Compléments et autres types de données

type enumération

```
typedef enum {ROUGE, NOIR, VERT,
              JAUNE, BLEU} Couleur;
Couleur c1, c2;
```

définit des constantes (de type `int`) pour les valeurs acceptables du type `Couleur`. On pourra ensuite tester l'égalité des valeurs de ces variables avec les constantes énumérées :

`if (c1==JAUNE) ...` ou
`swich (c2){ case ROUGE: ...`), etc.

type enumération

Les valeurs des constantes sont sans importance. Elles sont définies par le compilateur de 1 en 1 à partir de zéro. Mais

```
typedef enum {ROUGE, NOIR=3, VERT,
              JAUNE, BLEU=8} Couleur;
```

donnera la valeur `0` à `ROUGE`, `3` à `NOIR`, `4` à `VERT`, `5` à `JAUNE` et `8` à `BLEU`.

Remarque: les constantes étant de type `int`, elles peuvent être négatives.

Plan

Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Déclaration de variables

Une déclaration de variable indique surtout son type. Toute variable doit être déclarée avant d'être utilisée. Une définition ajoute une allocation mémoire à la déclaration et peut être suivie d'une initialisation.

attribut qualificatif type listeVar;

ex:

```
static const double pi=3.14;
register int a1,a2;
```

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Visibilité des variables

Il y a 4 attributs `extern`, `static`, `auto` et `register` déterminant la durée de vie et la visibilité des variables :

- Les variables locales ne sont visibles que depuis leur bloc de définition.
- Les variables globales sont visibles dans toute l'unité de compilation.
- Les variables globales peuvent être rendues visibles dans d'autres unités ou partout.

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Attribut `extern`

Les variables globales à un fichier ont une adresse fixe et sont statiques. Déclarées en tête, elles sont visibles depuis toutes les fonctions du fichier.

Mais attention: les variables globales de mêmes noms déclarées dans des fichiers différents ne sont pas les mêmes, et ont des adresses différentes.

Pour partager une variable globale, on utilise l'attribut `extern`.

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Attribut `extern`

En effet, une variable globale (ou une fonction) ne peut être **définie** qu'une seule fois dans un programme.

Si on veut l'utiliser dans plusieurs unités, il faut la **déclarer extern**, et c'est l'éditeur de lien qui se charge de déterminer son adresse pour créer le code exécutable.

Il ne peut y avoir dans ce cas qu'un seul fichier définissant la variable.

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

fichier.h

Les fichiers d'entête que l'on inclut en début de fichier pour faire appel aux fonctions d'une bibliothèque (ou à une fonction définie ailleurs) contiennent les déclarations `extern` de variables globales ou de fonctions pouvant être utilisées.

ex:

```
extern double f2(float x);
exterm int globalMax;
```

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

L'attribut `static`

Les variables locales déclarées `static` ne sont pas allouées dynamiquement dans la pile et ont une adresse fixe dans l'exécutable.

Elles sont initialisées lors du premier passage dans leur bloc de définition, et conservent ensuite la valeur acquise au précédent passage.

Leur durée de vie est celle du programme exécutable.

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

`register` et `volatile`

Ces qualificatifs ont destinés à l'attribution du stockage dans la génération de code optimisé.

- `register` demande à optimiser le code en utilisant le stockage dans un registre (mais cela ne produit pas toujours l'effet escompté!).
- `volatile` demande la suppression des optimisations pouvant nuire à la validité de codes asynchrones accédant à cette variable.

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

déclaration `const`

Ce qualificatif indique qu'une variable ne pourra pas être modifiée. La déclaration doit être accompagnée de son initialisation :

```
static const double pi=3.14159;
```

On peut aussi qualifier `const` les paramètres formels d'une fonction. Ils sont alors initialisés à l'appel, mais ne peuvent pas être modifiés dans le bloc de définition (corps) de la fonction.

Lecture et Ecriture dans des fichiers

en langage C

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Un programme doit souvent

- lire des données (texte, nombres, images, sons, etc.)
- sauvegarder des résultats.

On peut le faire en lisant ou en écrivant ces données dans des fichiers.

Un **fichier** est un ensemble d'octets stockés en mémoire. Le support mémoire peut être le disque dur, une clef usb, etc.

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Fichiers

La communication entre un programme et un fichier commence toujours par l'ouverture du fichier. Pour clore la communication on effectue ensuite la fermeture du fichier.

L'envoi et la lecture de données se fait via un jeu de fonctions définies dans la librairie standard (ou une autre librairie).

Les Flots

Il y a 2 modes de communication avec les fichiers : une communication de haut niveau, et une communication de bas niveau.

La communication de haut niveau s'effectue par l'intermédiaire d'un flot à l'aide d'une structure `FILE`. (Celle de bas niveau, à l'aide d'un `FileDescriptor`).

Certains flots sont prédéfinis : le flot d'entrée standard `stdin`, le flot de sortie standard `stdout` et celui de sortie d'erreurs `stderr`.

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Type FILE

La structure de données associée à un flot est de type `FILE`. Elle est manipulée avec un pointeur :

```
FILE *f ;
```

On parle parfois abusivement de f comme d'un pointeur de fichier. Elle contient des informations comme l'indicateur de position dans le fichier, etc.

Ouverture d'un flot

```
FILE *fopen(const char *filename,
            const char *opentype)
```

- "r" : ouvre un fichier existant en lecture seule
- "w" : ouvre un fichier en écriture seule (l'écrase s'il existe, sinon il est créé)
- "a" (*append*): écrit à la fin d'un fichier existant.

L'ajout d'un + est l'ouverture en lecture/écriture. En cas de succès la fonction retourne un pointeur sur une structure FILE, sinon elle retourne NULL.

Fermeture d'un flot

```
int fclose (FILE * filename)
```

La fonction retourne `0` en cas de succès et `EOF` en cas d'erreur (constante définie dans `stdio.h`). Par exemple, s'il n'y a plus d'espace mémoire disponible.

Il est important de fermer un fichier quand on a fini de l'utiliser.
Cela permet d'économiser la mémoire.

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Modes d'accès aux données

Les données qui transitent entre les fichiers et le programme sont des suites d'octets. Il y a deux modes d'accès à ces données :

- le mode binaire : on lit/écrit les octets par paquets de taille correspondant aux données qu'on veut lire.
- le mode texte : on lit/écrit les octets comme suite de caractères

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Mode texte

Les caractères sont stockés dans les fichiers par leur code ASCII (son code de caractère `char`).

Avec un langage évolué tel que le C :

- pas besoin de connaître le code du caractère
- pas besoin de connaître la représentation des nombres dans les mémoires

Plan

Variables & compilation

Lecture /Ecriture dans des fichiers

Piles, Files et Arbres

Mode texte

On va lire ou écrire des valeurs de variables dans des fichiers en mode texte, à l'aide des fonctions formatées

- `fprintf`
- `fscanf`

La syntaxe est la même que celle de `printf` et `scanf` qui utilisent `stdin` et `stdout`.


```
int fprintf(FILE *stream,
            const char *format, ...)
```

Envoie sur le flot la chaîne de caractères formatée selon la chaîne de format et les valeurs des variables passées en arguments. Retourne le nombre de caractères écrits en cas de succès (sans compter le caractère `\0`).

```
int fscanf(FILE *stream,
           const char *format, ...)
```

Lit les entrées obtenues à partir du flot selon la chaîne de format. Assigne les valeurs lues aux variables dont les adresses sont passées en argument. Renvoie le nombre d'assignation effectuées avec succès.

Exemples

- **Écriture**

```
fprintf(FILE *filename,
          const char *format,…);

Ex: double a;
    fprintf(filename,"%lf", a);
```

- **Lecture**

```
fscanf(FILE *filename,
          const char *format,…);

Ex: double a;
    fscanf(fichier,"%lf",&a);
```



```
#include <stdio.h>
void main() {
   int i;
   double tab[10];

   FILE *fichier;
/* Ouverture du fichier essai.txt en
   lecture */
   fichier = fopen("essai.txt","r");
   if (fichier != NULL) {
        for(i=0;i<10;i++)
             fscanf(fichier,"%lf\n",tab+i);
        fclose(fichier);
   }
   for(i=0;i<10;i++
        printf("%lf\n",tab[i]);
}
```

Lire caractère par caractère

```
int fgetc(FILE *stream)
```

Lit le caractère suivant sur le flot et le retourne sous forme d'un `unsigned char` typé en `int`. Renvoie `EOF` si la fin du fichier est atteinte ou si une erreur est survenue.

On peut replacer le caractère lu dans le fichier avec `ungetc` :

```
int ungetc(int caractere,
           FILE *stream);
```

(Le caractère est converti en `unsigned char`).

Ecrire caractère par caractère

```
int fputc(int aCharVal,
          FILE *stream)
```

Ecrit le caractère `aCharVal` sur le flot en le typant comme `unsigned char`.
Retourne le caractère ou EOF en cas d'erreur.

Lire des chaines de caractères

```
char * fgets(char *s, int n,
                    FILE *stream)
```

- Lit sur le flot jusqu'à (n – 1) caractères, ou moins si on rencontre un caractère `\n`.
- Recopie les caractères lus à partir de l'adresse pointée par s (la mémoire doit être allouée) et rajoute un `\0` à la suite des caractères recopiés depuis le flot.

Retourne `s` en cas de succès et NULL en cas d'erreur ou si la fin du fichier est atteinte.


```
#include <errno.h>
char *s;
FILE *fichier;
char ligne[240];

s = (char *) malloc(sizeof(char) * 240);
fichier = fopen("file.txt", "r");
if (fichier == NULL) {
     printf("can't open file.txt ! %d:
\ %s\n", errno, strerror(errno));
   return errno;
}
while ((s = fgets(ligne,240,fichier)) != 0)
   printf("%s\n", ligne);
if (fclose(fichier) == EOF)
   printf("closing error file.txt\n");
```

Ecrire des chaines de caractères

```
int fputs(const char *s, FILE *stream)
```

Ecrit la chaîne de caractère dans le flot en omettant le `\0` final.
Renvoie zéro en cas de succès et EOF en cas d'erreur.

Mode binaire

Le format binaire occupe beaucoup moins de place en mémoire que le format texte.

En effet, soit le nombre écrit en mode texte avec les caractères `1.345643355454E-18`

- En mode texte, il est stocké au format ASCII avec `20` octets (un octet par caractère)
- En mode binaire, il est stocké en tant que `double` et occupera seulement 8 octets.

Ouverture en binaire

En mode binaire, comme il s'agit d'une recopie directe de la mémoire, on n'a pas non plus à savoir comment est codé chaque nombre.

Mais pour les fichiers en format binaire, on utilise un autre mode d'ouverture :

- "rb" (read) : lecture
- "wb" (write) : écriture (le fichier est écrasé s'il existe)
- "ab" (append) : écriture à la fin d'un fichier existant

Lecture en mode binaire

```
int fread(void *destination,
          int taille_type,
          int nombre, FILE *flot)
```

Lit, en un seul appel, un bloc d'un certain nombre de données d'un type d'une certaine taille, le range dans un tableau de données de ce type (destination) et renvoie un entier qui est le nombre d'éléments effectivement lus sur le flot.

Ecriture en mode binaire

```
int fwrite(void *source,
           int taille_type,
           int n, FILE *flot)
```

Ecrit, en un seul appel, tout un bloc de données de taille `taille_type` rangées dans le tableau `source`, et retourne un entier qui est le nombre d'éléments effectivement écrits dans le flot.

Types abstraits
Piles et files

Type abstrait

Un type abstrait ne décrit pas la manière dont il est représenté dans un algorithme (son implémentation ou réalisation) mais :

- un nom de type
- un ensemble d'opérations avec
  1. un nom d'opérateur (ou fonction)
  2. une signature de l'opérateur
- des propriétés concernant les résultats des opérateurs

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Le type abstrait Pile

Une pile est un ensemble d'objets ordonnés, de type élément (le type abstrait élément est supposé défini) .

Les éléments sont posés les uns par dessus les autres, et seul l'élément situé en haut de la pile est directement accessible.
ex: une pile d'assiettes.

Opérateurs et propriétés d'une Pile

- nom : *estPileVide*
  signature : *Pile* → *Booleen*
  estPileVide(p) est *Vrai* si la pile p ne contient aucun element.
- nom : *sommet*
  signature : *Pile* → *Element*
  sommet(p) renvoie la valeur de l'élément situé en haut de la pile.

Opérateurs et propriétés d'une Pile

- nom : *empiler*

signature : *Element* x *Pile* → *Pile*
empiler(e,p) renvoie la pile après l'ajout d'un élément e au sommet de la pile.

- nom : *depiler*

signature : *Pile* → *Pile*
depiler(p) renvoie la pile après suppression de l'élément au sommet.

Opérateurs et propriétés d'une Pile

- nom : *hauteur*

signature : *Pile* → *Entier*
hauteur(p) renvoie le nombre d'éléments dans la pile.

Il y a un ensemble de propriétés comme :
hauteur(empiler(e,p))=hauteur(p)+1
hauteur(depiler(p)) = hauteur(p)-1

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Le type abstrait File

Le type Pile est parfois appelé FILO (First In, Last Out) car le premier élément empilé sera le dernier à être dépilé.

Le type File est aussi un ensemble d'éléments ordonnés (ils sont en file), mais sa gestion est différente d'une pile et qualifiée de FIFO (First Int, First Out).

Plan

Variables & compilation

Lecture /Ecriture dans des fichiers

Piles, Files et Arbres

Le type abstrait File

Les deux types sont très voisins. La différence concerne le retrait d'un élément. Pour une pile il s'effectue du même côté que l'ajout (en haut, ou en tête si l'on préfère l'appeler ainsi), alors que pour une file l'ajout se fait en tête, et le retrait en queue.

Opérateurs et propriétés d'une File

Les opérateurs sont donc les mêmes, (sauf pour l'extraction), mais on peut changer leurs noms :

- *estFileVide*, qui retourne *Vrai* si la file ne contient pas d'élément est l'analogue de *estPileVide*.
- pour *sommet*, on dira *dernier*, qui renvoie le dernier élément ajouté.

Opérateurs et propriétés d'une File

- *ajouter(e,f)* renvoie la file *f* après ajout d'un élément *e* en tête de file (il correspond à *empiler(e,p)*).
- *retirer(f)* renvoie la file *f* après suppression de l'élément situé en queue.
- *longueur(f)* renvoie le nombre d'éléments dans la file (correspond à *hauteur* pour une pile).

Plan

Variables & compilation

Lecture /Ecriture dans des fichiers

Piles, Files et Arbres

Implémentations

Les deux types étant très voisins, on peut en donner des implémentations similaires, utilisant une structure de données commune.

En particulier, on peut les implémenter avec un tableau et un indice, ou encore, avec une liste chaînée d'éléments.

Plan

Variables & compilation

Lecture /Ecriture dans des fichiers

Piles, Files et Arbres

Implémentations

On peut implémenter les piles et les files à partir d'une liste chaînée de structures, en maintenant en sus un pointeur sur le début, et un pointeur sur la fin.

Si elles sont bornées, on peut utiliser un tableau et un indice et allouer directement la mémoire.

structures pour une liste

```
struct element {
  int val;
  struct element *suiv;
}
typedef struct element  element_t;

struct liste_e {
  struct element *premier;/*firstIn*/
  struct element *dernier;/*lastIn*/
  int taille;
}
typedef struct liste_e liste_t;
```


fonctions à implémenter (en TP)

```
element_t *  creerElement(int v);
void detruireElement (element_t * e);
liste_t * creerListe();
void detruireListe(liste_t* l);
int listeVide(liste_t* l);
int listeTaille(liste_t* l);
liste_t* listeAjouterDeb(liste_t* l,
                         int x);
void listeAjouterFin(liste_t* l, int x);
element_t * listeExtrDeb(liste_t * l);
void afficher(liste_t* l);
```


et ensuite pour une Pile

```
#include "liste.h"

typedef liste_t  pile_t;

pile_t* pile_creer();
int pile_vide(pile_t* p);
int pile_taille(pile_t* p);
pile_t * empiler(pile_t* p, int x);
pile_t * depiler(pile_t* p);
void pile_detruire(pile_t* p);
```

et ensuite pour une File

```
#include "liste.h"

typedef liste_t  file_t;

file_t* file_creer();
int file_vide(file_t* f);
int file_taille(file_t* f);
file_t* ajouter(file_t* f, int x);
file_t* retirer(file_t* f);
void pile_detruire(file_t* f);
```


Arbres et arbres binaires

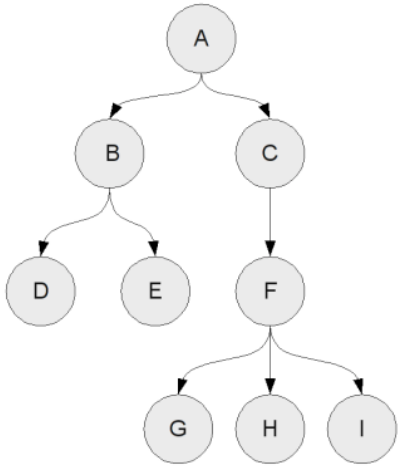
Le type abstrait Arbre

Vocabulaire:

Un *arbre* est une structure constituée de *noeuds*, qui peuvent avoir des *enfants* (qui sont d'autres *arbres*). Certains noeuds portent un nom particluer:

- Un *noeud* qui n'a pas d'enfant s'appelle une *feuille*
- Le *noeud* initial, qui n'est l'enfant d'aucun autre, s'appelle la *racine*.

Type abstrait Arbre



- Ici, la *racine* est le *noeud* A.
- Le *noeud* B a deux enfants, C un seul, et le noeud F, trois.
- Les *noeuds* D et E, G,H,et I sont des *feuilles*.

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Type ABin (arbre binaire)

Un *arbre binaire* est un arbre dont chaque noeud a au plus deux enfants: un *sous-arbre gauche* et un *sous-arbre droit*, appelés aussi *fils gauche* et *fils droit*.

Un arbre binaire peut aussi être *vide*, et dans ce cas il n'a ni valeur, ni sous-arbres.

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Type ABin (arbre binaire)

Opérateurs :

- *Vide: {} → ABin*
- *Noeud: ABin × ABin → ABin*
- *EstVide: ABin → Booleen*
- *SAG, SAD : ABin → ABin*

Préconditions :
SAD(t) et *SAG(t) sont définis seulement si t n'est pas vide, i.e. si non(EstVide(t)).*

Plan
Variables & compilation
Lecture /Ecriture dans des fichiers
Piles, Files et Arbres

Type ABin (arbre binaire)

Propriétés :

- *EstVide(Vide()) =VRAI*
- *EstVide(Noeud(g,d)) =FAUX*
- *SAG(Noeud(g,d)) = g*
- *SAD(Noeud(g,d)) = d*
- *Noeud(SAG(t),SAD(t)) = t si non(EstVide(t)).*