UE Structures de données (LU2IN006)

Rappels de C

Nawal Benabbou

Licence Informatique - Sorbonne Université

2022-2023

Contenu de ce cours de rappels

- Cases mémoires et pointeurs
- Tableau C et allocation mémoire (statique VS dynamique)
- Manipulation du type `struct` et allocation mémoire (statique VS dynamique)
- Entrées/sorties (saisie clavier, écriture/lecture de fichiers, etc.)
- Compilation séparée (Makefile)

Adresse mémoire, type et pointeur

Adresse mémoire et type

La mémoire d'un ordinateur peut être vue comme une séquence de bits (0 ou 1) mis les uns après les autres. Un emplacement dans la mémoire est ainsi repéré par son **adresse physique** et sa **taille** (en bits, octets...). Si l'on ne connaît pas le codage utilisé à cet emplacement, on ne pas lire/utiliser le contenu stocké à cet emplacement. On appellera **type** le codage d'un emplacement (qui correspond à une taille pré-définie). On appellera **case mémoire** un emplacement typé.

Pointeur

Un pointeur est une variable contenant l'adresse d'un emplacement mémoire (on dit qu'il "pointe" vers un emplacement mémoire). Un **pointeur sur un type** est un pointeur dont la valeur correspond à une case mémoire (c'est-à-dire un emplacement typé). Un pointeur sur un type se déclare en ajoutant une étoile derrière le type.

Exemple : l'instruction `int* p;` déclare un pointeur sur un entier.

Remarque : l'accès à une case mémoire est en $\Theta(1)$.

Manipulation de pointeurs

Les opérateur & et $*$

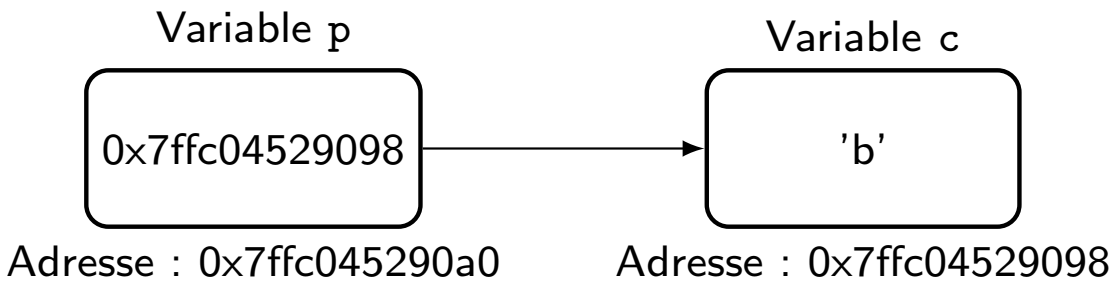
En langage C :
- l'opérateur & permet de récupérer l'adresse de la case mémoire d'une variable.
- l'opérateur $*$ permet d'accéder à la case mémoire vers laquelle pointe un pointeur (on parle de déréférencement).

Exemple :

- `char c = 'a';` permet de réserver une case mémoire de la taille d'un `char` (1 octet), contenant le caractère 'a' (codé en binaire).
- `char* p;` permet de réserver une case mémoire de la taille d'un pointeur sur un caractère (ce qui correspond souvent à la taille d'un entier).
- `p = &c;` permet de récupérer l'adresse de la variable `c` et de la stocker dans `p` (on dit que `p` pointe vers la case mémoire de `c`).
- `printf("%c\n",*p);` permet d'afficher la valeur stockée à la case mémoire vers laquelle pointe `p` (ici cela affiche le caractère 'a').
- `*p = 'b';` permet de changer la valeur de la case mémoire pointée par `p` (ici cela revient à écrire `c = 'b'`).

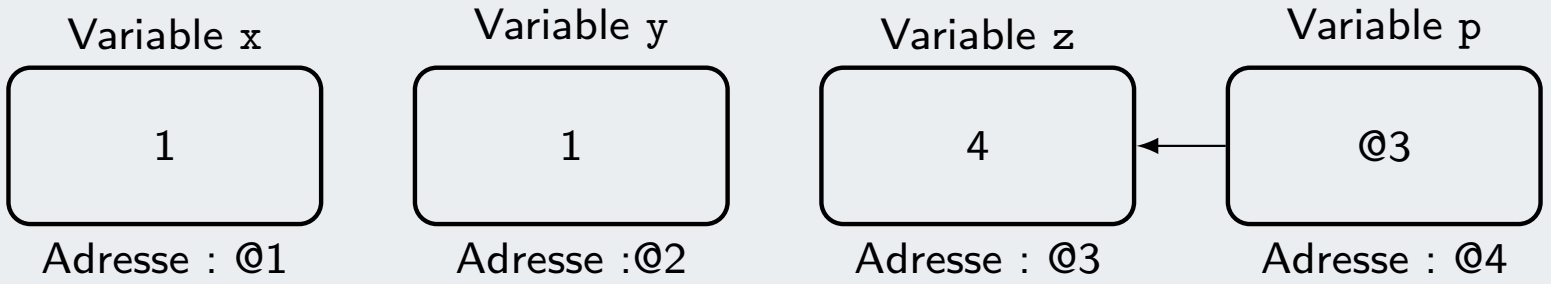
Représentation graphique de la mémoire

On représente souvent les cases mémoires par des boîtes (contenant la valeur des variables), et les pointeurs par des flèches. Pour l'exemple précédent, on obtient à la fin la représentation suivante :



Exercice : Donner la représentation graphique

```
1  int x=1,y=2,z=3;
2  int* p = &x;
3  y = *p;
4  p = &z;
5  *p = 4;
```



Mémoire contiguë et tableaux

Définition : tableau

En langage C, un tableau est un pointeur sur une **zone contiguë** de la mémoire composée d'éléments **du même type**. Plus précisément, un tableau est représenté par un pointeur sur le premier élément, les autres éléments pouvant être obtenues par l'arithmétique des pointeurs.

Remarque : l'instruction `int* t;` permet de déclarer un pointeur sur un entier, ou un tableau d'entiers... la différence se fera au moment de son initialisation.

Sur l'arithmétique des pointeurs

Si t est un tableau de type `int*`, alors `t+i` est un pointeur sur le `i`ème entier du tableau `t`. En effet, l'opération `+i` correspond à se déplacer dans la mémoire de `i` fois le nombre d'octets permettant de représenter un entier. Ainsi, pour accéder à la valeur du `i`ème élément, on peut écrire `t[i]` ou encore `*(t+i)`.

Exemple : représentation graphique du tableau d'entiers `int t[3] = {2,5,1};`

Allocation mémoire d'un tableau (version statique)

Attention : écrire `int* t;` déclare un tableau d'entiers, mais ne réserve pas l'espace mémoire pour les éléments du tableau. Il faut réserver de la mémoire, ce qui s'appelle l'allocation mémoire.

Allocation statique

Un tableau peut être alloué statiquement de la manière suivante :

- `int t[4];` alloue l'espace mémoire pour un tableau de 4 entiers, sans initialiser les valeurs.
- `int t[4]={2,4,7,1};` permet de faire les deux.
- `int t[4]={2,4};` alloue le tableau en entier et initialise seulement les deux premières valeurs (les autres sont mises à zéro).
- `int t[4]={};` alloue le tableau en entier et le remplit avec des zéros.

Remarque

- On ne peut pas écrire `int t[n];` quand `n` est une variable, car l'allocation statique réserve l'espace mémoire lors de la compilation (or la valeur de `n` est connue uniquement à l'exécution).
- On peut écrire `int[N_MAX] t;` quand N_MAX est une constante définie avec #define (pas avec `const`) car #define est une commande préprocesseur qui remplace toutes les instances de N_MAX par sa valeur avant la compilation du programme.

Allocation mémoire d'un tableau (version dynamique)

Allocation dynamique

Un tableau peut être alloué dynamiquement en utilisant la fonction `malloc` qui prend en paramètre le nombre d'octets à réserver. Par exemple :

- `malloc(12);` réserve un espace de 12 octets.
- `malloc(5*sizeof(int));` réserve l'espace nécessaire pour stocker 5 entiers de type `int`.

Attention : la fonction `malloc` alloue de l'espace non typé (elle retourne un pointeur de type `void*`). Pour donner un type à la mémoire allouée, il faut "caster" le retour de la fonction `malloc`. Par exemple, on peut écrire :

- `int* t1 = (int*) malloc(5*sizeof(int));`
- `double* t1 = (double*) malloc(n*sizeof(double));`

Remarques :

- Dans le dernier exemple, la taille de `t1` est définie à partir d'une variable (nommée `n`), ce qui n'aurait pas été possible avec un allocation statique. Par exemple, cela permet à un utilisateur de saisir sa taille au clavier.
- Par ailleurs, allouer la mémoire dynamiquement permet de changer la taille d'un tableau en cours d'exécution (une ou plusieurs fois, avec la fonction `realloc`), et donc permet une meilleure gestion de la mémoire.

Libération mémoire d'un tableau alloué dynamiquement

Quand un espace alloué dynamiquement n'est plus utilisé, il faut libérer la mémoire pour permettre au programme (ou à d'autres programmes) de l'utiliser. Dans certains langages, ce travail est réalisé par un "ramasse-miette" (garbage collector), ce qui n'est pas optimal car seul le programmeur sait exactement ce qui est à libérer et quand.

Désallocation (libération de la mémoire)

En langage C, on désalloue avec la fonction `free` :

- `free(p);` permet de libérer l'espace mémoire pointé par `p`. Il peut contenir un type simple, ou une structure plus élaborée.
- `free(t);` permet de libérer la zone contiguë (tableau) pointée par `t`.

Gestion des problèmes d'allocation

Comme l'allocation mémoire a lieu pendant l'exécution, des problèmes peuvent surgir si le système ne peut plus allouer de mémoire à un moment donné. Cela peut arriver quand l'espace demandé est trop grand ou que des zones mémoires n'ont pas été libérées. Pour éviter les *segfault*, on traite les éventuelles erreurs :

```
1 double* p = (double*) malloc(n*sizeof(double));
2 if (p == NULL){ // ou p == 0
3     printf("Erreur durant l'allocation m'emoire");
4     return NULL;
5 }
```

Le type `struct` en C

Les tableaux permettent de stocker des données de même type. Comment faire pour rassembler des variables qui peuvent avoir des types différents ?

Une réponse : le type `struct`

En langage C, le type `struct` permet de grouper dans une case mémoire des variables appelées *champs* ou *membres*, potentiellement de types différents. En particulier, cela permet au programmeur de créer un nouveau type de variable, composé de variables de types différents, mais qui ont un lien sémantique fort pour le programmeur. Par exemple, on peut définir le type `struct personne` :

```
1 typedef struct personne {
2     char* nom;
3     char* prenom;
4     int* dateDeNaissance;
5     int sexe;
6     struct personne* conjoint;//un pointeur sur le meme type
7 } Personne;
```

Remarque : Le mot clé `typedef` permet de créer des types synonymes. Ce n'est pas nécessaire, mais sans lui, il faudrait écrire "`struct personne`" à chaque définition de variable, ce qui est un peu lourd... En ajoutant "`typedef [...] Personne;`", on peut simplement écrire "`Personne`" à la place.

Manipulation de struct et allocation mémoire

Accès aux champs

Pour une variable s de type struct, l'accès aux champs se fait avec un point.
Pour une variable p de type pointeur sur struct, on utilise une flèche :

```
1 Personne s; // declaration d'une variable de type personne
2 s.sexe = 0; // initialisation du champ ''sexe''
3 Personne* p = &s; // recuperation de l'adresse de s
4 p->sexe = 1; // modification du champ ''sexe''
```

Allocation statique VS allocation dynamique

Pour faire une allocation entièrement statique, il faut que dans la déclaration du nouveau type, les tableaux soient alloués statiquement.

```
1 typedef struct personne {
2     char nom[20];
3     char prenom[20];
4     int dateDeNaissance[3];
5     int sexe;
6     struct personne* conjoint;
7 } Personne;
```

Pas besoin pour les allocations dynamiques.

Allocation mémoire de `struct` (version statique)

Allocation statique

On peut faire des allocations mémoire et initialiser les champs comme suit :

```
1  struct personne p1 = {"A", "B", {12,5,1995}, 0, NULL};
2  Personne p2 = {"D", "E", {}, 1, &p1};
3  memcpy(p2.dateDeNaissance, dateAcopier, 3*sizeof(int));
4  p1.conjoint = &p2;
5
6  Personne p3;
7  (&p3)->sexe = 1; //on pourrait ecrire p3.sexe = 1;
8  strcpy(p3.nom,"F");
9  strcpy(p3.prenom,"G");
10 p3.dateDeNaissance[0]= 15;
11 p3.dateDeNaissance[1]= 10;
12 p3.dateDeNaissance[2]= 1998;
13 p3.conjoint = NULL;
```

Cette solution n'étant pas très souple au niveau de la gestion mémoire, on peut préférer l'allocation dynamique pour allouer des types élaborées.

Allocation mémoire de struct (version dynamique)

```
1  typedef struct personne {
2      char* nom;
3      char* prenom;
4      int* dateDeNaissance;
5      int sexe;
6      struct personne* conjoint;
7  } Personne;
```

Allocation dynamique : allouer aussi les champs contenant des tableaux

On peut faire des allocations mémoire et initialiser les champs comme suit :

```
1  Personne* p = (Personne*) malloc (sizeof(Personne));
2  p->nom = (char*) malloc (sizeof(char)*tailleChaine);
3  strcpy(p->nom,chaineACopier);
4  p->prenom = "B"; //plus court, quand la chaine est donnee
5  p->dateDeNaissance = (int*) malloc (3*sizeof(int));
6  memcpy(p->dateDeNaissance, dateACopier, 3*sizeof(int));
7  p->sexe = 0;
8  p->conjoint = NULL;
```

Remarque : on peut mixer allocation statique et dynamique selon les besoins. Par exemple, le tableau dateDeNaissance sera toujours de taille 3. L'allouer statiquement (en écrivant int dateDeNaissance[3] ;) permet de retirer la ligne 5.

Tableau de struct

Grouper des struct

On a vu que le type `struct` permet de créer des nouveaux types de variables, comme par exemple le type `Personne`. On peut maintenant grouper des variables de ce type dans des tableaux, par exemple pour créer une base de données contenant les personnes inscrites sur un site de rencontre en ligne.

Un exemple de tableau de `struct` :

```
1 Personne tab[2]={{"A","B",{},0,NULL}, {"C","D",{},1,NULL}};
```

Observation

Si chaque élément `struct` est consommateur de mémoire, on ne choisira pas d'allouer de la mémoire contiguë pour stocker ces éléments. On préférera un "tableau de pointeurs sur struct" (le tableau ne contenant que des pointeurs).

Un exemple de tableau de pointeurs sur `struct` :

```
1 Personne p1 = {"A","B",{},0,NULL};
2 Personne p2 = {"C","D",{},1,NULL};
3 Personne** tab = (Personne **) malloc(2*sizeof(Personne*));
4 tab[0]=&p1;
5 tab{1]=&p2;
```

Entrées/sorties

Entrées/sorties en langage C

On appelle entrées/sorties les échanges de données entre la mémoire de la machine et ses périphériques (clavier, écran, disque dur, imprimante, réseau...). Le type FILE permet de stocker les informations relatives à la gestion d'un flux de données, comme par exemple la position actuelle de la tête de lecture/écriture, le type d'accès, l'état d'erreur etc.

Entrées/sorties standards

Trois flux pré-ouverts :

- `stdin` : flux standard d'entrée de l'application (clavier par défaut).
- `sdtout` : flux standard de sortie (console de l'application par défaut).
- `stderr` : flux standard d'erreur (console de l'application par défaut).

Saisie au clavier : scanf

La fonction scanf

La fonction `scanf` est une fonction de la bibliothèque `<stdio.h>` permettant de lire des données formatées, en provenance du flux `stdin` (entrée standard). Cette fonction prend en paramètre une chaîne de caractères décrivant le format du flux entrant attendu, ainsi qu'une liste de pointeurs pour stocker le résultat de la lecture. L'acquisition s'arrête après avoir rencontré le caractère `'\n'` (saut de ligne). Pour chaque conversion réussie, `scanf` place l'objet converti dans l'espace pointé par le pointeur associé.

Exemple : lecture d'un entier suivi d'une chaîne de caractères.

```
1 int i;
2 char chaine[20];
3 scanf("%d %s", &i, chaine);
```

Observation : la fonction scanf est plus complexe qu'il ne paraît.

Par exemple, on peut avoir des problèmes quand :

- on fait une petite erreur de saisie (interruption de lecture).
- on souhaite lire une chaîne de caractères avec des espaces (problèmes liés aux caractères "blanc").
- on entre une chaîne de caractères trop longue (saisie non sécurisée).

Scanf et les erreurs de saisie

Si en cours d'analyse, une des conversions échoue, la fonction `scanf` s'arrête brutalement, sans effectuer les conversions suivantes. Le morceau de chaîne de caractères non traité reste dans la mémoire tampon. Par exemple :

```
1  int code, id;
2  char nom[20];
3  printf("Entrer votre code postal et identifiant.\n");
4  scanf("%d %d",&code,&id); //Saisie : 75001 258p5 (Entree)
5  printf("Saisie : %d %d\n", code, id); //Affiche : 75001 258
6  printf("Entrer maintenant votre nom de famille.\n");
7  scanf("%s",nom); // le programme ne s'arrete pas...
8  printf("Bonjour %s\n", nom); //Affiche "Bonjour p5"
```

Solution possible : vider la mémoire tampon après chaque appel à la fonction `scanf`, et vérifier le nombre de conversions réussies.

```
1  int n = scanf("%d %d",&code,&id);
2  char c = getchar();//lit un caractere dans la memoire tampon
3  while (c!='\n'){
4      c = getchar();
5  }
6  if (n!=2){
7      printf("Erreur durant la lecture");
8      return 0; //on pourrait demander une nouvelle saisie
9  }
```

Remarque : Sur l'exemple, le test en ligne 6 ne suffit pas à détecter l'erreur...

Scanf et la gestion des blancs

Différences entre les formateurs %s et %c

Le formateur %s :

- s'arrête au premier caractère "blanc" (espace, tabulation, saut de ligne).
- supprime les caractères "blanc" en début de chaîne.
- ajoute le caractère de fin '\0'.

Le formateur %c :

- n'ignore pas les caractères "blanc".
- nécessite de connaître la taille exacte de la chaîne.
- n'ajoute pas le caractère de fin '\0'.

Exemple :

```
1  char nom[20];
2  printf("Entrer votre nom et prenom.\n");
3  scanf("%s",nom); //Saisie : abc def (Entree)
4  printf("%s\n", nom); //Affiche "abc"
5
6  char ville[20];
7  printf("Entrer votre ville de naissance.\n");
8  scanf("%3c",ville); //le programme ne s'arrete pas
9  printf("%s\n", ville); //Affiche " de<:V"
```

Scanf et la saisie non sécurisée

Dépassement mémoire

Lors de la saisie d'une chaîne de caractères, la fonction `scanf` ne fait aucune vérification sur la taille de l'espace mémoire disponible pour stocker la chaîne. Si la chaîne à stocker est trop grande, `scanf` écrit les caractères en trop à la suite. La fonction écrit alors dans des zones mémoires qui n'étaient pas prévues pour cela, écrasant d'autres informations en mémoire potentiellement importantes pour le programme ou pour d'autres. On parle de *dépassement mémoire*, ou encore de *buffer overflow* en anglais.

Exemple :

```
1  char nom[5]; //4 lettres maximum (pour le '\0' final)
2  printf("Entrer votre nom\n");
3  scanf("%s",nom); //Saisie: abcdefgh (Entree)
4  printf("Bonjour %s\n", nom); //Affiche : "Bonjour abcdefgh"
5  //Affiche a la fin "*** stack smashing detected ***"
6  //Pas de warning si l'allocation etait dynamique...
```

Une solution possible, mais peu satisfaisante : allouer un très grand tableau de caractères (ne protège pas des attaques dites "par buffer overflow"...).

$\Rightarrow$ Pour une version plus sécurisée, on peut utiliser la fonction `fgets` en remplacement de la fonction `scanf` (ou de `fscanf` qui permet notamment de lire dans un fichier, avec encore d'autres défauts...).

Saisie au clavier avec la fonction `fgets`

Signature : `char* fgets(char* chaine, int tailleMax, FILE* flux );`

La fonction `fgets` est une fonction de la librairie standard qui permet de lire des données (non formatées) provenant d'un flux (`stdin`, ou autre). La lecture s'interrompt après :

- avoir lu `tailleMax-1` caractères (pour pouvoir ajouter le '\0' final),
- ou avant si la fonction rencontre un caractère de saut de ligne ('\n') ou un caractère fin de flux (EOF).

```
1  char nom[20];
2  printf("Entrer votre nom et prenom.\n");
3  fgets(nom,20,stdin); //Saisie : a bc  (Entree)
4  printf("%s\n", nom); //Affiche bien "a bc" mais...
```

Petit hic : `fgets` ajoute un saut de ligne à la fin de la chaîne lue, correspondant à l'appui sur la touche "Entrée".

Une solution possible : remplacer le '\n' par le caractère de fin '\0'.

```
1  char nom[20];
2  printf("Entrer votre nom et prenom.\n");
3  fgets(nom,20,stdin); //Saisie : a bc (Entree)
4  nom[strlen(nom)-1]='\0'; // il faut inclure <string.h>
5  printf("%s\n", nom); //Affiche "a bc" sans saut
```

La fonction fgets et la mémoire tampon

Observation : fgets et mémoire tampon

Bien que fgets permette d'éviter les dépassements mémoire, rien n'empêche l'utilisateur de taper trop de caractères. Dans ce cas, les caractères en trop se retrouvent dans la mémoire tampon après l'appel à la fonction (dans le cas où le flux est stdin). Si on ne souhaite pas les récupérer, il faut penser à vider la mémoire tampon juste après l'appel (comme avec scanf).

```
1  char nom[5];
2  printf("Entrer votre nom.\n");
3  fgets(nom,5,stdin); //Saisie : abcdefghijklmn (Entree)
4  printf("%s\n", nom); //Affiche "abcd" sans saut
5  char c = getchar();
6  while (c!='\n' && c!=EOF){
7      c = getchar();
8  }
```

La fonction fgets et la lecture de texte formaté

Lecture d'un texte formaté avec fgets et sscanf

Pour pouvoir lire un texte formaté, on peut utiliser la fonction fgets suivie de la fonction sscanf, qui elle lit du texte formaté dans la chaîne de caractères qu'on lui fournit en paramètre.

```
1  char lus [20];
2  char nom[20];
3  char prenom[20];
4  printf("Entrer votre nom et prenom.\n");
5  fgets(lus,20,stdin); //Saisie : ab cde (Entree)
6  printf("%s\n", lus);//Affiche "ab cde" avec saut
7  sscanf(lus,"%s %s", nom, prenom);
8  printf("%s %s\n", nom, prenom); //Affiche "ab cde" sans saut
```

Remarque : le saut de ligne contenu dans la chaîne lus est ignoré par le sscanf, donc pas besoin de le retirer.

Attention : sscanf peut aussi conduire à des débordements mémoire... par exemple ici, que se passe-t-il quand prenom est de taille 2 ? (sur l ordinateur de Nawal, aucun problème détecté...).

Note : la fonction fgetc ne lit pas du texte formaté, mais permet de lire caractère par caractère (permet de faire des vérifications).

Lecture/écriture de fichier

Ouverture de fichier

L'ouverture se fait avec la fonction `fopen`. Différents modes d'ouverture :

- "r" : lecture seule (le fichier doit exister).
- "w" : écriture seule (si le fichier n'existe pas, il est créé).
- "a" : ajout en fin (si le fichier n'existe pas, il est créé).
- "r+" : lecture et écriture (le fichier doit exister).
- "w+" : lecture et écriture (si le fichier existe, destruction avant création).
- "br" : lecture en binaire (le fichier doit exister).
- etc.

La fonction `fopen` retourne NULL si l'ouverture échoue.

Fermeture de fichier

Tout fichier ouvert doit être fermé avec la fonction `fclose` (comme `malloc` et `free`), qui libère la mémoire en le supprimant de la mémoire vide. La fonction `fclose` retourne 0 si la fermeture s'est bien passée, et EOF sinon.

Lecture de fichier avec la fonction fgets

```c
#include <stdio.h>
#include <stdlib.h>

int lecture(char* nomFichier){
    char ligne[256];
    char nom[256];
    int id;
    FILE* fic = fopen(nomFichier, "r"); // "r" pour read
    if (fic == NULL){
          printf("Probleme ouverture fichier");
          return 0;
    }
    while (fgets(ligne, 256, fic)!=NULL){// lit une ligne
        /* Traitement de la ligne avec sscanf */
        if (sscanf(ligne,"%s%d", nom, &id) == 2){
            printf("Nom : %s , ID :%d", nom, id);
        }else{
            printf("Format de ligne incorrect");
            return 0;
        }
    }
    fclose(fic); //on ferme le fichier avant de quitter
    return 1;
}
```

Remarque : la fonction `fgetc` permet aussi de lire dans un fichier.

Écriture dans un fichier

Fonctions d'écriture : `fputc`, `fputs` et `fprintf`

La fonction `fputc` permet d'écrire un caractère à la fois, tandis que `fputs` permet d'écrire une chaîne de caractères. La fonction `fprintf` permet d'écrire une chaîne formatée dans un flux.

```c
#include <stdio.h>
#include <stdlib.h>

int ecriture(char* nomFichier){
    int i=0;
    FILE* fic = fopen(nomFichier, "w"); // "w" pour write
    if (fic == NULL){
          printf("Probleme ouverture fichier");
          return 0;
    }
    fputs("Debut Fichier\n",fic);
    while (i<100){
        fprintf(fic, "Ligne %d",i);
        fputc('\n',fic);//'\n' peut etre ajoute dans fprintf
        i=i+1;
    }
    fclose(fic); //on ferme le fichier avant de quitter
    return 1;
}
```

Compilation et programmation modulaire

Programmation modulaire

Pour un programme, la modularité est le fait d'être composé en plusieurs parties, relativement indépendantes les unes des autres. La programmation modulaire offre la possibilité de découper un gros programme en modules (réunissant des fonctionnalités particulières) répartis dans différents fichiers.

Avantages

La programmation linéaire :

- rend le code plus compréhensible,
- permet la réutilisation de code,
- supprime les risques d'erreurs en reprogrammant la même chose,
- permet de développer et d'améliorer des parties de code indépendamment,
- facilite le travail collaboratif,
- simplifie la compilation en se concentrant sur les fichiers modifiés.

$\Rightarrow$ Nécessite une réflexion sur le découpage du code avant de se lancer.

Module en langage C et exemple

Module

Un module est composé de :

- un fichier d'en-tête (extension ".h"), décrivant l'interface du module . Il contient des déclarations de fonctions, de type (`struct`), de variables et éventuellement des inclusions de bibliothèques.
- un fichier source (extension ".c") contenant le code des fonctions déclarées dans le fichier d'en-tête.

```c
/* Fichier tableau.h */
#ifndef TABLEAU_H
#define TABLEAU_H

typedef struct tableau {
    char* tab;
    int taille;
} Tableau;

int recherche(int x, Tableau* t);

#endif
```

```c
/* Fichier tableau.c */
#include "tableau.h"

int recherche(int x, Tableau* t){
    int i=0;
    while (i<t->taille){
        if (t->tab[i]==x){
            return i;
        }else{
            i=i+1;
        }
    }
    return -1;
}
```

Exemple

```c
/* Fichier lecture.h */
#ifndef LECTURE_H
#define LECTURE_H
#include "tableau.h"
#include <stdio.h>

Tableau* lire (int n, FILE* f);

#endif
```

```c
/* Fichier lecture.c */
#include "lecture.h"
#include <stdlib.h>

Tableau* lire (int n, FILE* f){
    Tableau* p = (Tableau*)malloc(sizeof(Tableau));
    p->tab = (char*)malloc(sizeof(char)*n);
    p->taille=n;
    if (fgets(p->tab,n,f)!= NULL){
        return p;
    }else{
        printf("Probleme de lecture");
        return NULL;
    }
}
```

Exemple

```c
/* Fichier main.c */
#include "tableau.h"
#include "lecture.h"
#include <stdio.h>

void main(){
    FILE* f = fopen("nomFic.txt", "r");
    if (f!=NULL){
        Tableau* p = lire(256, f);
        printf("%s", p->tab);
    }
}
```

Compilation séparée

On peut maintenant compiler les fichiers sources séparément, puis créer l'exécutable à partir des fichiers compilés. Sur l'exemple, on peut faire :

- `gcc -c tableau.c` (crée le fichier "tableau.o")
- `gcc -c lecture.c` (crée le fichier "lecture.o")
- `gcc -c main.c` (crée le fichier "main.o")
- `gcc -o main main.o tableau.o lecture.o` (crée l'exécutable `main`)

Makefile

L'outil `make`

L'outil `make` est un programme permettant d'automatiser la génération de fichiers à partir de code source, en exécutant un certain nombre de commandes `shell`. Contrairement à un simple script `shell`, `make` exécute les commandes uniquement si elles sont nécessaires. Dans notre cas, cela permet de recompiler uniquement les fichiers qui ont été modifiés depuis la dernière compilation.

Makefile

L'outil `make` utilise des fichiers, appelés `Makefile`, qui spécifient les règles à suivre pour créer les fichiers désirés. Les règles sont sous la forme :

```
1 nom_cible : dependances
2     commandes
```

où `nom_cible` est le nom du fichier à créer, `dependances` est l'ensemble des fichiers nécessaires à sa création, et `commandes` donne les commandes à exécuter (précédées d'une tabulation).

Fonctionnement de l'outil `make`

Quand on tape la commande `make nom_cible` dans le terminal, l'outil `make` évalue la règle permettant de créer `nom_cible`. Si `nom_cible` n'est pas donné en argument de `make`, alors l'outil évalue la première règle rencontrée dans le `makefile`.

Fonctionnement : une évaluation récursive

Évaluer une règle R se fait de récursivement en analysant ses dépendances de la manière suivante :

- Toute dépendance qui est la cible d'une autre règle doit être évaluée avant de pouvoir exécuter les commandes de la règle R.
- Une fois que toutes les dépendances de R ont été évaluées, les commandes de la règle R sont exécutées uniquement si la cible est plus ancienne que l'une de ses dépendances (au moins).

Retour sur l'exemple

```
1  all : main //pas necessaire ici car un seul executable
2
3  main : main.o lecture.o tableau.o
4      gcc -o main main.o lecture.o tableau.o
5
6  main.o :  main.c lecture.h tableau.h
7      gcc -c main.c
8
9  lecture.o : lecture.c lecture.h tableau.h
10     gcc -c lecture.c
11
12 tableau.o : tableau.c tableau.h
13     gcc -c tableau.c
14
15 clean :
16     rm -f *.o main
```