Note d'Application

Mécanismes avancés du C et leur application dans la rédaction de librairies génériques

1 Table des matières

Introduction

Cette note d'application s'intéresse au langage C, et plus particulièrement à certains mécanismes avancés permettant la simplification d'écriture de code complexe, ainsi que la flexibilité des codes obtenus.

Parmi ces mécanismes, on s'intéressera dans un premier temps aux pointeurs et leurs utilisations diverses, puis nous verrons en détail les types composés (`enum`, `struct` et `union`).

Chaque partie sera introduite par un chapitre de rappel plus global revenant sur les bases du C et donc plutôt destiné aux novices.

A terme, cette note a pour but de montrer comment combiner ces deux mécanismes pour obtenir des codes flexibles sans trop sacrifier les performances, dans le but de décrire des librairies génériques indépendantes du hardware.

2 Les pointeurs

2.1 Rappels sur les pointeurs et la mémoire

Le langage C permet d'écrire du code exécutant des fonctionnalités avancées tout en restant proche de l'implémentation physique, et notamment au niveau de la mémoire. En effet, lors de l'écriture d'un code en C, le codeur peut connaitre son impact sur la mémoire à chaque instant et chaque type s'étend sur un nombre fixe d'octets.

Le deuxième aspect du C lié à cette proximité est la possibilité d'accéder à la mémoire à partir d'adresses plutôt qu'à travers des variables. Dans certaines situations, il n'est pas nécessaire de connaitre la position en mémoire d'une variable. Pour certains algorithmes, c'est principalement la valeur contenue qui va nous intéresser, et la chaîne de compilation s'occupe des aspects mémoires sans impacter sur la rédaction du code. Mais dans d'autres circonstances, par exemple quand on désire éviter de travailler sur une copie de la valeur mais bien sur la variable elle même, il est nécessaire de pouvoir accéder directement à l'emplacement mémoire où se trouve la donnée.

2.1.1 Généralités sur les pointeurs

En C, ces accès à la mémoire sont possible grâce aux pointeurs. Alors qu'une variable contient une donnée, un pointeur va contenir une adresse. Comme chaque donnée possède une adresse, et qu'un adresse sert à pointer sur une donnée, des opérateurs permettent de passer d'une variable à un pointeur et inversement.

Pour bien comprendre le fonctionnement du système de pointeur, il est important de maitriser le concept de typage et en particulier son impact sur la mémoire. Au niveau des types de bases, on en retrouve en particulier quatre : `char`, `short`, `long` et `int`.

Ces types sont utilisés pour gérer des entiers. On ne parle pas ici de `float` et de `double` qui sont plus complexes à appréhender.

Dans la plupart des implémentations du C, les trois premiers types correspondent à des zones mémoires de taille croissante : un octet, deux octets et quatre octets. Le type `int` est plus particulier, car sa taille dépend plus du système sur lequel le code est destiné à être utilisé

plutôt que sur l'implémentation du C. Il pourra donc parfois être égal à un `short`, parfois à un `long`.

Sachant cela, on peut considérer le code suivant :

```
char c; // déclare une variable prenant un octet en mémoire
short s; // déclare une variable prenant deux octets en mémoire
long l; // déclare une variable prenant quatre octets en mémoire
```

La déclaration d'une variable va donc déclencher des opérations en mémoire afin de trouver le nombre de cases consécutives correspondant à son type. Lorsque cette variable est utilisée plus tard dans le code, le compilateur utilisera toujours les fonctions assembleurs agissant sur toutes les cases nécessaires.

La déclaration d'un pointeur est quant à elle un peu différente. En effet, un pointeur, quel que soit le type pointé, demandera toujours le même nombres de cases mémoires : celles nécessaire à coder une adresse. Par exemple, sur un processeur au bus d'adresse de 16 bits, on aura :

```
char * cp; // Pointeur sur un char. Place mémoire : 2 octets
short * sp; // Pointeur sur un short. Place mémoire : 2 octets
long * lp; // Pointeur sur un long. Place mémoire : 2 octets
```

Ici, le type passé devant le pointeur ne définit plus la taille requise en mémoire, mais aura un impact sur les opérations réalisées sur le pointeur. Par exemple, sur une mémoire organisée en mot de 8 bits, on observera le comportement suivant :

```
char * cp = 0x00;
cp+=1; // Ici, cp sera égal à 0x01, car on l'incrémente de la taille d'un char
long * lp = 0x10;
lp+=1; // Bien qu'on fasse seulement +1, lp vaut désormais 0x14 car on l'incrémente de
       // la taille d'un long (4 octets)
```

Ce comportement permet à un pointeur de `long` de parcourir la mémoire de 4 en 4 afin de ne pas réutiliser les octets du long précédemment pointé.

2.1.2 Remarque sur le type d'un pointeur

Une considération importante sur la différence fondamentale entre une variable et un pointeur est la suivante :

Un pointeur sur un type ne garantit pas la présence d'une donnée de ce type à l'adresse pointée

Ainsi, lorsque l'utilisateur tape le code suivant :

```
short a = 0;
```

Il est garanti que les futures utilisations de la variable a agiront sur les deux octets composant a. En comparaison,

```
short * ap = 0x00;
```

ne garantit pas qu'un short se trouve à l'adresse 0x00, ni que la variable ap pointera toujours sur une variable de type short. En fait le type donné à un pointeur définit non pas la donnée qui se trouve à l'adresse pointée, mais la façon dont le code (et donc le compilateur) doit considérer les cases mémoires pointées. Par exemple :

```
char a[2] = {128,0};
short * bp = (short *) &a[0];
```

Ici, on déclare un tableau de deux variables de type char. Puis, on déclare un pointeur vers un short. Ce pointeur est assigné à la première case du tableau. Bien qu'à cet endroit en mémoire, on retrouve les deux char de notre tableau (128 et 0), la variable pointée par bp sera un short et contiendra donc la valeur 32167 (ou 128 selon l'endianess). Cette situation sera grandement exploitée dans la suite de cette note d'application.

2.1.3 Opérations sur les pointeurs

Afin de manipuler les pointeurs, et en particulier de passer des adresses aux données, le C propose deux opérateurs :

- & : permet de récupérer l'adresse d'une variable (passage variable -> pointeur), on parle d'opérateur d'indirection
- * : permet de récupérer la valeur contenue à une adresse (passage pointeur -> variable), on parle d'opérateur de déréférencement

L'opérateur * cause parfois des soucis lorsqu'on débute avec les pointeurs, car il s'agit également du signe permettant de déclarer un pointeur. Cependant, il est possible de voir les choses différemment :

```
char * c;
```

Ici on voit une variable c de type char *

```
char * c;
```

Ici on voit une variable * c de type char

Il devient dès lors naturel de considérer que c est un pointeur sur un char (de type char *), et que *c est la variable pointée (de type char).

2.1.4 Utilité des pointeurs

Les utilisations les plus courantes des pointeurs sont les tableaux et les arguments de fonction modifiables. Dans la première utilisation, la notion de pointeur est cachée derrière un opérateur particulier, `[x]`. En pratique, cette opérateur agit comme l'opérateur `*` : il permet donc de déclarer un pointeur, et de récupérer la valeur de la variable par déréférencement. La différence avec `*`, est que lors du déréférencement un entier est passé entre les crochets et permet de décaler temporairement le pointeur. L'utilisation en déclaration possède également une différence, car elle va allouer de la mémoire pour autant de variables du type souhaité que passé entre les crochets.

```
int p[0]; // equivalent à int * p; mais une variable est créée et le pointeur pointe
          // dessus.
p[5]; // equivalent à *(p+5)
```

Note : dans l'exemple ci-dessus, p[5] est incorrect car le tableau déclaré possède moins de six éléments. Cependant, *(p+5) n'est pas incorrect au yeux du compilateur. Seul le comportement sera imprévisible.

S'il est souvent peu utile de considérer un tableau comme un pointeur, il est intéressant de noter que l'opérateur `[x]` peut être utilisé indifféremment sur un tableau ou un pointeur.

La deuxième utilité réside donc dans les paramètres de fonction. En effet, une variable passée en paramètre d'une fonction est copiée dans l'espace mémoire de celle-ci. Les modifications apportées à cette copie au sein de la fonction ne sont pas reportées sur la variable d'origine. Cependant, la valeur d'un pointeur est une adresse en mémoire, et les actions réalisées à cette adresse sont donc permanents. On obtient donc le code suivant :

```
void fonction(int a, int * p) {
  a *= 2; // a est multiplié par 2
  (*p) *= 2; // la variable pointée par p est multipliée par 2
}

void main(void) {
  int a = 1, b = 1;

  fonction(a, &b); // à la sortie de la fonction, a = 1 et b = 2;
}
```

Les pointeurs peuvent donc être utilisés pour déclarer des tableaux de valeurs, ou encore pour modifier des variables dans une autre fonction que leur espace de déclaration. Mais la vraie importance des pointeurs est de permettre de manipuler directement une case précise en mémoire, et surtout de pouvoir, à travers le type pointé, changer la façon dont le compilateur et le code traitent le contenu de cette mémoire.

2.2 Les pointeurs de fonction

2.2.1 Présentation des pointeurs de fonction

Le premier type de pointeur intéressant dans la rédaction d'une librairie générique est le type "pointeur de fonction". En effet, dans le chapitre de rappel, nous avons vu qu'un pointeur comportait un type, permettant au compilateur de savoir comment traiter son déréférencement. Mais il est également possible de créer un pointeur vers une fonction plutôt que vers un simple type.

En C, une fonction est définie par 3 éléments : ses paramètres (ce qu'on lui donne au début de son exécution), son type de retour (ce qu'on obtient à la fin de son exécution) et son déroulement (les instructions appelées lors de son exécution). Lorsqu'une fonction est appelée, l'environnement dans lequel l'appel est effectué n'a besoin que des deux premiers éléments : c'est pour cela qu'on peut avoir recours à des .H ne contenant que les prototypes, et délaissant l'implémentation.

Mais là où le prototype dit "Voici ce que veut la fonction et ce qu'elle retourne, elle sera définie par la suite", ce à quoi le linker répond "Voici où se trouve la fonction voulue", un pointeur de fonction dit simplement "A cette adresse se trouve une fonction comme la fonction voulue". Le principal inconvénient de cette méthode est que rien ne garantit qu'une telle fonction se trouve effectivement en mémoire à cette adresse. Mais les avantages sont multiples : le même pointeur peut, à différents instants, pointer vers des fonctions différentes, et la fonction n'a pas besoin d'être écrite dans la librairie elle-même.

2.2.2 Déclaration d'un pointeur de fonction

Pour saisir facilement l'écriture d'un pointeur de fonction, il faut d'abord rappeler la structure d'un prototype de fonction classique :

```
int fonction(char, short);
```

Ce prototype est composée de trois morceaux : le premier est le mot-clé "int", qui correspond au type de la variable renvoyée par la fonction. Si on ne s'intéresse pas à l'implémentation de la fonction ni à son exécution, son utilisation dans le code est identique à l'utilisation d'une variable de ce type.

Le second morceau est "fonction". Cette partie peut être remplacée par n'importe quel nom, construit comme un nom de variable. Il s'agit du nom qui sera utilisé par la suite pour appeler la fonction.

La dernière partie se trouve entre les parenthèse et correspond à une liste de variables passée en paramètres à la fonction. Vu qu'on parle ici du prototype et non de l'implémentation,

il est inutile de donner des noms à ces paramètres.

Si on ignore les paramètres, on se rend compte qu'une fonction se déclare exactement de la même manière qu'une variable : un type suivi d'un nom. Il est donc naturel de penser qu'un pointeur de fonction se déclare comme un pointeur de variable : un type, suivi d'une étoile puis d'un nom :

```
int * fonction(char, short);
```

Le problème ici est que `int *` est un type, celui d'un pointeur sur int. On se retrouve donc encore avec un format type – nom – paramètres, ce qui est une simple fonction. Pour contourner ce problème, il suffit donc d'entourer la partie nom par des parenthèses :

```
int * (fonction1)(char, short); // fonction renvoyant un int *
int (* fonction2)(char, short); // pointeur vers une fonction renvoyant un int
```

En utilisant cette norme d'écriture, il est donc possible de déclarer des pointeurs de fonction. Leur utilisation peut ensuite être fait de deux manières :

- Soit en considérant le pointeur directement équivalent à une fonction :

```
int fonction(void);

void main(void) {
  int (* fp)(void); // Déclaration du pointeur
  fp = fonction; // Assignation directe
  fp(); // Utilisation directe
}
```

- Soit en traitant le pointeur comme un pointeur classique :

```
int fonction(void);

void main(void) {
  int (* fp)(void); // Déclaration du pointeur
  fp = &fonction; // Assignation par indirection
  (*fp)(); // Utilisation par déréférencement
}
```

A la compilation ces deux méthodes sont strictement équivalentes.

2.2.3 Utilisation des pointeurs de fonction dans une librairie générique

Dans la rédaction de librairies, les pointeurs de fonction sont donc utilisés lorsqu'une fonction est nécessaire mais que son implémentation varie par exemple en fonction des périphériques de l'utilisateur. Ainsi le type de retour et les paramètres de la fonction sont déjà définis, et celle-ci peut être utilisée dans le code même si elle n'est écrite nul part. Il suffit par la suite de décrire à l'utilisateur comment réaliser la fonction par rapport à sa situation particulière

et le code sera complet.

Une seconde utilisation est pour la réaction à des événements : les fonctions de callback en anglais. En utilisant un pointeur de fonction, il est possible de créer un type structure (décrit en détail dans la deuxième partie) possédant un pointeur vers une fonction de réaction, par exemple en cas de click. Cette fonction, par définition, varie d'un élément à l'autre et ne peut donc pas être décrite statiquement. L'utilisation de pointeurs de fonctions permet de rendre l'assignation de ces callbacks dynamique.

2.3 Les pointeurs sur void

Les pointeurs permettent donc de passer directement par la mémoire et donc de s'affranchir des barrières entre fonctions, et peuvent également rendre l'implémentation d'une fonction dynamique plutôt que statique, en laissant la possibilité à l'utilisateur de décrire sa propre implémentation personnalisée.

Mais il reste toutefois un inconvénient : un pointeur de fonction est limité dans la mesure où la liste des paramètres et le type de retour doit être connu d'avance. Cependant, il peut arriver que le type des paramètres puisse varier d'une implémentation à l'autre, nécessitant ainsi un type "neutre" permettant par la suite à l'utilisateur de personnaliser sa fonction.

2.3.1 Utilité d'un pointeur sur void

Dans l'introduction sur les pointeurs, nous avions vu la remarque suivante :

Un pointeur sur un type ne garantit pas la présence d'une donnée de ce type à l'adresse pointée

C'est cette vérité que nous allons utiliser ici pour rendre nos pointeurs de fonction encore plus personnalisables. Le C propose un type particulier `void`, plutôt destiné aux fonctions pour pouvoir par exemple préciser qu'une fonction ne prend pas de paramètre ou ne renvoie pas d'information. `void` ne peut pas être utilisé pour le type d'une variable car le compilateur ne saurait pas quoi réserver en mémoire.

Cependant, il est possible de déclarer un pointeur sur `void` : en effet, la déclaration d'un pointeur n'initialise pas le type en mémoire, juste la ou les cases nécessaires pour stocker l'adresse. En déclarant un pointeur sur `void`, on crée une adresse sans pour autant donner d'indication au compilateur sur la façon dont traiter la variable contenue à cette adresse. Il sera par la suite nécessaire de transtyper le pointeur avant de le déréférencer pour que le compilateur sache quelle type est supposé se trouver à l'adresse pointée.

Un pointeur sur `void` permet donc à l'utilisateur de passer en paramètre d'une fonction non pas une variable classique, mais une simple adresse sans a priori sur le contenu de la mémoire. A cette adresse, il pourra s'y trouver n'importe quelle type, y compris le début d'un tableau.

2.3.2 Utilisation

La principale limitation d'un pointeur sur `void` est l'impossibilité de lui passer des constantes littérales (sans passer par une variable). Il est donc nécessaire avant d'appeler la fonction de stocker les paramètres à passer en `void *` dans des variables.

Pour utiliser un pointeur sur `void`, il suffit de passer à sa fonction un paramètre de type `void *` :

```
void (*ma_fonction_indefinie)(void * param_indefini);
```

Lorsque l'utilisateur définira la fonction, il pourra alors récupérer le type qui l'intéresse à l'aide d'un transtypage :

```
void fonction_utilisateur(void * param_indefini) {
  short mon_param = *((short*) param_indefini);
  ... // Travail sur le short récupéré
}
```

Cette opération commence par dire au compilateur de considéré `param_indefini` comme un pointeur sur un `short`, puis procède au déréférencement pour récupérer la valeur contenue en mémoire. `mon_param` peut par la suite être utilisé comme un `short` normal.

L'appel à la fonction se fait également par un transtypage, mais cette fois dans l'autre sens :

```
short mon_short = 10;
ma_fonction_indefinie = fonction_utilisateur;
ma_fonction_indefinie( (void*) &mon_short);
```

Ici l'opérateur `&` permet de récupérer un pointeur sur `short`, qui est transtypé vers un pointeur sur `void` pour pouvoir être passé en paramètre. Avec cette technique, il est possible d'avoir une implémentation avec un autre type que `short` qui utilise le même pointeur, même dans la suite du code :

```
void fonction_tableau(void * param_indefini) {
  mon_tableau = (char *) param_indefini;
  ... // Travail sur le tableau de char récupéré
}

...

short mon_short = 10;
char mon_tableau[] = "Hello World!";

ma_fonction_indefinie = fonction_utilisateur;
ma_fonction_indefinie( (void*) &mon_short);

ma_fonction_indefinie = fonction_tableau;
ma_fonction_indefinie( (void*) mon_tableau);
```

2.4 Conclusion

En combinant les pointeurs de fonction pour laisser l'implémentation à l'utilisateur, et les pointeurs sur `void` pour laisser également le type des paramètres, il est possible de laisser de grandes options de personnalisation dans une librairie.

Souvent, en plus des callbacks, les pointeurs de fonction permettent de faire abstraction de la partie matérielle sur laquelle la librairie fonctionnera. Il faut cependant noter que l'appel à une fonction à travers un pointeur de fonction est nécessairement plus lent qu'un appel statique, puisqu'il faut tout d'abord aller chercher en mémoire l'adresse pointée par notre pointeur.

Dans le cas d'une fonction d'abstraction, il est donc important de limiter au maximum son ampleur, en essayant de réduire à son strict minimum la dépendance au hardware. Par exemple, dans le cas d'une librairie graphique, toute la connexion à la partie matérielle peut se résumer en une seule fonction : être capable d'envoyer une couleur donnée à un pixel de coordonnées connues. Cette fonction pourra donc être codée dans un pointeur de fonction, ce qui permettra à la librairie d'être compatible avec tout écran répondant à cette spécification.

3 Les types composés

En plus des types de base permettant de manipuler des entiers et des flottants, le C propose trois types supplémentaires dédiés à des données plus complexes : `enum`, `union` et `struct`.

3.1 Rappel sur les types composés

Les trois types composés du C permettent de faire face à plusieurs situation où des données plus complexes que de simples nombres sont requises. On les appelle types 'composés' car contrairement aux autres types, ils contiennent plus d'une information à la fois.

3.1.1 Les énumérations

Le premier type composé est l'énumération, ou `enum` dans le code. Un `enum` consiste en une liste finie de valeur que peut prendre une variable. Contrairement aux entiers, les valeurs d'une `enum` ne sont pas des chiffres (ou des caractères) mais des mots écrits comme des noms de variable. L'`enum` le plus connu et le plus parlant est le booléen, permettant de rajouter le type `bool` du C++ dans un code C :

```
enum bool {
  false,
  true
};

enum bool monBooleen = false;
```

Au niveau de la valeur, chaque champ d'un `enum` est associé à un entier, avec un incrément de 1 entre chaque champ. Par défaut la première valeur d'une liste est associée à l'entier 0, mais il est possible de forcer la valeur d'un champ :

```
enum bool {
  false = 0x00,
  true = 0xFF
};
```

3.1.2 Les unions

Le second type composé est le type `union`. Ce type permet d'utiliser le même espace mémoire pour des variables différentes. Cela peut être utile dans deux situations : lorsqu'on sait que plusieurs variables ne seront jamais utilisés en même temps, les stocker au même endroit peut économiser de la mémoire ; ou lorsque notre variable peut avoir plusieurs représentations en fonction de son utilisation. Par exemple :

```
union twoChar {
  char c[2];
  short s;
};

union twoChar ma_variable;
```

Ici, le tableau de deux `char` nécessite deux octets en mémoire, tout comme le `short`. `ma_variable.s` permet donc d'utiliser un `short` classique, tandis que `ma_variable.c[0]` et `ma_variable.c[1]` permet d'accéder séparément au MSB et au LSB de notre variable (attention à l'endianess !).

Contrairement à dans cet exemple, les différentes composantes d'une `union` ne font pas forcément la même taille en mémoire. Dans ce cas, la taille de l'`union` en mémoire est celle prise par sa plus grande composante.

3.1.3 Les structures

Le troisième et dernier type composé, et sans doute le plus utilisé, est le type `struct`. Sa déclaration est similaire à celle d'un union, mais différentes variables sont stockées en mémoire à la suite et non au même endroit. Cela permet de rassembler en une seule variable un ensemble de valeurs portant sur le même objet. On peut par exemple utiliser une `struct` pour décrire une piste de musique :

```
struct song {
  char title[32];
  unsigned char track;
  unsigned short duration;
};

struct song ma_chanson;
```

Les `struct` permettent donc d'encapsuler une logique plus complexe qu'un simple nombre dans une seule variable. Chaque composante d'une `struct` ayant sa propre position en mémoire,

il est également facile pour le compilateur de faire une copie ou un test d'égalité entre deux variables créée depuis une même `struct`, champ par champ. De plus, grâce à cette capacité de copie, il est possible de passer une `struct` en paramètre d'une fonction de la même manière qu'une simple variable.

3.1.4 Le mot clé typedef

Lors de la déclaration de variables en utilisant les types composés, on remarque qu'il faut, par défaut, répéter le mot-clé `enum`, `union` ou `struct` en plus du nom associé à notre composition spécifique. Pour éviter cela, il est possible d'utiliser le mot-clé `typedef` permettant d'associer un alias à un type composé :

```
typedef struct song {
  char title[32];
  unsigned char track;
  unsigned short duration;
} song_t;

struct song ma_chanson_1;
song_t ma_chanson_2;
```

Dans cet exemple, le type `struct song` et l'alias `song_t` sont complètement équivalents. L'utilisation de `typedef` permet donc d'alléger les codes utilisant des types composés.

Note : il existe une légère différence entre le type et son alias : le nom du type est défini avant la description du contenu du type, contrairement à son alias qui est défini après. Par conséquent, le nom du type, contrairement à l'alias, peut être utilisé dans la description sous forme de pointeur, par exemple pour créer une liste chainée.

3.2 Des enum pour une personnalisation plus claire

Au sein d'un code, les `enum` peuvent être utilisés pour rendre certains aspects plus clairs, par exemple dans un `switch ... case`. Cela simplifie la relecture et la vérification du code.

Lors de la rédaction d'une librairie, l'utilisation d'`enum` est d'autant plus vitale que le code est destiné à être utilisé non pas par le ou les codeurs initiaux, mais par d'autres développeurs. Les `enum` permettent donc des options de personnalisation plus compréhensibles.

L'un des exemples les plus courants d'utilisation d'`enum` dans une librairie est pour les codes d'erreurs renvoyés par les fonctions. Non seulement l'`enum` garantit dans une certaine mesure que le bon code d'erreur est renvoyé, mais permet aussi à l'utilisateur de se faire une idée sur l'erreur survenue sans avoir besoin de relire la fonction pour comprendre le contexte.

Il est aussi intéressant d'utiliser des `enum` pour certains paramètres de fonction, par exemple si ce paramètre sert de flag pour modifier le comportement de la fonction.

Une autre utilisation des `enum` dans la rédaction d'une librairie est décrite dans le chapitre 3 et permet d'implémenter un semblant d'héritage en C.

3.3 Des struct pour des modèles de données complexes

La structure est le type composé permettant de tirer au maximum profit des capacités du C. En plus des avantages apportés par l'encapsulation des données, `struct` propose d'autres possibilités de gestion mémoire permettant d'écrire des codes très complexes et efficaces.

3.3.1 Organiser ses données

L'intérêt premier d'une structure est d'organiser correctement ses données. Ceci est particulièrement important dans une librairie : les `struct` définies formeront la grande majorité des types utilisés par l'utilisateur pour faire fonctionner la librairie.

Souvent une librairie sera amenée à utiliser des données complexes : audio, dessins, canaux de communication... Tous ces éléments gagnent en clarté d'utilisation une fois encapsulé dans une ou plusieurs structures.

Les composantes d'une structure pouvant être de n'importe quel type (y compris un type défini par l'utilisateur), une structure peut en contenir une autre, permettant ainsi d'encore affiner la représentation d'une donnée.

Un autre élément intéressant des structures est leur capacité à former des listes chainées. Une liste chainée est une alternative aux tableaux offrant plus de flexibilité, au prix d'une efficacité réduite dans certaines situations.

Le principe d'une liste chainée repose sur les structures et les pointeurs. Il s'agit d'ajouter comme composante d'une structure un pointeur vers une instance de cette structure. Chaque instance de la structure pourra donc posséder un lien vers une prochaine instance, créant ainsi la "chaine" :

```
typedef struct short_chaine {
  unsigned short valeur;
  struct short_chaine * suivant;
} short_chaine_t;
```

Le type `short_chaine_t` créé permet donc de former une liste de `unsigned short` dont la taille n'est pas connue à l'avance. Pour chaque élément, la valeur est obtenue par `elem.valeur`, tandis que la prochaine valeur sera accessible par `(*elem.suivant).valeur`.

Le système de liste chainée permet de créer des tableaux de longueurs variables, très utile dans le cas où la longueur de ce tableau varie en fonction des besoins particulier d'un utilisateur ou d'une situation.

3.3.2 Union et structure

Il est également possible de combiner les structures et les unions afin de simplifier certaines situations. Par exemple, il est possible de faire une union entre une structure et un entier de la même taille. Ceci peut être utile par exemple pour des communications :

```
typedef union donnee {
  struct {
    unsigned char rw;
    unsigned char addr;
    unsigned short value;
  } field;
  unsigned long complete;
} donnee_t;
```

Le type ci-dessus permet de contrôler une trame de données. Cette trame est composée d'une information "read-write" sur 8 bits, d'une adresses sur 8 bits, et d'une donnée sur 16 bits. Au total, une trame est formée de 32 bits. L'union permet l'accès direct à la trame complète sous la forme d'un entier de 32 bits (`unsigned long`) à travers l'expression `trame.complete`, ou à un champ particulier grâce à la structure "field", par exemple `trame.field.value` pour accéder à la donnée.

Reposant à peu près sur le même principe, le C permet d'utiliser les structures pour manipuler des données plus petites qu'un *char*. En situation normale, le `char` est la plus petite donnée manipulable en C, et équivaut dans la plupart des cas à 8 bits. Pourtant, on voit souvent des `char` utilisés non pas pour stocker des caractères ou des entiers de 0 à 255, mais de simple flags ne possédant que 2 ou 3 valeurs significatives. Si une donnée nécessite plus d'un flag de ce type, il est tentant de les stocker dans la même variable afin d'économiser de la mémoire. Il est ensuite possible de travailler sur un flag particulier à l'aide de masques et d'opérations bit à bit, mais cette méthode peut vite être source d'erreur. Le C propose donc la formulation suivante :

```
typedef struct flags {
  unsigned char flag1 : 2;
  unsigned char flag2 : 5;
  unsigned char flag3 : 1;
} flags_t;
```

Le type défini ci-dessus permet de manipuler 3 flags de taille limitée inférieure à celle d'un `char` : le flag1 est codé sur 2 bits (4 valeurs), le flag2 sur 5 bits (32 valeurs), et le flag3 sur un seul bit (2 valeurs). Faisant partie de la même structure, ces flags sont en réalités codées sur un seul `unsigned char` et concaténés. Il est cependant possible de les utiliser dans le code comme des variables indépendantes, le compilateur s'occupera de mettre les masques en place.

Le type passé devant une variable suffixée d'un `:` (appelée bit-field) ne sert pas ici à définir sa taille (qui est définie par le chiffre final), mais à donner différentes indications au compilateur :

- Décrire comment traiter la variable lorsqu'elle est manipulée (par exemple en cas de comparaison)
- Donner le signe de la variable contenue
- Définir la taille totale à allouer, dans laquelle les bits nécessaires seront prélevés

Le dernier point est particulièrement important. En effet, lorsqu'un second bit-field est déclaré utilisant la même taille de mot, le compilateur va d'abord vérifier s'il reste la place nécessaire dans la variable précédente. Si c'est le cas, les bits restants de cette variables seront utilisés ; sinon, une nouvelle zone mémoire sera réservée. Il est d'autant plus important d'être rigoureux dans la taille des zones mémoires que chaque compilateur peut traiter les bit-fields différemment. Le code suivant en est un exemple :

```
typedef struct color_16bits {
  unsigned char red : 5;
  unsigned char green : 6;
  unsigned char blue : 5;
} color_16bits_t
```

Ici, on demande au compilateur de réserver 16 bits en mémoire, pour contenir le code RGB565 d'une couleur. Pour certains compilateurs, cette structure sera répartie sur deux octets de la manière suivante :

1. La variable `red` nécessite 5 bits. Un espace de 8 bits est alloué, il en reste 3.
2. La variable `green` nécessite 6 bits. Il reste 3 bits libres. Les 3 bits de poids fort de `green` y sont placés, et une nouvelle zone de 8 bits est allouée, dans laquelle 5 bits restent vacants.
3. La variable `blue` nécessite 5 bits. Il reste 5 bits libres, qui sont donc utilisés.

Au terme de ces étapes, un espace de 16 bits est alloué et chaque bit est utilisé. Cependant, d'autres compilateurs agiront plutôt comme suit :

1. La variable `red` nécessite 5 bits. Un espace de 8 bits est alloué, il en reste 3.
2. La variable `green` nécessite 6 bits. Il n'y a pas assez de place dans la variable précédente. Un nouvel espace de 8 bits est créé, et les 6 bits de poids fort sont utilisés. Il en reste 2.
3. La variable `blue` nécessite 5 bits. Il n'y a que deux bits libres dans la variable précédente, un nouvel espace de 8 bits est donc créé.

Dans cette seconde interprétation, la taille finale de notre structure est de trois octets, comme si les bit-fields n'avaient pas été utilisés. Dans chaque variable, plusieurs bits sont ignorés et rendus inutilisables.

Ici la façon correcte de déclarer notre structure aurait été de déclarer chaque champ comme faisant partie non pas d'un `char` mais d'un `short` :

```
typedef struct color_16bits {
  unsigned short red : 5;
  unsigned short green : 6;
  unsigned short blue : 5;
} color_16bits_t
```

Dans cette situation, une seule interprétation est possible, et l'espace réservé est bien de 16 bits.

Il est bien évidemment possible de combiner les bit-fields avec les unions :

```
typedef union color_16bits {
  struct {
    unsigned short red : 5;
    unsigned short green : 6;
    unsigned short blue : 5;
  } field;
  unsigned short code;
} color_16bits_t
```

Ici, notre type `color_16bits_t` permet d'accéder au code de la couleur soit dans son intégralité grâce à `color.code`, soit champ par champ avec `color.field.red`, `color.field.green` et `color.field.blue`.

3.3.3 Structure anonyme

Les structures anonymes sont une solution pour alléger un code utilisant une union contenant une structure. Dans les deux exemples précédents, les composantes indépendantes de nos unions étaient accessibles à travers le champ "field". Cependant, ceci peut rapidement alourdir le code en créant des noms de variables assez long.

Les structures anonymes permettent d'alléger le code en évitant un niveau de profondeur d'accès. Ainsi, les champs de la structures deviennent directement accessibles comme s'il s'agissait de champs de l'union. Pour déclarer une structure anonyme, il suffit de ne pas donner de nom à notre variable. Ainsi, les structures des exemples précédents peuvent être écrites comme suit :

```
typedef union donnee {
  struct {
    unsigned char rw;
    unsigned char addr;
    unsigned short value;
  };
  unsigend long complete;
} donnee_t;
```

```
typedef union color_16bits {
  struct {
    unsigned short red : 5;
    unsigned short green : 6;
    unsigned short blue : 5;
  };
  unsigned short code;
} color_16bits_t
```

Dans cette situation, une variable de type `donnee_t` peut être utilisée comme suit :

```
donnee_t data;
data.rw = 1;
data.addr = 0xAA;
data.value = 42;

send32bit(data.complete);
```

On remarque que les champs `rw`, `addr` et `value` sont utilisés de la même manière que le champ `complete`, sans passer par la structure, allégeant ainsi le code. Il en va de même pour le type `color_16bits_t`.

3.3.4 Les structures en mémoire

Dans ce dernier chapitre sur les structures, l'accent est porté sur leur représentation en mémoire. En effet, en C, la position en mémoire d'une structure est régie par des règles précises qu'il faut connaitre afin d'optimiser l'impact mémoire d'un code, mais qui peuvent aussi être tournées à notre avantage dans des cas précis.

Le premier élément à prendre en compte est la continuité des données d'une structure. En effet, la norme du C requiert que les données d'une structure soient stockées dans l'ordre dans lequel les champs sont déclarés. Cependant, ceci peut rentrer en conflit avec le principe d'alignement des variables. Ce principe du C repose sur la formule suivante :

```
(adresse d'une variable) % (taille en mémoire de la variable) == 0
```

Ce principe assure entre autre qu'un pointeur sur un `short` par exemple ne pourra jamais pointer sur une moitié de `short`. Il s'agira soit d'un `short` entier, soit d'un autre type.

Cela signifie également qu'un `char` suivi d'un `long` pourra causer des cases mémoires vides :

```
unsigned char mychar; // Adresse 0x0000;
unsigned long mylong; // Adresse 0x0004, car un long ne peut pas être stocké en 0x0001
                      // ce qui crée un vide de 3 octets
```

Cette règle est évidemment vraie également au sein d'une structure. L'ordre des déclaration est donc important. De plus, une règle supplémentaire oblige la taille totale d'une structure à être un multiple de la taille de sa plus grande variable.

Ces deux contraintes et leurs conséquences peuvent être visualisées par un simple

exemple.

Prenons la structure suivante contenant un `char`, un `short`, un `long` et un nouveau `char`. On obtient la structure suivante :

```
typedef struct test {
  char c1;
  short s2;
  long l3;
  char c4;
} test_t;
```

En oubliant les contraintes citées précédemment, on pourrait imaginer que cette structure requiert 1 + 2 + 4 +1 = 8 octets en mémoire. Si on imagine maintenant un tableau de deux éléments composé de cette structure, voici ce que donnerait le plan mémoire :

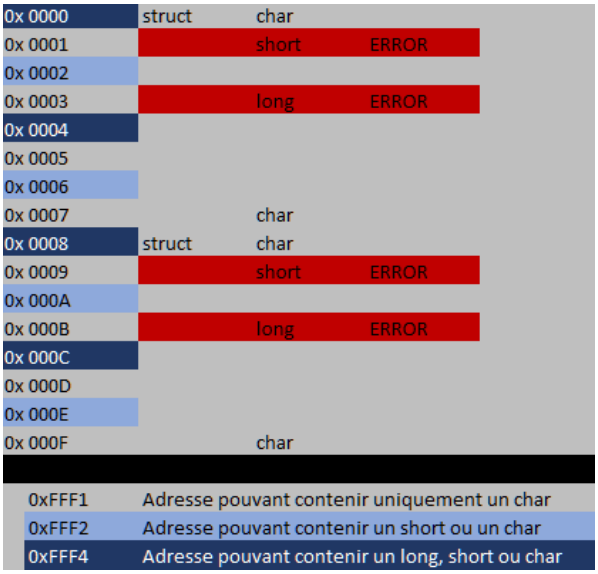

Fig. 1. Plan mémoire sans alignement

On voit clairement sur la figure que dès la première instance de structure, un problème survient sur le plan mémoire : des `short` et des `long` sont déclarés dans des cases mémoires incompatibles.

Pour contrer cela, le C procède à un alignement automatique en insérant des cases de 'padding'. Ces cases seront inutilisées, mais permettent d'aligner les éléments en mémoire. Une fois ces cases intégrées, on se retrouve avec le plan mémoire suivant :

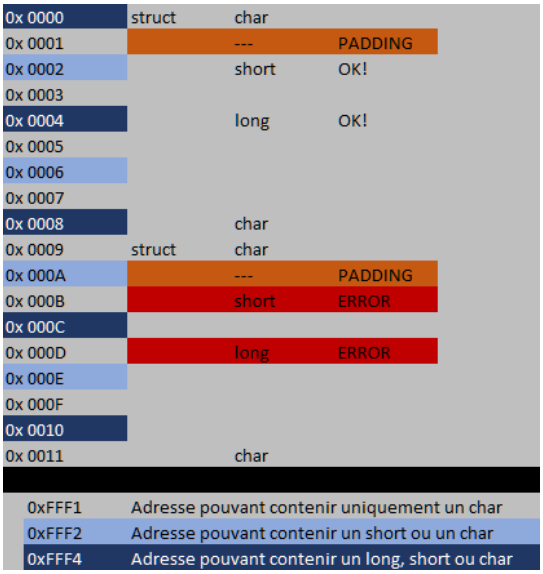

Fig. 2. Plan mémoire avec alignement

Ici, la première instance de la structure est alignée correctement. On remarque d'ores et déjà qu'on utilise 9 octets au lieu de 8. De plus, la seconde instance n'est pas alignée comme il faut. Cela est dû au fait que les éléments d'un tableau sont directement contigus. Pour éviter ce problème, la taille de la structure devient un multiple de la taille de sa plus grande variable. Ici, il s'agit d'un `long` sur 4 octets. Il faudra donc que notre structure prennent comme taille un multiple de 4 supérieur à 9, soit 12. Trois octets d'alignement sont donc ajoutés à la fin de notre structure :

Fig. 3. Plan mémoire avec alignement et taille

Dans ce plan final, notre structure possède une organisation mémoire compatible avec la norme C. Cependant, elle prend en mémoire 12 octets, mais n'en représente que 8 en termes de valeur. Cela peut être évité en réorganisant notre structure pour remplacer les octets d'alignement par des valeurs utiles. Ici, le dernier `char` peut être placé à la place du premier octet d'alignement : ainsi le `short` et le `long` seront alignés correctement et notre structure ne prendra que 8 octets en mémoire. Les compilateurs n'ont pas le droit par défaut de réaliser ce déplacement eux-mêmes.

Il est important de bien avoir conscience de ces octets d'alignement, surtout lorsque notre structure est intégrée dans une union comme dans la partie précédente.

En dépit de la difficulté supplémentaire d'organisation apportée par ces règles, la norme du C en matière de mémoire pour les structures permet cependant des actions poussées tel qu'une forme d'héritage, évoquée dans le chapitre suivant.

4 Exemple : une forme d'héritage en C

Dans ce dernier chapitre, nous allons nous intéresser à un exemple concret mettant en application les pointeurs et les types composés : une forme d'héritage comme existant en C++. Cet exemple est directement tiré du projet "Développement d'une librairie graphique pour microcontrôleur" réalisé en GE5A en 2015-2016.

4.1 Le problème des drawables

L'un des éléments 'haut-niveau' de base d'une librairie graphique sont ce qu'on appelle en anglais des drawables. Il s'agit d'une collection d'élément partageant un point commun, le fait de pouvoir être dessiné à l'écran. Le problème posé par les drawables est qu'ils présentent en effet tous des attributs communs, mais possèdent également chacun des informations particulières. Par exemple, une image et un texte partagent des informations comme la position de dessin à l'écran, la taille de la zone impactée par leur présence, etc., mais diffèrent en d'autres points, comme le fait qu'un texte nécessite une chaîne de caractère et une police d'écriture, tandis qu'une image possède un format et une description de l'image en mémoire.

Dans les langages orientés objets comme le C++ ou le Java, cette situation est souvent réglée par l'utilisation du concept d'héritage. Ce concept permet de créer une classe (ou un type) de base contenant uniquement les informations communes, puis de définir un type précis pour chaque drawable héritant de ce premier élément, c'est à dire contenant toutes ses caractéristiques, et pouvant se comporter comme lui en mettant de côté temporairement les composantes spécifiques.

En C, il n'existe pas de mécanisme de classe ou d'héritage, il n'est donc pas possible par défaut de réaliser un tel comportement. Le développeur est donc souvent obligé de se répéter pour chacun des éléments. Cependant dans le cas d'une librairie graphique, cela pose un second problème : une fois un élément créé, il est important de pouvoir l'ajouter/le retirer facilement du tampon d'écran sur lequel nous travaillons. Or, si chaque élément est défini à part, il est également nécessaire de multiplier les fonctions d'ajout et de retrait au tampon... Ce qui n'est bien sûr pas viable.

Deux solutions s'offrent donc à nous. Il est possible de créer un type `drawable` contenant les informations de tous les éléments voulus, et créer des fonctions de dessins n'utilisant que les informations pertinentes pour le type de drawable voulu. Cette solution permet d'avoir une unique fonction d'ajout et de retrait d'élément au tampon, mais signifie également que l'impact mémoire de chaque élément sera décuplé. La seconde solution consiste à jouer sur le fonctionnement des pointeurs et des structures pour implémenter notre propre système d'héritage.

4.2 Héritage de structures

Dans le chapitre sur les structures, nous avons appris que la seule liberté du compilateur au niveau de l'organisation mémoire d'une structure est l'insertion d'octets de padding. Ces insertions sont dans certains cas contournables et de toute façon prévisibles et répétables. Ce qui va nous aider ici, est que la norme du C empêche les compilateurs de réorganiser l'ordre des données : nous pouvons donc savoir avec précision où se trouve chaque composante d'une structure par rapport à son adresse de début. Cette première information sera la base de notre système d'héritage.

La deuxième information utile est le fait qu'un pointeur n'a pas besoin de garantie que le type de variable pointée est bien présent en mémoire. Partant de ce principe, nous allons pouvoir faire croire au compilateur qu'un certain type de structure est présent en mémoire tout en sachant que cela n'est pas – complètement – vrai.

La première étape est de définir les attributs communs à chacun de nos éléments. Ici, nous allons également utiliser cette méthode pour créer une liste chainée dont les éléments sont de types différents.

Dans cette exemple, la structure de base peut être définie comme suit :

```
typedef struct base_elem {
  struct base_elem * next;
  BufferType * buffer;
  unsigned short x;
  unsigned short y;
} base_Elem;
```

Pour notre exemple, l'élément de base contient donc un lien vers un prochain élément, un lien vers le tampon d'écran auquel il est associé, et une position représentée par le couple (x; y).

Nous allons maintenant pouvoir construire les deux types cités en exemples précédemment : un pour gérer des textes, et un pour gérer des images. Dans notre situation, les polices sont décrites par le type `Font`, et les tableaux d'images par le type `Bitmap`.

Le premier type, `text_Elem`, doit pouvoir passer pour un `base_Elem` dans certains cas, mais doit également contenir les informations permettant de gérer le texte. Pour se faire, on peut construire le type `text_Elem` comme suit :

```
typedef struct text_elem {
  struct base_elem * next;
  BufferType * buffer;
  unsigned short x;
  unsigned short y;

// Début des composantes spécifiques
  char * string;
  Font font;
} text_Elem;
```

En déclarant notre type `text_Elem` de cette façon, nous sommes assurés que sa représentation en mémoire commencera exactement comme celle de `base_Elem`. En particulier, en utilisant le code suivant, on pourra demander au compilateur de n'utiliser que la partie "de base" de notre `text_Elem` :

```
text_Elem monTexte;
base_Elem * adresseElem;

adresseElem = (base_Elem *) &monTexte;
```

A partir de là, l'élément contenu à l'adresse de `monTexte` sera considéré par le compilateur comme une variable de type `base_Elem` tant qu'on y accédera par le pointeur. Ce pointeur pourra par exemple être passé en paramètre d'une fonction qui prendrait un `base_Elem *` en paramètre.

De la même manière, on peut désormais construire notre type `image_Elem` :

```
typedef struct image_elem {
  struct base_elem * next;
  BufferType * buffer;
  unsigned short x;
  unsigned short y;

// Début des composantes spécifiques
  unsigned char format;
  Bitmap bitmap;
} image_Elem;
```

Désormais nous possédons deux types différents pouvant tout deux être interprétés par le compilateur comme des `base_Elem`, permettant ainsi de les utiliser comme paramètre d'une même fonction, à condition que cette fonction n'utilise que les éléments communs.

En effet, comme l'ordre des éléments est intouché à la compilation et que la position de chaque attribut est déterministe, `base_Elem.next`, `text_Elem.next` et `image_Elem.next` se trouvent exactement à la même position relativement à l'adresse de la structure, et il en va de même pour tous les attributs communs, puisque les attributs spécifiques se situent après les attributs communs.

Cette méthode manque cependant de flexibilité pour deux raisons : il est nécessaire de réécrire dans le bonne ordre sans se tromper la structure de base à chaque nouvel élément codé, et si un attribut commun doit être ajouté, il faudra modifier la déclaration de tous les types héritant de notre élément commun.

Ces deux limitations peuvent cependant être facilement contournées en utilisant des macros pré-processeurs. En effet, il suffit d'encapsuler la description de l'élément de base dans une macro :

```
#define BASE_HERIT struct base_elem *next; \
BufferType * buffer; \
unsigned short x; \
unsigned short y;

...
```

```
typedef struct base_elem {
  BASE_HERIT
} base_Elem;

...

typedef struct text_elem {
  BASE_HERIT
// Début des composantes spécifiques
  char * string;
  Font font;
} text_Elem;
```

Note: l'antislash en fin de ligne permet d'écrire une macro sur plusieurs lignes

4.3 Récupération du type d'origine

Maintenant que nous avons un moyen de gérer l'héritage avec des structures, il peut être intéressant de trouver un moyen de connaitre le type d'origine lorsqu'on manipule un pointeur sur `base_Elem`. Pour se faire, il suffit d'associer à chaque type hérité un identifiant unique. Heureusement, la C propose déjà un mécanisme permettant de généré un tel identifiant : les `enum` ! En effet, par définition chaque entrée d'une `enum` possède une valeur unique.

On commence donc par créer une `enum` listant les identifiants de types :

```
typedef enum elem_id {
  baseId = 0,
  textId,
  imageId,
  maxId
} elem_Id;
```

Par la suite, on ajoute un attribut de type `elem_Id` dans la macro de déclaration de l'élément de base. Il sera dès lors possible d'utiliser cette attribut par exemple dans un `switch ... case` pour pouvoir récupérer le type d'origine d'un élément.

La dernière étape consiste à s'assurer que l'identifiant est initialisé correctement pour chaque élément. Pour cela, nous allons profiter de la capacité d'une structure à être copiée pour créer des fonction de constructions.

En effet, contrairement à un tableau dont la valeur est l'adresse de la première case, la valeur d'une structure est directement composée de la concaténation de chacun de ses attributs, permettant la comparaison champ par champ d'une structure mais également sa copie directe. Et si une variable peut être copiée, elle peut aussi être la valeur de retour d'une fonction.

On pourra alors imaginer une fonction permettant de construire un élément de type `text_Elem` comme suit :

```
text_Elem textElem(char * string, Font font) {
  test_Elem tmp_text;
  tmp_text.id = textId; // On s'assure que l'id est correct
  tmp_text.string = string;
  tmp_text.font = font;

  return tmp_text;
}
```

Grâce à cette méthode, tant que l'utilisateur crée bien ses éléments à l'aide des constructeurs, l'identifiant sera correct.