Cours de Programmation en C

Mendy Fatnassi

10 décembre 2020

Table des matières

Chapitre 1 : Généralité

Il existe plusieur type de langage :

-Langage declaratif : Il n'y aurait pas d'algorithme , par exemple : Quel est actuellement le president de la republique francaise. La machine doit trouver la reponse toute seule par analyse très fine de la question et s'appuyer sur des bases de connaissance.

-Langage imperatif : Un algorithme decrit comment faire : Donnez moi la valeur de B en faisant B=5*3. La reponse de la machine sera B=15. On a donc dit comment il faut faire pour trouver la solution.

Chapitre 2 : Les Fichiers

2.1 Flux E/S

Le flux d'entrée est appelé "entrée standard" (standard input), le flux de sortie "sortie standard" (standard output), et le flux d'erreur est souvent appelé "erreur standard" (standard error).
Ces noms sont abrégés dans les dénominations symboliques de ces fichiers : stdin, stdout et stderr.

Chacun de ces symboles est une macro de stdio de type pointeur sur un FILE, et peut être utilisé dans des fonctions comme fprintf ou fread.

Comme les FILE sont simplement des coquilles entourant les descripteurs de fichiers en ajoutant une mémoire tampon, il est également possible d'accéder aux fichiers UNIX "bruts", avec des fonctions comme read et lseek.

Au démarrage du programme, les descripteurs de fichier associés aux flux stdin(0), stdout(1) et stderr(2) sont 0,1 et 2.

2.2 Ouverture d'un fichier

Mode d'ouverture :

-**"r"** : lecture seule. Vous pourrez lire le contenu du fichier, mais pas y écrire. Le fichier doit avoir été crée au préalable.

-**"w"** : écriture seule. Vous pourrez écrire dans le fichier, mais pas lire son contenu. Si le fichier n'existe pas, il sera crée.

-**"a"** : mode d'ajout. Vous écrirez dans le fichier, en partant de la fin du fichier. Vous ajouterez donc du texte a la fin du fichier. Si le fichier n existe pas, il sera crée.

-**"r+"** : lecture et écriture. Vous pourrez lire et écrire dans le fichier. Le fichier doit avoir ete crée au prealable.

-**"w+"** : lecture et ; avec suppression du contenu au préalable. Le fichier est donc d abord vide de son contenu, vous pouvez y écrire, et le lire ensuite. Si le fichier n'existe pas, il sera crée.

-**"a+"** : ajout en lecture / écriture à la fin. Vous ecrivez et lisez du texte a partir de la fin du fichier. Si le fichier n'existe pas, il sera crée.

Ouverture d'un fichier :
FILE fopen(const char* nomDuFichier, const char* modeOuverture) ;*

```
int main(int argc, char *argv[])
{
FILE* fichier = NULL;
fichier = fopen("test.txt", "r+");

  fclose(fichier);
return 0;
}
```

Ne pas oublier de fermé le fichier une fois sont utilisation fini grace a la fonction :
int fclose(const char nomDuFichier) ;*

2.3 Lecture/Ecriture dans les fichiers

Ecriture :

-**fputc** : écrit un caractère dans le fichier (UN SEUL caractère à la fois) .
int fputc(int caractere, FILE pointeurSurFichier) ;*

-**putchar** : écrit un caractere sauf que le flux imposé est stdout.
int putchar(int character) ;

-**fputs** : écrit une chaîne dans le fichier .
char fputs(const char* chaine, FILE* pointeurSurFichier) ;*

-**fprintf** : écrit une chaîne « formatée » dans le fichier, fonctionnement quasi-identique à printf.
S'ecrit comme printf mais avec le nom du fichier avant :
fprintf(fichier, "Le Monsieur qui utilise le programme, il a %d ans", age) ;

Lecture :

-**fgetc** : lit un caractère dans un fichier.
int fgetc(FILE pointeurDeFichier) ;*

-

2.3.1 Saisie sécurisé

Supposons qu'on demande une chaîne de caractères à l'utilisateur, mais que celui-ci insère un espace dans sa chaîne :

```
printf("Quel est votre nom ? ");
scanf("%s", nom);
printf("Ah ! Vous vous appelez donc %s !\n\n", nom);
```

Resultat :

Quel est votre nom ? Jean Dupont
Ah ! Vous vous appelez donc Jean !

Pourquoi le « Dupont » a disparu ?
Parce que la fonction scanf s'arrête si elle tombe au cours de sa lecture sur un espace, une tabulation ou une entrée.
Vous ne pouvez donc pas récupérer la chaîne si celle-ci comporte un espace.

En fait, le mot"Dupont" se trouve toujours en mémoire, dans ce qu'on appelle le buffer. La prochaine fois qu'on appellerascanf, la fonction lira toute seule le mot "Dupont" qui était resté en attente dans la mémoire.

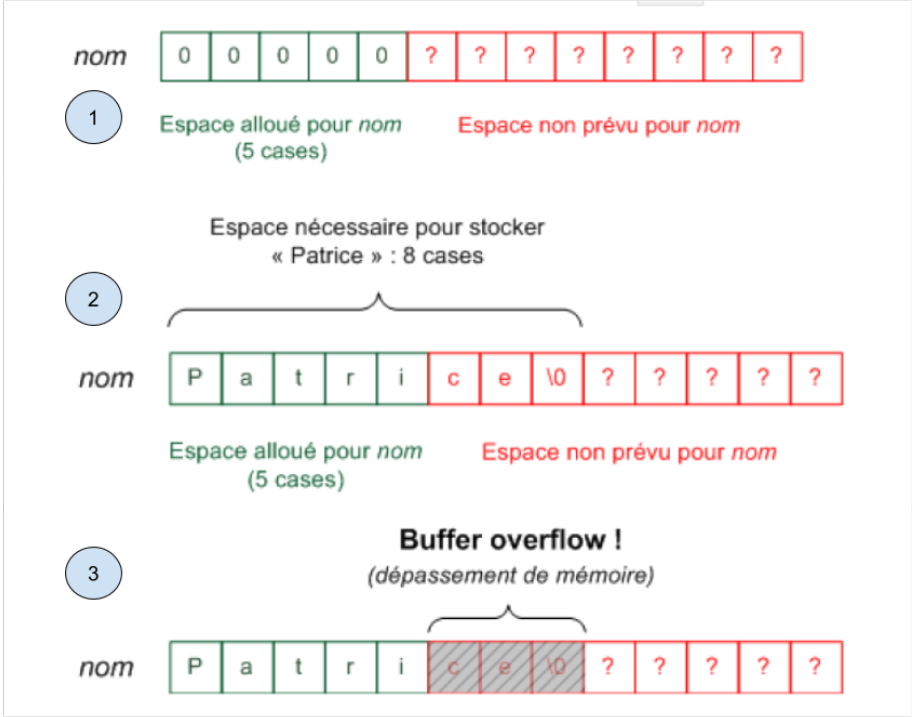
Autre probleme, beaucoup plus grave encore : celui du dépassement de mémoire.
Dans le code que nous venons de voir, il y a la ligne suivante : char nom[5] = 0 ;

Vous voyez que j'ai alloué 5 cases pour mon tableau de char appelé nom (fig.1). Cela signifie qu'il y a la place d'écrire 4 caractères, le dernier étant toujours réservé au caractère de fin de chaîne `\0`.

Si jamais on ecrit plus de caractere que prévue a l'allocation , il y aura un depassement de memoire.
Exemple :
Quel est votre nom ? Patrice
Ah ! Vous vous appelez donc Patrice !

A priori, il ne s'est rien passé. Et pourtant, on vient de faire un dépassement de mémoire, aussi appelé buffer overflow en anglais.
Comme vous le voyez sur la fig.2 , on avait alloué 5 cases pour stocker le nom, mais en fait il en fallait 8.
Qu'a fait la fonction scanf ? Elle a continué à écrire à la suite en mémoire comme si de rien n'était ! Elle a écrit dans des zones mémoire qui n'étaient pas prévues pour cela.

Les caractères en trop ont « écrasé » d'autres informations en mémoire. C'est ce qu'on appelle un buffer overflow (fig.3).



Sans entrer dans les détails, il faut savoir que si le programme ne contrôle pas ce genre de cas, l'utilisateur peut écrire ce qu'il veut à la suite en mémoire. En particulier, il peut insérer du code en mémoire et faire en sorte qu'il soit exécuté par le programme. C'est l'attaque par buffer overflow, une attaque de pirate célèbre mais difficile à réaliser.

On peut utiliser la fonction scanf de telle sorte qu'elle lise les espaces, mais c'est assez compliqué.
A la place nous allons voir comment faire des saisie sécuriser.

On peux utiliser gets une fonction qui lit toute une chaine de caractere mais n'est pas sécuriser ou alors on peux utiliser fgets qui agit comme gets mais en version sécuriser.
fgets peux lire dans un fichier mais egalement lire sur l'entrée standard stdin (donc le clavier).

Exemple :

```
printf("Quel est votre nom ? ");
fgets(nom, 10, stdin);
printf("Ah ! Vous vous appelez donc %s !\n\n", nom);
```

Resultat :
Quel est votre nom ? Mateo Ah ! Vous vous appelez donc Mateo !
Ça fonctionne très bien, à un détail près : quand vous pressez "Entrée", fgets conserve le `\n` correspondant à l'appui sur la touche "Entrée". Cela se voit dans la console car il y a un saut à la ligne après "Mateo".

Chapitre 3 : Fonction

3.1 Fonction Connue

Il existe des fonction predefinie propre au langage C tel que :

Conversion de type :

atoi : Convertit un caractere ou chaine en entier .
*int atoi(const char *str) ;* includestring.h

itoa : *Convertitunentierenchaineoucarcatere.*
*char * itoa (int value, char * str, int base) ;*

Operationsurleschainesdecaractere :

strcat : *Cettefonctionpermetderajouterunechanedecaractresdjexistantelecontenud'unese*
*char * strcat(char * destination, const char * source) ;#includestring.h*

strcmp : *Cettefonctionpermetdecomparer*2*chainedecaractere.*
*int strcmp(const char * first, const char * second) ;#includestring.h*

strpy : *Cettefonctionpermetdecopierunechainesourcedansuneautrechainedestination.*
*char * strcpy(char * destination, const char * source) ;#includestring.h*

3.2 Nombre Aleatoire

Pour cela on utilise la fonction int rand() qui delivre un entier compris entre 0 et 32 000 (includestdlib.h).
Pour initialiser le tirage aleatoire on fait appel a la fonction strand(time(0)).

Chapitre 4 : Les tableaux

Les tableaux sont une suite de cellule contigue en memoire stockant le meme type de valeur (int,char ...).
Ce sont des lvalue c-a-d qu'un tableau ne peut pas être affecté, autrement dit, il ne peut pas être la valeur à gauche (left value) d'une affectation.
Il se declare avec des crochet `[nb_elem]` et le type devant :

Declaration : *int tab[10]; -> type nomTab[nbElem];*

Pour utiliser un tableau il faut d'abord l'initialiser pour evite d'avoir a rechercher une case qui serait pas definie donc egale a NULL .
L'initialisation et l'affichage d'un tableau peux s'effectuer grace a une boucle (while,for,...) .

Passage en parametre d'une fonction :

Voci une belle transition pour passer au pointeur , lorsque l'on passe un tableau a une fonction celui ci n'st pas passer par valeur mais par adresse , ducoup il faut simplement faire :

```
void fonction (int tab[10]){ du code };\\
main(){
  int tab[10];
  fonction(tab);
}
```

Chapitre 5 : Les Pointeurs

5.1 Les Pointeurs

Un pointeur est une variable a pour valeur l'adresse d'une autre variable .Un pointeur se differencie d'une variable par son operateur etoile .

Declaration et affectation :

```
main(){
  int var;
  int *pointeur;

  pointeur=&var;
}
```

-Le déréférencement (ou indirection) :

C'est l'opération la plus simple sur les pointeurs. Comme son nom l'indique, il s'agit de l'opération réciproque au *référencement* (&) qui permet d'affecter un pointeur.
L'opérateur associé est l'étoile (), qui est aussi utilisé pour déclarer un type pointeur.Cet opérateur permet donc de transformer un pointeur de type T *, en un objet de type T, ainsi si l'on veux connaitre la valeur sur lequel le pointeur pointe on utilisera l'etoile .

Exemple :

```
main(){
  int var=15;
  int pvar=&var;
}
```

	valeur	adresse @
var	=15	&var=0x123
pvar	=0x123	&pvar=0x456

Donc ici la valeur de pvar est l'adresse de var mais pour savoir la valeur de cette adresse il faut utiliser l'operateur etoile *pvar.

On peux cree aussi des double pointeur **pvar . C'est a dire que ce pointeur auras l'adresse d'un autre pointeur.
Exemple :

```
main(){
  int var=15;
  int *pvar=&var;
  int **ppvar=&pvar;
}
```

5.2 Arithmetique des pointeurs (Tableaux)

Il faut savoir que l'on peux addition , multiplier , soustraire et diviser des pointeurs entre eux mais cela ne presente pas beaucoup d'interet par contre on peut plutot le faire mais avec une constante et un pointeur .
Cela est utile pour parcourire et manipuler un tableau par exemple .

Exemple :

```
int i;
int T[N];
for(i=0;*(T+i)<N;i++){
  printf("%d\n",*(T+i));
}
```

Que fait ce bout de code , *(T+i) represente le debut du tableau pointé T a l'adresse i si i=0 alors on commence au debut du tableau a l'adresse 0 , $T[i] = *(T + i)$.
En realite cela ce fait automatiquement par le compilateur , il traduite *(T+i) par :
Pour i=0 , $(T + i) = (@T + i*\text{taille\_objet\_pointe}) = (T + 0 * 4) = (T + 0)$ debut du tableau , ceci est l'adresse si on veut le contenue $*(T + i)$
taille_objet_pointe = 4 car c'est la taille du type int tout simplement.
Pour i=1 , $(T+i) = (T+1) = (T+1*4) = (T+4)$ 2ieme element du tableau
...

Pour i=N , $(T+i) = (T+N) = (T+N*4) = (T+N)$ Nieme element du tableau

On se deplace de la taille de l'objet a chaque fois pour se deplacer dans le tableau .

5.3 Gestion de la Memoire

Il faut savoir que lorsque l'on crée des variables ou fonction elle peuvent etre soit statique ou dynamique .

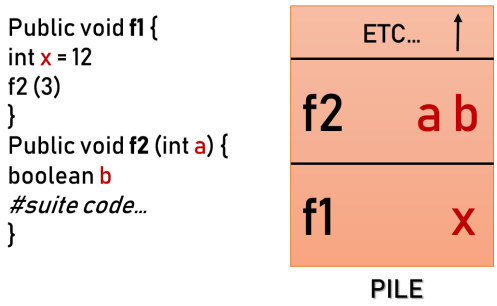
Statique : Elle se fait avant même l'exécution du programme, car elle concerne les variables globales, déclarées en dehors d'une fonction, entièrement connues, fixées et auxquelles on doit pouvoir accéder n'importe où dans le programme. Le compilateur allouera alors la variable en question de manière statique dans un segment .

Dynamique : Ensuite vient l'allocation dynamique de memoire gerer cette fois par la pile d'execution(stack) et le tas(heap).Il faudra donc veuiller a bien allouer de la memoire pour la variable dynamique et a la restituer grace au fonction malloc(t_size) et free() .

Pile/Stack :

Lors de l'exécution d'un programme, chaque appel de fonction implique la création d'un nouveau bloc au sommet de la pile, correspondant à la fonction en question.
L'intérêt du LIFO prend alors tout son sens : cela permet de connaitre l'endroit où chaque fonction active (dont l'exécution n'est pas terminée) doit retourner à la fin de son exécution.

La pile stocke également les données associées à chaque fonction, telles que les variables locales, les paramètres, etc... Lorsqu'une fonction se termine, le bloc de la pile correspondant, nécessairement au sommet de celle-ci, est libéré, donc les variables placées dans la pile par cette fonction sont supprimées.



Tas/Heap :

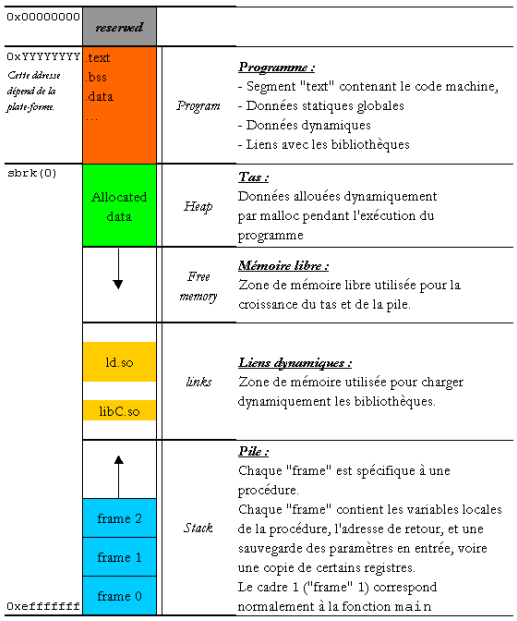
Le tas lui correspond au second segment de mémoire utilisé pour l'allocation dynamique. A la différence de la pile, vous êtes responsable de l'allocation de la mémoire sur le tas, avec l'utilisation de malloc() ou calloc() (fonctions C intégrées). Évidemment, vous devez également gérer l'utilisation de free () pour libérer cette mémoire dès que vous n'en avez plus besoin. Il est très important de le faire afin d'éviter les fuites de mémoire, qui provoquent la saturation de la mémoire machine.

Les variables créées sur le tas sont accessibles par n'importe quelle fonction et n'importe où dans le programme, rendant leur portée globale. Il n'en fallait pas plus pour ajouter une contrainte supplémentaire : la gestion de l'accès concurrentiel (Quand 2 utilisateur veulent acceder a une meme ressource en meme temps).

Pile ou Tas :

La mémoire sur le tas est plus grande et plus durable que sur la pile qui elle requiert une gestion plus complexe et donc plus lente .Le tas offre une bien plus grande liberté : au-delà de la portée globale des variables, celles-ci peuvent être redimensionnées à l'aide de la fonction realloc() ,la taille des variables dans le tas n'est pas limitée, du moins seulement par la limite physique de la mémoire.

On peux montrer le schema suivant pour resumer cette partie sur la gestion de la memoire :



5.4 Tableau heterogene

Pour ce deplacer dans un tableau heterogene , il suffit de se deplacer de la taille de l'objet .
Pour connaitre l'adresse de debut du tableau , peut importe le type ca sers toujour en (T+0) , mais si on veux aller sur le debut de la case du INT alors on se deplace de sizeof(char) pour arriver au debut de la case du INT.

L'odrinateur calcule automatiquement les indice de la facons suivant : (T+i*Taille_objet_poi
.

Exemple :

```
char *T=malloc(sizeof(char));
printf("%c\n",*(T)); \\lecture 1er element
T=T+1; \\char = 1
printf("%d\n",*(int *)T); \\lecture 2ieme element
T=T+sizeof(int);
printf("%d\n",*(double*)T); \\lecture 3ieme element
```

```
T=T+sizoef(double) \\ deplacement a l'element suivant
...
```

5.5 Pointeur de Fonction

-Declaration e :

type_ t (ptfonction) (...parametres ...) ;

-Exemple :

double(F2)(void) ; /Sans parametre/
int(F3)(int, S_t) ; /avec 2 parametres/

Pour creer un affichage (ou autre) générique on peux déclarer des pointeurs de fonction dans la structure de l'objet :

```
typedef struct entier_s entier_t ;

struct entier_s
{
  /* Attributs */
  int i ;
  /* Methodes */
  void (*afficher)(entier_t *) ;
  void (*incrementer)(entier_t *) ;
  void (*decrementer)(entier_t *) ;
} ;
```

Ou alors on peux creer une enumeration et faire une fonction d'aiguillage :

```
typedef enum message_s { AFFICHER , INCREMENTER , DECREMENTER } message_t ;

typedef struct objet_s objet_t ;

struct objet_s
{
  /* Attributs (vide) */
```

```
  /* Methodes (1 seule : aiguillage) */
  void (*switching)(objet_t * , message_t) ;
} ;

static
void afficher_int( entier_t * entier )
{
  printf( "%d\n" , entier->i ) ;
}

static
void decrementer_int( entier_t * entier )
{
  entier->i-- ;
}

static
void incrementer_int( entier_t * entier )
{
  entier->i++ ;
}

/* Corps aiguillage specifique a entier_t */
static
void switching_int( objet_t * self , message_t message )
{
  switch(message)
    {
    case AFFICHER :
      ((entier_t *)self)->afficher((entier_t *)self) ;
      break ;
   case INCREMENTER :
      ((entier_t *)self)->incrementer((entier_t *)self) ;
      break ;
    case DECREMENTER :
      ((entier_t *)self)->decrementer((entier_t *)self) ;
      break ;
    }
}
```

5.6 Encapsulation / Pointeur generique

Le type void* permet de pointé sur n'importe qu'elle type d'objet (int,char,struct_t,...) .

Exemple :

```
int entier=5;
int *pent=&entier;
void *pgen;

pgen=pent; //ok
pent=pgen; //ko deferencement
```

Un pointeur generique void* ne peux pas etre deferencé c'est a dire etre a droite de l'affectation .

Utilité :
Le principe d'utiliser des pointeur generique est de permettre de rendre des fonction generique capable de prendre en parametre n'importe qu'elle type d'objet on appelle cela l'encapsulation/call back.
On utilise une fonction classique qui fait sont traitement sur des int par exemple "void afficher_int(int *entier)" et ensuite on cree une 2ieme fonction qui vas venir encapsuler la premiere grace aux pointeur generique "void afficher_int (void *obj)" ou l'on retournera la 1ere fonction . Il faut que "int entier" soit de type pointeur lors du passsage en parametre de la fonction "int *entier" sinon on ne pourras pas referencer le void* .

Exemple :

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

#include "test.h"
#define N 10

void afficher_int (int *ent){
  printf("%d\n",*ent);
  printf("Entier (int) : %d \n",*ent);
```

```
}

//Fonction qui encapsule et qui retourne la fonction pour le bon type d'objet
void afficher_int_cb (void * ent){
  return (afficher_int(ent));
}

void afficher_char(char *car){
  printf("Caractere (char) : %s \n",car);
}

void afficher_char_cb(void *car){
  return (afficher_char(car));
}

void afficher_string(char *string){
  printf("Chaine de caractere (char *) : %s \n",string);
}

void afficher_string_cb(void *string){
  return (afficher_string(string));
}

//Fonction generique qui permet de choisir les fonction d'encapsulation pour
void afficher (void * obj,int val){
  if(val == 0){
    afficher_int_cb(obj);
  }
  else if (val == 1){
    afficher_string_cb(obj);
  }
  else{
    afficher_char_cb(obj);
  }
}

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

  int ent;
  char car;
  char *string=malloc(sizeof(char *));
```

```
  printf("Entrez entier : ");
  scanf("%d",&ent);

  printf("Entrez chaine de caractere : ");
  scanf("%s",string);

  printf("Entrez un caracter : ");
  scanf("%s",&car);

  printf("\n\n");

  afficher(&ent,0);
  afficher(string,1);
  afficher(&car,2);

  free(string);
  string=NULL;

  return 0;
}
```

Chapitre 6 : Structure

6.1 Type Structure

:

Declaration :

```
#DEFINE MAX 20

typedef struct s_individu {
      char nom[MAX] ;
      char prenom[MAX] ;
      struct individu * pere ;
      struct individu * mere ;
} t_individu ;
```

De plus cette structure est recusif c-a-d que dans ca declaration elle s'appelle elle meme (pere,mere).
Ducoup si on declare une variable :
Avec typedef : t_individu Nom_var ;
Sans typedef : struct s_individu Nom_var ;

6.1.1 Passage en parametre de fonction

Si on declare une variable de type t_individu ,on feras un passage par addrese a la fonction pour utilise notre structure sans perdre de donnee.
exemple :

```
int indiv_creer(t_individu *Indiv){

}

void main(){
```

```
  t_individu Indiv;
  indiv_creer(&Indiv);
}
```

6.2 Enumeration et Union

Enumeration : *enum nom_ enum val1, val2,..., valN;*

Union : Se declare et s'utilise comme une structure .Contrairement a une structure une union ne peux contenir qu'un seul de ces membres a la fois.

```
typedef struct nombre_s{
    unsigned entier : 1;
    unsigned flottant : 1;
    union
    {
        int e;
        double f;
    } u;
}nombre;

static void affiche_nombre(struct nombre n)
{
    if (n.entier)
        printf("%d\n", n.u.e);
    else if (n.flottant)
        printf("%f\n", n.u.f);
}
```

Chapitre 7 : Recusitivité

7.1 Recusitivité

Definition :
Une fonction est dite recursive quand elle faut appele a elle meme pour traiter certaine operation .
Une fonction recursive doit avoir un "test d'arret" pour sortir de l'appel recursif .

On creer autant de contexte d'execution qu'il y a d'appel de la fonction , il y a donc une place memoire reserver pour les variable local de la fonction a chaque appel, c'est a dire que pour un meme appele de fonction on auras differente variables local meme si elle portent le meme nom .

Exemple :
fact(n)–> sauvegarde de n en var local pour le 1er appel
fact(n)–> sauvegarde de n en var local pour le 2ieme appel
Pourtant les 2 n sont different ils portent le meme nom meme pas a la meme adresse .

Rappel : Variable Global = 1 exemplaire $\neq$ Variable local = n exemplaire en memoire

Un exemple de fonction recursive , la factoriel :

```
/* fonction qui renvoie n! */
int fact(int n) {
  if (n == 1)         //Condition d'arret
    return 1;
return n * fact(n-1); // Appel recursif
}

Appel : printf(“%i”, fact(3));
```

7.1.1 Recusitivite Terminal/Non-Terminal

On dit qu'une fonction est Terminal quand la derniere instruction finit par l'appel de la fonction $\neq$ Non-Terminal .

Exemple :

La fonction fact du dessus est non Terminal car la derniere instruction se finit par une multiplication (n*fact(n-1)) , le derniere terme est n et non fact() .

Si on prend une version Terminal de la fonction fact cela donnerai :

```
/* fonction qui renvoie n! */
int fact2(int n, int result){
  if (n == 1)
    return result;
return fact2(n-1, n*result);
}

Appel : printf(“%i”, factorielle(3,1));
```

7.1.2 Derecursivation

Toutes fonction recursive a une eccriture equivalente en **iterative** (en generale grace a une boucle while) on appele cela la **derecursivation** .

On peux derecursiver une fonction non-terminal mais il faut prendre en compte la gestion d'une pile (stack) .

Chapitre 8 : Complexité

8.1 definition

L'idée en deux mots de la complexité , c'est :

Si je donne a mon programme une entrée de taille N , quel est l'ordre de grandeur,enfonction de N,du nombre d'opérations qu'il vas effectuer ? .

Complexité temporelle : Une fonction de N qui mesure le temps de calcul pour une donnée de taille N .

Complexité en mémoire : Une fonction de N qui mesure laplace mémoire utilisé pour le calcul sur une donnée N .

Pour la complexité temporelle , on parle soit dans le "pire des cas" , cad on donne une borne superieure sur le temps de clacul pour toutes les données de taille n. Ou on parle de moyenne „ on fait la moyenne des temps de calculs pour toutes les données de taille N. Il faut savoir que le temps de calcul peut varier d'une machine a l'autre (il s'agit donc d'une estimation du temps de calcul) .

Pour calculer la complexité d'un algorithme on calcule le nombre d'opérations "élémentaires" faites (additions,multiplications,etc.).

8.2 Complexité Exponentiel O(10^n)

Prenons un cadenas à trois chiffres,le but est de trouver la combinaisions pour ouvrire le cadenas. Nous décidons de tester, une à une, toutes les combinaisons.

Nous essayeons le nombre le plus petit (000) puis le nombre suivant (001) et ainsi de suite jusqu'à atteindre 999. En effet, nous calculons que cette stratégie nous prendra, dans le pire des cas, 30 minutes.

Effectivement, quelques minutes plus tard, le cadenas s'ouvre. Hourra ! Le code était 123. Facile ! Notre stratégie était efficace et nous nous félicitons d'être si intelligents.

Mais si nous prenons un cadenas à 4 chiffres. Forts de notre premier succès, nous nous basons sur le même algorithme pour trouver le code. Tester toutes les combinaisons possibles aurait pris plus de 5 heures.

Comment le temps de calcul peut-il passer de 30 minutes à 5 heures en ajoutant un simple chiffre ? Le secret, c'est la complexité de notre algorithme.

Si le code a 3 chiffres, il faut tester 1000 combinaisons (eh oui, tous les nombres entre 000 et 999). En revanche, s'il en a 4, il faut en tester 10 000. S'il en avait eu 5, il aurait fallu en tester 100 000, et ainsi de suite. Nous nous apercevons donc que notre algorithme d'ouverture de coffre dépend du nombre de chiffres du code donc de la taille de la donnée N.

8.3 Complexité Linéaire O(n)

Prenons le meme cadenas a 4 chiffres , afin de trouver le premier chiffre du code, on tente 10 combinaisons. Quand on l'a trouvé, on passe au suivant et teste de nouveau 10 combinaisons Et ainsi de suite pour chaque chiffre ! On n'aura donc à tester que 40 combinaisons (10 + 10 + 10 + 10, soit 10 x 4) pour ce cadenas à quatre chiffres (ce qui est mieux que les 10 000 combinaisons que nous nous apprêtions à essayer...).

Si l'on y regarde de plus près, on teste 10 nouvelles combinaisons pour chaque nouveau chiffre du cadenas. Autrement dit, si n est le nombre de chiffres, on teste 10 x n combinaisons. Vous l'avez dans le mille, la "complexité" de cette algorithme était bien meilleure que celui d'avant (complexité exponentiel).

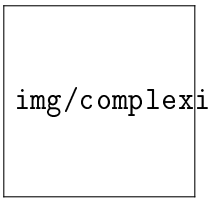
8.4 Complexité en Temps Constant O(1)

Maintenant prenons un cadenas à 500 chiffres et essayons l'aglorithme linéaire , cela nous prendrais environ 3h pour trouver le code , nous revoila au meme problèmes qu'au début.
Aussi efficace que soit la technique, le nombre de chiffres du cadenas est trop

important. Autrement dit, la taille des données du problème excède la capacité de notre algorithme.

Une complexité en temps constant , quel que soit le nombre de chiffres, il prendra toujours le même temps.

Voici un petit recapitulatif des complexité :



Chapitre 9 : Trie

Les algorithme de trie peuvent etre tres efficace comme tres lents , c'est pourquoi pour comparer 2 algorithme de trie on regarde leur complexite en O(n) , nombre d'instruction executer.