**I.S.I.T.V.**

# PROGRAMMATION AVANCÉE EN C

**J.L DAMOISEAUX**

# Table des matières

# Récursivité

## 1. Généralités

La récursivité est une technique puissante qui permet de décrire un ensemble infini d'objets au moyen d'une phrase finie. Par exemple :

- l'ensemble des entiers naturels se définit par : `0` est un entier naturel, le successeur d'un entier naturel est un entier naturel,
- la factorielle se définit par `0!` égale `1`, et pour `n > 0`, `n!` égale `n*(n-1)!`

On voit immédiatement au travers de ces deux exemples que tout objet récursif se contient dans sa définition. Au niveau d'un langage comme le C, cela se traduit par le fait qu'une fonction puisse s'appeler elle-même.

Prenons comme exemple, le calcul du pgcd de deux nombres `a` et `b` obtenu par la définition suivante :

- si `b` est nul alors le pgcd de `a` et `b` c'est `a`
- sinon le pgcd de `a` et `b` c'est le pgcd de `b` et du reste de la division de `a` par `b.`

La traduction en C de cette définition donne la fonction suivante :

```
int pgcd(int a, int b)
{
int v;
if (b == 0)
      {v = a;     return v;}  /* cas terminal */
else
      {v = pgcd(b,a%b); return v;   }     /* appel récursif */

}

void main(void)
{
int x;
x = pgcd(18,12);
}
```

Dors et déjà, il convient de signaler que tout programme récursif :

- doit comporter un test sans appel récursif (cas terminal),

- doit être écrit de telle manière que chaque appel récursif est plus proche du cas terminal que du problème initial.

## 2. Fonctionnement de la récursivité

D'une manière générale, lorsqu'une fonction est appelée, l'ordinateur réserve une zone mémoire destinée à la table d'activation de la fonction, aux paramètres et aux variables locales. Si, par la suite, une deuxième fonction est appelée alors que la première n'est pas encore terminée, l'ordinateur réserve, au dessus de la zone mémoire allouée précédemment, une seconde zone mémoire. Lorsque cette deuxième fonction se termine, la place allouée est rendue disponible et l'on se retrouve au niveau de l'appel de la première fonction. Schématiquement, l'ordinateur fonctionne comme une pile d'assiettes dont l'accès se ferait uniquement par le sommet.

Lorsqu'une fonction est appelée récursivement, tout se passe comme si une autre fonction était appelée ; un nouvel espace mémoire est réservé pour les variables locales et les paramètres qui sont donc redéfinis.

La figure suivante :

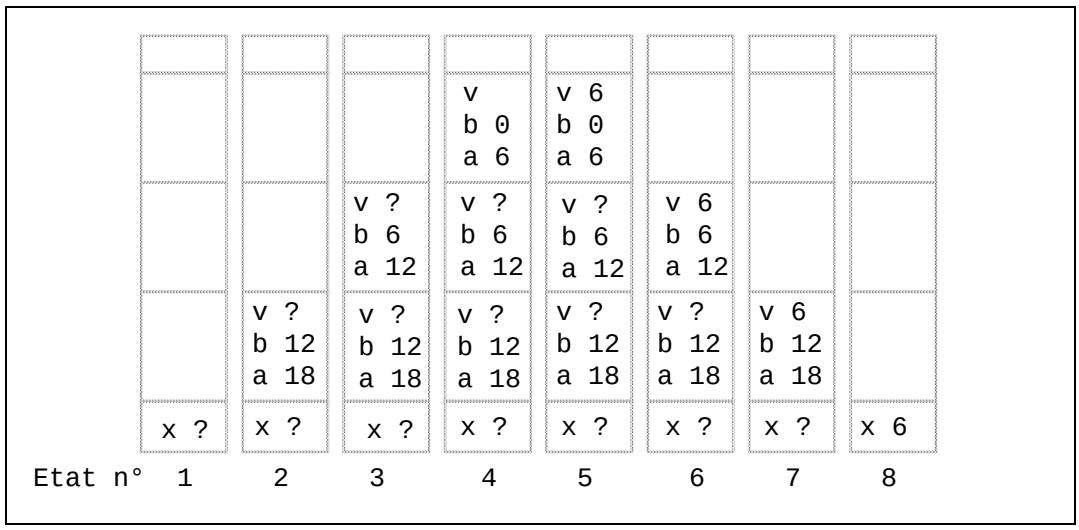



Figure 1 : Etats successifs de la mémoire
après l'appel initial de la fonction `pgcd`

illustre ceci en décrivant les appels récursifs déclenchés par l'appel de `pgcd(18,12)`. Cette fonction comporte 3 variables locales, et la succession des états de la pile est la suivante :

Etat 1 : la mémoire contient le contexte de la fonction `main`, c'est-à-dire uniquement la variable locale `x` dont la valeur est indéterminée ; on appelle alors la fonction pgcd avec pour arguments `18` et `12`.

Etat 2 : la mémoire contient toujours le contexte de la fonction `main`, plus celui de la fonction `pgcd` nouvellement appelée, c'est-à-dire uniquement la variable locale `x` dont la valeur est indéterminée et les variables locales `a`, `b`, `v` dont les valeurs respectives sont `18`, `12` et indéterminée ; `b` est différent de zéro donc on rappelle la fonction `pgcd` avec comme arguments la valeur de `b(12)` et la valeur du reste de la division de `a` par `b(6)`.

Etat 3 : la mémoire contient les contextes précédents plus celui de la fonction `pgcd` nouvellement appelée ; `b` est différent de zéro donc on appelle la fonction `pgcd` avec comme arguments la valeur de `b(6)` et la valeur du reste de la division de `a` par `b(0)`,

Etats 4 et 5 : la mémoire contient les contextes précédents, plus celui de la fonction `pgcd` nouvellement appelée ; `b` est égal à zéro donc on affecte à `v` la valeur de `a` correspondant à l'état courant de la mémoire, soit `6` ; cet appel de la fonction se termine par la transmission à la fonction appelante de la valeur de `v`.

Etat 6 : on se retrouve dans le contexte de l'état 3, et on affecte à `v` la valeur reçue ; cet appel de la fonction `pgcd` se terminera donc par la transmission à la fonction appelante de `v`.

Etat 7 : voir état 6.

Etat 8 : on se retrouve dans le contexte de l'état 1, et on affecte à `x` la valeur reçue ; la fonction `main` peut donc se terminer.

## 3. Récursivité directe et indirecte

Il existe deux types de récursivité : la récursivité directe et la récursivité croisée. Dans la récursivité directe, une fonction s'appelle elle-même.

```
void recursivite_directe(void)
      {
      …
      recursivite_directe();
      …
      }
```

Dans la récursivité croisée, une fonction `A` appelle une fonction `B` qui elle-même appelle la fonction `A`.

```
void fonctionA(void)
      {
      …
      fonctionB();
      …
      }

void fonctionB(void)
      {
      …
      fonctionA();
      …
      }
```

Dans cette situation, il est très important de ne pas avoir oublié, en tête du programme, les prototypes des fonctions. En effet, les identificateurs `fonctionA` et `fonctionB` sont connus depuis la fin de leur définition jusqu'à la fin du fichier. Par conséquent, dans la première fonction l'identificateur `fonctionB` n'est pas connu, mais vicieusement le compilateur C suppose qu'il correspond à l'appel d'une fonction renvoyant un `int`, d'où les problèmes de redéfinition lors de la rencontre de l'entête de la deuxième fonction.

## *4. Deux exemples classiques*

### 4.1. Hanoï

Le problème des tours de Hanoi est le suivant : on dispose de `n` disques de taille croissante empilés les uns sur les autres, et l'on désire déplacer ces `n` disques du piquet de départ sur le piquet d'arrivé. Pour cela on dispose d'un troisième piquet, et les règles de déplacement des disques sont les suivantes : on ne peut déplacer qu'un disque à la fois, et on ne peut poser un disque que sur un disque plus grand (figure n° 2). La légende veut que des moines tibétains essaient de trouver la solution pour `n` égal `64`, ce qui à un déplacement par seconde correspond à trois milliards de siecles, soit la fin du monde.

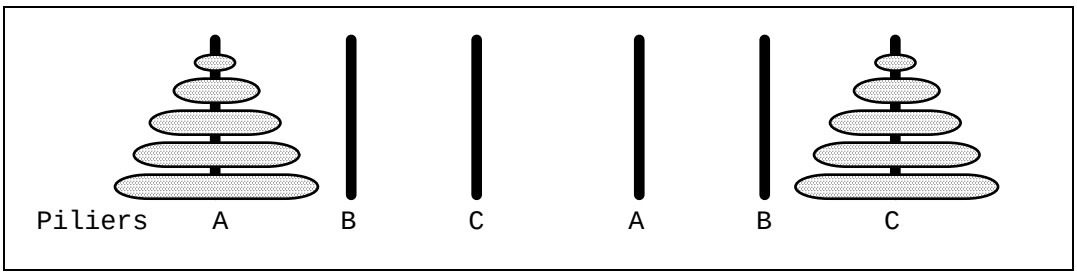



Figure n° 2 : Les tours de Hanoï

Le programme ci-dessous respecte l'algorithme suivant :

- pour un disque il est trivial d'arrivé à l'état final,
- pour 2 disques il est également facile d'arrivé à l'état final,
- pour 3 ou n disques, la récursivité apparaît : on déplace `n-1` disques du pilier de départ sur le pilier temporaire, on déplace alors le dernier disque du pilier de départ sur le pilier d'arrivé, puis on déplace enfin `n-1` disques du pilier temporaire sur le pilier d'arrivé.

```
void hanoi(int n, char dep, char arr, char tpr)
{
if (n == 1) printf("%c -> %c",dep,arr);
else
      {
      hanoi(n-1,dep, tpr, arr);
      hanoi(1,dep, arr, tpr);
      hanoi(n-1,tpr,arr,dep);
      }
return;
}
void main(void)
{
hanoi(3,'a','b','c');
}
```

Remarquez la présence du cas terminal, et le fait que chaque appel récursif est plus proche de la solution que le cas initial[1].

## 4.2. Analyseur syntaxique

Dans de nombreuses situations, on désire décider si une phrase d'un certain langage est syntaxiquement correcte ou incorrecte[2]. Pour cela, on se donne les règles de la grammaire définissant la syntaxe du langage et l'on construit un analyseur en fonction de ces règles. Pratiquement, il convient :

- d'associer à chaque symbole non-terminal on associe une fonction C,
- de continuer l'analyse de la chaîne que si l'on a effectivement reconnu le caractère.

Prenons à titre d'exemple la grammaire suivante :

[1]Il est possible d'optimiser le nombre des appels récursifs, et donc d'écrire un autre programme.

[2]Il existe sous UNIX des outils comme Lex et Yack qui vous permettent de faire facilement ce travail

```
S -> 'a' S 'b'
S -> 'c'
```

que l'on interprète comme le symbole non-terminal `S` peut se réécrire en `'a' S 'b'` ou `'c'`. Les phrases engendrées par cette grammaire sont de la forme $a^n\ c\ b^n$ avec $n \bullet 0$.

```
S -> 'c'
S -> 'a' S 'b' -> 'a' 'c' 'b'
S -> 'a' S 'b' -> 'a''a' S 'b''b' -> 'a''a' 'c' 'b''b'....
```

L'algorithme proposé pour l'analyse d'une phrase de ce langage est le suivant :

```
analyser un mot du langage c'est
soit reconnaitre la lettre c,
soit
      reconnaitre la lettre a,
      puis analyser un mot du langage,
      pour enfin reconnaitre la lettre b.
```

La traduction en C de cet algorithme est immédiate et donne l'analyseur ci-dessous pour lequel :

- `chaine_analysee` est un tableau de caractères contenant la chaine à analyser (ce tableau sera initialisé dans la fonction `main`),
- `caran` est un entier qui indique la position dans le tableau `chaine_analysee` du caractère en cours d'analyse,
- `erreur` est une fonction affichant un message d'erreur approprié et provoquant la terminaison du programme.

```
char chaine_analysee[80];
int caran = 0;

void axiome (void)
{
if (chaine_analysee[caran] = 'c')
      caran++;    // j'ai reconnu la lettre c
else if (chaine_analysee[caran] = 'a')
            {
            caran++;    // j'ai reconnu la lettre a
            axiome();   // je cherche à reconnaitre une phrase S
            if (chaine_analysee[caran] = 'b')
                  caran++;    // j'ai reconnu la lettre b
            else erreur();
            }
else erreur();
return;
}
```

# Listes chaînées

## 1. Généralités

La liste chaînée est une structure de donnée très importante en informatique. Bien que la liste chaînée ne soit pas une primitive du langage C, ce langage offre toutefois des opérations élémentaires facilitant la conception et l'utilisation d'une telle structure.

Le principal avantage d'une liste chaînée est que sa taille varie en fonction des besoins du programme. En outre, le réarrangement efficace de leurs éléments leur donne une grande souplesse d'utilisation, souplesse toutefois acquise au détriment de l'accès direct (donc de la rapidité) à l'information.

Une liste chaînée est une collection d'éléments organisés séquentiellement. Cependant, contrairement au tableau, cette organisation est explicite puisque chaque élément appartient à un maillon qui contient un lien sur le maillon suivant (figure n°1).

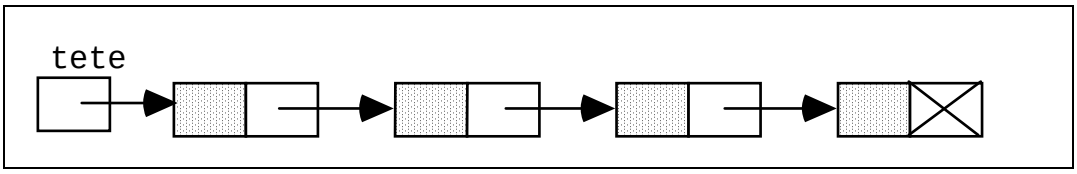



Figure n° 1 : Représentation d'une liste chaînée

Cette représentation met en évidence deux éléments importants à prendre en considération :

- chaque maillon possède un lien ; le dernier maillon de la liste pointera sur rien, ce que l'on symbolisera par une constante appropriée (en l'occurence `NULL`),
- l'utilisation d'une liste nécessitera un lien particulier pour accéder au premier maillon de celle-ci ; ce lien sera assuré par un pointeur de tête pointant sur le premier maillon de la liste.

## 2. Définition d'une liste en C

La définition d'une liste passe par la définition d'un maillon. En C, cette définition repose sur l'utilisation d'une structure :

```
struct maillon
      {
      type_info info;
      struct maillon *suiv;
      };
```

où `type_info` est le type de la donnée (qui peut-être quelconque) stockée dans le maillon, et `suiv` un pointeur qui réalisera le lien sur le maillon suivant.

---

Rappel, si `p` est un pointeur contenant l'adresse d'une structure de type `maillon`, alors `p->info` et `p->suiv` désignent respectivement les champs `info` et `suiv` de la structure pointée par `p`.

---

Attention, les maillons d'une liste ne seront pas rangés consécutivement en mémoire.

## *3. Traitements élémentaires dans une liste chaînée*

Les traitements élémentaires abordés dans cette partie sont l'insertion d'un maillon dans la liste, la suppression d'un maillon, le parcours de la liste, la recherche d'une information. Comparativement à la réalisation de ces mêmes opérations sur un tableau, la représentation explicite de l'ordre induira pour ces opérations une plus ou moins grande efficacité.

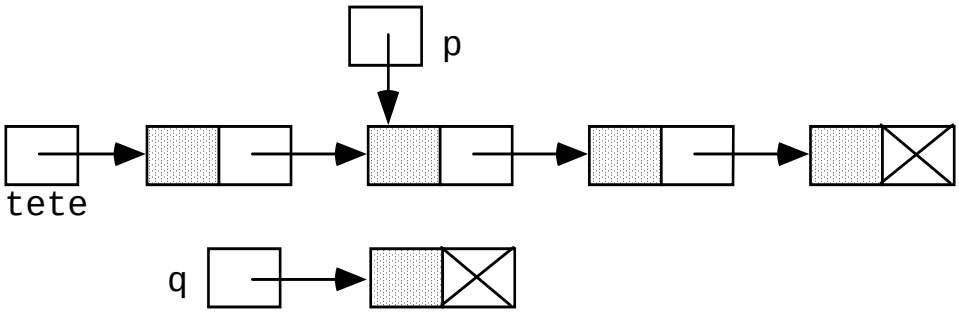
### 3.1. Insertion d'un maillon dans une liste

Supposons que l'on veuille insérer dans une liste un maillon pointé par un pointeur `q` à la suite d'un maillon pointé par un pointeur `p`. L'insertion du maillon sera réalisée par le programme suivant :

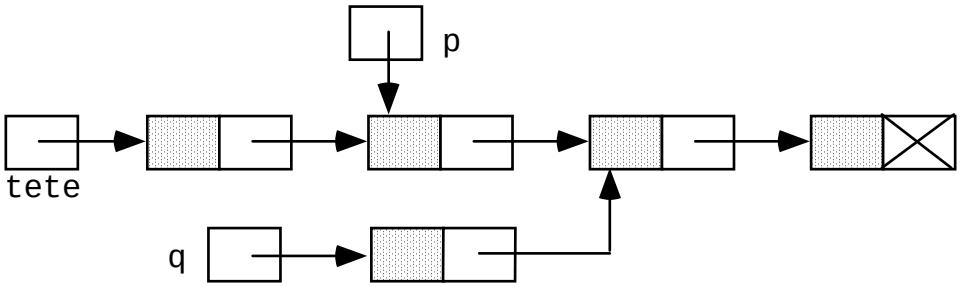
```
      q->suiv = p->suiv
      p->suiv = q
```

dont voici l'illustration :

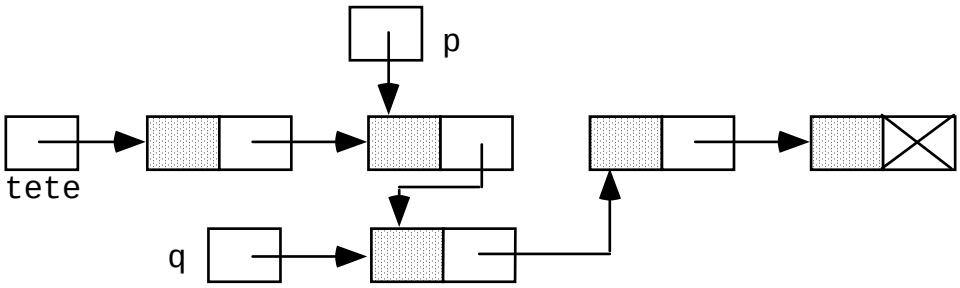
- état de la liste avant les instructions




- état de la liste après l'instruction `q->suiv = p->suiv`




- état de la liste après l'instruction `p->suiv = q`



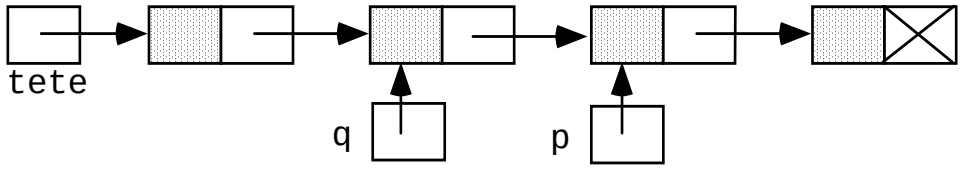

## 3.2. Suppression d'un maillon dans une liste

La suppression d'un maillon de la liste nécessite l'adresse du maillon à supprimer, mais également l'adresse de son prédécesseur. Soit `p` le pointeur sur le maillon à supprimer et `q` l'adresse de son prédécesseur, la suppression d'un maillon dans une liste est réalisée par l'instruction suivante :

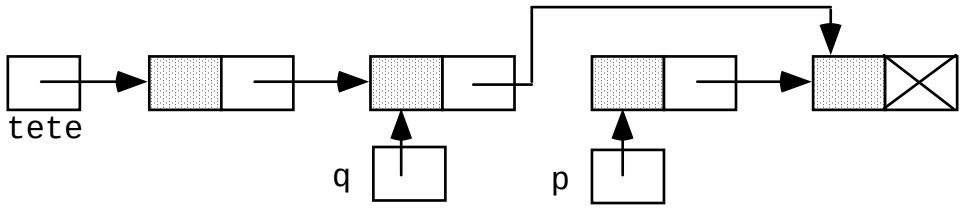
```
q->suiv = p->suiv;
```

dont voici également l'illustration :

- état de la liste avant l'instruction




- état de la liste après l'instruction




---

L'élimination physique du maillon à supprimer sera généralement réalisée par l'instruction `free`.

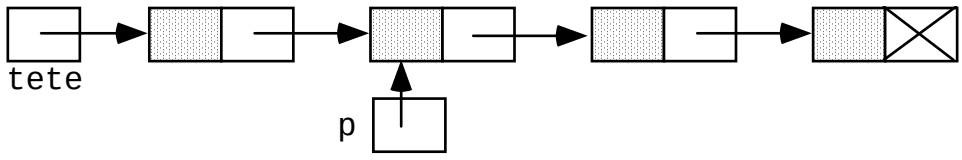
## 3.3. Parcours d'une liste

Le parcours d'une liste se fait tout simplement en passant successivement d'un maillon à l'autre. Ce passage d'un maillon à l'autre est réalisé par l'instruction suivante :

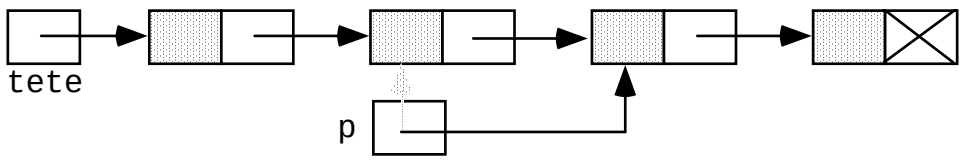
```
p = p->suiv;
```

dont l'illustration est la suivante :

- état de la liste avant l'instruction




- état de la liste après l'instruction



Il suffit maintenant pour parcourir une liste de mettre l'instruction précédente dans une boucle dont la condition d'arrêt sera la rencontre de la fin de la liste. Une traduction possible en C serait la suivante :

```
p = tete;
while (p != NULL)
      p = p->suiv;
```

On remarquera d'une part le cas de la liste vide est traité correctement puisque l'on sort immédiatement de la boucle, et d'autre part que l'on ne travaille pas sur le pointeur `tete`, mais sur une copie, ceci afin de ne pas toucher à la structure de nôtre liste, et ainsi perdre irrémédiablement l'accès aux maillons qui la composent.

## 3.4. Recherche d'un élément dans une liste

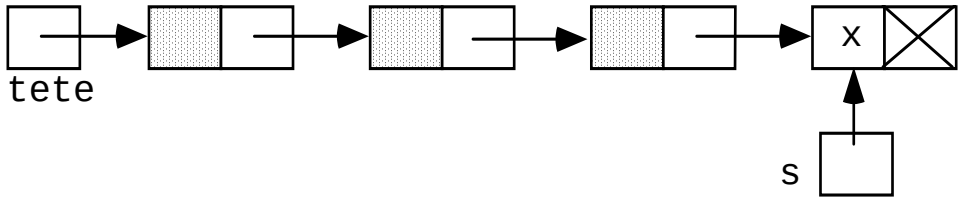
La recherche d'un élément dans une liste se fait en parcourant la liste jusqu'à avoir trouver l'élément en question. Le programme proposé est le suivant :

```
if ① (  p != NULL)
      {
      p = tete;
      while (②(p->suiv != NULL) &&
             ③  (p->info != x)) p = p->suiv;
      if ④  (p->info == x) printf("x est dans la liste");
      }
else
      printf("la liste est vide");
```

Plusieurs problèmes importants sont résolus par ce programme :

- dans le cas où la liste est vide, il importe de ne pas se déplacer puisqu'il n'existe pas de maillons. La condition ① remédie à ce problème.
- l'élément `x` n'est pas forcément présent dans la liste, et il importe de ne pas sortir de la liste lors de sa recherche. La condition ② sera fausse quand `p` pointera sur le dernier maillon, ce qui nous évitera ainsi de continuer notre recherche. Toutefois, il convient de tester si la sortie de boucle s'est faite par la condition ② ou la condition ③. La condition ④ nous permet de vérifier cela. En effet, si l'on est sortie de la boucle par la condition ③ alors la condition ④ est vraie aussi. D'autre part, si l'on est sorti de la boucle par la condition ② et que le dernier maillon contient la valeur `x`, alors la condition ④ est encore vraie.

Bien que le programme précédent soit juste, nous allons améliorer son efficacité. Lors de la recherche d'un élément dans une liste, notre condition de sortie de boucle porte sur un double test logique (conditions ② et ③). Il est possible de supprimer la condition ② en étant sûr que la valeur `x` soit toujours présente dans la liste. Pour cela, nous allons rajouter un maillon particulier à notre liste[3], et ainsi nous placer dans la configuration suivante :




Ce maillon pointé par `s` est baptisé sentinelle. Son champ `info` sera initialisé à la valeur recherchée, ce qui rend certain sa présence dans la liste. Il conviendra donc juste de tester si la recherche s'est terminée sur le maillon `s`, ce qui signifiera que la valeur `x` est absente de la liste (puisqu'elle n'a pas été trouvée avant).

```
s->info = x;      /* initialisation de la sentinelle */
p = tete;
while (p->info != x) p = p->suiv;
if   (p == s) printf("x n'est pas dans la liste");
else printf("l'élément x a été trouvé");
```

## 3.5. Construction d'une liste

Nous allons maintenant terminer notre étude par une méthode de construction de liste. Cette méthode est fondée sur le fait que à tout moment de la construction, nous disposerons de deux pointeurs :

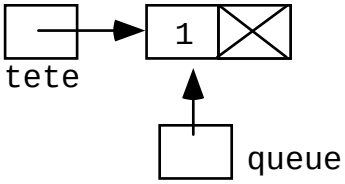
- le premier appelé `tete` pointera toujours sur le premier maillon de la liste,
- le second baptisé `queue` pointera toujours sur le dernier maillon de la liste.

Le processus de construction de notre liste est illustré par les schémas suivants :

- étape ① début de construction, la liste est vide (le pointeur `tete` a la valeur `NULL`)

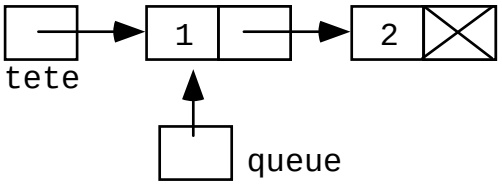
[3]Dans cette situation, une liste vide contiendra toujours un élément : la sentinelle.


- étape ② l'insertion du premier maillon est un cas à part qui consiste simplement à faire pointer `tete` et `queue` sur le maillon à insérer

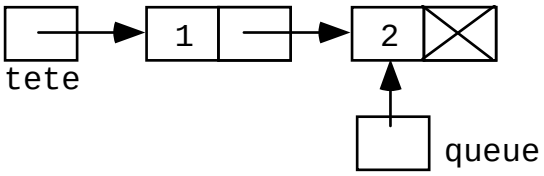



- étape ③ l'insertion du maillon suivant, à la suite du premier, se fera en deux étapes.

  ③-a Tout d'abord, on lie le nouveau maillon au maillon pointé par `queue`




  ③-b Ensuite, on déplace le pointeur `queue` sur ce dernier maillon (on se retrouve ainsi prêt pour une nouvelle insertion)




Le programme qui en découle suit scrupuleusement ce qui vient d'être écrit :

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

struct maillon
      {
      type_info info; /* le type de l'info dépend du problème */
      struct maillon *suiv;
      };

struct maillon * allouer(type_info i, struct maillon * s)
      {
      struct maillon * p;

      p = (struct maillon * ) malloc(sizeof(struct maillon));
      p->info = i;
      p->suiv = s;
      return p;
      }

struct maillon * creer_liste(void)
      {
      struct maillon * t = NULL,q = NULL;
      type_info i;

      t = q = NULL;     /* étape ①  */
      while (scanf("%d",&i), i >= 0)/* lecture d'une valeur */
            if (t == NULL)
                  t = q = allouer(i,NULL); /* étape ②      */
            else /* étape ③   */
                  {
                  q->suiv = allouer(i,NULL); /* étape ③-a  */
                  q = q->suiv;            /* étape ③-b     */

                  }
      return t;
      }
```

avec en plus la fonction `allouer` qui créé et initialise une structure de type maillon.

# Types de données abstraits

## 1. Généralités

Un type de données abstrait est un type pour lequel on ne connait pas son implantation, et qu'on ne peut utiliser autrement que par les opérations qui lui sont associées. Des exemples de types abstraits classiques sont les entiers, les réels, les structures, etc.

Le concept de type de données abstrait est un élément essentiel dans la construction de gros programmes, et ce pour trois raisons :

- l'interface entre les structures de données et les programmes les utilisant est minimisé,
- le programme utilisant ces structures s'affranchit des détails concernant leur gestion,
- les modifications du programme sont aisées.

Afin de mieux comprendre cette notion, nous allons étudier deux exemples classiques de types de données abstraits, à savoir la pile et la file.

## 2. Le type pile

La pile est un concept extrêmement utilisé en informatique (évaluation d'expression, récursivité, mode de fonctionnement de processeur, langage de programmation, etc.). Le principe est celui d'une pile d'assiettes pour laquelle on ne voit que la dernière assiette : les informations à stocker sont toujours placées sur le sommet de la pile, et celles à extraire le sont toujours à partir du sommet : l'information la première arrivée sera la dernière sortie (Last In First Out).

### 2.1. Implémentation

Nous proposons ici une implémentation de ce concept par un tableau et un entier indiquant le sommet de la pile. Les opérations associées à la pile seront `initialiser`, `empiler` et `depiler`.

Commençons par la définition des constantes de fonctionnement de la pile

```
#define TAILLE_PILE 100
#define PILE_VIDE -1
#define PILE_PLEINE (TAILLE_PILE - 1)
```

Le type abstrait `T_PILE` est réalisé grâce à la notion de structure :

```
typedef int T_INFO;
struct Pile
{
T_INFO tampon[TAILLE_PILE];
int i_tampon;
};
typedef struct Pile T_PILE;
```

Enfin, voici la définition des fonctions de gestion de la pile.

```
void initialise(T_PILE *pile)
{
pile->i_tampon = PILE_VIDE;
return;
}

int empiler(T_PILE *pile,T_INFO info)
{
if (pile->i_tampon != PILE_PLEINE)
      {
      pile->i_tampon++;
      pile->tampon[pile->i_tampon] = info;
      return 1;
      }
printf("erreur la pile est pleine\n");
return 0;
}

int depiler(T_PILE *pile,T_INFO *info)
{
if (pile->i_tampon == PILE_VIDE)
      {
      printf("erreur la pile est vide\n");
      return 0;
      }
*info = pile->tampon[pile->i_tampon];
pile->i_tampon--;
return 1;
}
```

Il est important de bien comprendre que l'utilisateur de ce type ne pourra en principe que déclarer un objet de type T_PILE, et ne devra en principe l'utiliser qu'au travers des opérations `initialise`, `empiler` et `depiler`.

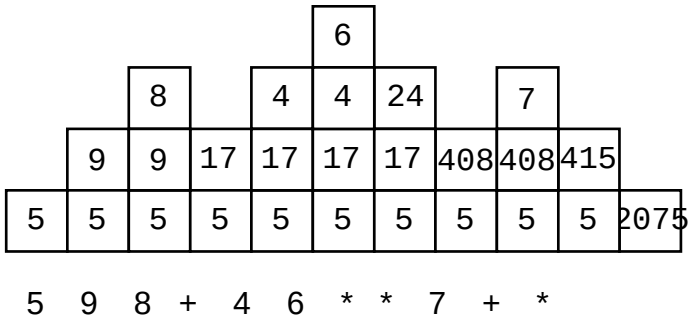
## 2.2. Exemple d'application

La manière habituelle d'écrire des expressions arithmétiques s'appelle la notation infixée. Il existe cependant une autre notation (la notation postfixée où notation polonaise inversée) qui permet de se dispenser de toutes parenthèses dans une expression. Par exemple, l'expression infixée `5*(((9+8)*(4*6))+7)` se note `5 9 8 + 4 6 * * 7 + *` en écriture postfixée, et l'expression infixée `((5*9)+8)*((4*6)+7)` se note toujours en écriture postfixée `5 9 * 8 + 4 6 * 7 + *`.

L'évaluation d'une expression postfixée est triviale à l'aide d'une pile, et suit le principe suivant :

- lorsque l'on rencontre un nombre, on l'empile,
- lorsque l'on rencontre un opérateur, on dépile les deux premiers éléments de la pile et on empile le résultat de l'opération.

A titre d'exemple, voici l'évolution de la pile lors de l'évaluation de l'expression arithmétique[4]
`5 9 8 + 4 6 * 7 + *`




Le programme d'évaluation d'une expression postfixée est donc le suivant :

[4]Par mesure de simplicité, les opérandes et les opérateurs sont séparés par un espace, et les opérandes ne sont que des chiffres.

```
int evalue(char expression[])
{
int v, r1, r2, i;
T_PILE p;
char c;

initialise(&p);
for (i = 0;(c = expression[i]) != '\0'; i++)
   switch(c)
        {
        case ' ' :   break;
        case '+' :   depiler(&p,&r1);
                                depiler(&p,&r2);
                                empiler(&p,r1+r2);
                                break;
        case '-' :   depiler(&p,&r1);
                                depiler(&p,&r2);
                                empiler(&p,r2-r1);
                                break;
        case '*' :   depiler(&p,&r1);
                                depiler(&p,&r2);
                                empiler(&p,r1*r2);
                                break;
        case '/' :   depiler(&p,&r1);
                                depiler(&p,&r2);
                                empiler(&p,r1/r2);
                                break;
        default  :   r1 = 0;
                                while (isdigit(c))
                                      {
                                      r1 = r1 * 10+ c-'0';
                                      c = expression[++i];
                                      }
                                empiler(&p,r1);
        }
depiler(&p,&v);
return v;
}
```

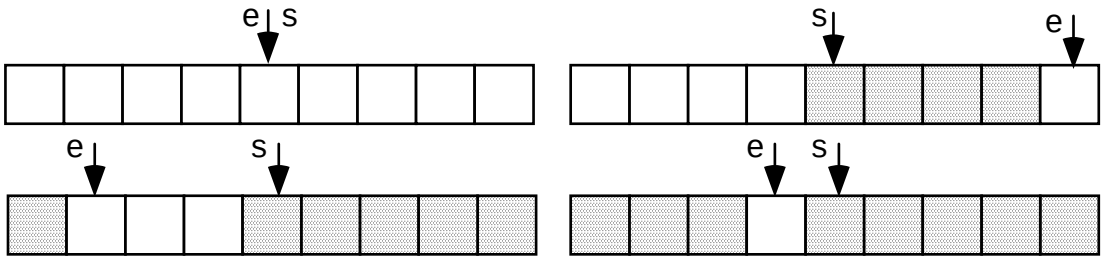
L'intêret du type abstrait apparaît immédiatement, car si je change l'implémentation de ma pile (par exemple en la codant sous la forme d'une liste chaînée -cf Modularité-), le programme d'évaluation restera identique.

## 3. Le type file

La file est également un concept extrêmement utilisé en informatique, et son principe est celui d'une liste d'attente chez le médecin : le premier malade arrivé sera le premier soigné[5]. Plus informatiquement parlant, on traduit ceci par la première information arrivée sera la première sortie (First In First Out).

[5]Sauf urgence bien sûr....

Nous proposons ici une implémentation de ce concept par un tableau et deux entiers `e` et `s`, l'un indiquant l'entrée de la file (la prochaine case dans laquelle on écrira), l'autre indiquant la sortie de la file. Cette file sera considéré comme un tampon circulaire, et les opérations associées sont `initialiser`, `enfiler` et `defiler`. Lorsque `e` et `s` seront égaux alors la file sera considéré comme vide, et quand `e` sera une case avant `s` alors la file sera considérée comme pleine.




Commençons par la définition de la taille de la file

```
#define TAILLE_FILE 100
```

Le type abstrait `T_FILE` est réalisé grâce à la notion de structure :

```
typedef int T_INFO; /* dépend des données traitées */
struct File
      {
      T_INFO tampon[TAILLE_FILE];
      int e, s;
      };
typedef struct File T_FILE;
```

Enfin, voici la définition des fonctions de gestion de la file d'attente

```
void enfiler(T_FILE *file, T_INFO info)
      {
      if (suivant(file->s) == file->s)
            printf("la file est pleine\n");
      else
            {
            file->tampon[file->s] = info;
            file->s = suivant(file->s);
            }
      }

void defiler(T_FILE *file, T_INFO *info)
      {
      if (file->e == file->s)
            printf("la file est vide\n");
      else
            {
            *info = file->tampon[file->s];
            file->s = suivant(file->s);
            }
      }

void initialiser(T_FILE *file)
      {
      file->e = file->s = 0;
      }
```

où la fonction `suivant` renvoie l'indice de la position suivante de celle qui lui est donnée en argument.

```
int suivant(int p)
      {
      return (p+1) % TAILLE_FILE;
      }
```

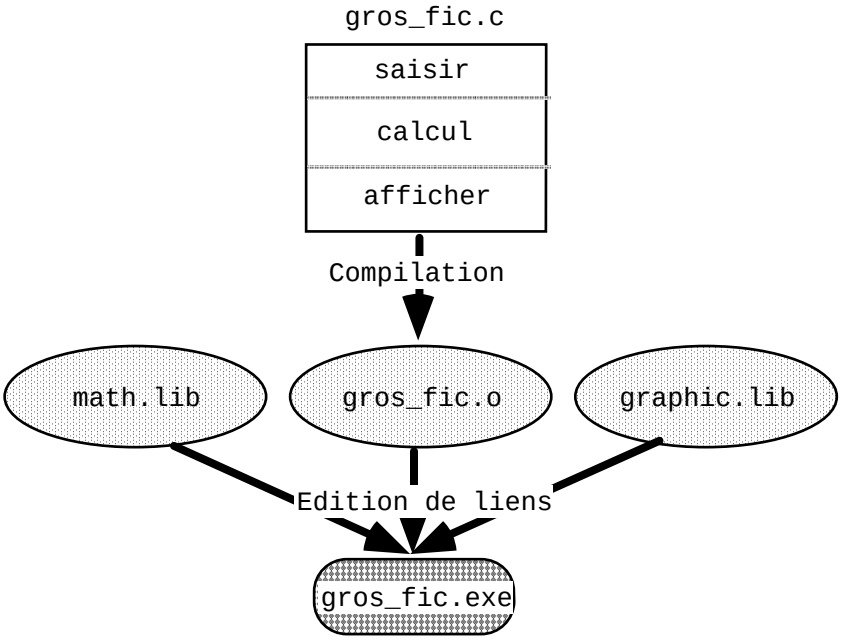
# Modularité

## 1. Généralités

La modularité est une technique de programmation très utilisée dans l'industrie du logiciel. Puissante, elle permet de décomposer l'écriture d'un programme volumineux en un ensemble de fichiers, qui seront compilés indépendamment des uns des autres, puis réunis lors de l'édition de liens, pour finalement ne former qu'un seul code exécutable.

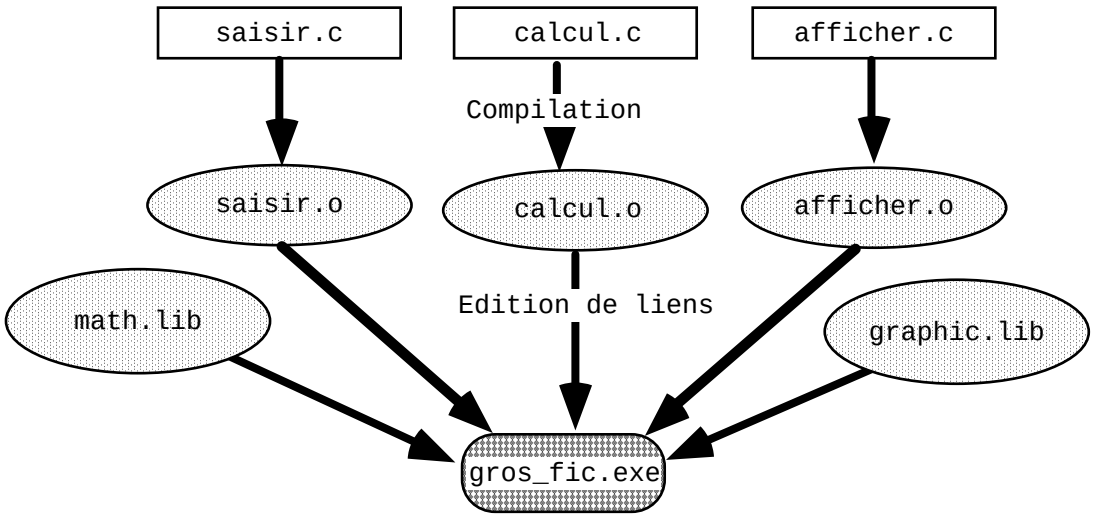
L'intérêt d'une telle technique est évident. Supposons que nous devions écrire un programme comportant les trois parties suivantes : saisie des données à traiter, calcul, et affichage graphique des résultats.

Une première approche pour mener à bien ce projet consisterait à regrouper ces trois parties dans un seul fichier gros_fic.c, qui serait ensuite compilé, puis lié aux librairies mathématique et graphique, pour enfin obtenir un fichier exécutable.




La deuxième approche possible serait de définir un fichier indépendant pour chacune des trois parties (calcul.c, saisir.c, afficher.c), chacun de ces fichiers étant ensuite compilés

séparément, pour être liés tous ensembles aux librairies mathématique et graphique, afin d'obtenir le même fichier exécutable.




Si vous modifiez une ligne de code dans une fonction de la partie du programme concernant l'affichage des résultats, que se passe-t-il ? Dans la première approche, vous relancerez une nouvelle compilation du fichier `gros_fic.c`, pour ensuite refaire l'édition de liens. Dans la deuxième approche, vous recompilerez uniquement le fichier `afficher.c`, puis vous relancerez l'édition de liens. A votre avis, quelle est l'approche la plus efficace ?

## 2. Visibilité d'un objet dans le langage C

Lors de la mise en oeuvre de la compilation séparée, ils se posent de nombreux problèmes relatifs à l'utilisation de variables et de fonctions communes à plusieurs fichiers. Afin d'éviter de sérieux désagréments, il convient de bien préciser les notions afférentes à la visibilité d'un objet, comme par exemple sa définition, sa déclaration.

La visibilité d'un objet est l'espace du programme dans lequel on peut le référencer. Jusqu'à présent, vous saviez qu'un objet était connu depuis la fin de sa définition jusqu'à la fin de l'espace auquel il appartenait, exception faite des zones où il était redéfini.

La définition d'un objet précise sa classe (variable, fonction), ses caractéristiques (type, valeur initiale, etc.), éventuellement les traitements qui lui sont associés, et alloue en mémoire la place de l'objet en question.

```
/* définition d'une fonction */
int max(int a, int b)
{
return a > b ? a : b;
}

/* définition de variables */
double a, table[100];
```

La déclaration d'un objet postulera l'existence d'un objet dont on supposera qu'il a été défini ailleurs (éventuellement dans un autre fichier).

```
/* déclaration d'une fonction */
int max(int a, int b);

/* déclaration de variables */
extern double a, table[];
```

A ces deux notions, il convient d'ajouter les notions d'objets public et d'objets privés :

- un objet sera dit public s'il est défini dans un fichier et que l'on peut l'utiliser, via une déclaration, dans d'autres fichiers. Par défaut tout objet global est public.
- un objet qui n'est pas public sera dit privé.

Le qualificatif `static` placé devant la définition d'un objet rend celui-ci privé, tandis que qualificatif `extern` placé devant la déclaration d'un objet précise que cet objet a été défini dans un autre espace (fichier, bloc) que celui où se trouve cette déclaration.

Par exemple :

```
#include <stdio.h>

int i;
static int k;
extern int r;

static int max(int a, int b);
extern int inc(int *v);
int plus(int z, int k);/* plus est une fonction publique */
```

- la variable `i` est globale donc par défaut publique,
- la variable `k` est globale mais rendue privée par le qualificatif `static`,

- la variable `r` est définie publique dans un autre fichier (cf le qualificatif `extern`),

- la fonction `max` est rendue privée par le qualificatif `static`,
- la fonction `inc` est définie publique dans un autre fichier (cf le qualificatif `extern`),
- la fonction `plus` est par défaut publique.

## 3. Mise en oeuvre de la compilation séparée en C

La modularité en C, bien que faisant partie intégrante du langage, n'est pas du tout fiable en ce qui concerne la sécurité du logiciel produit. En fait, elle est soumise à l'autodiscipline du programmeur qui s'imposera certaines règles de programmation, dont voici donc les plus importantes :

- à chaque fichier module (fichier d'extension `.c`) est associé un fichier en-tête (fichier d'extension `.h`) regroupant la déclaration des objets publics de ce module. Ce fichier en-tête servira d'interface pour l'utilisation du fichier module, et sera inclus dans tous les autres modules désireux d'utiliser les fonctionnalités du module en question,
- dans un fichier module, tous les objets qui ne seront pas publics doivent être rendus privés grâce au qualificatif `static`.
- il est préférable de rendre une variable privée et de l'utiliser via des fonctions spécifiques, plutôt que de laisser cette variable publique et de l'utiliser directement dans les autres modules (notion d'encapsulation).

Illustrons nos propos par la reprise du concept de pile (Last In First Out), en l'implémentant à partir d'une liste chaînée (l'insertion d'un maillon se fera en tête de liste). Les opérations associées sont toujours `initialiser`, `empiler` et `depiler`. La conception de notre application se fera autour :

- du fichier en-tête `pile.h` qui propose un ensemble d'outils pour manipuler une pile d'entiers ; ce fichier sera inclus dans les fichiers qui souhaitent travailler avec une pile.

```
typedef int T_INFO;
struct Pile
{
T_INFO info;
struct Pile *suiv;
};
typedef struct Pile * T_PILE;

extern void initialiser(T_PILE *);
extern int empiler(T_PILE *, T_INFO );
extern int depiler(T_PILE *, T_INFO *);
```

- du fichier `essai` qui utilise la structure de pile uniquement au travers des objets proposés dans le fichier `pile.h` .

```
#include "pile.h"
void main(void)
{
int i;
T_PILE p;
initialiser(&p);
while (scanf("%d",&i), empiler(&p, i));
while (depiler(&p, &i));printf("%d\n", i);
return
}
```

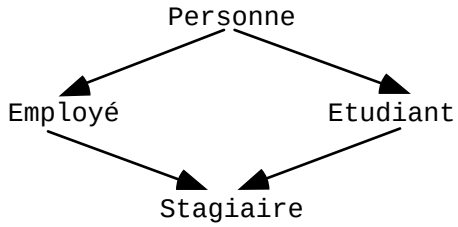
- et du fichier `pile.c` qui détaille le fonctionnement d'une pile :

```
#include "pile.h"

static T_PILE allouer(T_PILE, T_INFO);
static T_PILE allouer(T_PILE s, T_INFO i)
      {
      T_PILE p;

      p = (T_PILE)      malloc(sizeof(struct Pile));
      p->info = i; p->suiv = s;
      return p;
      }

void initialiser(T_PILE * p)  {     *p = NULL;  }

int empiler(T_PILE *p, T_INFO info) { *p = allouer(*p,info); }
```

```
int depiler(T_PILE *p,T_INFO *info)
      {
      T_PILE s = *p;

      if (*p == NULL) return 0;
      *p = (*p)->suiv;  *info = s->info;
      free(s);
      }
```

## 4. L'inclusion conditionnelle

Il arrive couramment lors de l'écriture de gros programmes que l'on inclut plusieurs fois un même fichier, ce qui outre le fait d'augmenter inutilement la taille du code, peut parfois provoquer des erreurs de compilation ou de liens désagréables. Par exemple, si l'on réalise la modélisation du monde du travail, on peut être amener à établir le graphe suivant :




qui définit un `Stagiaire` comme un `Employé` et un `Etudiant`. Dans le fichier des caractéristiques d'un `Stagiaire`, on inclut les fichiers relatifs aux `Employés` et aux `Etudiant`, qui eux-mêmes incluent le fichier relatif aux `Personne`. De ce fait, le fichier des caractéristiques des `Personne` est donc inclus deux fois dans le fichier relatif aux `Stagiaire`.

Afin d'éviter les possibles problèmes liés à ces inclusions multiples, on utilise les fonctionnalités du préprocesseur (cf Chapitre 6). Le contenu du fichier qui doit être inclus sera placé entre les instructions conditionnelles suivantes du préprocesseur :

```
#if !defined(ENTETE)
#define ENTETE

texte du fichier qui sera inclus

#endif
```

Ainsi, lors de la première inclusion du fichier, le préprocesseur analyse et traite la ligne #if !defined(ENTETE). Comme la macro `ENTETE`[6] n'a jamais été définie auparavant dans le fichier en cours de traitement, le préprocesseur continu son travail et analyse maintenant toute les lignes du programme qui sont incluses entre le `#if` et `#endif`. La première de ces lignes consiste à définir la macro ENTETE. Ainsi, lors d'une ré-inclusion future du fichier, la macro ENTETE ayant déja été définie, les lignes du fichier ne seront plus incluses.

[6]Le nom de la macro est complètement libre.

# Retour arrière ou Backtracking

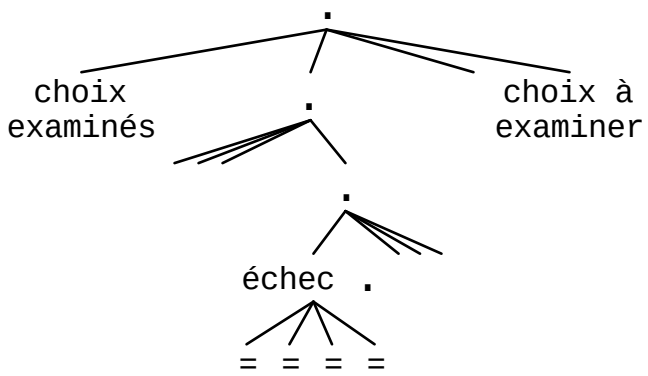
La technique de programmation exposée ci-dessous propose une solution pour résoudre des problèmes qui n'ont pas d'algorithmes connus. Ces problèmes, relevant souvent du domaine de l'intelligence artificielle, sont reconnaissables par le fait que :

- la solution au problème peut être obtenue par étape. Chaque étape est appelée un état qui peut être considéré comme une solution partielle,
- il est possible d'identifier un état final qui est la solution.
- si un état n'est pas la solution, alors il conduit par des transitions à d'autres états (on augmente ainsi la solution partielle),
- parmi les transitions possibles, certaines sont plus ou moins acceptables ; on officialise ici le fait que les transitions n'ont éventuellement aucun sens ; c'est le couple énumérateur de choix + test d'acceptabilité qui définit le vrai constructeur de transitions,
- un état sans transition et qui n'est pas la solution est un échec.

D'autre part, même dans le cas où ces conditions seraient remplies, la combinatoire du problème peut être telle, qu'aucun ordinateur ne sera assez puissant pour un jour le résoudre.

## *1. Méthodologie*

Si l'on dessine la résolution du problème en schématisant les états par lesquels on passe, on obtient un arbre :



dont certaines branches mènent aux solutions. Le parcours de cet arbre se fera en profondeur d'abord et de gauche à droite. Lorsqu'on arrivera à un échec, on remontera alors à l'état immédiatement au dessus (l'état père) pour essayer un nouveau choix (état frère). Si l'on s'intéresse à toutes les solutions, il suffira de les afficher au fur et à mesure, sinon il est possible de s'arrêter dès la première. Enfin, il est important de bien comprendre que cet arbre n'est jamais construit dans sa totalité.

Voici donc un algorithme de principe dans lequel l'état est représenté par une variable locale :

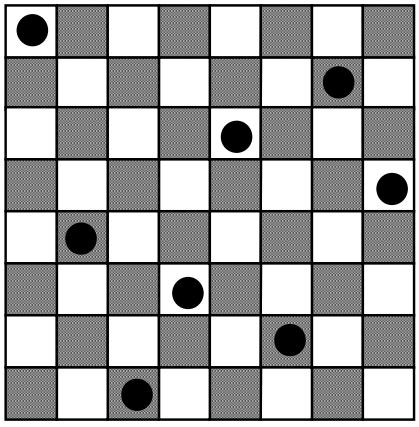
```
parcourir(etat)
      si etat est une solution alors afficher cette solution
      sinon
            pour tout les choix c_i possibles à partir de cet état
                  si la transition etat + c_i est acceptable alors
                        état' <- etat + c_i
                        pacourir(etat')
```

Le passage au fils d'un état est effectué par un appel récursif de la fonction, et la remontée est assurée par une boucle qui relance sur les états frères.

# *2. Exemples*

## 2.1. Les huit reines

Le premier exemple que nous donnerons est le positionnement sur un échiquier de 8 reines sans qu'aucune ne soit en prise avec une autre. Voici une solution parmi les 64 existantes :



La représentation d'un état du jeu sera la suivante : $(n, (x_1, x_2, \ldots, x_n))$ où $n$ est le nombre de reines placées et $x_i$ indique que sur la colonne $i$ il y a une reine sur la ligne $x_i$. La représentation d'un choix sera la suivante : $(n+1, (x_1, x_2, \ldots, x_n), pos)$.

S'il est acceptable cela se traduira par un nouvel état :
$(n+1,(x_1, \ldots, x_n, x_{n+1} = pos)$.

Voici donc le programme C :

- l'état du jeu est codé par un tableau `solution` et une variable `n` qui correspond au nombre de reines placées sur l'échiquier

```
int solution[9], n = 1;
```

- la fonction `en_prise` détermine si deux reines sont sur la même colonne ou diagonale

```
int en_prise(int x1, int y1, int x2, int y2)
      {
      return (x1 == x2) || (abs(x2 - x1) == abs(y2 - y1);
      }
```

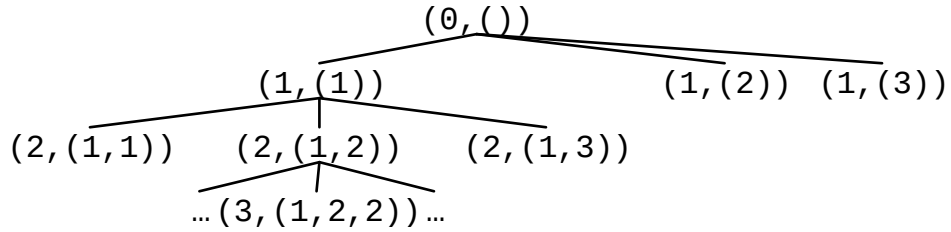
- la fonction `acceptable` vérifie que la reine que l'on souhaite positionner sur l'échiquier n'est pas en prise avec les reines précédemment placées.

```
int acceptable(int x, int y)  {
      int ci;
      for (ci = n - 1; ci >= 1; ci--)
            if (en_prise(solution[ci], ci, x, y)) return 0;
      return 1;
      }
```

- la fonction `parcours` énumère récursivement l'ensemble des solutions en positionnant l'une après l'autre les reines.

```
   void parcours(void) {
        int l;
        if (n == 9) afficher_solution();
        else
              for (l = 1; l <= 8; l++)
                   if (acceptable(l,n)) {
                         solution[n] = l; n = n + 1;
                         parcours(); n = n - 1;
                         }
        return;
        }
```

L'arbre de recherche obtenu par ce programme sera donc de la forme :




## 2.2. Le labyrinthe

Voici maintenant le deuxième exemple qui consiste à trouver un parcours dans un labyrinthe. Ici le labyrinthe est codé sous la forme d'un graphe (à chaque croisement on associe un noeud du graphe), lui même représenté par une matrice d'incidence `M` (`M[i][j]` est égal à `1` si il y a une arête entre le noeud `i` et le noeud `j`). Le programme C est le suivant :

```
int labyrinthe[N][N];
int entree,sortie;
int solution[N];

int acceptable(int n, int x)  {
int i;
for (i = 0; i <= n; i++)
      if (solution[i] == x) return 0;
return 1;
}

void essayer(int n, int i){
int d, x;
solution[n] = i;
if (sortie == i) afficher(n);
else
      for (d = 0; d < N; d++)
            if (labyrinthe[i][d] && acceptable(n,d))
                  essayer(n+1,d);
return;
}

void main(void) {
definir_labyrinthe();
essayer(0, entree);
return;
}
```

# Rappel et compléments

## *1. Le préprocesseur*

Le préprocesseur travaille, avant la compilation, sur le texte du fichier source écrit en C. En fonction des directives rencontrées au cours de la lecture du programme, il traite le fichier source et génère un texte intermédiaire[7]. Les directives du préprocesseur commence toujours par le caractère # et se terminent toujours à la fin de la ligne, sauf si cette dernière est terminée par le caractère \ .

### 1.1. Inclusion de fichier

L'inclusion de fichier dans un programme C est l'un des outils de modularité du langage. Elle permet d'introduire dans le fichier source des références à des objets situés dans d'autres fichiers. L'inclusion de fichier se fait par l'une des directives suivantes :

```
#include <nom_de_fichier>
#include "nom_de_fichier"
```

Dans les deux cas, le fichier inclus est un programme C. La première forme d'inclusion indique que le fichier à inclure se trouve dans les répértoires standard de travail du compilateur. La seconde indique que le fichier à inclure doit d'abord être cherché dans le répertoire contenant le programme source, puis dans les répertoires standard de travail du compilateur. Généralement, la première forme est utilisée pour les fichiers standard, et la deuxième pour les fichiers définis par l'utilisateur.

### 1.2. Définition de macro

Une macro est une instruction du préprocesseur associant un nom à une suite de caractères. Avant la compilation, le nom sera remplacé textuellement par la suite de caractères auquel il est associé.

#### 1.2.1. Macro sans argument

Les macros sans arguments se définissent par la directive

[7] Ce texte intermédiaire est toujours du C.

```
#define nom texte_de_remplacement
```

Le nom utilisé dans une directive `define` a la même forme qu'un nom de variable. Le texte de remplacement est quelconque ; il est en principe constitué du reste de la ligne, mais on peut le prolonger sur plusieurs lignes en plaçant le caractère \ à la fin de chaque ligne incomplète.

Voici quelques exemples de macros sans arguments :

```
#define     NB_LIGNE    20
#define     BOUCLE_INFINIE    for(;;)
#define     BEGIN {
```

### 1.2.2. Macro avec arguments

Il est également possible de définir des macros avec arguments, de telle sorte que le texte de remplacement diffère suivant les appels de la macro. La définition d'une macro avec arguments est assez similaire à celle d'une macro sans arguments :

```
#define nom(ident1, … , identn) texte_de_remplacement
```

De telles macros s'utilisent sous une forme similaire à un appel de fonction :

```
nom(texte1, …, texten)
```

et lors de la substitution de la macro par son texte de remplacement, $ident_i$ sera remplacé par $texte_i$.

Voici deux exemples de macros avec arguments :

```
#define     PERMUT(a,b) {int t; t=a;a=b;b=t;}
#define     toupper(c)  (islower(c) ? (c) + ('A'-'a') : (c))
```

Lorsqu'on définit une macro avec arguments, il est important de prendre les précautions suivantes :

- il ne doit pas y avoir de blanc entre le nom de la macro et la parenthèse ouvrante,
- les arguments de la macro, s'ils sont utilisés dans le texte de remplacement, doivent être entourés de parenthèses pour éviter de nombreux problèmes,
- si le texte de remplacement est une expression, il est fortement conseillé de l'entourer aussi de parenthèses,
- pas d'expression à effet de bord dans une macro.

Illustration par l'exemple du non respect de ces mesures :

```
#define TVA (p) 0.186*p
l'expression TVA(10) est expansée par (10) 0.186*p   erreur

#define TVA(p) 0.186*p
l'expresion TVA(10+2) est expansée par 0.186*10+2 ce qui ne
correspond pas à notre pensée

#define TVA(p) 0.186*(p)
l'expresion (int) TVA(10) est expansée par (int)0.186*10 ce qui
ne donne comme résultat 0

#define TVA(p)    (0.186*(p)) est donc la bonne écriture

PERMUT(i++,j++) est expansée par {int t; t=i++;i++=j++;j++=t}
on voit ici que i et j sont incrémenté deux fois.
```

## 2. Les arguments de la fonction `main`

Il est souvent intéressant de paramétrer l'exécution d'un programme par des arguments fournis au moment de son exécution. Un programme C commençant toujours par l'exécution de la fonction `main`, c'est par elle et par ce que vous tapez sur la ligne de commande, que se fera la communication. Le prototype de la fonction `main` est le suivant :

```
int main (int argc, char *argv[]);
```

où `argc` est un entier indiquant le nombre d'arguments fournis, et `argv` est un tableau de chaînes de caractères contenant les arguments du programme (le nom du programme est le premier paramètre).

## 2.1. Premier exemple

Illustrons nos propos en écrivant une fonction `echo` qui affiche les arguments reçus sur la ligne de commande. Le programme est donc le suivant :

```
int  main (int argc, char *argv[])
{
int i;

for (i = 0; i < argc; i++)
      printf("%s\n",argv[i]);
return 0;
}
```

Lors de son lancement, il se trouve dans la configuration de la figure n° 1 :

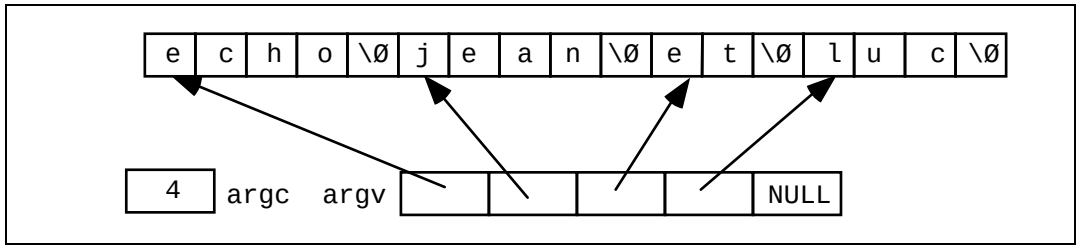



Figure n° 1 : Initialisation des arguments de la fonctions `main` lors de la commande `echo jean et luc`

---

A la fin de l'exécution d'un programme, il est courant que celui-ci indique s'il s'est ou non correctement terminé. La première des fonctions lancée dans un programme C étant la fonction `main`, c'est à elle que va incomber cette tâche. Par convention, si la fonction `main` renvoie 0 le programme s'est correctement déroulé, sinon elle renvoie un code d'erreur.

## 2.2. Deuxième exemple

Un deuxième exemple pour bien comprendre le mécanisme d'initialisation, par le système, du tableau `argv` et de la variable `argc`, est le programme de copie d'un fichier :

```
#include <stdio.h>

FILE *src, *dest;

int main(int argc, char *argv[])
{
char tampon[512];
int n;

if (argc != 3) {printf("nb arguments incorrect"); return 1;}
if ((src = fopen(argv[1],"rb"))== NULL)
      {printf("erreur d'ouverture"); return 2; }
if ((dest = fopen(argv[2],"wb"))== NULL)
      {printf("erreur d'ouverture"); return 3; }
while ((n = fread(tampon,sizeof(char),512,src))>0)
      fwrite(tampon,sizeof(char),n,dest));
fclose(dest);fclose(src);
return 0;
}
```

dont l'utilisation sur la ligne de commande (DOS ou UNIX) sera `copie toto.c tutu.c`

## 3. Les fonctions avec un nombre variable d'arguments

Le C ANSI permet de définir des fonctions dont le nombre d'arguments n'est pas fixé et peut être même variable d'un appel à l'autre.

Une fonction avec un nombre d'arguments variable est déclarée en explicitant au moins un argument fixe, puis en remplaçant les autres arguments par **...** . A l'interieur de la fonction, on accède aux arguments anonymes au moyen des macros `va_arg` , `va_start`, `va_end` et `va_list` définies dans `stdarg.h` :

- `va_list ident` déclare une variable permettant l'accès au arguments anonymes de la fonction ; cette variable sera utilisée par les macros `va_start` et `va_arg`
- `va_start(ident,dernier_argument)` initialise le pointeur sur le dernier argument explicitement nommé;
- `va_arg(ident,type)` le premier appel de cette macro donne le premier argument anonyme, les appels suivants donnent les arguments suivants ; `type` décrit le type de l'argument référencé ;
- `va_end(ident)` remet tout en ordre pour un fonctionnement normal.

Voici maintenant un exemple d'utilisation de cette technique :

```
#include <stdarg.h>

int max(int n, int x, ...)
{
va_list les_autres;
int i, m = x,t;

va_start(les_autres,x);
for(i = 2; i <= n; i++)
      if ((t=va_arg(les_autres,int)) > m)
            m = t;
va_end(les_autres);
return m;
}
```

et des appels possibles à cette fonction :

```
max(3,1,4,9);
max(5,a,c,v,f,h);
```

## 4. La macro `assert`

La macro `assert` est une aide à la mise au point d'un programme. Elle permet d'arrêter un programme pendant son exécution lorsqu'une condition posée n'a pas été respectée. Incluse dans le fichier `assert.h`, cette macro s'utilise sous la forme suivante :

```
assert(expression);
```

où `expression` est la condition à vérifier. Si cette condtion est fausse, le programme s'arrêtera.

L'exemple naïf suivant illustre le fonctionnement de cette macro :

```
#define TAILLE 100
int table[TAILLE];
void zero(int i)
{
assert(i >=0 && i < TAILLE);
table[i] = 0;
return;
}
```

Dans la fonction `zero` (appartenant au fichier `essai.c`) chargé de mettre à zéro l'élément `i` du tableau `table`, si la valeur de `i` sort des limites du tableau (la condition ne sera donc pas vérifiée), alors le message suivant apparaîtra

```
Assertion failed : i >=0 && i < TAILLE, file essai.c, line 5
```

---

Lorsque le programme fonctionne parfaitement (la phase de mise au point est donc terminée), il convient de supprimer tous les appels à la macro `assert`. Pour cela, il suffit de définir dans le programme l'identificateur `NDEBUG`[8].

```
#define NDEBUG
#define TAILLE 100

int table[TAILLE];

void zero(int i)
{
assert(i >=0 && i < TAILLE);
table[i] = 0;
return;
}
```

## 5. La notion de `lvalue`

Toute expression possède au moins deux attributs : son type et sa valeur. En C, certaine expression possède un troisième attribut qui est leur adresse. De telles expressions sont appelées des `lvalue`[9], leur valeur étant alors la valeur contenue à l'adresse en question. Il est important de savoir reconnaitre les `lvalue` puisqu'elles seules peuvent se trouver à gauche du symbole d'affectation (`=`).

Concrètement et très schématiquement:

- sont des lvalue les nom de variables (et ceci quelque soit leur type),
- ne sont pas des lvalue les constantes, les noms de tableaux et les noms de fonctions.

---

[8]La valeur de cet identificateur n'aura strictement aucune importance, seule son existence compte.

[9]Une expression qui n'est pas une `lvalue` est appelée une `rvalue`