case SDL_KEYUP : {
    /* gestion événement touche relâchée */
  } break;
  case SDL_MOUSEBUTTONDOWN : {
    /* gestion événement bouton enfoncé */
  } break;
  case SDL_MOUSEBUTTONUP : {
    /* gestion événement bouton relâché */
  } break;
  case SDL_MOUSEMOTION : {
    /* gestion événement déplacement souris */
  } break;
  default : break;
}
```

## 19.2.2 Clavier

### Clés

Pour récupérer la touche du clavier associée à l'événement, nous accèderons au champ `key.keysym.sym` de l'événement. Les principales clés nommées sont les suivantes :

- `SDLK_ESCAPE` : touche 'Échap'.
- `SDLK_SPACE` : touche espace.
- `SDLK_RETURN` : touche entrée.
- `SDLK_BACKSPACE` : touche d'effacement du caractère précédent.

- `SDLK_DELETE` : touche 'Suppr'.
- `SDLK_0`, ..., `SDLK_9` touches numériques.
- `SDLK_a`, ..., `SDLK_z` touches alphabétiques.
- `SDLK_KP0`, ..., `SDLK_KP9` touches du keypad.
- `SDLK_UP`, `SDLK_DOWN`, `SDLK_LEFT` et `SDLK_RIGHT` flèches directionnelles.
- `SDLK_LSHIFT` et `SDLK_RSHIFT` shift.
- `SDLK_LCTRL` et `SDLK_RCTRL` 'Ctrl'.
- `SDLK_LALT` et `SDLK_RALT` 'Alt'.

Pour d'avantage de touches, il est possible de consulter la [documentation à ce sujet](#).

Le traitement des différentes touches peut ensuite se faire dans un `switch` imbriqué dans celui concernant l'état de pression de la touche :

```
switch(event.type) {
  case SDLK_QUIT : {
    /* fin du processus */
    active = 0;
  } break;
  case SDLK_KEYDOWN : {
    switch(event.key.keysym.sym) {
      case SDLK_ESCAPE : {
        active = 0;
      } break;
      /* ... */
      default : break;
    }
  } break;
  case SDLK_KEYUP : {
    /* gestion événement touche relâchée */
  } break;
  default : break;
}
```

### Activation de répétition de prise en compte

Il est possible d'activer la répétition d'une clé enfoncée pour un `SDL_KEYDOWN`. Ceci se fait par la fonction `SDL_EnableKeyRepeat`. Cette fonction prend en paramètres un délai pendant lequel la clé doit être activée pour répétition et un intervalle auquel elle sera à nouveau répétée. On place cette fonction avant la



boucle principale. Des valeurs par défaut sont fournies pour l'utilisation de cette fonction :

```
SDL_EnableKeyRepeat(SDL_DEFAULT_REPEAT_DELAY,  
↳ SDL_DEFAULT_REPEAT_INTERVAL);
```

### 19.2.3 Boutons de souris

Lorsque l'événement est de type `SDL_MOUSEBUTTONDOWN` ou `SDL_MOUSEBUTTONUP`, il est possible de récupérer le bouton enfoncé sur la souris par l'accès à `event.button.button`. Les principales valeurs sont les suivantes :

- `SDL_BUTTON_LEFT` : clic gauche.
- `SDL_BUTTON_RIGHT` : clic droit.
- `SDL_BUTTON_MIDDLE` : bouton du milieu.
- `SDL_BUTTON_WHEELUP` : molette de la souris vers l'avant.
- `SDL_BUTTON_WHEELDOWN` : molette de la souris vers l'arrière.

### 19.2.4 Coordonnées de souris

Il est ensuite possible de récupérer les coordonnées de la souris lors d'une événement de type `SDL_MOUSEBUTTONDOWN` ou `SDL_MOUSEBUTTONUP` par les appels à `event.button.x` et `event.button.y` :

```
switch(event.type) {  
    case SDL_QUIT : active = 0; break;  
    case SDL_MOUSEBUTTONDOWN : {  
        switch(event.button.button) {  
            case SDL_BUTTON_LEFT : {  
                printf("clic gauche : (%d, %d)\n", event.button.x,  
↳ event.button.y);  
            } break;  
            default : break;  
        }  
    } break;  
    default : break;  
}
```

À noter que par exemple dans le cas d'un clic qui aurait été enfoncé pour déplacer un élément à l'écran, nous ne pouvons plus gérer la position de la souris dans `SDL_MOUSEBUTTONDOWN`. La souris peut être gérée lorsque déplacée par un

événement de type `SDL_MOUSEMOTION`. Pour récupérer les coordonnées de la souris, ceci demandera d'accéder aux champs `event.motion.x` et `event.motion.y`.

```
int active = 1;
SDL_Event event;
int grab = 0;
int gx, gy;
int update = 0;
while(active) {
    SDL_WaitEvent(&event);
    switch(event.type) {
        case SDL_QUIT : active = 0; break;
        case SDL_KEYDOWN : active = 0; break;
        case SDL_MOUSEBUTTONDOWN : {
            if(event.button.button == SDL_BUTTON_LEFT) {
                gx = event.button.x;
                gy = event.button.y;
                grab = 1;
                update = 1;
            }
        } break;
        case SDL_MOUSEBUTTONUP : {
            if(event.button.button == SDL_BUTTON_LEFT) {
                grab = 0;
                update = 1;
            }
        } break;
        case SDL_MOUSEMOTION : {
            if(grab) {
                gx = event.motion.x;
                gy = event.motion.y;
                update = 1;
            }
        } break;
        default : break;
    }
    if(update && grab) {
        printf("mouse grab active : (%d, %d)\n", gx, gy);
    }
    update = 0;
}
```

Imaginons que vous souhaitez que votre souris fasse tourner une caméra dans un FPS. Il est possible de récupérer le déplacement relatif de la souris par `event.motion.xrel` et `event.motion.yrel`. Vous pouvez ensuite replacer la souris au centre de l'écran avec `SDL_WarpMouse` :

```
int active = 1;
SDL_Event event;
int gx, gy;
int update = 0;
while(active) {
    SDL_WaitEvent(&event);
    switch(event.type) {
        case SDL_QUIT : active = 0; break;
        case SDL_KEYDOWN : active = 0; break;
        case SDL_MOUSEMOTION : {
            gx = event.motion.xrel;
            gy = event.motion.yrel;
            SDL_WarpMouse(ecran->w / 2, ecran->h / 2);
            update = 1;
        } break;
        default : break;
    }
    if(update) {
        printf("mouse moving : (%d, %d)\n", gx, gy);
    }
    update = 0;
}
```

## 19.3 Résumé

Récupération d'événements :

```
/* variable événement : */
SDL_Event event;
/* récupération bloquante : */
SDL_WaitEvent(&event);
/* récupération non bloquante : */
SDL_PollEvent(&event);
```

Analyse d'événements :

```
/* type de l'événement : */
event.type
/* touche clavier concernée : */
event.key.keysym.sym
/* bouton souris concerné : */
event.button.button
/* coordonnées position souris */
event.button.x
event.button.y
/* coordonnées position souris */
event.motion.x
event.motion.y
/* déplacement relatif souris */
event.motion.xrel
event.motion.yrel
```

Types d'événements :

```
SDL_QUIT /* croix fenêtre */
SDL_KEYDOWN /* enfoncement d'une touche du clavier */
SDL_KEYUP /* relâchement d'une touche du clavier */
SDL_MOUSEBUTTONDOWN /* enfoncement d'un bouton de la souris */
SDL_MOUSEBUTTONUP /* relâchement d'un bouton de la souris */
SDL_MOUSEMOTION /* déplacement de la souris */
```

Pour plus de détails sur les clés du clavier, se référer à la [documentation](#).

Boutons de la souris :

```
SDL_BUTTON_LEFT /* clic gauche */  
SDL_BUTTON_RIGHT /* clic droit */  
SDL_BUTTON_MIDDLE /* bouton du milieu */  
SDL_BUTTON_WHEELUP /* molette vers l'avant */  
SDL_BUTTON_WHEELDOWN /* molette vers l'arrière */
```



Sixième partie

Examens de l'année  
précédente





# Table des matières

<b>A</b>	<b>Données statistiques partiels 2021 - 2022</b>	<b>347</b>
A.1	Semestre 1 . . . . .	348
A.1.1	Alternance . . . . .	348
A.1.2	Janvier . . . . .	348
A.2	Semestre 2 . . . . .	349
A.2.1	Alternance . . . . .	349
A.2.2	Janvier . . . . .	349
<b>B</b>	<b>Sujets partiels 2021 - 2022</b>	<b>351</b>
B.1	Semestre 1 - Alternance . . . . .	352
B.2	Semestre 1 - Janvier . . . . .	358
B.3	Semestre 2 - Sujet zéro . . . . .	364
B.4	Semestre 2 - Alternance . . . . .	370
B.5	Semestre 2 - Janvier . . . . .	376



---

# A Données statistiques partiels 2021 - 2022

---

Pour vous permettre d'appréhender au mieux les examens terminaux des semestres 1 et 2, vous retrouverez ici des informations sur leur déroulé les années précédentes :

- Des statistiques sur les résultats obtenus par les candidats sur l'année précédente.
- Les sujets sur lesquels les candidats ont été évalué pour vous tester en amont de l'examen.

Vous pouvez vous référer aux conseils promulgués pour réussir ces examens en section [2.1](#).

Les données suivantes représentent les notes obtenues aux examens finaux correspondants aux sujets donnés en section [B](#).

---

## A.1 Semestre 1

### A.1.1 Alternance

91 étudiants présents

Moyenne :	14.7802
Écart type :	4.56649

Minimale :	0
Premier quartile :	12
<b>Médiane :</b>	16
Troisième quartile :	18
Maximale :	20

### A.1.2 Janvier

38 étudiants présents

Moyenne :	13.6316
Écart type :	4.51559

Minimale :	3
Premier quartile :	10
<b>Médiane :</b>	14
Troisième quartile :	17
Maximale :	20

---

## A.2 Semestre 2

### A.2.1 Alternance

72 étudiants présents

Moyenne :	9.77778
Écart type :	4.80515

Minimale :	0
Premier quartile :	6
<b>Médiane :</b>	10
Troisième quartile :	13
Maximale :	20

### A.2.2 Janvier

36 étudiants présents

Moyenne :	7
Écart type :	4.40328

Minimale :	0
Premier quartile :	4
<b>Médiane :</b>	7
Troisième quartile :	9
Maximale :	17

---

---

## B Sujets partiels 2021 - 2022

---

Les sujets donnés ici sont les sujets tombés à l'examen de fin de semestre l'année 2021 - 2022 pour les formations en alternance et la formation en initial entrée en Janvier.

## B.1 Semestre 1 - Alternance

Note

..... / 20

Prénom NOM : .....

Exercice 1	Exercice 2	Exercice 3	Exercice 4	Exercice 5	Exercice 6
..... / 3	..... / 3	..... / 3	..... / 4	..... / 5	..... / 2

## Modalités

- L'étudiant a 2 heures pour composer sur ce sujet et rendre une copie papier de sa production.
- Les 6 exercices peuvent être traités indépendamment les uns des autres.
- Chaque exercice doit être représenté par un code source en langage C tel qu'il pourrait être fourni à un compilateur.
- Les attendus à utiliser pour réussir les exercices ne dépassent pas les notions étudiées dans le support de cours jusqu'à la section 4 (structures conditionnelles) : l'utilisation de notions externes au cours ne donne pas de points supplémentaires et n'est pas nécessaire (au risque de se complexifier une tâche qui peut être simple).
- Le barème utilisé vérifiera et valorisera des éléments de code (entrées, sorties, logique conditionnelle, validité d'opérations, inclusions et autres éléments pertinents liés au langage C et à la logique du code proposé).
- Les exercices sont ordonnés par ordre croissant en fonction des notions vues en classe et le temps / complexité qui peut être requis pour atteindre les objectifs de chaque exercice.
- Des exemples de sorties peuvent être donnés pour aider à comprendre la logique attendue par l'exercice.

Bon courage !



## Exercice 1 (..... / 3 points)

**Objectif :** adapter un code en langage C.

Un élève débutant en langage C essaie d'affecter 42 à un entier puis de l'afficher. Le compilateur refuse de compiler son code. Adapter le code suivant en langage C pour qu'il soit fonctionnel :

```
include <<studio>>
main :
    entier = 42
    print(entier)
```

## Exercice 2 (..... / 3 points)

**Objectif :** vérifier un mot de passe.

- **Entrée 1 :** un entier (nombre)

Lire un entier, selon sa valeur :

- S'il vaut 1235, afficher un accès autorisé.
- Sinon, afficher un accès refusé.

**Exemple de sortie :**  
Accès autorisé.

```
Entrez un mot de passe : 1235
Accès autorisé !
```

**Exemple de sortie :**  
Accès refusé.

```
Entrez un mot de passe : 0420
Accès refusé !
```

## Exercice 3 (..... / 3 points)

**Objectif :** lire un entier et visualiser un dépassement de capacité.

- **Entrée 1 :** un entier sur 8 octets (nombre)

Lire un entier sur 8 octets puis afficher sa visualisation :

- décimale sur 4 octets.
- décimale sur 8 octets.
- hexadécimale sur 4 octets (sur 8 caractères, complés par des zéros).
- hexadécimale sur 8 octets (sur 16 caractères, complés par des zéros).

**Exemple de sortie :**

Petite valeur.

```
Entrez un entier : 42
Decimal sur 4 octets : 42
Decimal sur 8 octets : 42
Hexadecimal sur 4 octets : 0000002a
Hexadecimal sur 8 octets : 000000000000002a
```

**Exemple de sortie :**

Négative en 4 octets.

```
Entrez un entier : 4000000000
Decimal sur 4 octets : -294967296
Decimal sur 8 octets : 4000000000
Hexadecimal sur 4 octets : ee6b2800
Hexadecimal sur 8 octets : 00000000ee6b2800
```

**Exemple de sortie :**

Dépassement de capacité du 8 octets.

```
Entrez un entier : 80000000000000
Decimal sur 4 octets : 1939144704
Decimal sur 8 octets : 80000000000000
Hexadecimal sur 4 octets : 73950000
Hexadecimal sur 8 octets : 000048c273950000
```

## Exercice 4 (..... / 4 points)

**Objectif :** déterminer un minimum et un maximum.

- **Entrée 1 :** un entier (nombre 1)
- **Entrée 2 :** un entier (nombre 2)
- **Entrée 3 :** un entier (nombre 3)

Lire trois nombres entiers puis :

- Afficher le minimum des trois nombres.
- Afficher le maximum des trois nombres.

**Exemple de sortie :**

Ordre croissant.

```
Entrez 3 entiers : 1 2 3
min(1, 2, 3) = 1
max(1, 2, 3) = 3
```

**Exemple de sortie :**

Ordre décroissant.

```
Entrez 3 entiers : 3 2 1
min(3, 2, 1) = 1
max(3, 2, 1) = 3
```

**Exemple de sortie :**

Ordre quelconque.

```
Entrez 3 entiers : 5 7 2
min(5, 7, 2) = 2
max(5, 7, 2) = 7
```

## Exercice 5 (..... / 5 points)

**Objectif :** calculatrice d'entiers.

- **Entrée 1** : un entier (nombre 1)
- **Entrée 2** : un caractère (opérateur)
- **Entrée 3** : un entier (nombre 2)

Lire un nombre, un opérateur et un autre nombre puis afficher le calcul pour les opérations :

- '+' pour l'addition d'entiers.
- '-' pour la soustraction d'entiers.
- '\*' pour la multiplication d'entiers.
- '/' pour la division fractionnaire (à virgule).
- 'd' pour la division entière (quotient de la division euclidienne).
- 'm' pour le modulo (reste de la division euclidienne).

**Exemple de sortie :**  
Addition.

```
>>> 5 + 4
= 9
```

**Exemple de sortie :**  
Multiplication.

```
>>> 100000000 * 10000000
= 1000000000000000
```

**Exemple de sortie :**  
Division fractionnaire.

```
>>> 7 / 5
= 1.4
```

**Exemple de sortie :**  
Division entière.

```
>>> 7 d 5
= 1
```

**Exemple de sortie :**  
Modulo.

```
>>> 7 m 5
= 2
```

**Exemple de sortie :**  
Gestion division par zéro.

```
>>> 7 / 0
= inf
```

## Exercice 6 (..... / 2 points)

**Objectif :** déterminer hauteur et arrivé d'un lancer de projectile.

- **Entrée 1 :** un réel (coordonnée position initiale abscisses x)
- **Entrée 2 :** un réel (coordonnée position initiale ordonnées y)
- **Entrée 3 :** un réel (coordonnée vecteur vitesse initiale abscisses x)
- **Entrée 4 :** un réel (coordonnée vecteur vitesse initiale ordonnées y)

La trajectoire d'un projectile à un temps  $t$  donné est exprimée en fonction d'une position initiale  $(P_x, P_y)$ , d'un vecteur de vitesse initiale  $(V_x, V_y)$  et de la valeur de pesanteur  $g \simeq 9.81$  par la relation :

$$\begin{cases} x &= V_x \cdot t + P_x \\ y &= -\frac{1}{2}g \cdot t^2 + V_y \cdot t + P_y \end{cases}$$

Le projectile atteint sa hauteur maximale pour :

$$t_{max} = \frac{V_y}{g}$$

et touche le sol pour :

$$t_{impact} = \frac{V_y + \sqrt{V_y^2 + 2g \cdot P_y}}{g}$$

Dans votre programme, indiquez depuis la position initiale et le vecteur de vitesse initiale :

- La relation qui guide la trajectoire.
- La position et temps où la hauteur maximale est atteinte.
- La position et temps où le projectile touche le sol.

**Exemple de sortie :**

Depuis la position (1,2) avec pour vitesse initiale (8,4).

```
Entrez les coordonnées de départ du projectile : 1 2
Entrez le vecteur vitesse initial : 8 4
Le projectile va suivre la trajectoire en fonction du temps t par :
x = 8 t + 1
y = -4.905 t^2 + 4 t + 2
Arrive à hauteur maximale à t = 0.407747 et coordonnées (4.26198, 2.81549)
Arrive au sol à t = 1.16538 et coordonnées (10.323, 0)
```

## B.2 Semestre 1 - Janvier

Note

..... / 20

Prénom NOM : .....

Exercice 1	Exercice 2	Exercice 3	Exercice 4	Exercice 5
..... / 3	..... / 3	..... / 4	..... / 5	..... / 5

## Modalités

- L'étudiant a 1 heure et 30 minutes pour composer sur ce sujet et rendre une copie papier de sa production.
- Les 5 exercices peuvent être traités indépendamment les uns des autres.
- Chaque exercice doit être représenté par un code source en **langage C** tel qu'il pourrait être fourni à un compilateur.
- Les attendus à utiliser pour réussir les exercices ne dépassent pas les notions étudiées dans le support de cours jusqu'à la section 4 (structures conditionnelles) : l'utilisation de notions externes au cours ne donne pas de points supplémentaires et n'est pas nécessaire (au risque de se complexifier une tâche qui peut être simple).
- Le barème utilisé vérifiera et valorisera des éléments de code (entrées, sorties, logique conditionnelle, validité d'opérations, inclusions et autres éléments pertinents liés au langage C et à la logique du code proposé).
- Les exercices sont ordonnés par ordre croissant en fonction des notions vues en classe et le temps / complexité qui peut être requis pour atteindre les objectifs de chaque exercice.
- Des exemples de sorties peuvent être donnés pour aider à comprendre la logique attendue par l'exercice.
- Le sujet est à rendre avec la copie pour notation.

Bon courage !

## Exercice 1 (..... / 3 points)

**Objectif :** adapter un code en langage C.

Un élève débutant en langage C essaie de faire saisir un entier à l'utilisateur puis de l'afficher. Le compilateur refuse de compiler son code. Adapter le code suivant en langage C pour qu'il soit fonctionnel :

```
import studio

Sub prog
    read "Entrez un entier : ", entier
    print "Voici un entier : ", entier
End Sub
```

## Exercice 2 (..... / 3 points)

**Objectif :** donner le plus grand nombre.

- **Entrée 1 :** un entier (nombre)
- **Entrée 2 :** un entier (nombre)

Lire deux entiers, afficher la valeur du plus grand.

**Exemple de sortie :**  
Premier plus grand.

```
Saisir deux entiers : 1337 111
1337 est le plus grand.
```

**Exemple de sortie :**  
Second plus grand.

```
Saisir deux entiers : 42 1235
1235 est le plus grand.
```

## Exercice 3 (..... / 4 points)

**Objectif :** sommer des éléments sous condition.

- **Entrée 1 :** un entier (nombre 1)
- **Entrée 2 :** un entier (nombre 2)
- **Entrée 3 :** un entier (nombre 3)

Lire trois nombres entiers puis :

- Afficher la somme des nombres pairs.
- Afficher la somme des nombres impairs.

**Exemple de sortie :**  
Impairs uniquement.

```
Entrez 3 nombres : 1 3 5
Somme des éléments pairs : 0
Somme des éléments impairs : 9
```

**Exemple de sortie :**  
Pairs uniquement.

```
Entrez 3 nombres : 2 4 8
Somme des éléments pairs : 14
Somme des éléments impairs : 0
```

**Exemple de sortie :**  
Pairs et impairs mélangés.

```
Entrez 3 nombres : 7 10 3
Somme des éléments pairs : 10
Somme des éléments impairs : 10
```



## Exercice 4 (..... / 5 points)

**Objectif :** calculatrice d'entiers.

- **Entrée 1 :** un entier (nombre 1)
- **Entrée 2 :** un caractère (opérateur)
- **Entrée 3 :** un entier (nombre 2)

Lire un nombre, un opérateur et un autre nombre puis afficher le calcul pour les opérations :

- '+' pour l'addition d'entiers.
- '-' pour la soustraction d'entiers.
- '\*' pour la multiplication d'entiers.
- '/' pour la division fractionnaire (à virgule).
- 'd' pour la division entière (quotient de la division euclidienne).
- 'm' pour le modulo (reste de la division euclidienne).

**Exemple de sortie :**  
Addition.

```
>>> 5 + 4
= 9
```

**Exemple de sortie :**  
Multiplication.

```
>>> 100000000 * 10000000
= 1000000000000000
```

**Exemple de sortie :**  
Division fractionnaire.

```
>>> 7 / 5
= 1.4
```

**Exemple de sortie :**  
Division entière.

```
>>> 7 d 5
= 1
```

**Exemple de sortie :**  
Modulo.

```
>>> 7 m 5
= 2
```

**Exemple de sortie :**  
Gestion division par zéro.

```
>>> 7 / 0
= inf
```

## Exercice 5 (..... / 5 points)

**Objectif :** lire un entier et visualiser un dépassement de capacité.

- **Entrée 1 :** un entier sur 8 octets (nombre)

Lire un entier sur 8 octets puis afficher sa visualisation :

- hexadécimale et décimale sur 8 octets.
- hexadécimale et décimale sur 4 octets.
- hexadécimale et décimale sur 2 octets.
- hexadécimale et décimale sur 1 octets.
- découpée en hexadécimal (comblée automatiquement de zéros et séparée par des tirets-bas) :
  - 4 premiers octets (octets de poids fort d'un entier sur 8 octets)
  - 2 octets suivants (octets de poids fort d'un entier sur 4 octets)
  - 1 octet suivant (octets de poids fort d'un entier sur 2 octets)
  - 1 octet suivant (octets d'un entier sur 1 octet)

**Exemple de sortie :**

Petite valeur.

```
Entrez en entier : 42
entier sur 8 octets :          2a (42)
entier sur 4 octets :          2a (42)
entier sur 2 octets :          2a (42)
entier sur 1 octet :           2a (42)
découpe : 00000000_0000_00_2a
```

**Exemple de sortie :**

Limite sur 4 octets.

```
Entrez en entier : 4000000000
entier sur 8 octets :          ee6b2800 (4000000000)
entier sur 4 octets :          ee6b2800 (4000000000)
entier sur 2 octets :           2800 (10240)
entier sur 1 octet :            0 (0)
découpe : 00000000_ee6b_28_00
```

**Exemple de sortie :**

Dépassement de capacité du 8 octets.

```
Entrez en entier : 8000000000000000
entier sur 8 octets : 48c273950000 (8000000000000000)
entier sur 4 octets : 73950000 (1939144704)
entier sur 2 octets : 0 (0)
entier sur 1 octet : 0 (0)
découpe : 000048c2_7395_00_00
```

## B.3 Semestre 2 - Sujet zéro

Note

..... / 20

Prénom NOM : .....

Exercice 1	Exercice 2	Exercice 3	Exercice 4	Exercice 5
..... / 4	..... / 3	..... / 4	..... / 5	..... / 4

### Modalités

- L'étudiant a 1 heure et 30 minutes pour composer sur ce sujet et rendre une copie papier de sa production.
- Les 5 exercices peuvent être traités indépendamment les uns des autres.
- Chaque exercice doit être représenté par un code source en **langage C** tel qu'il pourrait être fourni à un compilateur.
- Les attendus à utiliser pour réussir les exercices ne dépassent pas les notions étudiées dans le support de cours jusqu'à la section 8 (pointeurs) : l'utilisation de notions externes au cours ne donne pas de points supplémentaires et n'est pas nécessaire (au risque de se complexifier une tâche qui peut être simple).
- L'évaluation vérifiera et valorisera des éléments de code (entrées, sorties, logique conditionnelle, validité d'opérations, inclusions et autres éléments pertinents liés au langage C et à la logique du code proposé).
- Les exercices sont ordonnés par ordre croissant en fonction des notions vues en classe et le temps / complexité qui peut être requis pour atteindre les objectifs de chaque exercice.
- Des exemples de sorties peuvent être donnés pour aider à comprendre la logique attendue par l'exercice.
- Le sujet est à rendre avec la copie pour notation.

Bon courage !

## Exercice 1 (..... / 4 points)

**Objectif :** déterminer minimum et maximum d'une suite d'entiers.

- **Entrée 1 :** séquence d'entiers non nuls
- **Entrée 2 :** entier nul (marqueur de fin)

Lire un nombre non déterminé d'entiers non nuls, afficher la valeur du plus petit et du plus grand.

Un entier nul marque la fin de la séquence.

**Exemple de sortie :**  
Entiers positifs.

```
Entrez des valeurs non nulles : 1 2 3 4 5 0
min = 1
max = 5
```

**Exemple de sortie :**  
Petits entiers.

```
Entrez des valeurs non nulles : -1 1 -2 2 0
min = -2
max = 2
```

**Exemple de sortie :**  
Grands entiers.

```
Entrez des valeurs non nulles : 2000000000 -2000000000 0
min = -2000000000
max = 2000000000
```

**Exemple de sortie :**  
Entiers négatifs.

```
Entrez des valeurs non nulles : -100 -10 0
min = -100
max = -10
```

## Exercice 2 (..... / 3 points)

**Objectif :** Calculer une suite numérique.

La suite de Pell est donnée par la relation suivante :

$$\mathcal{P}_n = \begin{cases} 0 & \text{Si } n = 0 \\ 1 & \text{Si } n = 1 \\ 2 \times \mathcal{P}_{n-1} + \mathcal{P}_{n-2} & \text{Sinon} \end{cases}$$

Votre objectif est d'afficher les 10 premiers éléments de la suite de Pell :

**Exemple de sortie :**

```
[0, 1, 2, 5, 12, 29, 70, 169, 408, 985]
```

## Exercice 3 (..... / 4 points)

**Objectif :** manipuler des tableaux.

Un collègue a commencé le code commun suivant et vous laisse terminer l'implémentation des fonctionnalités.

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

/* TODO : afficher une liste */
void afficherListe(int liste[]);

/* TODO : concaténer deux listes */
void listeCat(int res[], const int first[], const int second[]);

/* TODO : multiplier chaque élément de la première liste avec chaque
↪ élément de la seconde */
void listeProduct(int res[], const int first[], const int second[]);
```

```
int main() {  
    int liste1[] = {2, 4, 6, 8, -1};  
    int liste2[] = {4, 5, 6, 7, 8, -1};  
    int resCat[10];  
    int resProd[21];  
    printf("liste 1 : ");  
    afficherListe(liste1);  
    printf("liste 2 : ");  
    afficherListe(liste2);  
    listeCat(resCat, liste1, liste2);  
    printf("liste cat : ");  
    afficherListe(resCat);  
    listeProduct(resProd, liste1, liste2);  
    printf("liste prod : ");  
    afficherListe(resProd);  
    exit(EXIT_SUCCESS);  
}
```

Il s'attend à ce que le résultat soit le suivant :

```
liste 1 : [2, 4, 6, 8]  
liste 2 : [4, 5, 6, 7, 8]  
liste cat : [2, 4, 6, 8, 4, 5, 6, 7, 8]  
liste prod : [8, 10, 12, 14, 16, 16, 20, 24, 28, 32, 24, 30, 36, 42, 48,  
→ 32, 40, 48, 56, 64]
```

## Exercice 4 (..... / 5 points)

**Objectif :** concaténer des chaînes de caractères.

Un élève débutant en langage C a vu qu'il peut être simple de concaténer deux chaînes de caractères avec d'autres langages de programmation.

Cependant le compilateur C refuse de le laisser additionner deux chaînes et il ne comprend pas l'erreur :

```
exo4_err.c: In function 'spaceCat':
exo4_err.c:8:15: error: invalid operands to binary + (have 'const
↳ char *' and 'const char *')
   return first + second;
               ^
```

Expliquer la problématique.

Puis, adapter le code suivant en langage C pour qu'il soit fonctionnel :

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

char * spaceCat(
    const char * first,
    const char * second) {
    first += ' ';
    return first + second;
}

int main() {
    printf("%s !\n", spaceCat("Hello", "ESGI"));
    exit(EXIT_SUCCESS);
}
```



## Exercice 5 (..... / 4 points)

**Objectif :** afficher un nombre en bases de 2 à 16.

- **Entrée :** un entier (nombre)

Lire un nombre, puis l'afficher dans les base allant de la base 2 à la base 16.

**Exemple de sortie :**

1.

```
Entrez une valeur : 1
(1)_2
(1)_3
(1)_4
(1)_5
(1)_6
(1)_7
(1)_8
(1)_9
(1)_10
(1)_11
(1)_12
(1)_13
(1)_14
(1)_15
(1)_16
```

**Exemple de sortie :**

16.

```
Entrez une valeur : 16
(10000)_2
(121)_3
(100)_4
(31)_5
(24)_6
(22)_7
(20)_8
(17)_9
(16)_10
(15)_11
(14)_12
(13)_13
(12)_14
(11)_15
(10)_16
```

**Exemple de sortie :**

42.

```
Entrez une valeur : 42
(101010)_2
(1120)_3
(222)_4
(132)_5
(110)_6
(60)_7
(52)_8
(46)_9
(42)_10
(39)_11
(36)_12
(33)_13
(30)_14
(2c)_15
(2a)_16
```

**Exemple de sortie :**

8000000000.

```
Entrez une valeur : 8000000000
(11101110011010110010100000000000)_2
(202122112102002220022)_3
(13130311211000000)_4
(1123410000000000)_5
(3401455433012)_6
(402150610106)_7
(73465450000)_8
(22575362808)_9
(8000000000)_10
(3435878453)_11
(1673225768)_12
(9a6543115)_13
(55c6a7c76)_14
(31c4ea585)_15
(1dcd65000)_16
```

## B.4 Semestre 2 - Alternance

Note

..... / 20

Prénom NOM : .....

Exercice 1	Exercice 2	Exercice 3	Exercice 4	Exercice 5
..... / 4	..... / 3	..... / 3	..... / 4	..... / 6

### Modalités

- L'étudiant a 1 heure et 30 minutes pour composer sur ce sujet et rendre une copie papier de sa production.
- Il est conseillé de lire l'intégralité du sujet avant de composer : les 5 exercices peuvent être traités indépendamment les uns des autres.
- Chaque exercice doit être représenté par un code source en **langage C** tel qu'il pourrait être fourni à un compilateur.
- Les attendus à utiliser pour réussir les exercices ne dépassent pas les notions étudiées dans le support de cours jusqu'à la section 8 (pointeurs) : l'utilisation de notions externes au cours ne donne pas de points supplémentaires et n'est pas nécessaire (au risque de se complexifier une tâche qui peut être simple).
- L'évaluation vérifiera et valorisera des éléments de code (entrées, sorties, logique conditionnelle, validité d'opérations, inclusions et autres éléments pertinents liés au langage C et à la logique du code proposé).
- Les exercices sont ordonnés par ordre croissant en fonction des notions vues en classe et le temps / complexité qui peut être requis pour atteindre les objectifs de chaque exercice.
- Des exemples de sorties peuvent être donnés pour aider à comprendre la logique attendue par l'exercice.
- Le sujet est à rendre avec la copie pour notation.

Bon courage !

## Exercice 1 (..... / 4 points)

**Objectif :** Connaître le langage C et identifier des erreurs.

Pour démontrer le concept des boucles en langage C, un camarade propose les codes suivants. Chaque code doit afficher les entiers de 0 à 4.

- Validez ou non les codes proposés.
- Dans le cas où vous l'invalidiez, indiquez le **problème rencontré**.
- Puis proposez une version **corrigée** du programme en C ANSI.

```
/* 1) Boucle Pour */
#include <stdio.h>
#include <stdlib.h>

int main() {
    for(i = 0; i < 5; ++i) {
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);
}
```

```
/* 2) Boucle Tant que */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;
    while(i < 5) {
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);
}
```

```
/* 3) Boucle Faire Tant que */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;
    loop {
        printf("%d\n", i);
    } while i < 5
    exit(EXIT_SUCCESS);
}
```

## Exercice 2 (..... / 3 points)

**Objectif :** Déterminer une moyenne avec coefficients.

- **Entrée 1 :** séquence de couples (note, coefficient) dont note et coefficient sont séparés par une étoile
- **Entrée 2 :** valeur négative (marqueur de fin)

Lire une suite de taille non déterminée de notes suivies d'une étoile et de leur coefficient.  
Puis en calculer la moyenne.

Une valeur négative marque la fin de la séquence.

Rappel : une moyenne de valeurs  $(x_i)_{i \in [n]}$  avec coefficients  $(c_i)_{i \in [n]}$  se calcule de la manière suivante :

$$\bar{x} = \frac{x_1 \times c_1 + x_2 \times c_2 + \dots + x_n \times c_n}{c_1 + c_2 + \dots + c_n}$$

**Exemple de sortie :**

Coefficients différents :

```
Entrez une suite de (note * coeff) :  
10 * 2 20 * 1 5 * 4 12 * 2 -1  
Moyenne : 9.33333
```

$$\frac{10 \times 2 + 20 \times 1 + 5 \times 4 + 12 \times 2}{2 + 1 + 4 + 2} \simeq 9.33333$$

**Exemple de sortie :**

Mêmes coefficients :

```
Entrez une suite de (note * coeff) :  
10 * 1 11 * 1 12 * 1 -1  
Moyenne : 11
```

$$\frac{10 \times 1 + 11 \times 1 + 12 \times 1}{1 + 1 + 1} = 11$$

## Exercice 3 (..... / 3 points)

**Objectif :** Calculer une suite numérique.

La suite de Fibonacci est donnée par la relation suivante :

$$\mathcal{F}_n = \begin{cases} 0 & \text{Si } n = 0 \\ 1 & \text{Si } n = 1 \\ \mathcal{F}_{n-1} + \mathcal{F}_{n-2} & \text{Sinon} \end{cases}$$

Votre objectif est d'afficher les 10 premiers éléments de la suite de Fibonacci. Puis si vous supposez que votre méthode ne donnerait pas les 100 premiers éléments de cette suite dans un temps raisonnable, proposez une méthode qui le permettrait.

**Exemple de sortie :**

```
[0, 1, 2, 5, 12, 29, 70, 169, 408, 985]
```

## Exercice 4 (..... / 4 points)

**Objectif :** Manipuler des chaînes de caractères.

Un collègue a commencé le code commun suivant et vous laisse terminer l'implémentation des fonctionnalités.

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

/* passe chaque lettre en majuscule */
void makeUpper(char * chaine);

/* passe chaque lettre en minuscule */
void makeLower(char * chaine);

/* affiche le nombre d'occurrence de chaque lettre */
```

```
void afficheOccurrencesLettres(char * chaine);

int main() {
    char texte[] = "Bienvenue a l'examen de Langage C";
    makeUpper(texte);
    printf("%s\n", texte);
    makeLower(texte);
    printf("%s\n", texte);
    afficheOccurrencesLettres(texte);
    exit(EXIT_SUCCESS);
}
```

Il s'attend à ce que le résultat soit le suivant :

```
BIENVENUE A L'EXAMEN DE LANGAGE C
bienvenue a l'examen de langage c
'a' : 4
'b' : 1
'c' : 1
'd' : 1
'e' : 7
'g' : 2
'i' : 1
'l' : 2
'm' : 1
'n' : 4
'u' : 1
'v' : 1
'x' : 1
```

## Exercice 5 (..... / 6 points)

**Objectif :** Gérer une liste dynamique, la trier et y rechercher.

Un ami aimerait tenir un historique d'une suite de numéros et vérifier si un nouveau numéro est présent dans cette liste. Il a commencé un programme mais a rapidement abandonné. Complétez le code suivant :

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

int * lireListe(int * taille);

void afficherListe(const int * liste, int taille);

void trierListe(int * liste, int taille);

void swap(int * a, int * b);

int recherche(int valeur, const int * liste, int taille);

int main() {
    int * liste = lireListe(/* ??? */);
    /* Les pointeurs c'est trop difficile, need help ! */
    exit(EXIT_SUCCESS);
}
```

Il s'attend à ce que le résultat soit le suivant :

```
12335 54875 65264 13347 98566 94585 48514 -1
[12335, 54875, 65264, 13347, 98566, 94585, 48514]
[12335, 13347, 48514, 54875, 65264, 94585, 98566]
Quel numéro rechercher ?
>>> 13347
Numéro trouvé.
```

## B.5 Semestre 2 - Janvier

Note

..... / 20

Prénom NOM : .....

Exercice 1	Exercice 2	Exercice 3	Exercice 4	Exercice 5
..... / 4	..... / 3	..... / 4	..... / 4	..... / 5

## Modalités

- L'étudiant a 1 heure et 30 minutes pour composer sur ce sujet et rendre une copie papier de sa production.
- Il est conseillé de lire l'intégralité du sujet avant de composer : les 5 exercices peuvent être traités indépendamment les uns des autres.
- Chaque exercice doit être représenté par un code source en **langage C** tel qu'il pourrait être fourni à un compilateur.
- Les attendus à utiliser pour réussir les exercices ne dépassent pas les notions étudiées dans le support de cours jusqu'à la section 8 (pointeurs) : l'utilisation de notions externes au cours ne donne pas de points supplémentaires et n'est pas nécessaire (au risque de se complexifier une tâche qui peut être simple).
- L'évaluation vérifiera et valorisera des éléments de code (entrées, sorties, logique conditionnelle, validité d'opérations, inclusions et autres éléments pertinents liés au langage C et à la logique du code proposé).
- Les exercices sont ordonnés par ordre croissant en fonction des notions vues en classe et le temps / complexité qui peut être requis pour atteindre les objectifs de chaque exercice.
- Des exemples de sorties peuvent être donnés pour aider à comprendre la logique attendue par l'exercice.
- Le sujet est à rendre avec la copie pour notation.

Bon courage !



## Exercice 1 (..... / 4 points)

**Objectif :** Connaître le langage C et identifier des erreurs.

Pour démontrer le concept des boucles en langage C, un camarade propose les codes suivants. Chaque code doit afficher les entiers de 0 à 4.

- Validez ou non les codes proposés.
- Dans le cas où vous l'invalidiez, indiquez le **problème rencontré**.
- Puis proposez une version **corrigée** du programme en C ANSI.

```
/* 1) Boucle Pour */
#include <stdio.h>
#include <stdlib.h>

int main() {
    for(i = 0; i < 5; ++i) {
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);
}
```

```
/* 2) Boucle Tant que */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;
    while(i < 5) {
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);
}
```

```
/* 3) Boucle Faire Tant que */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;
    loop {
        printf("%d\n", i);
    } while i < 5
    exit(EXIT_SUCCESS);
}
```

## Exercice 2 (..... / 3 points)

**Objectif :** Calculer une suite numérique.

La suite de Fibonacci est donnée par la relation suivante :

$$\mathcal{F}_n = \begin{cases} 0 & \text{Si } n = 0 \\ 1 & \text{Si } n = 1 \\ \mathcal{F}_{n-1} + \mathcal{F}_{n-2} & \text{Sinon} \end{cases}$$

Votre objectif est d’afficher les 10 premiers éléments de la suite de Fibonacci.  
Puis si vous supposez que votre méthode ne donnerait pas les 100 premiers éléments de cette suite dans un temps raisonnable, proposez une méthode qui le permettrait.

**Exemple de sortie :**

```
[0, 1, 2, 5, 12, 29, 70, 169, 408, 985]
```

## Exercice 3 (..... / 4 points)

**Objectif :** Manipuler une chaîne de caractères.

- **Entrée :** Un mot

Lire un mot puis lui appliquer les traitements suivants :

- Mettre en majuscules.
- Changer le sens de lecture : renverser la chaîne de caractères.
- Mélanger les lettres aléatoirement.

**Exemple de sortie :**

Nom : Fibonacci.

```
Entrez un nom : Fibonacci
majuscules : FIBONACCI
renversé : ICCANOBI F
mélangé : BAOICNIF
```

## Exercice 4 (..... / 4 points)

**Objectif :** Calculer une exponentiation matricielle 2x2.

Un matheux vous affirme que la suite de Fibonacci peut se calculer comme la mise à une puissance donnée de la matrice  $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$  et par conséquent que ceci demanderait un nombre d'étapes logarithmique pour ce calcul.

A priori, une matrice 2x2 serait un tableau à deux dimensions. Il vous précise aussi comment calculer la multiplication entre deux matrices :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

Proposer un programme en langage C qui calcule les puissances de  $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$  jusqu'à la puissance 9 :

```
| 1 0 |
| 0 1 |
---
| 1 1 |
| 1 0 |
---
| 2 1 |
| 1 1 |
---
| 3 2 |
| 2 1 |
---
...
---
| 21 13 |
| 13 8 |
---
| 34 21 |
| 21 13 |
---
| 55 34 |
| 34 21 |
```

## Exercice 5 (..... / 5 points)

**Objectif :** Mini-interpréteur sur liste d'entiers.

- **Entrée 1 :** Un entier (taille de la plage mémoire)
- **Entrée 2 :** Nombre non déterminé de caractères (commandes)

Un ami a entendu parler du langage Brainfuck. C'est un langage où vous pouvez vous déplacer dans la mémoire et y changer des valeurs. Il aimerait que vous en codiez un mini-interpréteur dans une plage de taille donnée par l'utilisateur en suivant les règles suivantes :

- '+' ajoute 1 à la case mémoire regardée par le curseur.
- '-' retire 1 à la case mémoire regardée par le curseur.
- '=' égalise toutes les cases mémoire à la valeur de la case mémoire regardée.
- '>' déplace le curseur vers la droite (retour au début si dépasse de la plage mémoire).
- '<' déplace le curseur vers la gauche (retour à la fin si dépassé de la plage mémoire).
- '.' affiche le contenu de la plage mémoire.

```
taille de la mémoire : 4
.
[0, 0, 0, 0]
+.
[1, 0, 0, 0]
=.
[1, 1, 1, 1]
>++.
[1, 3, 1, 1]
=.
[3, 3, 3, 3]
<--.
[1, 3, 3, 3]
<++++.
[1, 3, 3, 7]
-->>-.
[1, 2, 3, 5]
```



## Épilogue

Bon, vous avez digéré ces quelques pages de cours et ceci fait donc de vous un programmeur chevronné ?

Le langage C n'est qu'une porte d'entrée pour initier à la programmation. En effet, ce cours vous apporte des bases de programmation sur lesquelles vous appuyer pour apprendre d'autres langages et leurs paradigmes ou pour associer un contexte d'utilisation à votre apprentissage.

Le Langage C vous permet de poursuivre sur de la programmation système, logiciel embarqué ou encore logiciel d'infographie.

Des bibliothèques intéressantes comme OpenGL pour la programmation graphique sous GPU existent et peuvent être intéressantes à coupler avec du C++. Le C++ vous offre la possibilité d'utiliser des paradigmes orientés objet tout en utilisant du code en langage C.

Un autre langage répandu pour sa portabilité, son intérêt logiciel et appliqué au web qui peut faire suite au langage C est le langage Java.

De même, vous pouvez vous intéresser aux langages interprétés comme Python 3 dont l'interpréteur est écrit en langage C et vous épargnera des lignes de code.

Cette liste de possibilités est loin d'être exhaustive. C'est désormais à vous de profiter de l'apport de ce cours pour votre réussite par la suite !

## Remerciements

Mes premiers remerciements vont à Messieurs Kamal HENNOU et Frédéric SANANES pour la confiance qu'ils m'accordent pour la formation de leurs étudiants.

Je remercie également les étudiants qui par leurs retours aident à l'amélioration de cours. Puisqu'en effet, il leur est destiné.

Mes autres remerciements vont à mes amis qui ont participé à la relecture de ce cours sans fin dont particulièrement Alexandre LAIRAN et Victor VEILLERETTE.

Et finalement, je vous remercie vous, lecteur, qui avez accepté de prendre le temps d'acquérir la connaissance que j'ai voulu partager.



# À propos

Cet ouvrage est conçu pour accompagner l'étudiant de l'ESGI dans son apprentissage du langage C lors de sa première et seconde année de Bachelor en informatique.

Nous proposons une initiation au langage C dont les fondamentaux sont la base d'un grand nombre de langages de programmation aujourd'hui.

Nous donnerons ici les clés pour s'initier au langage C : les éléments élémentaires d'algorithmique tels que les entrées-sorties, variables, expressions, conditions, boucles, fonctions. Mais nous étudierons aussi des concepts plus propres au langage C tels que les tableaux, la structuration de la mémoire, les directives pré-processeur, la programmation modulaire, les opérations bit-à-bit et les possibilités offertes par les pointeurs sur la mémoire, les fonctions et la généricité.

© 2021 - 2023

Kevin TRANCHO



école supérieure de  
génie informatique