Université de Strasbourg

J.M. Dischler
2024-25

Structures de données et algorithmes II

Enoncé - Devoir de TP libre

Cadre du problème : gestion d'un arbre généalogique

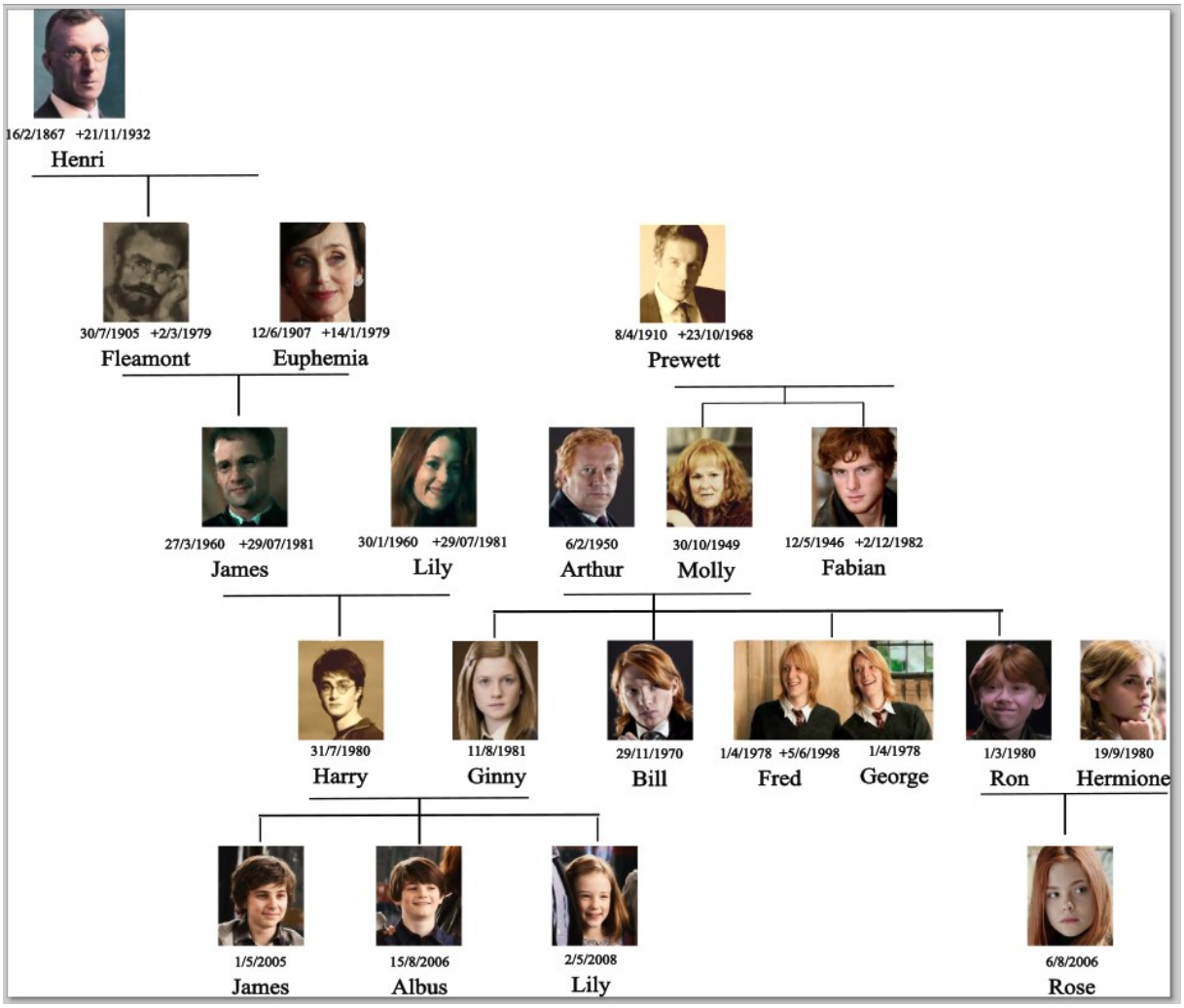


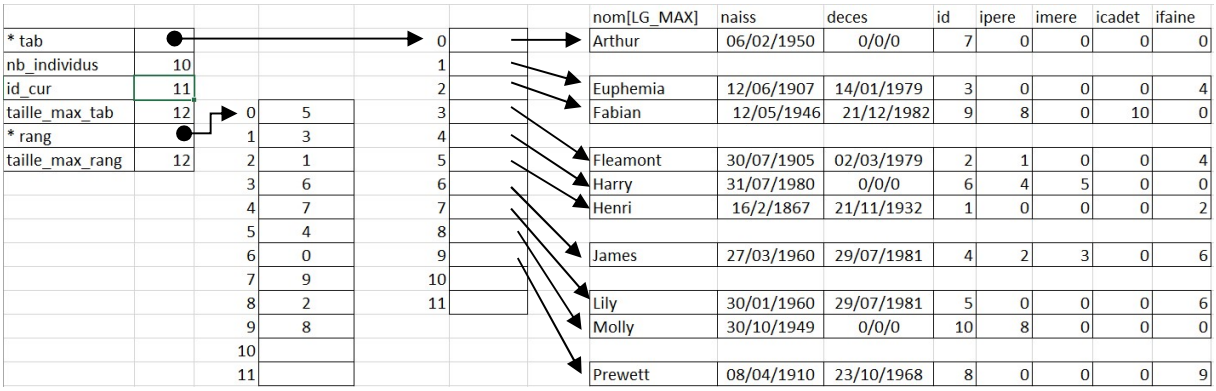
FIGURE 1 – Un exemple d'arbre généalogique (fictif).

Descriptif : Nous nous plaçons dans *un cas parfaitement idéal* : on suppose que chaque individu a toujours exactement deux parents (père et mère). On ne considère pas les familles recomposées : il n'y a pas de beau-père, belle-mère, demi-frères ou demi-sœurs. Chaque couple père-mère se partage la même liste d'enfants. Cependant un parent peut être inconnu.

Chaque individu possède un numéro (ou identifiant) strictement supérieur à 0. Si un parent est inconnu l'identifiant correspondant vaudra 0, noté aussi ω.

Pour représenter la généalogie, on utilise **une table** T qui associe à un identifiant *i* unique (de type *Ident*) toutes les données d'un individu par adressage associatif. On note *T(i)* la référence vers l'individu d'identifiant *i*. La table est représentée par un tableau contigüe alloué

dynamiquement : c'est donc simplement une liste d'individus. Attention : le rang de l'individu dans la liste n'est pas égal à son identifiant : voir la structure ci-dessous.



Nous mettons en œuvre dans cette table, **deux indexations** pour rendre certains accès plus rapides: un index par rapport à l'identifiant pour permettre un accès immédiat à *T(i)* et un index par rapport au nom pour permettre une recherche dichotomique d'un individu par son nom. Les identifiants commencent à 1. Voici les structures C que vous devez compléter en utilisant le schéma :

```
typedef Nat Ident;
#define omega  0
#define LG_MAX  64

typedef struct s_date { Nat jour, mois, annee; } Date;

typedef struct s_individu {
       Car nom[LG_MAX];
       Date  …;                    // à compléter
       Ident  …;                   // à compléter
} *Individu;

typedef struct s_genealogie {
                                   // à compléter
} *Genealogie;
```

Observez bien sur le schéma les types "Genealogie" et "Individu". Remarquez que * signifie tableau dynamique. Déduire du schéma le rôle des différents champs. Remarquez aussi le tri par ordre alphabétique. Indication : ***rang*** est tel que : *T(i) = tab[rang[i-1]]*. Respecter **obligatoirement** – et strictement - les **noms des champs** donnés sur le schéma, p.ex « taille_max_rang » n'est pas « TailleMaxRang ». Par ailleurs, **aucun autre champ ne peut être ajouté**.

Lorsque *nom[0]* vaut '\0' l'individu est invalide. Lorsque l'individu est en vie, sa date de décès vaut {0,0,0}. Lorsqu'un parent est inconnu (père ou mère), l'identifiant correspondant vaut *omega*. De même lorsqu'il n'y a pas de fils ou frère cadet (resp. fille ou sœur).

Dans la figure 1, les parents de James sont Fleamont et Euphemia et les parents de Ginny sont Arthur et Molly. Pour Molly, **`ipere`** vaut l'identifiant de Prewett, **`imere`** vaut omega, **`ifaine`** vaut l'identifiant de Bill, étant le fils le plus agé. Le champ **`icadet`** de Fabian vaut Molly, Fabian étant le frère ainé de Molly.

1. Construction de l'arbre généalogique (7pts)

1.1 Ecrire les générateurs de base de ***Genealogie*** et ***Individu***, ainsi que leur destructeur respectif (libération de la mémoire). On utilisera les prototypes suivants :

```
void genealogieInit(Genealogie *g);
void genealogieFree(Genealogie *g);
Individu nouvIndividu(Ident i, Chaine s, Ident p, Ident m, Date n, Date d);
void freeIndividu(Individu id);
```

1.2 Ecrire les sélecteurs suivants :

```
Chaine nomIndividu(Individu ind);
Date naissIndividu(Individu ind);
Nat cardinal(Genealogie g);            // nombre total d'individus
Individu kieme(Genealogie g, Nat k);   // renvoie tab[k], k n'est pas un Ident !
```

1.3 Ecrire une fonction qui permet l'accès à un individu dont on donne l'identifiant :

```
Individu getByIdent(Genealogie g, Ident i);
```

Cette fonction correspond exactement à *T(i)*. Elle doit utiliser l'indexation par identifiant. Sa complexité doit être constante $\Theta(1)$. Elle renvoie NULL si aucun individu d'identifiant *i* existe.

1.4 Ecrire une fonction qui permet l'accès à un individu dont on donne le nom :

```
Individu getByName(Genealogie g, Chaine name, Date naissance);
```

Cette fonction doit utiliser le fait que les individus sont triés par nom. Sa complexité doit être en $\Theta(\log(n))$ – recherche dichotomique. Elle renvoie NULL si aucun individu n'est trouvé. Si *naissance* vaut {0,0,0} alors cette fonction renvoie l'individu « le plus jeune » ayant pour nom *name*.

Pour écrire *getByName*, on définit une fonction intermédiaire de recherche dichotomique :

```
Nat getPos(Genealogie g, Chaine name);
```

qui renvoie la position de la **première occurrence** d'un individu dans le tableau *tab* ayant pour nom *name*. Ou à défaut la position à laquelle il devrait être.

1.5 Ecrire une fonction qui permet d'ajouter un nouvel individu à la généalogie.

```
//PRE:  (p==omega || getByIdent(g,p)!=NULL) && (m==omega || getByIdent(g,m)!=NULL) &&
//      !freres_soeurs(g,p,m) && getByIdent(g,p)->ifaine == getByIdent(g,m)->ifaine &&
//      compDate(getByIdent(g,p)->naiss,n)<0 && compDate(getByIdent(g,m)->naiss,n)<0
   Ident adj(Genealogie g, Chaine s, Ident p, Ident m, Date n, Date d) ;
```

Comme on peut voir sur ce prototype, on n'indique pas en paramètre les frères, ni les enfants. Ces liens sont automatiquement mis à jour dans la fonction à partir des parents. C'est un procédé automatique de mise à jour des liens de parenté qui impose qu'on ne peut pas ajouter d'individu avant d'avoir ajouté préalablement ses parents. On ne donne pas non plus

l'identifiant de l'individu. Celui-ci doit être généré automatiquement, et correspondra à la valeur renvoyée par la fonction. Il peut y avoir deux individus de même nom dès lors qu'ils n'ont pas la même date de naissance dans la généalogie.

Lorsque l'individu est ajouté à la généalogie, il est important de penser à mettre à jour :

1) les champs **`ifaine`** et **`icadet`** en fonction des enfants déjà connus du père et de la mère (ce sont les mêmes car : *getByIdent(g,p)->ifaine == getByIdent(g,m)->ifaine*).
2) la liste des fils/filles. Celle-ci doit être **ordonnée du plus âgé (en premier) au plus jeune (en dernier)**. Il faut donc pouvoir comparer les dates de naissance. Pour cela, on écrira une fonction ***`Ent compDate(Date, Date)`*** qui renvoie 0 si les deux dates sont identiques, une valeur <0 si la première est antérieure à la seconde et >0 dans le cas contraire.
3) le tableau *rang* de la structure généalogie.

Pour écrire *adj*, on se servira de *getPos*. Puis, on utilisera deux fonctions auxiliaires :
a) une fonction d'insertion dans le tableau *tab* d'un individu en position *pos* (attention *pos* n'est pas de type *Ident*):

```
//PRE: (pos>=1 => strcmp(g[pos-1]->nom,s)<=0)
//     && (pos<g->nb_individus-1 => strcmp(g[pos+1]->nom,s)>=0)
void insert(Genealogie g, Nat pos, Chaine s, Ident p, Ident m, Date n, Date d);
```

Cette fonction crée un nouvel individu, qui a pour identifiant la valeur de *id_cur*, et le place dans le tableau *tab* en position *pos*. Si nécessaire, elle réalloue des tableaux plus grands. Elle met aussi à jour le tableau *rang[ ]*.

b) une fonction d'insertion d'un individu *idx* dans la liste des fils :

```
// PRE: getByIdent(g,x)!=NULL) && getByIdent(g,fils)!=NULL &&  (pp!=omega || mm!=omega)
void adjFils(Genealogie g, Ident idx, Ident fils, Ident p, Ident m) ;
```

fils est le fils ainé du père *p* ou de la mère *m*, dont au moins l'un des deux est connu (càd différent de *omega*). *idx* est ajouté en fonction de la date de naissance : soit en tête et devient le fils ainé de *p* et/ou *m*, soit ailleurs dans la liste, dont la tête restera *fils*. En effet, les enfants doivent être triés du plus ancien (l'ainé) au plus jeune.

1.6 Construire la généalogie donnée en exemple.

Le code suivant permet de construire une partie de l'arbre généalogique de la figure 1 et doit générer **exactement** la structure donnée sous forme de schéma :

```
Genealogie g;
genealogieInit(&g);
Date dnull = { 0,0,0 };
// Famille Potter
Date jhen = { 16,2,1867 }; Date jhed = { 21,11,1932 };
Ident ihep = adj(g, "Henri", 0, 0, jhen, jhed);

Date jfn = { 30,7,1905 }; Date jfd = { 2,3,1979 };
Ident ijfl = adj(g, "Fleamont", ihep, 0, jfn, jfd);

Date jeu = { 12,6,1907 }; Date jed = { 14, 1,1979 };
Ident ijm = adj(g, "Euphemia", 0, 0, jeu, jed);
```

```
Date jpn = { 27, 3, 1960 }; Date jpd = { 29, 7, 1981 };
Ident ijp = adj(g, "James", ijfl, ijm, jpn, jpd);

Date lpn = { 30, 1, 1960 }; Date lpd = { 29, 7, 1981 };
Ident ilp = adj(g, "Lily", 0, 0, lpn, lpd);

Date hn = { 31, 7, 1980 };
Ident ih = adj(g, "Harry", ijp, ilp, hn, dnull);

// Famille Weasley
Date an = { 6, 2, 1950 };
Ident iaw = adj(g, "Arthur", 0, 0, an, dnull);

Date dpre = { 8,4, 1910 }; Date ddpre = { 23, 10, 1968 };
Ident ipre = adj(g, "Prewett", 0, 0, dpre, ddpre);

Date dfab = { 12, 5, 1946 }; Date ddfab = { 21,12, 1982 };
Ident ifab = adj(g, "Fabian", ipre, 0, dfab, ddfab);

Date mn = { 30, 10, 1949 };
Ident imw = adj(g, "Molly", ipre, 0, mn, dnull);
```

Compléter ce code pour construire la généalogie de la figure 1 **dans son intégralité**. Respecter les noms et les dates indiqués sur cette figure !

2. Affichage (5pts)

Attention : Aucune de ces fonctions ne fait d'affichage avec ***printf*** (voir les consignes)! Elles placent le résultat sous forme de texte dans *buff*, une variable de type *Chaine* passée en paramètre et modifiée par effet de bord. L'affichage du texte stocké dans *buff* se fera uniquement et exclusivement dans le *main()* avec: `printf("%s\n", buff);`

2.1 Ecrire une fonction permettant de récupérer dans la chaine *buff* les noms de TOUS les frères et sœurs d'un individu.

```
void affiche_freres_soeurs(Genealogie g, Ident x, Chaine buff);
```

2.2 Ecrire une fonction permettant de récupérer les noms de TOUS les enfants d'un individu.

```
void affiche_enfants(Genealogie g, Ident x, Chaine buff);
```

2.3 Ecrire une fonction permettant de récupérer TOUS les cousins / cousines d'un individu

```
void affiche_cousins(Genealogie g, Ident x, Chaine buff);
```

2.4 Ecrire une fonction permettant de récupérer TOUS les oncles / tantes d'un individu

```
void affiche_oncles(Genealogie g, Ident x , Chaine buff);
```

3. Créer des liens de parenté (2pts)

Comme nous l'avons vu dans la question 1, on n'indique pas en paramètre de la fonction *adj* les frères et les enfants. Ces liens devaient être automatiquement mis à jour dans la fonction à partir des parents. Ceci imposait qu'on ne pouvait pas – a priori – ajouter d'individu avant d'avoir ajouté préalablement ses parents. Or nous souhaitons à présent pouvoir ajouter des parents **après** avoir ajouté leurs enfants et recréer les bons liens de parenté.

3.1 Ecrire une fonction permettant de lier deux individus par fratrie :

```
void deviennent_freres_soeurs(Genealogie g, Ident x, Ident y);
```

Ici x et y deviennent des frères ou sœurs. Notez qu'ils peuvent avoir eux-mêmes un champ **`icadet`** non nul. Dans ce cas les deux fratries sont fusionnées. Quelles sont les préconditions de cette fonction concernant **`ipere`** et **`imere`** de x et y sachant que chaque couple père-mère doit se partager la même liste d'enfants.

3.2 Ecrire une fonction permettant de lier deux individus par parenté :

```
void devient_pere(Genealogie g, Ident x, Ident y);
```

Ici x devient le père de y. Notez que y peut avoir lui-même un champ **`icadet`** non nul. Dans ce cas les fratries sont fusionnées. Quelles sont les préconditions de cette fonction notamment concernant **`ipere`** `et` **`imere`** de y ? Faire de même pour la mère :

```
void devient_mere(Genealogie g, Ident x, Ident y);
```

Ici x devient la mère de y. Quelles sont les préconditions ?

4. Parcours de l'arbre généalogique (6pts)

4.1 Ecrire une fonction permettant de tester si un individu x est un ancêtre d'un individu y.

```
Bool estAncetre(Genealogie g, Ident x, Ident y);
```

Dans l'exemple si x est Fleamont et y est Albus alors la fonction renvoie vrai. Par contre, Fleamont n'est pas un ancêtre de Rose. Attention il faut remonter toutes les générations possibles et connues.

4.2 Ecrire une fonction permettant de tester si deux individus ont un ancêtre commun.

```
Bool ontAncetreCommun(Genealogie g, Ident x, Ident y);
```

Dans l'exemple si x est Rose et y est James (le plus jeune) alors la fonction renvoie vrai. En revanche, Harry et Bill n'ont pas d'ancêtre commun.

4.3 Ecrire une fonction permettant de récupérer l'ancêtre le plus ancien d'un individu (ayant la date de naissance la plus ancienne).

```
Ident plus_ancien(Genealogie g, Ident x);
```

Dans l'exemple si x est Albus la fonction renvoie l'identifiant de Henri.

4.4 Ecrire une fonction permettant de récupérer dans la chaine *buff* toute la parenté d'un individu : parents avec leurs frères et sœurs (oncles et tantes) , grands-parents avec leurs frères et sœurs (grand-oncle/grand-tante), les arrières grands-parents avec leurs frères et sœurs, etc.

```
void affiche_parente(Genealogie g, Ident x , Chaine buff);
```

On précisera la génération : -1 parents + oncles/tantes, -2 grands-parents + grand-oncles/tantes, -3 arrière-grands-parents, etc. Par exemple si x est Albus, on récupère dans *buff* le texte suivant :

```
- 1 :
Harry Bill George Fred Ron Ginny
 - 2 :
James Lily Arthur Fabian Molly
 - 3 :
Fleamont Euphemia Prewett
 - 4 :
Henri
```

4.5 Ecrire une fonction permettant de récupérer toute la descendance d'un individu : enfants, petits-enfants, arrière-petits-enfants, etc. On indiquera la génération.

```
void affiche_descendance(Genealogie *g, Ident x , Chaine buff);
```

Par exemple si x est Euphemia, le texte de la Chaine *buff* sera :

```
- 1 :
James
- 2 :
Harry
- 3 :
James Albus Lily
```

CONSIGNES

Le TP libre est à déposer – au plus tard – le **dimanche 4 mai 2025, 23h59 sur moodle**. Le fichier déposé est un unique fichier de nom ***genealogie.c*** contenant l'intégralité du code source commenté, en respectant exactement le modèle du fichier ***canevas.c*** qui vous est fourni (pas de header séparé, un seul fichier, tous les types de données et prototypes des fonctions seront placés dans cet unique fichier *genealogie.c*).

Le projet est à réaliser seul.

La note est fonction de la quantité et qualité ***du travail personnel*** réalisé.

Consignes spécifiques concernant le code source :

- Le programme doit être écrit en C, pas en C++.
- Il est interdit de faire d' « include » autre que « base.h ».
- Il est interdit de modifier « base.h ».
- Il est interdit d'utiliser directement les fonctions d'allocation mémoire du système. Il faut obligatoirement utiliser les macros MALLOC, MALLOCN, CALLOCN, REALLOC et FREE définies dans le fichier « base.h ».
- Il est interdit d'utiliser les types prédéfinis : *int, unsigned, float, char, union, enum*, etc. du langage C. Il faut obligatoirement **utiliser les seuls types** définis dans « base.h », c'est-à-dire : Bool, Nat, Reel, Car, Chaine, etc.
- Il est interdit de définir d'autres structures ou types de données que ceux fournis dans *canevas.c* : *Date, Ident, Genealogie* et *Individu*. Il est aussi interdit d'ajouter à ces structures d'autres champs ou de modifier les *typedef* de quelque manière que ce soit (enlever le * par exemple, ou remplacer *Individu *tab ;* par *Individu tab[100] ;*). Si on souhaite définir d'autres types (comme liste, pile ou file, il faut faire une *implémentation directe par tableau* sans définir de type spécifique).

- Il est interdit de modifier le prototype des fonctions : il faut conserver exactement les prototypes tels que définis dans *canevas.c*.
- Il est autorisé d'ajouter des opérations ou fonctions auxiliaires dans la rubrique prévue à cet effet.
- Aucune fonction ou variable « externe » n'est autorisée. Donc **pas de *printf*** - dans aucune fonction (sauf dans le *main*), pas de variables globales, pas d'appel à *qsort* non plus ou tout autre fonction du système Unix. Pour les opérations sur les chaînes de caractères, utiliser les macros de « base.h ».
- Les noms de fonctions ET de variables suivants sont interdits : *new, delete, volatile, static, mutable, register, namespace, using, asm, inline, template, typename, operator, explicit, constexpr, decltype, concept, requires, class, protected, public, private, explicit, friend, virtual, override, final, try, catch, throw*. Ce sont des mots réservés du langage C++ qui perturbent la compilation.
- Il est interdit d'utiliser des directives de précompilation # et ## : donc pas de *#undef, #if, #elif, #else, #endif, #ifdef, #ifndef, #pragma, #error, #warning*.
- L'opérateur de séquencement « virgule » est interdit même lorsqu'il s'agit d'initialiser des variables : `Nat a=0, b=1, c=2 ;` ou `for (i=0,j=5 ; i<5 ; i++,j--)`. Il faudra écrire par exemple : `Nat a=0 ; Nat b=1 ; Nat c=2 ;`
- Les opérateurs ternaire « ? : » et unaires « ++ » et « -- » sont autorisés.
- Les structures de « sauts » suivantes sont interdites : *break, continue, goto, switch, case*. Seul le *return* est autorisé.
- Pour définir des constantes dans la rubrique prévue à cet effet, on utilisera *#define*. Par exemple : `#define omega 0 ;`
- Il est interdit d'utiliser *#define* pour définir autre chose que des **constantes** numériques. Donc *#define* ne doit pas être détourné pour définir des fonctions. Vous devez toujours définir des « vraies » fonctions (auxiliaires), **jamais des macros**. Les seules macros autorisées sont celles de « base.h ». Donc par exemple : #define max(a,b) ((a)>(b) ?(a):(b)) est interdit !
- Le cast (conversion de type) utilisera la syntaxe du C et non celle du C++ : par exemple : `Nat i = (Nat)x ;` et non pas : `Nat i = Nat(x) ;`
- Les commentaires // et /* */ sont autorisés et fortement recommandés !
- Le code doit compiler sur Turing avec gcc sans erreur, ni warning.
- Les fuites mémoire seront vérifiées et donneront, le cas échéant, lieu à une pénalité.

Le non-respect de l'une de ces consignes peut entrainer une lourde pénalité, car elle empêchera l'interpréteur et vérificateur automatique de programme de fonctionner correctement.