1 Langage C

La présente annexe liste limitativement les éléments du langage C (norme C99 ou plus récente) dont la connaissance, selon les modalités de chaque sous-section, est exigible des étudiants. Ces éléments s'inscrivent dans la perspective de lire et d'écrire des programmes en C ; aucun concept sous-jacent n'est exigible au titre de la présente annexe.
À l'écrit, on travaille toujours sous l'hypothèse que les entêtes suivants ont tous été inclus : `<assert.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, `<stdio.h>`, `<stdlib.h>`. Mais ces fichiers ne font pas en soi l'objet d'une étude et aucune connaissance particulière des fonctionnalités qu'ils apportent n'est exigible. Le sujet doit préciser si `<string.h>` a été inclus.

1.1 Traits et éléments techniques à connaître

Les éléments et notations suivants du langage C doivent pouvoir être compris et utilisés par les étudiants sans faire l'objet d'un rappel, y compris lorsqu'ils n'ont pas accès à un ordinateur.

Traits généraux

- Commentaires `/* ... */` et commentaires ligne `//`.
- Les retours à la ligne et indentations ne sont pas signifiants mais sont nécessaires pour la lisibilité.
- Compilation, et compilation séparée. Il est fortement recommandé aux étudiants de compiler avec les options `-Wall -Wextra -fsanitize=address`.
- Notion de fichier d'en-tête. Directive `#include "`*fichier*`.h"`. Utilisation de `#define`, `#ifndef` et `#endif` lors de l'écriture d'un fichier d'en-tête pour rendre son inclusion idempotente.

Définitions et types de base

- Typage statique. Types indiqués lors de la déclaration ou définition.
- Types entiers signés `int8_t`, `int16_t`, `int32_t` et `int64_t`, entiers non signés `uint8_t`, `uint16_t`, `uint32_t` et `uint64_t`. Lorsque la spécification d'une taille précise pour le type n'apporte rien à l'exercice, on utilise les types signé `int` et non signé `unsigned int`. On est tolérant quant aux comparaisons entre entiers de types différents, par exemple entre un `int` et un `size_t`.
- Opérations arithmétiques sur les entiers `+`, `-`, `/`, `*`, `%`. Ces opérations sont sujettes à dépassement de capacité. Seule l'arithmétique modulaire est toujours bien définie, les dépassement d'entiers signés entraînent un comportement non défini.
- Le type `char` sert exclusivement à représenter des caractères codés sur un octet. Notation `'\0'` pour le caractère nul, `'\n'` pour le retour à la ligne, `'\t'` pour la tabulation.
- Type `double` (on considère qu'il est sur 64 bits). Opérations `+`, `-`, `*`, `/`.
- Type `bool` et les constantes `true` et `false`. Opérateurs `!`, `&&`, `||` (y compris évaluation paresseuse). Les entiers ne doivent pas être utilisés comme booléens, ni l'inverse.
- Opérateurs de comparaison `==`, `!=`, `<`, `>`, `<=`, `>=`.
- Les constantes sont définies par `const` *type* *c* `=` *v*.
- Les conversions doivent être explicites : `(`*type*`)` *v*.

Types structurés

- Tableaux statiques : déclaration par *type* *T*`[`*s*`]` où *s* est une constante littérale entière (et surtout pas une variable). Lecture et écriture d'un terme de tableau par son indice *T*`[`*i*`]` ; le langage ne vérifie pas la licéité des accès. On identifie un tableau comme un pointeur sur sa première case.
- Définition d'un tableau statique par un initialisateur `{`t_0`,` t_1`,` ...`,` t_{N-1}`}`.
- Tableaux statiques multidimensionnels et leur linéarisation.
- Définition d'un type structuré par `struct` *nom_s* `{`$type_1$ $champ_1$`;` ... $type_n$ $champ_n$`;}` ; et ensuite `typedef struct` *nom_s* *nom*`;`. On respecte cette procédure en deux temps : déclaration de la structure *puis* alias de type. Lecture et écriture d'un champ d'une valeur de type structure par *v*`.`*champ*, ainsi que *v*`->`*champ* pour un pointeur.
- Définition d'une valeur de type structure par un initialisateur `{.`c_1 `=` v_1`, .`c_2 `=` v_2`, ...}`.

— Chaînes de caractères vues comme des tableaux de caractères avec sentinelle (caractère nul). Fonctions `strlen`, `strcpy`, `strcat`, `strcmp` de `<string.h>`. On peut faire comme si `strlen` renvoyait un entier signé.
— Fonctions de conversion de chaînes de caractères vers un type de base : `atoi`, `atof`.

Structures de contrôle

— Délimitation des portées par les accolades.
— Déclaration et définition de fonctions, uniquement dans le cas d'un nombre fixé de paramètres. Cas particuliers : passage de paramètre de type tableau, y compris multidimensionnels, et simulation de valeurs de retour multiples.
— Utilisation de `void` comme type de retour d'une fonction.
— Passage des paramètres par valeur et par référence.
— Conditionnelle `if (`c`)` s_T, `if (`c`)` s_T `else` s_F.
— Boucle `while (`c`)` s ; boucle `for (`*init*`;` *fin*`;` *incr*`)` s, possibilité de définir une variable dans *init* ; `continue` ; `break`.
— Utilisation de `assert` (notamment lors d'opérations sur les pointeurs).

Pointeurs et gestion de la mémoire

— Gestion de la mémoire : pile et tas, allocation statique et dynamique, durée de vie des objets.
— On considère que les pointeurs sont sur 64 bits. Pointeur `NULL`.
— Pointeur vers un objet alloué, notation *type*`*` p `= &`v.
— Déréférencement d'un pointeur valide, notation $*p$.
— Pointeurs comme moyen de réaliser une structure récursive.
— Création d'un objet sur le tas avec `malloc` et `sizeof` (on peut présenter `size_t` pour cet usage mais sa connaissance n'est pas exigible). On utilise `sizeof` uniquement pour des types et jamais pour des objets. Libération avec `free`. Sur machine, on fait les vérifications nécessaires concernant la gestion de la mémoire.
— En particulier : gestion de tableaux de taille non statiquement connue.

Gestion des ressources de la machine

— Utilisation élémentaire de `printf` et de `scanf`. La syntaxe des chaînes de format n'est pas exigible.
— Rôle des arguments de la fonction `int main(int argc, char* argv[])` ; utilisation des arguments à partir de la ligne de commande.
— Flux standard (`stdin`, `stdout`, `stderr`).
— Gestion de fichiers : `fopen` (dans les modes `r`, `w` ou `a`), `fclose`, `fscanf`, `fprintf` (avec rappel des formatages). Valeur `EOF`.
— (MPI) Fils d'exécution : inclusion de `pthread.h`, type `pthread_t`, commandes `pthread_create` avec attributs par défaut, `pthread_join` sans récupération des valeurs de retour.
— (MPI) Mutex : inclusion de `pthread.h`, type `pthread_mutex_t`, commandes `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_mutex_destroy`.
— (MPI) Sémaphores : inclusion de `semaphore.h`, type `sem_t`, commandes `sem_init`, `sem_destroy`, `sem_wait`, `sem_post`.

1.2 Traits et éléments techniques utilisables sauf mention contraire

Les éléments et notations suivants du langage C ne sont pas à connaître, mais les étudiants sont autorisés à les utiliser, sauf mention explicite du contraire par le sujet.

— Librairie `<limits.h>`, en particulier `INT_MAX` et `INT_MIN`.
— Opérateurs bit à bit, en particulier `<<` et `>>`, sans utilisation abusive.
— Définition d'un type énuméré par `enum` nom_e `{`val_1`, ..., `val_n`}` ; et ensuite `typedef enum` nom_e nom ;. On respecte cette procédure en deux temps : déclaration *puis* alias de type.
— Tableau statique avec initialiseur : on peut utiliser le fait qu'une initialisation est toujours complète et utiliser sciemment une initialisation incomplète.

— Pour les pointeurs, on privilégie la notation du programme `type* p`., mais on accepte l'étoile à droite (`type *p`) ou au centre (`type * p`).
— Pour les blocs (conditionnelles, boucles), on privilégie l'utilisation systématique des accolades même pour une seule instruction, mais on accepte sans.
— Syntaxe **`return;`** pour terminer une fonction de type de retour **`void`**.
— Arithmétique des pointeurs. Pour un tableau, on peut utiliser `t + i` et `*(t + i)`, mais sans abus.
— Transtypage de données depuis et vers le type **`void*`** dans l'optique stricte de l'utilisation de fonctions comme `malloc`. Notamment, les deux syntaxes *type*`* p = malloc(`**`sizeof`**`(`*type*`))` et *type*`* p = (`*type*`*)malloc(`**`sizeof`**`(`*type*`))` sont acceptées.
— Fonction `calloc`, sans abus.

1.3 Traits et éléments techniques interdits sauf mention contraire

Les éléments et notations suivants du langage C ne doivent jamais être utilisés par les étudiants, sauf mention explicite du contraire par le sujet.

— Toute librairie non mentionnée dans les parties précédentes, en particulier `math.h`.
— Opérateurs d'incrémentation `++` et `--`. Ils pourront éventuellement être tolérés dans des cas simples, par exemple comme troisième champ d'une boucle **`for`**.
— Directive du préprocesseur **`#define`**.
— Structure de contrôle **`switch`** … **`case`**.
— Opérateur ternaire (syntaxe `c ? p1 : p2`).
— Utilisation de VLA (par exemple, il est interdit d'écrire **`const int`** *c* `=` *v*`;` *type t*`[c];`).
— Fonction `realloc`.

2 Langage OCaml

La présente annexe liste limitativement les éléments du langage OCaml (version 4 ou supérieure) dont la connaissance, selon les modalités de chaque sous-section, est exigible des étudiants. Aucun concept sous-jacent n'est exigible au titre de la présente annexe.

2.1 Traits et éléments techniques à connaître

Les éléments et notations suivants du langage OCaml doivent pouvoir être compris et utilisés par les étudiants sans faire l'objet d'un rappel, y compris lorsqu'ils n'ont pas accès à un ordinateur.

Traits généraux

- Commentaires `(* ... *)`.
- Les retours à la ligne et indentations ne sont pas signifiants mais sont nécessaires pour la lisibilité.
- Utilisation du toplevel. Les commandes dans le toplevel terminent par `;;`.
- Compilation, et compilation séparée, avec `ocamlc` ou `ocamlopt`.
- Notion de fichier d'interface `.mli`, mot-clef `val`.
- Utilisation d'un module : notation $M.f$. Les noms des modules commencent par une majuscule.
- Gestion automatique de la mémoire.

Définitions et types de base

- Typage statique, inférence des types par le compilateur. On peut utiliser ou non des annotations de type.
- Notion de polymorphisme. On prononce `'a` « alpha », `'b` « beta », etc.
- Déclaration d'une variable immuable globale `let` v `=` e ou locale `let` v `=` e `in` e'.
- Type `int` et les opérateurs `+`, `-`, `*`, `/`, `mod`. Exception `Division_by_zero`.
- Type `float` et les opérateurs `+.`, `-.`, `*.`, `/.`, `**`.
- Type `bool`, les constantes `true` et `false` et les opérateurs `not`, `&&`, `||` (y compris évaluation paresseuse).
- Entiers et flottants sont sujets aux dépassements de capacité. L'arithmétique modulaire est toujours bien définie.
- Comparaisons sur les types de base : `=`, `<>`, `<`, `>`, `<=`, `>=`.
- Types `char` et `string` ; `'`x`'` quand x est un caractère imprimable, `"`x`"` quand x est constituée de caractères imprimables, `String.length`, `String.make`, `String.sub`, s`.[`i`]`, opérateur `^`. Existence d'une relation d'ordre total sur `char`. Immuabilité des chaînes. Caractère `'\n'` pour le retour à la ligne, `'\t'` pour la tabulation.
- Fonctions de conversions pour les types de base : `int_of_float`, `float_of_int`, `string_of_float`, `float_of_string`, `int_of_string`, `string_of_int`.

Types structurés

- n-uplets ; non-nécessité d'un `match` pour récupérer les valeurs d'un n-uplet (déconstruction avec `let`). Type d'un n-uplet.
- Listes immuables : type `'a list`, constructeurs `[]` et `::`, notation `[`x`;` y`;` z`]` ; opérateur `@` (y compris sa complexité). Motifs de filtrage associés. Fonctions du module `List` : `length`, `mem`, `exists`, `for_all`, `filter`, `map`, et `iter`.
- Tableaux mutables : type `'a array`, notations `[|`...`|]`, t`.(`i`)`, t`.(`i`) <-` v. Fonctions du module `Array` : `length`, `make`, `copy` (y compris le caractère superficiel de cette copie), `make_matrix`, `init`, `mem`, `exists`, `for_all`, `map` et `iter`.
- Type `'a option`.
- Déclaration d'alias, mot-clef `type`.
- Types sommes (ou unions), récursifs ou non, polymorphes ou non ; les constructeurs commencent par une majuscule. Motifs de filtrage associés.
- Types produits (ou enregistrements), mutables ou non, récursifs ou non, polymorphes ou non ; notation `{`c_0 `:` t_0`;` c_1 `:` t_1`;` ...`}`, `{`c_0 `:` t_0`;` `mutable` c_1 `:` t_1`;` ...`}` ; leurs valeurs, notation `{`c_0 `=` v_0`;` c_1 `=` v_1`;` ...`}`, accès $e.c$, modification $e.c$ `<-` v.
- Déclaration de types mutuellement récursifs (`and`).

— Piles et files mutables : fonctions `create`, `is_empty`, `push` et `pop` des modules `Queue` et `Stack` ainsi que l'exception `Empty`.
— Tableaux associatifs mutables implémentés par tables de hachage (sans liaison multiple ni randomisation) par le module `Hashtbl` : fonctions `create`, `add`, `remove`, `mem`, `find` (y compris levée de `Not_found`), `find_opt`, `iter`.

Structures de contrôle

— Expression conditionnelle `if` e `then` e_V `else` e_F.
— Filtrage : `match` e `with` p_0 `->` v_0 `|` p_1 `->` v_1 …; les motifs ne doivent pas comporter de variable utilisée antérieurement ni deux fois la même variable; plusieurs motifs peuvent être rassemblés s'ils comportent exactement les mêmes variables; notation `_`, mot-clef **`when`**, importance de l'ordre des motifs quand ils ont des instances communes. Les filtrages doivent être exhaustifs.
— Définition de fonctions : `let`, `let rec` (si récursivité), `let rec` … **`and`** …. On utilise **`and`** uniquement pour des définitions mutuellement récursives.
— Fonction locale à une autre fonction `let rec` f x `=` e `in` e'.
— Fonction anonyme `fun` x y `->` e.
— Notation `function` p_0 `->` v_0 `|` p_1 `->` v_1 …. On n'abuse pas de `function`. On se limite au cas ou la fonction n'a qu'un seul paramètre.
— Curryfication des fonctions.
— Fonction d'ordre supérieur.
— Portée lexicale : lorsqu'une définition utilise une variable globale, c'est la valeur de cette variable au moment de la définition qui est prise en compte.
— Exceptions : levée et filtrage d'exceptions existantes avec **`raise`**, `try` … `with` …; dans les cas irrattrapables, on utilise `failwith`. Utilisation de **`assert`** uniquement pour les vérifications concernant les préconditions.

Élements impératifs

— Absence d'instruction; la programmation impérative est mise en œuvre par des expressions de type `unit`; valeur `()`.
— Références : type `'a` **`ref`**, notations **`ref`**, `!`, `:=`. Les références doivent être utilisées à bon escient. L'usage de références dans des programmes n'en nécessitant pas est sanctionné. L'usage de références vers des structures de données non mutables (comme des listes) est sanctionné.
— Séquence `;`. La séquence intervient entre deux expressions, la première étant de type `unit`.
— Usage de `begin` … **`end`**.
— Boucle `while` c `do` b **`done`**; boucle `for` v `=` d `to` f `do` b **`done`**, boucle `for` v `=` f `downto` d `do` b **`done`**. Sortie de boucle à l'aide d'une exception.

Gestion des ressources de la machine

— `print_int`, `print_float`, `print_string`, `print_newline` `read_int`, `read_float`, `read_line`.
— Syntaxe `let _ = ...` dans un fichier destiné à être compilé.
— `Sys.argv`.
— Flux standard (`stdin`, `stdout`, `stderr`).
— Gestion de fichiers : fonctions `open_in`, `open_out`, `close_in`, `close_out`, `input_line`, `output_string`. Exception `End_of_file`.
— (MPI) Fils d'exécution : module `Thread`, fonctions `Thread.create`, `Thread.join`.
— (MPI) Mutex : module `Mutex`, fonctions `Mutex.create`, `Mutex.lock`, `Mutex.unlock`.

2.2 Traits et éléments techniques utilisables sauf mention contraire

Les éléments et notations suivants du langage OCaml ne sont pas à connaître, mais les étudiants sont autorisés à les utiliser, sauf mention explicite du contraire par le sujet.

— Définition d'une nouvelle exception (`exception`).
— Opérateurs bit à bit, en particulier `lsl` et `lsr`, sans utilisation abusive.
— Pour les séquences dans les instructions conditionnelles, on privilégie l'utilisation de `begin` … `end`, mais on autorise l'utilisation de parenthèses, sans abus (seulement deux expressions courtes dans la séquence).
— L'expression `if` e `then` e_V `else ()` est équivalente à `if` e `then` e_V. Dans le cas où l'expression dans le `then` est de type `unit`, le `else` est optionnel.
— Fonctions `max`, `min`, `print_endline`, `float`, `abs`, `abs_float`, `sqrt`, `log`, `exp`.
— Dans le module `List` : `hd`, `tl`, `sort` (sans abus), `rev` (sans abus), `init`.
— Dans le module `Array` : `sub`, `sort` (sans abus).
— Dans le module `Hashtbl` : `replace`.
— Dans les modules `Stack` et `Queue` : `top`, `iter` (sans abus).

2.3 Traits et éléments techniques interdits sauf mention contraire

Les éléments et notations suivants du langage OCaml ne doivent jamais être utilisés par les étudiants, sauf mention explicite du contraire par le sujet.

— Tout module non mentionné dans les parties précédentes.
— Opérateurs `==` et `!=`. Ils pourront éventuellement être tolérés si l'étudiant justifie précisément la nécessité de les utiliser, en explicitant leur fonctionnement.
— Opérateurs `|>` et `@@`.
— Fonctions `Array.of_list` et `Array.to_list`. De manière générale, les conversions de types sont interdites, sauf justification précise dans le cas où cela améliore la complexité.
— Fonctions `fold`, `fold_left`, `fold_right` (dans tous les modules). De manière générale, toute fonction hors programme issue d'un module au programme est déconseillée, en particulier lorsque son usage simplifie trop le code attendu.