Travail Personnel Encadré
Comment résoudre un problème avec un programme ?

LOPEZ Aliaume DESMARAIS Yann SÉVIGNÉ Alfred [1]

29 février 2012

[1]Mme Sirlin | Mr G[Pleaseinsert\PrerenderUnicode{ÅŞ}intopreamble]tz

Table des matières

Introduction

De nos jours l'utilisation de l'informatique est omniprésente... Dans de nombreux domaines scientifiques elle est utilisée comme un outil pratique qui offre une multitude d'opportunités. Les ordinateurs sont de plus en plus performants et offrent une puissance de calcul impressionnante qui peut maintenant aider dans de nombreux domaines. Nous avons donc inscrit la problématique du TPE dans cette logique de la résolution d'un problème scientifique, et plus particulièrement biologique, grâce à un programme de simulation. Au cours de ce rapport on suivra chronologiquement l'évolution du développement de ce programme : D'abord la première partie mettra en évidence les objectifs et les buts du logiciel. Elle sera consacrée à la vision globale des fonctionnalités du programme que l'on voulait implémenter, elle constitue en quelque sorte son cahier des charges. La deuxième partie est une description du développement du programme en lui-même, elle expliquera également les expériences et recherches auxquelles nous avons procédé pour obtenir les informations nécessaires à celui-ci. Enfin la troisième partie est en premier lieu la présentation du programme final mais également une explication des limites de la simulation et des problèmes qu'il peut éventuellement soulever.

Première partie

Mise en place du projet

Chapitre 1

Études préliminaires

1.1 Objectifs

Le but exact que nous nous sommes fixés pour le programme était l'étude du temps de vie et de division des cellules. Nous avons décidé pour plus de simplicité de nous en tenir à un seul type de cellule. Cela amena à des recherches qui nous orientèrent vers le choix des cellules de levures, qui sont des organismes eucaryotes dont le temps de division correspondait à notre échelle de temps. En effet les levures mettent deux heures pour se diviser une fois quand une cellule humaine en met vingt. Nos objectifs étaient qu'à l'aide du programme, à n'importe quel moment d'une culture de levure, on puisse savoir combien de cellules il y avait dans la colonie. Cela impliquait que le logiciel comprenne un compteur de temps et que l'on puisse stopper la simulation à tout moment pour voir l'état des cellules. A l'origine nous avions prévu deux environnements distincts dans l'interface ce qui permettait d'avoir une culture témoin et un autre test. Cette idée était motivée par le projet de soumettre aux levures des contraintes et de voir comment celle-ci y réagissait. Nous voulions en effet voir quelles étaient les différences entre une culture normale et une, par exemple, exposée à des agents mutagènes tels que les UV. Pour la représentation des cellules nous avions pensé à quelque chose de simple qui permettait une représentation sur un grand nombre de clones à l'échelle d'une colonie entière. Nous pension le menu comme une fenêtre classique avec une barre d'outil en haut ou apparaitraient les différentes fonctionnalités, les deux environnements de simulation et une barre dans le bas de la fenêtre qui afficherait les donnée de la simulation (nombres de cellules, temps de division etc).

1.2 Recherches

Pour réaliser cette vision première que nous avions du programme nous avons procédé il fallait réunir plusieurs conditions :

- Collecter les informations nécessaires à une simulation réaliste
- Faire nous même les expériences au cas où l'information n'est pas trouvée
- Trouver des outils informatiques performants et qui réponde à nos attentes
- Trouver les fonctions impossibles à implémenter et en proposer d'autres en remplacement

Pour collecter les informations nécessaires nous avons pu utiliser internet (voir sitographie), quelque livres notamment le livre scolaire de SVT de première et de seconde (voir bibliographie), en se renseignant auprès de notre professeur de SVT voir de nos parents (les parents étaient pour chacun de nous soit biologistes soit informaticiens) pour certains détails.

Pour certaines informations concernant surtout les UV et leurs effets nous avons eu l'occasion de faire une expérience pour tester les colonies de levures exposés à ces rayonnements. Cette expérience et plus détaillée dans la partie deux dans le chapitre consacré aux UV.

Pour le choix des outils numérique nous nous sommes orientés vers un langage qui possède de nombreux avantage dont la simplicité et la rapidité. Malgré tout il avait des problèmes de portabilité. Cela nous empêcha donc de fournir le programme avec ce document, il sera présent lors de la présentation oral et sera pleinement testable...

Ensuite pour les fonctions qui n'ont pas aboutie, nous avons procédé par tâtonnement, en essayant de voir laquelle serait la plus cohérente avec l'objectif du programme et celles qui le rendrait plus pratique et facile d'accès. Nous avons par exemple ajouté le fait de pouvoir placer une grille de comptage dans chaque

environnement pour mieux dénombrer les cellules, ou alors le fait de pouvoir zoomer pour voir le processus de division de plus près.

1.3 Problèmes

Enfin il y a la question des problèmes pratiques qui interviennent dès le début du projet dans la mise en place de celui-ci. Ces problèmes étaient d'abord dans la simulation que nous voulions faire : être le plus réaliste possible en restant dans la mesure du réalisable. Trouver certaines données a été assez difficile. Ensuite il y avait également la nécessité que le programme que nous voulions ne soit pas trop complexe à coder en définitive. En général ce ne fut pas une difficulté majeure, nous avons réussi à poser un projet réalisable et les rares fois où ce ne fut pas le cas nous nous sommes arrangés pour trouver des alternatives à ces fonctionnalités trop complexes. Enfin il y'avait les problèmes de choix personnels de chacun, nous avons dû nous mettre d'accord pour les orientations que nous souhaitions pour le logiciel. Au cours du développement et de la création du projet ce ne fut pas non plus très problématique du fait de la bonne entente du groupe et d'une communication assez régulière permettant la constatation des évolutions du programme quasiment en direct. En effet nous avions correspondances par mail importante qui se liait à l'utilisation d'une forge collaborative pour le code du programme : Github (voir sitographie).

Chapitre 2

Cahier des charges de la simulation

Avant de se lancer tête baissée dans une expérience, dans des recherches, ou dans un code, il faut impérativement avoir une idée de ce que l'on veut avoir. Cela comprend le design, les fonctionnalités, les représentations.

Même si cette première vision est large et peut-être très éloignée du programme que nous obtiendrons au final, il faut se fixer un objectif à atteindre.

Voici un aperçu du programme :

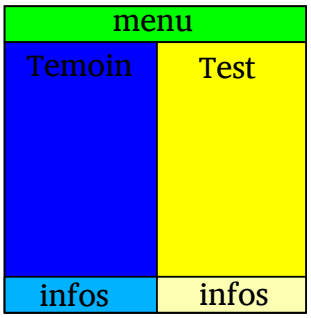

FIG. 2.1 – Une première maquette

Nous avons :

Un menu qui contrôle les différentes actions

Deux boites de pétri qui représentent le témoin et le test

Une barre d'information qui affiche les informations relatives aux simulations

C'est fini pour le design, maintenant il faut définir ce qu'on attend de la simulation :

- Des cellules qui se divisent
- Des UV qui les tuent / font muter

Deuxième partie

Réalisation du projet

Chapitre 3

Le programme

3.1 Présentation des outils

Pour réaliser notre programme, il faut des outils logiciels. Voici une liste exhaustive des outils dont nous avons besoin, et le nom ainsi que la description de ceux que nous avons choisi :

Un ordinateur : celui d'Aliaume

Un langage de programmation : nous avons choisi le Vala. Ce langage est en réalité un un méta-langage, un traducteur le transforme en un véritable langage de programmation : le C. Le C a des avantages, comme la rapidité, et des inconvénients, comme l'insécurité. Le Vala est une surcouche qui permet de simplifier et de clarifier le code, en utilisant des paradigmes de programmation que le C n'a pas à travers une bibliothèque nommée GObject, écrite en C.

Une bibliothèque pour afficher des pixels à l'écran : nous avons choisi la SDL, qui est une bibliothèque pour le C qui permet d'afficher des pixels à l'écran. Elle est plutôt basique, et par la même très simple. On peut donc faire tout ce que l'on veut ... si l'on y met le temps de réfléchir pixel par pixels.

Ces choix ont été fait pour des raisons de performances, mais aussi parce que le C est un langage qui est éprouvé, et dont les vices sont tous connus. Idem pour la SDL. Tous deux sont très bien documentés. Mais, aussi parce que nous avions déjà utilisé ces outils auparavant.

3.2 Programmer le contexte

À partir du cahier des charges de la première partie, nous savons déjà comment mettre en place tout ce qui n'est pas la simulation en elle-même. Car au départ, il n'y a pas encore d'expériences, ni de résultats, on ne peut donc que réaliser ce dont on est sûr d'avoir besoin. Malheureusement, dans la plupart des cas, c'est la chose de plus complexe à réaliser. Par exemple, il faut créer un menu, c'est évident. Or la SDL ne propose rien de tel. Nous allons donc créer notre propre menu, de manière à pouvoir le modifier facilement au cours du développement et le réutiliser dans des programmes ultérieurs. Le menu est en fait un « module ». L'idéal dans la programmation de nos jours, est non pas d'avoir le code avec le moins de lignes possibles, mais le code le plus lisible et le plus modulaire. Chaque module doit être indépendant des autres, être facilement modifiable et extensible. Voici la liste des modules qu'il faut absolument créer :

- Le menu
- La boite de pétri
- La cellule
- L'affichage[1]

Dans chaque module, il y aura différents éléments :

- Des fonctions
- Des variables gobales
- Des Classes ou des Structures

Une classe ou une structure est un patron, une définition formelle d'un nouveau type. Après les types classiques comme les nombres ou les lettres, on a la possibilité de créer des combinaisons de type (on peut aussi combiner des structures). Tout ceci est définit clairement dans l'annexe C.5 pour les structures, et **??** pour les classes.

[1] Qui simplifie l'utilisation de la SDL, par exemple une fonction « dessineCellule » qui exécute automatiquement une liste d'instruction SDL

Il faut impérativement que chaque module soit bien configurable, adaptable, réutilisable et modifiable facilement, car nous n'avons pour l'instant encore aucune idée des valeurs exactes de la simulation, il faut donc prévoir tous les cas impérativement.

3.3 Simulation et boucle principale

3.4 Rectangle, la base

Nous savons que la majorité des éléments de la simulation seront affichés à l'écran. Or pour afficher quelque chose sur une surface, il faut un certain nombre d'informations :

- Position (x, y) sur l'écran
- Taille (w, h)

Nous dessinerons uniquement des rectangles, pour des raisons de simplicité, et parce que nous n'avons pas besoin d'autres formes géométriques dans notre simulation[2].

Avec ces variables, on peut ajouter plusieurs méthodes qui seront utiles :

- déplacer(x,y) ; → redéfinit la position (x, y)
- move (x,y) ; → effectue une translation par le vecteur (x, y)

Nous avons donc notre première classe : « Rectangle ».

3.5 Le menu

L'exemple du menu étant simple et parlant, nous montrerons comment nous avons penser ce module. Dans un menu, il y a des boutons. Donc il faut un objet Menu[3], et un objet Bouton. Un Bouton est un objet affiché à l'écran, il faut donc le faire hériter de Rectangle.

De cette manière on a maintenant des classes comme ceci :

Ce qui n'est pas trop mal. Mais un menu n'est rien sans une gestion des clic et des évènements !

3.6 La gestion des évènements

3.7 La gestion du temps

L'ordinateur ne gère pas le temps nativement. Heureusement la SDL apporte certaines fonctions pour attendre un certain nombre de millisecondes. La gestion du temps dans une simulation est primordiale, et complexe.

Notre approche a été de raisonner en cycles cellulaires, ce qui est plus simple, mais moins fluide.

Plutôt que de mettre des « wait » SDL dans notre boucle principale, nous avons choisi de lancer deux processus en parallèle de la gestion des évènements grâce à la fonction SDL « timer » :

- L'affichage, lancé toutes les 20 millisecondes
- L'exécution de simulation, lancé toutes les 20 millisecondes aussi

Bien sûr il a fallu faire des conversions en heure, en minutes, en secondes. Du fait que le programme soit configurable, il a été d'autant plus difficile de le faire, car au lieu d'ajuster une bonne foi pour toutes les paramètres, il a fallu trouver toutes les opérations pour que les calculs fonctionnent tout le temps.

Par exemple quand on demande que toutes les N secondes (réelles) une cellule se divise, il faut :

- $N \times 50$ pour donner des secondes (20 millisecondes entre chaque tour)
- À chaque tour la cellule vérifie que son nombre de tour de boucle accumulé est divisible par le nombre de tour de boucle nécessaire à une division : $50N$.

[2] En effet, même si on définit les cellules comme des cercles, pour afficher une image, il faut le rectangle de la taille de l'image, même si l'image elle même est un rond sur fond transparent

[3] De manière à être utilisable dans d'autres programmes, ou pouvoir faire plusieurs menus

Chapitre 4

Simulation des Cellules

4.1 Choix des cellules

Les cellules sont la clef de voute de notre simulation, étant donné qu'elles sont les éléments principaux de celle-ci. La première difficulté à été de trouver quel type de cellule nous allions étudier. Pour cela nous avons effectué certaines recherches, pour trouver des cellules simples à comprendre, à manipuler et à représenter. Cela impliquait différents facteurs :

La taille des cellules : les bactéries sont environ 10 fois plus petites que les cellules eucaryotes

La durée de division : élément essentiel pour le programme, et qui varie beaucoup selon les espèces : de 20 minutes à 24h, respectivement pour les bactéries et les cellules humaines

Le métabolisme : les éléments nutritionnels, le fait de survivre en aérobie ou non, et autre facteurs environnementaux

Le déplacement des cellules : il existe tout types de cellules qui se déplacent ou non, et de différentes manières

L'organisation : cellules isolées, colonies, ou organismes multi-cellulaires

Initialement nous n'étions pas fixés sur le choix d'un type de cellule en particulier, il en découle que le programme, qui avait commencé à être développé en parallèle, peut être configuré pour certains des facteurs. Ainsi nous avons choisi les levures, qui sont des organismes unicellulaires qui forment des colonies, qui on fait l'objet de nombreuses études précédentes. Ce qui nous a permis d'accéder facilement à des informations les concernant. De plus les levures sont facilement mises en culture ce qui nous a permis de faire de nous en procurer facilement et de faire nos propres expériences.

4.2 Expérience personnelle

Nous avons fait cette expérience pour avoir une idée de l'effet des UV sur les colonies de levures. Notre but était de soumettre les colonies à différents temps d'exposition[1]. Nous avions pour projet l'observation au microscope des colonies afin de dénombrer les clones mutés et le taux de mortalité en fonction de la durée d'exposition. Nous voulions procéder à l'aide de lame de comptage telles que les « cellules de Malassez »**??**.

Pour cela nous avons eu besoin de l'aide de notre professeur référent en Sciences de la Vie et de la Terre. Ainsi à l'aide de Madame Sirlin et de X[2], nous avons réalisé cette expérience, selon le protocole définit ci-après.

4.2.1 Protocole

Matériel

- Six boites de pétri
- Des marqueurs

[1] Mais à puissance constante

[2] assistante ?

Fig. 4.1 – Cultures de levures personnelles

Fig. 4.2 – Les six cultures

- De la levure
- Deux lampes à alcool
- Une pipette râteau
- De la verrerie : bécher, tube à essai ...

Étapes

Tout d'abord il faut créer la gélose dans laquelle nous avons ajouté du glucose, afin de constituer un environnement riche pour le développement des levures. La gélose est une substance composée d'agar-agar, sur laquelle nous allons ensemencer les souches de levure.

Nous avons parallèlement produit une solution de levure de boulanger à faible concentration.

Nous avons répandu la gélose dans les boites de pétri et attendu qu'elle se solidifie. Puis, à l'aide d'une d'une pipette râteau, dans un environnement stérilisé par les lampes à alcool, nous avons répandu la solution de levure et d'eau chaude sur le milieu.

Nous avons crée 6 cultures :

- Deux témoins qui ne seraient pas exposés aux UV
- Deux test qui seraient exposés aux UV durant une nuit
- Deux autres test qui seraient exposés aux UV durant 24h

4.2.2 Interprétation et résultats

Inopportunément, nous n'avons pas eu accès à des microscopes pour analyser les résultats. De plus nous avions utilisé de la levure de boulanger, alors que pour observer des mutations visibles, il aurait fallu des levures ADE2. Malgré tout nous avons bien constaté à l'œil nu la grande disparité du nombre de colonies entre les différents test et témoins. Et donc, n'avons pu que constater la mortalité des UV, mais nous n'avons pas pu calculer des taux précis. Ce genre d'expérience est courante et nous avons pu trouver des résultats probants aussi bien que dans les livres de seconde et première[3], et des expériences similaires sur internet. Ce qui nous a permis d'enchaîner facilement sur la réalisation du programme.

4.3 Simulation

[3] ??

Chapitre 5

Simulation des UV

Troisième partie

Retrospective

Chapitre 6

Programme final

Chapitre 7

Limites et approfondissements

Annexe A

Définition de l'Algorithmie

Annexe B

Complexité algorithmique

B.1 Definition

> Quand les scientifiques se sont posé la question d'énoncer formellement et rigoureusement ce qu'est l'efficacité d'un algorithme ou son contraire sa complexité, ils se sont rendus compte que la comparaison des algorithmes entre eux était nécessaire et que les outils pour le faire à l'époque1 étaient primitifs. Dans la préhistoire de l'informatique (les années 1950), la mesure publiée, si elle existait, était souvent dépendante du processeur utilisé, des temps d'accès à la mémoire vive et de masse, du langage de programmation et du compilateur utilisé. Une approche indépendante des facteurs matériels était donc nécessaire pour évaluer l'efficacité des algorithmes. Donald Knuth fut un des premiers à l'appliquer systématiquement dès les premiers volumes de sa série The Art of Computer Programming.
>
> Wikipédia

La complexité algorithmique est donc une mesure indépendante de tout facteur matériel, ou même logiciel. Il existe plusieures manières de calculer la complexité algorithmique, nous nous contenterons d'utiliser la plus simpliste, celle qui définit l'ordre de grandeur du nombre d'opération nécessaires à la réalisation du résultat en fonction nombre d'entrée N, dans le pire des cas.

Par exemple le parcours d'un dictionnaire. Il y a différentes manières de coder, la méthode naïve est la suivante : parcourir tous les noms jusqu'à tomber sur le bon. Cette méthode demande au pire pour un dictionnaire de N entrées un nombre N d'opération (dans le pire des cas le nom est à la fin). Il existe aussi la méthode par dichotomie : prendre le nom du milieu. Regarder si le nom cherché est au dessus ou en dessous. prendre encore la moitié, et ainsi de suite. Le principe est de découper le dictionnaire en deux à chaque fois. Dans le pire des cas pour un dictionnaire de N éléments, il faudra couper le dictionnaire $log_2(N)$ fois (par définition du logarithme de base deux). Le deuxième algorithme est donc beaucoup plus efficace que le premier. C'est là tout le principe de mesurer la complexité algorithmique de différentes méthodes, savoir les-quelles seront les plus efficaces. Imaginons que nous voulions accéder à la valeur N d'un tableau en C : il faut effectuer une addition, puis récupérer la valeur du pointeur (cf C.3), soit une complexité de 1. Mais pour accéder à la même valeur N dans une liste, il faut effectuer N opérations (cf C.6).

C'est donc un bon indicateur afin de déterminer comment aborder un problème en fonction des utilisations que nous faisons des variables.

Annexe C

Archiver des données

En programmation il est très courant de vouloir conserver des données durant l'éxécution du programme. Nous parlerons ici uniquement des données qui sont dans la mémoire vive, c'est à dire la mémoire tampon de l'ordinateur, contrairement à la mémoire morte, qui est souvent un disque dur ou un CD. Pour conserver une information dans la mémoire vive, il faut généralement créer une variable. Nous détaillerons ici tout ce que l'on peut faire avec des variables.

C.1 Les Types

Une machine étant une suite de transistors, et la mémoire vive une suite de cases vides et pleines : tout est un nombre binaire. Mais les langages de programmation permettent de faire abstraction de cette complexité techinque. Il existe donc en Vala différents types :

- int : entier, avec des variantes commes : « uint » (entier uniquement positif), ou « int32 » (grand entier)
- float : nombre à virgule
- double : nombre à virgule à double précision
- char : caractère (en réalité c'est un entier positif de maximum 256, qui est traduit en caractères ensuite)
- string[1] : tableau de char, donc du texte vu qu'un char est un caractère.

Ces types sont la base du langage, et on les retrouve souvent car c'est la forme la plus simple de conserver une donnée. Mais il existe un autre type, légèrement plus complexe, qui permet de faire beaucoup plus de choses.

C.2 Le pointeur

Pour commencer il faut comprendre comment fonctionne la mémoire de notre ordinateur :

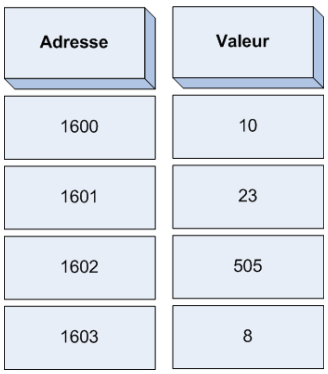

Fig. C.1 – La mémoire

Ce code demande une case mémoire avec une adresse, et met le nombre 3 dedans.

```
1  int a = 3;
```

[1]String = chaîne, sous entendu chaîne de caractères

On peut récupérer l'adresse d'une variable. Par exemple pour afficher la valeur de a, puis son addresse en mémoire :

```
1  printf ("La valeur est %d",a);
2  printf ("L'adresse est %p",&a);
```

Le plus intéressant est qu'une adresse est un nombre, donc on peut la conserver dans une variable aussi ! La syntaxe est la suivante (avec type étant type du langage) :

```
1  Type age = valeur;
2  Type* pointeurSurAge = &age;
```

Ce qui donne en mémoire :

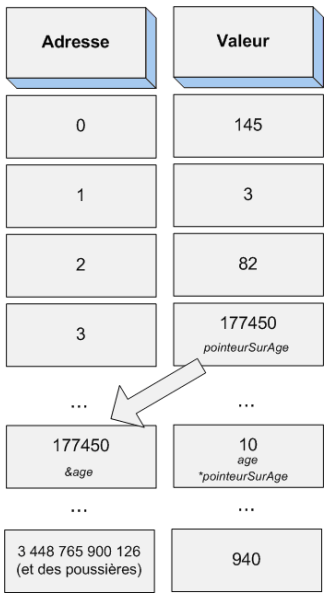

FIG. C.2 – Un exemple de pointeur

On peut ensuite accéder à la valeure pointée par un pointeur avec la syntaxe suivante :

```
1  *pointeurSurAge; // renvoie la valeure de age
```

Les pointeurs ne sont que des variables qui contiennent des adresses mémoire. Mais pour que l'ordinateur comprenne comment traiter la valeure pointée, le pointeur doit avoir le type de la variable (suivie de l'étoile du pointeur).

Une application directe du pointeur est le tableau.

C.3 Tableaux

Un tableau est une suite de cases mémoire du même type :

```
1  int tableau[4]; // Un tableau de 4 cases avec des int
2  printf ("%d", tableau[1]); // valeur de la case 1
3  tableau[10] = 2; // modifie une valeure du tableau
```

En réalité, voilà à quoi ressemble un tableau dans la mémoire de l'ordinateur[2] :

[2]Image tirée de : `http://www.siteduzero.com/tutoriel-3-14015-les-tableaux.html`

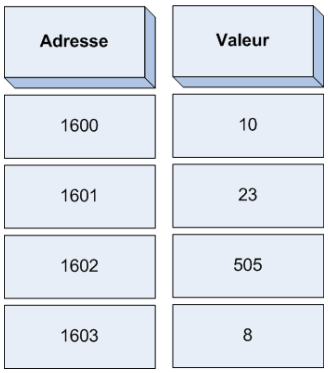

FIG. C.3 – Un tableau dans la mémoire

Lorsqu'un tableau est créé, il prend un espace contigu en mémoire : les cases sont les unes à la suite des autres. L'ordinateur « saura » que c'est un tableau, et quand on demande la 3ème valeur, il prend la 3ème en partant de la première case du tableau. C'est pour cela qu'il peut y avoir les dépassements et bugs définits plus haut !

Donc en réalité, en Vala, un tableau est un pointeur vers la première variable d'une suite de variables contigues. On peut donc considérer que :

```
1  *(tableau); // envoie la valeur de la case d'adresse 'tableau'
2  tableau[0]; // envoie la case 0
3  *(tableau + 3); //  envoie la valeur de la cases d'adresse 'tableau' + 3
4  tableau[3]; // envoie la case 3 ( quatrieme car on compte de 0 )
```

Les syntaxes sont différentes, mais mènent au même résultat, et expliquent un peu mieux le fonctionnement du tableau.

Il en découle que le tableau a une taille fixe. Et que pour l'agrandir, il faut trouver un espace suffisant de cases mémoire contigues.

> Le langage C n'impose pas à une implémentation de vérifier les accès, en écriture comme en lecture, hors des limites d'un tableau ; il précise explicitement qu'un tel code a un comportement indéfini, donc que n'importe quoi peut se passer. En l'occurence, ce code peut très bien marcher comme on pourrait l'attendre [...] ou causer un arrêt du programme avec erreur [...] ou encore corrompre une autre partie de la mémoire du processus [...] ce qui peut modifier son comportement ultérieur de manière très difficile à prévoir.
>
> Wikipédia

Il faut donc faire très attention à ne jamais dépasser la taille d'un tableau !

C.4 Ennumération

L'énnumération permet de créer un nouveau type qui peut prendre un certain nombre de valeurs dans un ensemble fini, imaginons un type « Volume » qui peut avoir uniquement les valeurs suivantes : FORT, MOYEN, FAIBLE, NULL, INFINI :

```
1  enum Volume {
2    FORT, FAIBLE, MOYEN, NULL, INFINI
3  };
4
5  Volume a = Volume.FORT;
```

Ceci est utilisé pour clarifier le code, il est toujours plus lisible de faire des conditions avec des mots, plutôt qu'avec des nombres. Par exemple, le menu a un certain nombre d'actions, au lieu de leur associer un nombre, on leur associe une valeur de l'énumération « ActionMenu ».

Mais si ceci permet quasi-uniquement de clarifier le code, il existe un moyen de créer nos propres types plus complexes.

C.5 Structures

Une structure est un type qui permet de combiner des variables :

```
typedef struct Personne {
  int age;
  string nom;
};

Personne p;
p.age = 16;
p.nom = "jaques";
```

C'est à partir de cette structure que l'on peut définir des structures plus complexes.

C.6 Listes

Les listes sont des structures plus complexes, et qui ne sont malheureusement pas gérées directement par le C (mais par le Vala oui). On fait en réalité la chose suivante :

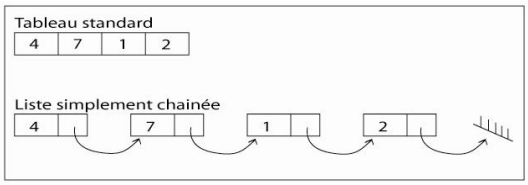

FIG. C.4 – Une liste chainée et un tableau

```
struct element
{
    int val;
    element *nxt; // pointeur vers l'element suivant
};
```

On a donc uniquement des structures qui se pointent les unes vers les autres, sans avoir besoin de zones contigues. Les opérations d'ajout et de suppression d'élément sont plus rapides :

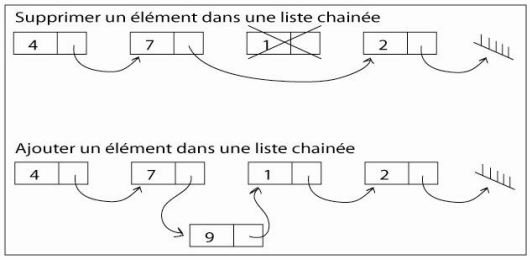

FIG. C.5 – Ajout et suppression d'un item à la liste

Mais pour récupérer un élément, on est obligé de parcourir la liste. Ce qui fait perdre du temps. Pour récupérer l'élément 3, il faut aller au un, puis au deux, et enfin regarder la valeure du numéro 3 ! Ce qui est bien moins efficace qu'un tableau. Il n'y a donc pas que des avantages, ni que des inconvénients. Les listes ne sont pas toujours simplement chainées, on peut aussi en avoir des doublement chainées, c'est à dire que chaque élément pointe vers le suivant ET le précédent. Il peut aussi y avoir des listes cycliques, c'est à dire que le dernier élément pointe vers le premier.