

Programmation Impérative 2

Introduction au langage C

Amaury Habrard

Faculté des Sciences et Techniques
Université Jean Monnet de Saint-Etienne

Licence Informatique 2 - Semestre 4
2017 - 2018

- Professeur des Universités
 - à l'Université Jean Monnet, St-Etienne, France
 - Responsable du Master Informatique, co-responsable parcours MLDM
 - au Laboratoire de recherche Hubert Curien
 - Responsable de l'équipe de recherche *Data Intelligence*
- Domaine de recherche :
 - Domaine principal : Apprentissage Automatique (*Machine Learning*)
 - aspects fondamentaux (statistiques, algorithmes, garanties en généralisation)
 - Applications à des problèmes de vision par ordinateur (*Computer Vision*), détection de fraudes et anomalies
- Comment me joindre :
Mail : `amaury.habrard@univ-st-etienne.fr`

- Concepts " avancés " de programmation impérative en langage Crécurtivité, Makefile, pré-processeur, debugueur, pointeurs, allocation dynamique, tableaux, structures, unions, listes chaînées, TAD Piles et Files, arbres, projet
- Objectifs : consolidés les acquis du premier semestre, maîtriser la gestion de la mémoire, création de types, et quelques concepts avancés

Références

- Les « slides » - la première partie de rappel est issue des transparents d'Emilie Morvant merci à elle !
- Poly de Anne Canteaut
- Claroline Connect (énoncés, corrections, exemples, polys, ...)
- Le Web ...

- Organisation : CM/TD - lundi/mardi matin + quelques séances vendredis matins
- TP : 3 groupes - 1 avec Leo Gautheron et 2 avec Christophe Moulin
Les TP sont à rendre.
- 2 Devoirs de contrôle
 - Vendredi 9 février 10h15
 - Jeudi 22 mars 13h30
- Un projet à partir d'avril.

Le langage C

Historique et caractéristiques

- Date : début années 1970
- Auteurs : Kernighan et Ritchie, Bell Labs / ATT
- But : proposer un langage impératif compilé, à la fois de haut niveau et “proche de la machine” (rapidité d'exécution)
- Conçu pour être le langage de programmation d'Unix, premier système d'exploitation écrit dans un langage autre qu'un langage machine
- Diffusé grâce à Unix
- Popularisé par sa concision, son expressivité et son efficacité
- Disponible actuellement sur quasiment toutes les plate-formes

- Proche de la machine :

(+) rapidité, programmation facile sur nouvelles architectures matérielles

(-) exécutable non portable

- Langage simple :

(+) compilateur simple

(-) (dans les 1ères version) peu de vérification à la compilation, plantages

- Langage vieux (1970) et populaire :

(+) beaucoup d'outils annexes, de bibliothèques réutilisables

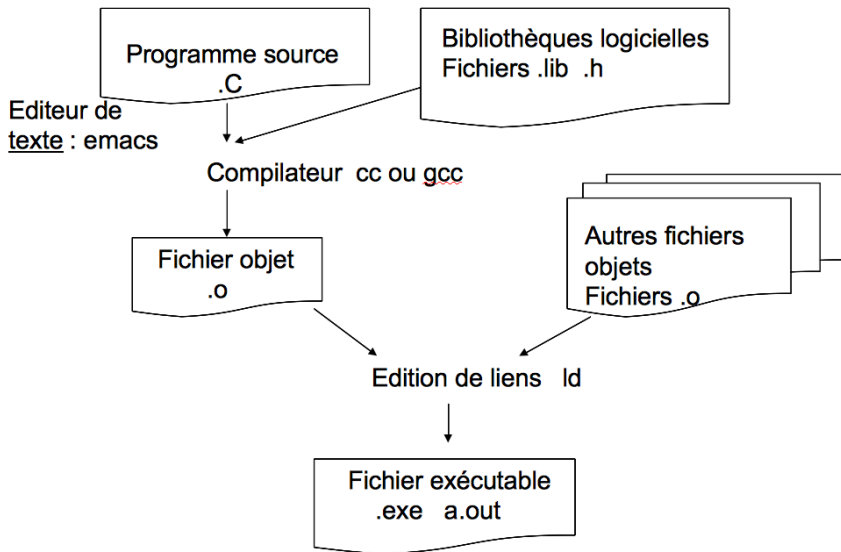
(-) ne supporte pas les “nouveauautés” (orienté objet, exception, ramasse-miettes...)

- Langage impératif et de haut niveau
 - Programmation structurée (= pas d'instruction "Go to" dans le programme, pour éviter le plat de spaghettis)
 - Organisation des données (regroupement en structures de données)
 - Organisation des traitements (fonctions/procédures avec paramètres)
Possibilité de programmer "façon objet"
- Langage de bas niveau :
 - Conçu pour être facilement traduit en langage machine
 - Gestion de la mémoire "à la main"
 - Attention : pas de gestion des exceptions

C'est un langage compilé

- Le programmeur écrit son programme sous la forme d'un code source, contenu dans un ou plusieurs fichiers texte d'extension ".c"
- Un programme appelé compilateur (habituellement nommé cc, ou gcc) vérifie la syntaxe du code source et le traduit en code objet, compris par le processeur
- Le programme en code objet ainsi obtenu peut alors être exécuté sur la machine (par défaut "a.out" si pas de nom spécifié)

Schéma de production de logiciel en C



Exemple de programme simple

Fichier texte "bonjour.c"

```
#include <stdio.h>      /* bibliothèque utilisée = liste de fonctions*/
#include <stdlib.h>      /* "" */

int main ()             /* mot clé obligatoire pour le début du prog*/
{                       /* début d'un ensemble d'instructions = bloc*/
    printf ("Bonjour!\n"); /* instruction d'affichage */

    return EXIT_SUCCESS;

}
```

Lignes de commandes

```
> gcc -Wall bonjour.c -o bonjour      Lancement de la compilation
                                     note : gcc fait aussi l'édition de liens
> ./bonjour                          Lancement du programme (nom par défaut)
Bonjour!                             Résultat de l'exécution
>
```

Les différentes phases de compilation

Les différentes phases de compilation sont :

- La compilation : le fichier engendré par le préprocesseur est traduit en assembleur i.e. en une suite d'instructions associées aux fonctionnalités du microprocesseur (faire une addition, etc.)
- L'assemblage : transforme le code assembleur en un fichier objet i.e. en instructions compréhensibles par le processeur
- L'édition de liens : assemblage des différents fichiers objets (nous en reparlerons)

A noter : la phase de compilation peut être précédée par une phase de traitement du préprocesseur (cf plus loin)

Comprendre un programme en C

Comprendre un programme

```
/* nom : diviseur.c
   entrée : entier n > 0      sortie : affiche les diviseurs de n */
#include <stdio.h>           /* Inclusion de fichiers */
#include <stdlib.h>
void usage (char *s) {      /* Déclaration d'une fonction */
    printf("Usage : %s <entier>",s);
    printf("<entier> positif\n");
}
int main (int argc, char *argv[]) { /* Déclaration d'une fonction */
    int n, i;                /* Déclaration de variables */
    if (argc < 2) {          /* Suite d'instructions dans */
        usage(argv[0]);      /* des blocs { ... } */
        exit(-1);           /* chaque instruction se */
    }                         /* termine par un ; */
    n= atoi(argv[1]);
    for (i=1 ; i <=n; i++)
        if (n%i == 0)
            printf(" %d ", i);
    printf("\n");
    return EXIT_SUCCESS;
}
```

Un programme C est classiquement composé de

- commentaires `/* obligatoire pour "lire" et comprendre */`
- directives d'inclusions de fichiers/librairies `# include`
- (variables globales)
- définitions de fonctions, dont `main()`

Les commentaires

- Encadrés par `/*` et `*/`
- Ignorés par le compilateur !!
- Texte libre, multi-ligne
- Utile pour relire et comprendre un programme (description en langage clair, choix effectués, etc.)

Exemple

```
/** /* tout ceci est du "commentaire"  
et ceci aussi : main( ) { i++;} */
```


Directives d'inclusions

- en début de ligne : `#include`
- Bibliothèques "système" :
`#include <nomBiblio>`
utilise un répertoire connu du compilateur, généralement :
`/usr/include`
Il existe :
`stdio, stdlib, ctype, string, time, math, locale, setjmp, etc.`
- Mes propres fichiers :
`#include "monProjet/nomFichier"`
utilise le chemin et le fichier indiqués

Définition de fonctions

- Un programme C est un ensemble de “fonctions”

Par convention les noms de fonctions commencent par une minuscule

- Structure :

```
type_sortie nom_fonction (type_1 argument_1, type_2 argument_2, ...){  
    type_variable_1 nom_variable_1 ;  
    type_variable_2 nom_variable_2 ;  
    ...  
    instruction_1 ;  
    instruction_2 ;  
    ...  
}
```

- Une fois définie (*i.e* “en dessous” dans le texte), la fonction peut-être appelée dans une autre fonction de la manière suivante : `nom_fonction(valeur_1,valeur_2,...) ;`

Définition de fonctions

- Une fonction est spéciale : `int main (in argc, char * argv[])`
 - Doit TOUJOURS être présente
 - C'est le point de commencement du programme :
c'est la fonction **principale, exécutée en premier**
 - Peut prendre 2 arguments : `argc` et `argv`
 - `argc` : nombre d'arguments sur la ligne de commande ("c" = count)
 - `argv` : liste des arguments ("v" = value)

Exemple

% diviseur 24

au début du programme, `argc = 2` et `argv = ["diviseur", "24"]`

Comprendre un programme

```
/* nom : diviseur.c
   entrée : entier n > 0      sortie : affiche les diviseurs de n */
#include <stdio.h>             /* Inclusion de fichiers */
#include <stdlib.h>
void usage (char *s) {        /* Déclaration d'une fonction */
    printf("Usage : %s <entier>",s);
    printf("<entier> positif\n");
}
int main (int argc, char *argv[]) { /* Déclaration dumain */
    int n, i;                  /* Déclaration de variables */
    if (argc < 2) {           /* verif nombre arguments */
        usage(argv[0]);       /* si probleme */
        exit(-1);             /* on sort du programme */
    }                          /*fin bloc if */
    n= atoi(argv[1]);          /* recuperation de n */
    for (i=1 ; i <=n; i++)      /* boucle de recherche */
        if (n%i == 0)          /* test diviseur */
            /*
                printf(" %d ", i); /* diviseur trouvé et affich */
            printf("\n");          /* affichage saut de ligne */
    return EXIT_SUCCESS;        /* tout s'est bien passé */
}
```

Exemples de fonctions pré-définies

- La fonction d'affichage

```
printf(<chaîne de caractère>{, <liste>})
```

- <chaîne> : chaîne de caractères, contenant aussi des formats comme :
 - %d entier, %f flottant, %c caractère, %s chaîne de caractères
 - ▶ caractères spéciaux : \n saut de ligne, \t tabulation, \\ caractère \
- <liste> : liste d'expressions (associées aux formats)
Chaque format correspond à une expression (dans l'ordre)
- Effet : affiche la chaîne de caractères en la "formatant" selon le format spécifié et les expressions calculées

Exemples de fonctions pré-définies

Exemple d'utilisation de la fonction `printf()`

```
int x;  
x = 5;  
printf ("Le carré de x est %d.\nMerci.\n", x*x);  
printf("Pour afficher \\, il faut écrire \\\\.\n");
```

Affiche :

Le carré de x est 25.

Merci.

Pour afficher \, il faut écrire \\\.

Remarque : `\n` en fin de chaîne permet d'“assurer” l'affichage (sinon risque de non affichage en cas d'erreur) et facilite la lisibilité.

Exemples de fonctions pré-définies

Une alternative, la fonction `puts(<chaîne de caractère>)`

- `<chaîne>` : chaîne de caractères simple **sans éléments de formatage**
- admet les caractères spéciaux : `\n` saut de ligne, `\t` tabulation, `\\` caractère `\`
- un saut de ligne est automatique ajouté à la fin.

```
puts ("Le carré de 5 est 25.\nMerci.");
```

Affiche :

Le carré de 5 est 25.

Merci.

Exemples de fonctions pré-définies

- La fonction de saisie de données
`scanf(<format>, <liste>)`
- La fonction `scanf` permet de récupérer des entrées de l'entrée standard, elle a des similitudes à `printf` dans la mesure où elle utilise les mêmes formats de conversion que `printf`,
 - `<format>` : format de lecture des données
 - `<liste>` : adresses des variables (pointeurs) auxquelles les données seront attribuées
- C'est une instruction bloquante :
le programme "attend" que l'utilisateur entre des valeurs, puis valide

```
int age;  
scanf("%d",&age);  
char prenom[20]; char nom[20];  
scanf("%19s %19s",prenom, nom); //lit au plus 19 lettres par t
```


Exemples de fonctions pré-définies

Exemples d'utilisation de la fonction `scanf()`

```
int jour, mois, annee;  
scanf("%d %d %d", &jour, &mois, &annee);
```

Lit trois entiers relatifs, séparés par des espaces, tabulations ou interlignes

Les valeurs sont attribuées respectivement aux variables `jour`, `mois`, `annee`

Si on entre : 06 10 2014

alors : `jour=6`, `mois=10` et `annee=2014`

Remarque : Une suite de signes d'espacement est évaluée comme un seul espace
On ne peut pas récupérer de chaînes contenant un espace avec `scanf`.

Problème avec scanf

```
char animal[10];  
printf("Entrez votre animal favori\n"),  
scanf("%s",animal);  
printf("Animal favori:  %s\n",animal);
```

Si la **taille limite** à saisir n'est **pas spécifiée**, il peut y avoir un problème de buffer overflow. Une alternative la fonctions fgets :

```
char animal[10];  
printf("Entrez votre animal favori\n"),  
fgets(animal,10,stdin);
```

Récupère 10 caractères depuis l'entrée standard, le **\0 est compris** ainsi que les espaces s'il y en a.

Si vous avez besoin de récupérer une chaîne avec un format spécifique, utilisez scanf, sinon pour une seule chaîne sans structure particulière, fgets est probablement plus adaptée.

Exemples de fonctions pré-définies

- La fonction de conversion d'une chaîne en entier

`atoi(<chaine>)`

- `<chaine>` : la chaîne de caractères que l'on désire convertir
- **Exemple** : `n = atoi("12");`

- Usage typique : conversion d'un nombre entré au clavier

Exemple

```
int main (int argc, char *argv[]) {  
    int n;  
    n=atoi(argv[1]);  
    ...  
}
```

Variables et types

Quelques généralités

- Objet de base en C - Un espace dans la mémoire où de l'information est stockée auquel est associé un identifiant dans le code pour manipuler cette donnée.
- Les variables sont typées et déclarées explicitement avant toute utilisation (permet d'éviter des erreurs via le compilateur qui contrôle)

Exemple : `type nomVariable ;`

- Triplet (type, nomVariable, valeur)
- La valeur peut changer
- L'initialisation peut se faire lors de la déclaration

Exemple : `int maVariable = 0;`

- Le nom des variables
 - Caractères de A à Z, a à z, 0 à 9 et _
 - Doit débiter par une lettre (convention : en minuscule)
 - Longueur quelconque (il existe une limite théorique)
 - Un nom significatif est impératif!!!!

Note sur les caractères

Le jeu de caractères normalisé par le langage C est le suivant :

A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' () * + , - . / :
; < = > ? [\] ^ _ { | } ~

A ceci s'ajoute l'espace, et les caractères de commande représentant :

- le saut de ligne
- le retour chariot
- la tabulation horizontale
- la tabulation verticale
- le saut de page.

L'utilisation d'autres caractères (comme les caractères accentués, par exemple) invoque un comportement défini par l'implémentation.

Dans la pratique, la plupart des compilateurs acceptent les extensions courantes comme IBM PC8 et ISO 8859-1 (aussi appelés respectivement OEM et ANSI dans le monde MS-DOS/Windows). Mais ils peuvent cependant être sujets à des problèmes de portabilité.

Le type des variables

- Le type d'une variable est une contrainte de sécurité :
 - “contrôle” les opérations et les valeurs admises
- 2 catégories de types
 - **Types simples**
 - Types construits à l'aide de structure

Les types simples (mots réservés)

- `char` : 1 seul caractère (8bits) (entre ' ')
- `char[]` : tableau de caractères = chaîne de caractères (entre " ")
ou `char *`

Attention : la manipulation est à apprendre

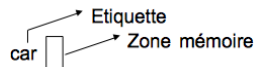
- `int` : entier (16 ou 32 bits, soit 2 ou 4 octets, selon l'architecture) - en 32 bits de -2 147 483 648 à 2 147 483 647
il existe aussi : `unsigned int`, `long int`, `unsigned long int` (0 à 4 294 967 295), `short int` (-32 767 à 32 767), ...
- `float` : flottants (réels - de $-3.4 * 10^{-38}$ à $3.4 * 10^{38}$) - `float x = 0.1;`
- `double` : flottants dit doubles (de $-1.7 * 10^{-308}$ à $1.7 * 10^{308}$) `double y = -.38; z= 4.25E4;`
`long double`
- `void` : aucune valeur

Les types simples

Exemples et notion de zone mémoire ( un octet
8 bits)

- Déclarations :

`char car;`



`int i;`

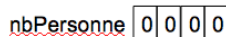


`double resultat;`



- Déclarations avec initialisation :

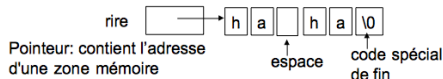
`int nbPersonne = 0 ;`



`char c = 'b';`



`char rire[] = "ha ha";`



Des variables constantes !

Mot clé : `const`

On l'utilise lors d'une déclaration de variable initialisée pour indiquer au compilateur d'**interdire tout changement de valeur de cette variable**

Exemple : `const int joursParSemaine = 7;`

Les (pseudo-)constantes (Pré-processeur)

- Définition en début de ligne juste après les inclusions (et en dehors de toute fonction)
- `#define NOM valeur`
 - `#define` est un nom réservé
 - `NOM` est en majuscule par convention

Exemple

```
#define PI 3.1415926  
#define MESSAGE_3176 "Bienvenue sur emacs"
```

- Intérêts
 - Lisibilité du code
 - Facilite la maintenance, la traduction, etc.
- Fonctionnement : le pré-compilateur remplace `NOM` par valeur

Expressions

- Formées d'opérandes et d'opérateurs (44 prédéfinis)
 - Opérandes : variables, constantes, appels de fonctions
 - Opérateurs : nous en utiliserons 22
 - Les opérateurs sont associatifs à gauche

Les opérateurs arithmétiques

- + addition
- - soustraction
- * multiplication
- / division (division entière avec des entiers)
- % modulo (reste de la division entière)
- ordre des priorités : *, /, % prioritaires à +, -

Les opérateurs relationnels

- == test d'égalité

Attention : ce n'est PAS =

= est l'opérateur d'affectation de base et est VRAI tout le temps !

- != test d'inégalité
- <, <= inférieur (ou égal)
- >, >= supérieur (ou égal)

Les opérateurs logiques

- && ET
- || OU
- ! NON
- Il n'y a pas de type booléen ! **MAIS**
0 (zéro) est considéré comme FAUX
'valeur non nulle' est considérée comme VRAI

Exemple

```
int maVariable;  
maVariable = ...;  
if (maVariable)           /* si variable != 0 */  
then ...
```

Les opérateurs d'affectation

- Forme générale : `[variable] <opérateur> [expression]`
- `[variable]` représente une variable existante
- L'évaluation de `[expression]` donne un résultat qui devient la nouvelle valeur de `[variable]`
- `<opérateur>`
 - = affectation simple
 - += affectation du résultat de `[variable] + [expression]`
idem avec -=, *=, %/, &=, |=

Exemple

```
int i;  
i=11*2;    /* i prend la valeur 22 */  
i*=4;      /* i prend la valeur de i*4, ie 22*4, ie 88 */
```


Les opérateurs pour incrémenter et décrémenter

- **Sur des variables entières**

- $[variable]++$ est équivalent à $[variable] += 1$
est équivalent à $[variable] = [variable] + 1$
- $[variable]--$ est équivalent à $[variable] -= 1$
est équivalent à $[variable] = [variable] - 1$
- Pré et post fixé :
si le ++ ou le -- est devant, alors l'opération est première

Exemple

```
int i,j,k;  
i=4;  
j= 10 + i++;    /* j vaut 14 et i vaut 5 */  
k= 10 + ++i;    /* k vaut 16 et i vaut 6 */
```

- Remarque : Peut être utilisé pour d'autres variables (pour programmeur confirmé ou le devenant)

Les instructions

Les instructions

- instruction vide : `;` ;
- instruction expression : `<expression>` ;
- instruction composée : un bloc = une suite d'instructions (vides, simples et/ou composées)
un bloc est délimité par `{` au début et `}` à la fin
- Remarque : l'indentation du code aide à la lecture

Exemple

```
{          /* début de bloc 1 */
int i;     /* instruction simple */
    {     /* début de bloc 2 */
        i=14; /* instruction simple */
    }     /* fin de bloc 2 */
;        /* instruction vide */
}        /* fin de bloc 1 */
```

Les instructions conditionnelles

Structure :

```
if (<expression>          /* evaluation Vrai (0) ou Faux (!=0) */
    <instruction>         /* exécution si Vrai (c'est le "then")*/
else
    <instruction>         /* exécution si Faux */
<instruction suivante> /* exécuté dans tous les cas */
```

La clause else <instruction> est optionnelle

Recommandations importantes

- bien présenter : indentation !
- travailler avec des blocs { }

Les instructions conditionnelles : Pourquoi indenter et utiliser { } ?

```
if(<expression>) <instruction> else <instruction>
```

<instruction> = instruction simple ou bloc

Exemple

```
if (C1 && C2 || C3)
<instruction simple 1>
<instruction simple 2>
else <instruction>
```

⇒ Provoque une erreur

Un autre exemple

```
if (C1 && C2 || C3)
<instruction>
else
<instruction simple 1>
<instruction simple 2>
```

⇒ ne provoque pas d'erreur mais peut en cacher une...

Les instructions conditionnelles : Emboîtements

Le else se rapporte au if le plus proche

Exemple

if (C1) if (C2) l1 else l2 se "lit" :

```
if (C1)
  if (C2)
    l1
  else
    l2
```

l2 est exécuté si C1 est vrai et C2 faux

Un autre exemple

On peut introduire des accolades, if (C1) {if (C2) l1} else l2 se "lit" :

```
if (C1){
  if (C2)
    l1
} else
  l2
```

l2 est exécuté si C1 est faux

Les instructions conditionnelles : Un exemple

Nombre de racines de $ax^2 + bx + c = 0$ et solutions

```
int main(int argc, char *argv[]){
    float a, b, c, delta, x1, x2;
    int nbRacines;
    a=atof(argv[1]); b=atof(argv[2]); /* Rq: atof(<chaine>):comme atoi() */
    c=atof(argv[3]);                /* mais pour les flottants      */
    delta = b*b-4*a*c;
    if (delta>0){
        nbRacines = 2;
        x1=(-b+sqrt(delta))/(2*a);
        x2=(-b-sqrt(delta))/(2*a);
        printf("Nombre de racines : %d\n solutions :
                %f et %f\n",nbRacines,x1,x2);
    }else
        if (delta==0){
            nbRacines = 1;
            x1 = x2 = -b/(2*a);
            printf("Nombre de racines : %d\n solution :
                    %f\n",nbRacines,x1);
        }
        else{
            nbRacines = 0;
            printf("Nombre de racines : %d\n",0);
        }
    exit(0);
}
```

Les instructions conditionnelles : Une autre écriture

Forme condensée du "if" :

`(condition) ? <expression1> : <expression2> ;`

est équivalent à

```
if (condition)
    <expression1>
else
    <expression2>
```

Exemples

- Stocker dans c le minimum entre deux nombres a et b :

`(a<b) ? c=a : c=b ;`

- Ajouter 's' en cas de pluriel :

```
int x;
x=...
printf("J'ai trouvé %d élément%c\n",x,(x>1) ? 's' : '');
```


Les instructions répétitives

Deux instructions répétitives “équivalente” : while et for

```
while(<expression>)  
    <instruction>
```

Quatre parties

- 1 conditions préparatoires
- 2 bloc répété
- 3 changement d' "état"
- 4 test (d'entrée/de reprise/de sortie)

```
int i=1, n=100;  
while(i<=n){  
    if n%i == 0  
        printf("%d",i);  
    i++;  
}
```

Les instructions répétitives

Deux instructions répétitives “équivalente” : while et for

```
for(<expression1>; <expression2> ; <expression3>)  
    <instruction>
```

Quatre parties

- ❶ conditions préparatoires
- ❷ bloc répété
- ❸ changement d'“état”
- ❹ test (d'entrée/de reprise/de sortie)

```
int i, n=100;  
for(i=1;i<=n;i++){  
    if n%i == 0  
        printf("%d",i);  
}
```

Les instructions répétitives

Il existe une autre instructions répétitives exécutée au moins une fois !

do

 <instruction>

while (condition)

Exemple

```
char rep;
do{                /* ici un traitement au moins */
    printf("On continue? O/N\n");
}
while( (rep=getchar()) != 'N') ;
```

Remarque : `getchar()` permet de lire un caractère au clavier

- Un programme : suite d'instructions
- Fichier source → compilation → fichier objet → programme exécutable
- Des fonctions : `main()`, `atoi()`, `printf()`, `laVotre()`
- Réservation d'une zone mémoire avec une étiquette
- Typages des variables (taille zone, mémoire, contrôles de l'usage)
- Instructions et bloc d'instructions `{ }`
- Opérateurs : relationnels, logiques, affectations, in/dé-crémentations
- Formes algorithmiques
 - Conditionnel : `si <condition> alors ... sinon... (if)`
 - Répétitifs :
 - tant que `<condition>` faire ... (`while`, `for`)
 - Faire ... jusqu'à (`do ... while`)

Les fonctions

Structure d'une fonction

`<type> <nom> <liste de paramètres>
<bloc de définition>`

- `<type>` : type de la valeur retournée par la fonction (int par défaut)
- `<nom>` : nom (explicite) de la fonction en commençant par une minuscule
la fonction <nom> est du type <type retourné>
- `<liste de paramètres>` :
 - entre (et)
 - pour chaque paramètre : `<type> <nomDeVariable>`
 - les paramètres sont séparés par une virgule ,

- `<bloc de définition>` : liste d'instructions entre { et }

si le type de la fonction **n'est pas** void, au moins une instruction :
`return <valeur ou variable>;`

ATTENTION `return` stoppe l'exécution de la fonction et renvoie la valeur indiquée (Rq : `return` ; ne renvoie rien (void))

Exemple :

```
float discriminant(float a, float b, float c){  
    return b*b-4*a*c;  
}
```

Déclaration et définition d'une fonction

- La déclaration = prototype, signature
 - **sans** le bloc de définition, **avec** un point virgule ;
 - Après la déclaration : possibilité de l'utiliser
= l'appeler, invoquer son nom dans le code

`<type> <nom> <liste de paramètres> ;`

- La définition : **avec** le bloc de définition
`<type> <nom> <liste de paramètres>`
`<bloc de définition>`

L'appel de fonction : `variable = nomFonction(param1, param2, etc);`

- `variable` : facultatif (le retour n'est pas utilisé ou fonction de type `void`)
- `variable` est du type de la fonction `nomFonction`
- il y a autant de paramètres effectifs que de paramètres formels avec respect des types et de l'ordre

Rq : le programme appelant la fonction est la fonction qui contient l'appel

Exemple

```
float discriminant (float a, float b, float c); /* déclaration */
float discriminant (float a, float b, float c){ /* définition */
    return b*b - 4*a*c;
}

void MaF( ){
    float result, v=1 , fxp = 12; /* déclarations variables */
    result = discriminant (14, v , fxp); /* appel de fonction */
} /* puis affectation */
```


Remarque importante sur la déclaration

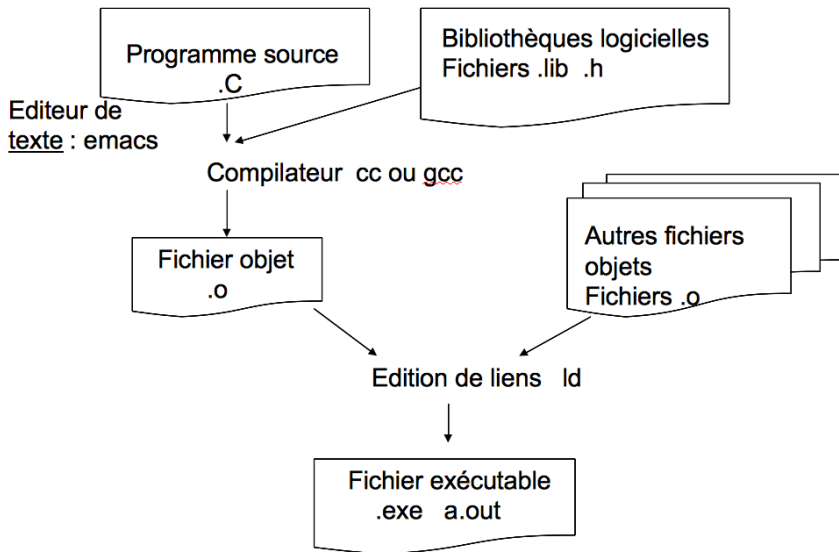
- La déclaration n'est pas obligatoire

MAIS parfois utile et parfois nécessaire

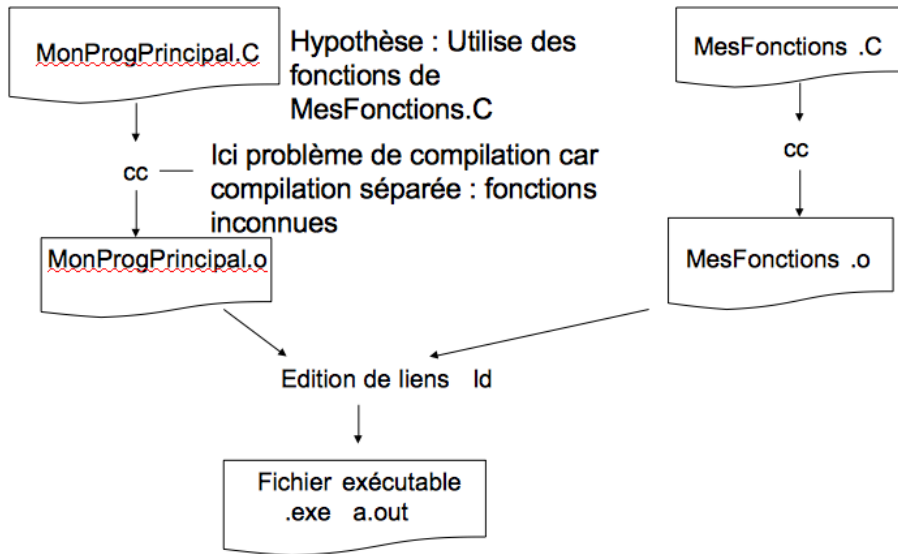
- **Utile** : le programme appelant (par exemple le main) est placé avant l'écriture de la fonction (question de composition du fichier .c)
- **Nécessaire** : le programme appelant utilise une fonction d'un autre module (autre .c, .o)

La compilation (qui est faite module par module) oblige à connaître la déclaration de la fonction (type retourné, types paramètres)

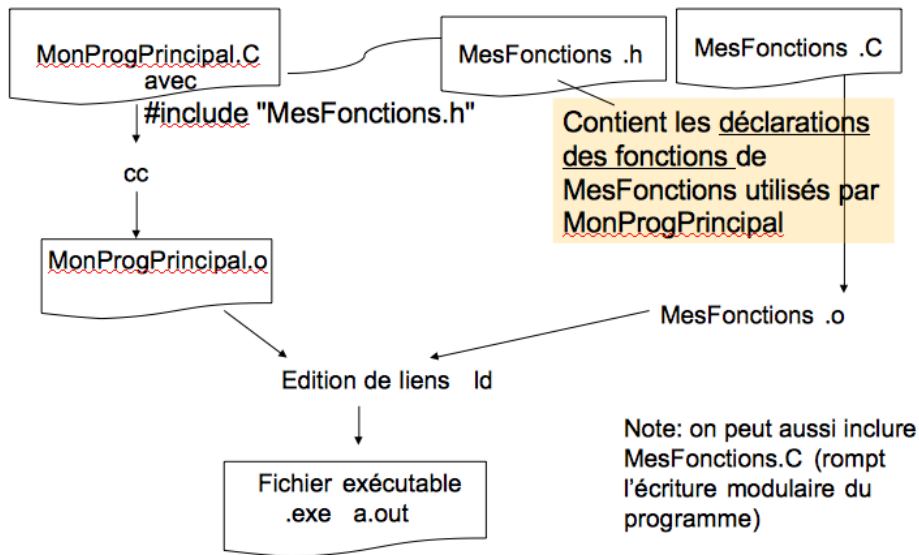
Schéma de production de logiciel C



Programme principal et module(s)



Programme principal et module(s)



Le passage de paramètres entre un programme appelant et une fonction ⇒ Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

Avant l'appel de maFonction ()

```
int maFonction(char param1,int param2)
{
    int i=1;
    printf("%c", param1);
    param2= 144;
    return i;
}

main ( ){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);
}
```

Zone mémoire

z
y
x

12
a
0

Type, taille en octets, adresse mémoire

int, 4, @3
char, 1, @2
int, 4, @1

Contexte
mémoire
d'exécution du
programme
appelant.

Le passage de paramètres entre un programme appelant et une fonction \Rightarrow Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

Exécution de `x = maFonction (y, z);`

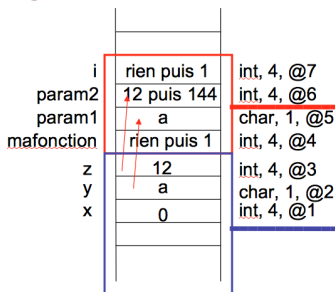
L'appel de la fonction \Rightarrow changement de contexte d'exécution

```
int maFonction(char param1,int param2)
```

```
{
    int i=1;
    printf("%c", param1);
    param2= 144;
    return i;
}
```

```
main ( ){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);
}
```

- ❶ Copie des valeurs données aux paramètres
 $\text{param1} \leftarrow y$ et $\text{param1} \leftarrow z$
- ❷ Exécution des instructions



Contexte mémoire d'exécution de la fonction.

Contexte mémoire d'exécution du programme appelant.

Le passage de paramètres entre un programme appelant et une fonction ⇒ Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

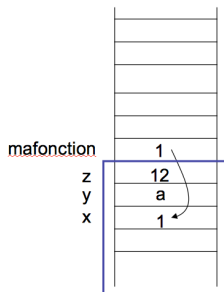
après l'exécution de `x = maFonction (y, z);`

Retour au programme appelant

⇒ retour au précédent contexte d'exécution
avec copie du résultat de `x=maFonction(y,z)`

```
int maFonction(char param1,int param2)
{
    int i=1;
    printf("%c", param1);
    param2= 144;
    return i;
}

main ( ){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);
}
```



Note : `y` et `z` n'ont pas changé de valeur

`int, 4, @3`
`char, 1, @2`
`int, 4, @1`

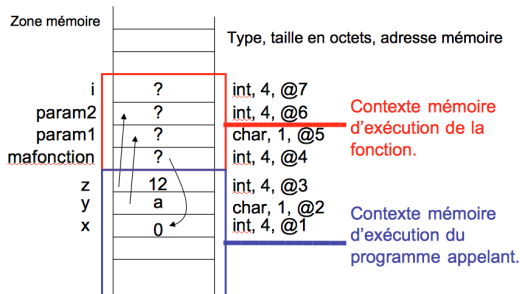
Contexte mémoire
d'exécution du
programme appelant.

Le passage de paramètres entre un programme appelant et une fonction ⇒ Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

```
int maFonction(char param1,int param2)
{
    int i=1;
    printf("%c", param1);
    param2= 144;
    return i;
}

main ( ){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);
}
```

Synthèse



Retour sur la fonction `atoi()` (et `atof()`)

Rappel

```
int atoi(char* param);
```

fonction prédéfinie dans `<stdio.h>`

Conversion de la chaîne de caractères
`param` en **entier**

Remarque :

```
double atof(char* param);
```

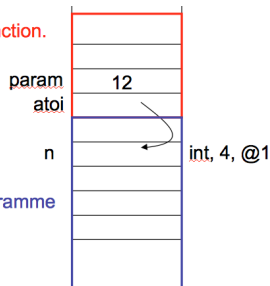
Fonction prédéfinie dans `<stdio.h>`

Conversion de la chaîne de caractères
`param` en **réel**

Exemple :

```
int n; n=atoi("12");
```

Contexte fonction.



Contexte programme
appelant

Retour sur la fonction printf()

Rappel

```
int printf(char* param1{,liste});
```

fonction prédéfinie dans <stdio.h>

Affiche la chaîne de caractères param1 en la formatant selon le(s) format(s) spécifié et les expressions calculées

Remarque :

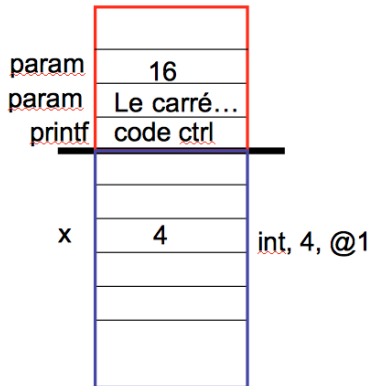
printf() retourne un entier

il correspond au nombre d'octets écrits

ou à la constante EOF (-1) en cas d'erreur

Exemple :

```
int x=4;  
printf("Le carré de  
x est % d \n",x*x);
```



Découverte de la fonction getchar()

```
char getchar();
```

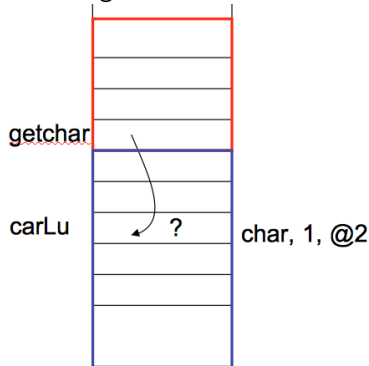
fonction prédéfinie dans <stdio.h>

Lit **un** seul caractère entré au clavier

C'est une fonction bloquante

Exemple :

```
char carLu;  
carLu = getchar();
```



Retour sur la fonction `scanf()`

Rappel

```
int scanf(char* format, att1, att2, att3,...);
```

fonction prédéfinie dans `<stdio.h>`

Lit des données rentrées par l'utilisateur

Remarque : `scanf()` retourne un entier

il correspond au nombre d'octets de variables lues ou à la cste EOF en cas d'erreur

Exemple d'utilisation :

```
int a;
```

```
scanf ("%d", a ); FAUX !!
```

```
scanf ("%d", &a );
```

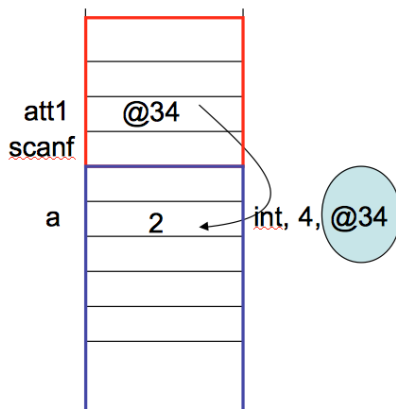
Correct ! mais pourquoi ?

&a correspond à l'adresse de la variable a
En fait, on veut que la valeur de a change !!

Retour sur la fonction `scanf()`

Exemple d'utilisation :

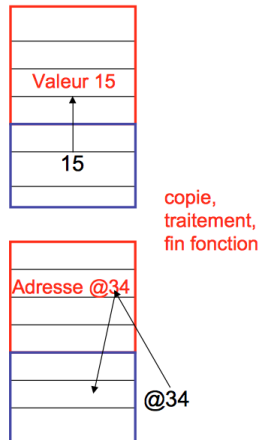
```
int a; scanf ("%d", &a );
```



Un petit bilan

Passage de paramètres

- soit par valeur
 - Copie de la valeur
 - Pas d'accès à la variable initiale copiée
 - Modification possible de la valeur locale
- soit par adresse (avec le signe **&**)
 - Copie l'adresse
 - Accès (modification possible) à la variable initiale via l'adresse



Il est tout à fait possible d'appeler depuis une fonction cette même fonction. On appelle cela un appel récursif.

Exemple

```
int fact(int n)
{
    if (n==0)
        return 1;
    else
        return n*fact(n-1);
}
```

- **Définir la condition d'arrêt**
- Lorsque la récursivité ne peut pas être faite directement, on utilise une fonction auxiliaire qui se charge de la récursivité (cf TD).

Les tableaux

Notions de bases

Définition

Représentation tabulaire des données de **même type** (liste, matrice)

Indicé par des entiers de 0 à `nbElements-1`

Déclaration

`<type> <nom>[<nbElements>];`

Exemple

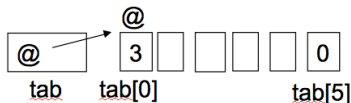
```
int tab[6];
```

Déclare la variable `tab` comme un tableau de 6 entiers

`tab` adresse de début du tableau

`tab[0]` valeur du premier entier `tab[0]=3;`

`tab[5]` valeur du dernier entier `tab[5]=0;`



Exemples basiques d'utilisations des tableaux

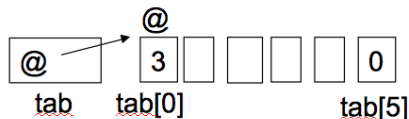
```
int tab[6];      /* declaration d'un tableau  
                  d'entier de taille 6   */
```

```
int i = 6, j ;
```

```
tab[0] = 3;
```

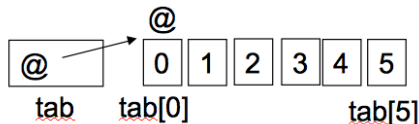
```
tab[5] = 0;
```

```
tab[i] = 14;    /* instruction possible mais a de grandes chances  
                  de provoquer une erreur à l'exécution */
```



```
for (j=0 ; j < i ; j++)
```

```
    tab[j] = j ;
```



Quelques propriétés importantes

- La taille du tableau DOIT être connue \Rightarrow prévoir la taille maximale utile !
On peut utiliser `#define` `#define TAILLE_MAX 300`
- **ATTENTION :** **pas** d'affectation entre tableaux ! (pas de ~~`tableau1 = tableau2`~~)
 \Rightarrow Il faut prévoir des boucles de recopie
- `double tab[TAILLE_MAX]; /* declare un tableau de TAILLE_MAX double */`

tab est l'adresse du début du tableau

tab[i] désigne l'élément (double) de rang i

tab[i] est défini pour $0 \leq i < \text{TAILLE_MAX}$

i indique le décalage par rapport au début du tableau
 \Leftrightarrow calculer un décalage de i double "au delà" de l'adresse de tab

Remplissage d'un tableau

Lorsque l'on déclare un tableau, il est “vide” (aucune valeur n'a été affectée aux cases)

⇒ Il faut donc le remplir

Exemple

- un entier n , un tableau `tab` de n “cases” (vides)
- Traitement : affecter aux n cases de `tab` les sommes partielles (0, 1, 3, 6, 10, 15, 21, *etc.*)

```
#define N 144;
int t[N];
int i;
t[0] = 0;
for (i=1 ; i < N ; i++)
    t[i] = t[i-1] + i ;
```

Exercice : ré-écrire la bout de code précédent à l'aide d'une boucle `while`

Exemples de déclarations (1/2)

- `int vecteur[100];` tableau à 1 dimension, taille 100
- `float matrice[10][10];` tableau à 2 dimensions
- `#define MAX 100`
`char prenom1[MAX];` chaîne (tableau) de caractères, taille MAX
- `char prenom2[] = "Jean";` chaîne de caractères initialisée à

'J'	'e'	'a'	'n'	'\0'
-----	-----	-----	-----	------
- `char prenom3[];` chaîne de caractères vide

ATTENTION : ici ni taille, ni valeur

⇒ une réservation spécifique de mémoire sera nécessaire pour entrer des valeurs

Exemples de déclarations (2/2)

- On peut aussi définir un nouveau type tab ou mat

```
#define MAX 100  
typedef int tab[MAX];  
typedef int mat[MAX][MAX];
```

Puis déclarer une variable d'un de ces types :

```
tab vecteur;  
mat matrice;
```

Une aparté sur la définition de types (typedef)

```
typedef <déclaration>;
```

↔ Permet de définir un nouveau type

Exemple :

```
typedef int kilometre; /* définit le type kilometre */  
    kilometre distance = 4;
```

Exemples d'algorithmes simples : La fonction afficher

Entrée : un entier n , un tableau d'entiers t

Sortie : affichage des n premiers éléments de t

ATTENTION : on suppose que t contient au moins n éléments

```
void afficher (int n, tab t){  
    int i;  
    for(i=0 ; i < n ; i++)  
        printf("%d ",t[i]);  
    printf("\n");  
}
```


Exemples d'algorithmes simples : La fonction `afficher_inverse`

Entrée : un entier `n`, un tableau d'entiers `t`

Sortie : affichage *inversé* des `n` premiers éléments de `t`

ATTENTION : on suppose que `t` contient au moins `n` éléments

```
void afficher_inverse (int n, tab t){
    int i;
    for(i=0 ; i < n ; i++)
        printf("%d ",t[(n-1)-i]);
    printf("\n");
}
```

OU

```
void afficher_inverse (int n, tab t){
    int i;
    for(i=n-1 ; i >=0 ; i--)
        printf("%d ",t[i]);
    printf("\n");
}
```

Les tableaux

Chaînes de caractères

Chaînes de caractères = tableau de caractères

- se termine par le caractère nul `'\0'`
- ⇔ le premier caractère du code ASCII (dont la valeur est 0)
c'est un caractère de contrôle (non affichable) qui indique la fin d'une chaîne de caractère
- ⇒ Une chaîne composée de n éléments est en fait un tableau de $n + 1$ éléments de type `char`

Rq : La chaîne débute à une adresse

'J'	'e'	'a'	'n'	'\0'
-----	-----	-----	-----	------

Rq : `char *argv[]` est un tableau de chaîne de caractères

Déclaration

- `char prenom[10];`

⇒ réserve 10 caractères : 9 explicitement utilisés + 1 pour le caractère de fin

prenom @ →

--	--	--	--	--	--	--	--	--	--

Exemples d'initialisation de la chaîne

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

```
void main(){
```

```
    char chaine[10];
```

```
    chaine[0]= 'J';
```

```
    chaine[1]= 'e';
```

```
    chaine[2]= 'a';
```

```
    chaine[3]= 'n';
```

```
    chaine[4]= '\0';
```

```
}
```

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

```
void main(){
```

```
    char chaine[10]={ 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };
```

```
}
```

- `char nom[] = "ZOLA";`

⇒ Taille indiquée par le nombre de caractères de la chaîne (+1)

Place automatiquement le `'\0'`

nom @ →

'Z'	'O'	'L'	'A'	'\0'
-----	-----	-----	-----	------

Attention à la gestion des chaînes

- Bien gérer les indices :
début à 0 et fin à taille-1, incrémenter pour parcourir le tableau, etc.
- Travailler avec une zone mémoire réservée
i.e., un nombre suffisant d'éléments

Rq : `char maChaine[]` ; ou `char * maChaine` ;

ne réserve que le “pointeur vers”, MAIS pas la zone pour les caractères

⇒ Le nombre d'éléments est ici inconnu !

- Insérer soi-même `'\0'` en fin de chaîne (si ce n'est pas fait automatiquement)
- **Pas d'affectation “globale” du type ~~`maChaine1 = maChaine2`~~ ;**

Exemple d'un algorithme simple

Recopie de la chaîne source vers la chaîne cible

```
#include<stdlib.h>
int main(){
    char source[]="ZOLA";
    char dest[10];                /* on travaille avec une zone mémoire réservée */
    int i=0;                      /* on gère les indices !! */
    while (source[i] != '\0'){
        dest[i]=source[i];
        i++;                      /* on incrémente pour parcourir le tableau */
    }
    dest[i]='\0';                 /* on insère le caractère de fin de chaine */
    exit(0);
}
```

Attention au piège !

Il faut recopier caractère par caractère ! (ou une fonction spéciale)
Sinon, c'est l'adresse que l'on recopie !

```
char rire[]="ha ha";
char *rireBis;                  /* equivalent à char rireBis[] */
rireBis = rire;                 /* copie de l'adresse de rire */
```

Fonctions prédéfinies

string.h contient des fonctions dédiées à la manipulation des chaînes
`#include <string.h>` (Rappel : chemin par défaut /usr/include)

strcpy	stricmp	strpbrk	strcat	strlen	strrchr	strchr
strlwr	strrev	strcmp	strncat	isxdigit	strcmpi	strncmp
strset	strcpy	strncmpi	strstr	strcspn	strncpy	strtok
strdup	strnicmp	strupr	strerror	strnset		


str : string	cmp : compare	n : n premiers caractères	len : length
i : ignore la casse	cpy : copie	cat : concaténation	chr : char r : reverse

IMPORTANT

Ne pas apprendre la liste des fonctions et paramètres par cœur
MAIS savoir les chercher et les utiliser correctement

Recopie de chaîne

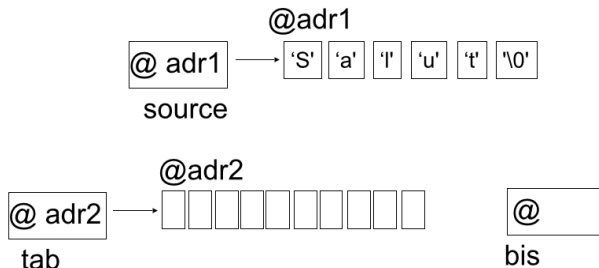
```
char *strcpy(char *cible, char *source);
```



- Recopie de la chaîne de caractère qui commence à l'adresse de **source** dans la chaîne qui commence à l'adresse de **cible**
- **ATTENTION** : il faut avoir réservé la place mémoire (*i.e.*, il faut correctement déclarer les chaînes)
- Le retour de la fonction “pointe” (est l'adresse) de la chaîne **cible**
- Le `'\0'` est placé en fin de chaîne

Recopie de chaîne - Un exemple (1/3)

```
char *source = "Salut" ;  
/* déclaration et initialisation d'une chaîne de taille 6 */  
char tab[10] ;  
/* déclaration d'une chaîne de taille 10 */  
char bis[] ;  
/* déclaration d'une chaîne sans réservation de mémoire */
```



Recopie de chaîne - Un exemple (2/3)

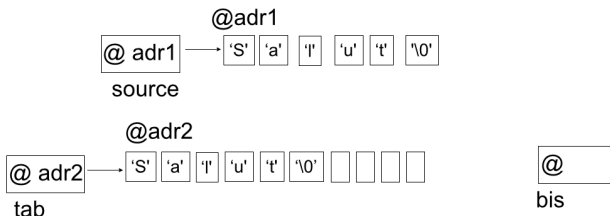
```
char *source = "Salut", tab[10], bis[] ;  
strcpy(tab,source);
```

Remarques

Ici, on veut copier la chaîne qu'il y a dans source dans la chaîne tab

On peut le faire puisque la taille de la chaîne est bien < 10

On n'utilise pas le retour de `strcpy()`



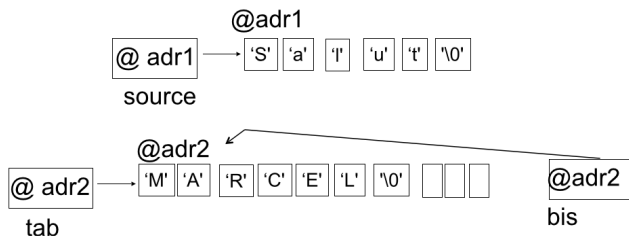
Recopie de chaîne - Un exemple (3/3)

```
char *source = "Salut", tab[10], bis[] ;  
bis = strcpy(tab, "MARCEL");
```

Remarques

Ici, on veut copier la chaîne 'MARCEL' dans la chaîne tab
ET récupérer l'adresse de tab dans bis

On peut le faire puisque la taille de la chaîne est bien < 10



Longueur de chaîne

```
int strlen(char *chaine);
```

↪ Retourne la longueur de la chaîne pointée par chaine

ATTENTION

strlen() ne compte pas le caractère de fin de chaîne '\0' !!

Exemple

```
char *chaine = "Salut";  
int taille;  
taille = strlen(chaine);  
printf("La taille de la chaîne \"%s\" est : %d\n",chaine,taille);
```

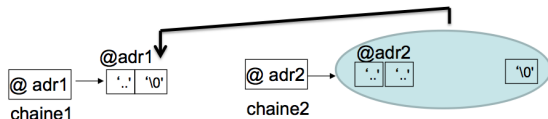
Affiche :

La taille de la chaîne "Salut" est : 5

Concatenation de chaînes

```
char *strcat(char *chaine1, char *chaine2);
```

- Recopie la chaîne pointée par chaine2 **à la fin** de la chaîne pointée par chaine1



- Le résultat est pointé par chaine1 et est retourné (pointeur sur la chaîne)

Remarques

- La place réservée pointée par chaine1 doit être suffisante !
`strlen(chaine1) + strlen(chaine2) < zone mémoire réservée pour chaine1`
- Le `\0` est (dé)placé en fin de chaîne

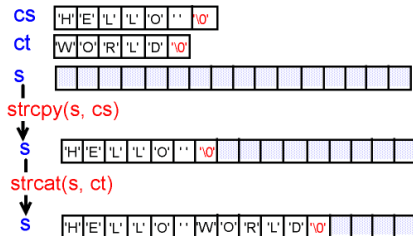
Concatenation de chaînes - Deux exemples

Exemple 1

```
char r[20], s1[20], s2[] = "nation";
strcpy(s1, "concaté");           /* copie de la chaîne concaté
                                   dans S1 de taille > à 7 ! */
R=strcat(s1, s2);                 /* concaténation de S1 et S2 */
printf("%s", r);                  /* Affiche "concaténation" */
printf("%s", s1);                 /* Affiche "concaténation" */
```

Exemple 2

```
char cs[] = "HELLO ";
char ct[] = "WORLD";
char s[16];
strcpy(s, cs);
strcat(s, ct);
```



Comparaison de chaînes

```
int strcmp(char *chaine1, char *chaine2);
```

- Comparaison des chaînes pointées caractère par caractère
- Retourne un entier négatif, nul ou positif, selon le classement alphabétique des 2 chaînes pointées

Exemple

```
char *s1 = "abcd", *s2 = "abz";  
if ( strcmp (S1, S2) == 0 ){  
    printf("Les deux chaines sont identiques\n");  
else if ( strcmp (S1, S2) < 0 ){  
    printf("%s est inférieure à %s\n", s1, s2);  
}
```

Des fonctions utiles sur les chaînes (dans `stdio.h`)

On a déjà vu

- `int printf(char* param1{,liste});`

↪ pour afficher une chaîne

- `int scanf(char* format, att1, att2, att3,...);`

↪ pour lire une chaîne(s) formatée(s) (**tout "espace" est délimiteur**)

Une nouvelle fonction : `char *gets(char *s);`

- Lit une chaîne **terminée par** `'\n'` sur l'entrée standart
- Le résultat est pointé par `s` et est retourné (renvoie `NULL` en cas d'erreur)
- `'\n'` n'est pas recopié dans la chaîne et un `'\0'` est placé à la fin de la chaîne

Rq : `scanf("%s",s);` ne permet pas de lire des chaînes contenant des espaces (l'espace est pour `scanf` un séparateur), tandis qu'avec `gets`, seul le caractère `'\n'` sert de délimiteur

Exemple d'utilisation de `char *gets(char *s);`

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char ligne[10];
    printf("Votre texte 1 ? ");
    gets(ligne);
    printf("Vous avez rentré le texte : %s\n",ligne);
    exit(0);
}
```

Résultat de l'exécution

```
> ./monprog
Votre texte 1 ? Salut
Vous avez rentré le texte : Salut
>
```

ATTENTION à l'espace mémoire réservé pour la chaîne... Par exemple :

[illegible]

- Tableau : regroupement de données de **même nature**

- `<type> <nom>[<nbElem>];`

↪ Déclaration d'un tableau de taille `<nbElem>`

- `<type> <nom>[<nbElem1>][<nbElem2>];`

↪ Déclaration un tableau à 2 entrées *ie* une matrice

- `<type> <nom>[];` ou `<type> *<nom>;`

↪ Réserve uniquement le pointeur vers le tableau (pas de réservation de zone mémoire)

- Les indices d'un tableau varient de 0 à `<nbElem>-1`

`<nom>[0]` est la première case du tableau — `<nom>[<nbElem>-1]` est la dernière

- Les chaînes de caractères sont des tableaux de caractères

`char chaine[<taille>;` ou `char *chaine;` ou `char chaine[];`

- Les chaînes se terminent par le caractère `'\0'`

- de nombreuses fonctions pour manipuler les chaînes sont pré-définies dans la bibliothèque `string.h`

Manipulation de fichiers

Lecture/Ecriture

Ce que l'on va voir

- Le type “fichier”
- Ouverture et fermeture de fichier
- Différentes méthodes de lecture/écriture

Le type “fichier” : FILE *

- **Déclaration** : `FILE * file;`

↪ réserve le pointeur vers une variable de type FILE

⇒ Il va falloir réserver la zone mémoire en “ouvrant” le fichier

- Cette structure se trouve dans la bibliothèque `stdio.h`

Rq : `EOF` est le marqueur de fin de fichier (*End Of File*)

Ouverture/Fermeture d'un fichier

Procédure à suivre

- **Ouverture de fichier** avec `fopen()` ; (renvoie un pointeur sur le fichier)
- **Vérification** de l'ouverture (est-ce que fichier existe ?)
 - ⇒ Si le pointeur vaut `NULL`, l'ouverture a échoué (afficher un message d'erreur)
 - ⇒ Si le pointeur n'est pas `NULL`, on peut lire et/ou écrire dans le fichier
- Quand on a finit, on **ferme** le fichier avec `fclose()` ;

Ouverture/fermeture d'un fichier

```
FILE* fopen(char* nom_fich, char * mode);
```

↪ Renvoie un pointeur vers le fichier en cas de succès, NULL sinon

- `nom_fich` est le chemin vers le fichier
- où `mode` est une chaîne de caractère :
 - **"r" : lecture seule.** Le fichier doit exister
 - **"w" : écriture seule.** Si le fichier n'existe pas, il est créé
 - **"a" : mode d'ajout à la fin.** Si le fichier n'existe pas, il est créé
 - **"r+" : lecture et écriture.** Le fichier doit exister
 - **"w+" : lecture et écriture, avec suppression du contenu** au préalable.

Si le fichier n'existe pas, il est créé

- **"a+" : ajout en lecture/écriture à la fin.** Si le fichier n'existe pas, il est créé

```
int fclose(FILE* fichier);
```

↪ renvoie un entier

- 0 : si la fermeture a marché
- EOF : si la fermeture a échoué

Ouverture/Fermeture d'un fichier — Un exemple

```
int main(){
    FILE* fichier = NULL;
    fichier = fopen("test.txt", "r+");
    if (fichier == NULL){ /* si le fichier n'existe pas */
        printf("Vous tentez d'accéder à un fichier inexistant\n");
        exit(-1);
    }
    /* sinon on peut lire et écrire dans le fichier */
    ..
    ..

    fclose(fichier);

    exit(0);
}
```

Rq : Si test.txt existe, le pointeur fichier devient un pointeur sur test.txt

Écriture dans un fichier

- d'un seul caractère `int fputc(int caractere, FILE* fichier);`

↪ renvoie le caractère écrit si succès, EOF sinon

Rq : caractere est un entier, mais revient à utiliser char (vous pouvez écrire 'A')

- d'une chaîne : `fputs(char* chaine, FILE* fichier);`

↪ renvoie un entier négatif si succès, EOF sinon

- chaine est la chaîne à écrire

- d'une chaîne formatée :

`int fprintf(FILE * fichier, char* chaine_formatée{,liste});`

↪ retourne le nombre de caractère écrits

Rq : s'utilise comme printf, excepté le 1^{er} paramètre qui est un pointeur de FILE

Lecture dans un fichier

- d'un seul caractère `int fgetc(FILE* fichier);`

↪ renvoie le caractère lu si succès, EOF sinon

- d'une chaîne : `char * fgets(char* chaine, int nb_car, FILE* fichier);`

↪ renvoie le pointeur vers la chaîne lue, EOF sinon

- `nb_car` est le nombre de caractères à lire

- d'une chaîne formatée :

`int fscanf(FILE * fichier, char* format{,liste});`

↪ retourne le nombre d'affectation(s) effectuée(s), EOF en cas d'erreur

Rq : s'utilise comme `scanf`, excepté le 1^{er} paramètre qui est un pointeur de FILE

Lecture dans un fichier — Un exemple

```
#include<stdio.h>
#include<stdlib.h>
#define TAILLE_MAX 1000

int main(){
    FILE* fichier = NULL;
    char chaine[TAILLE_MAX];
    fichier = fopen("test.txt", "r");
    if(fichier==NULL){
        printf("Vous tentez d'ouvrir un fichier inexistant\n");
        exit(-1);
    }
    /* On lit le fichier tant qu'on ne reçoit pas d'erreur (NULL) */
    while (fgets(chaine, TAILLE_MAX, fichier) != NULL){
        printf("%s",chaine); /* On affiche la chaîne qu'on vient de lire */
    }

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

Lecture dans un fichier — Un autre exemple

On suppose que le fichier `test.txt` contient 3 nombres séparés par un espace (ex : 15 20 30)

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

int main(){
    FILE* fichier = NULL;
    int score[3];
    fichier = fopen("test.txt", "r");
    if(fichier==NULL){
        printf("Vous tentez d'ouvrir un fichier inexistant\n");
        exit(-1);
    }

    fscanf(fichier, "%d %d %d", &score[0], &score[1], &score[2]);
    printf("Les meilleurs scores sont : %d, %d et %d",
           score[0], score[1], score[2]);

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

Écriture dans un fichier — Un exemple

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

int main(){
    FILE* fichier = NULL;
    int age ;
    fichier = fopen("test.txt", "w");
    if(fichier==NULL){
        printf("Impossible d'ouvrir le fichier\n");
        exit(-1);
    }

    fputs("Salut !\nComment allez-vous ?", fichier);
    fclose(fichier);
    exit(0);
}
```

Écriture dans un fichier — Un autre exemple

```
#include<stdio.h>
#include<stdlib.h>
#define TAILLE_MAX 1000

int main(){
    FILE* fichier = NULL;
    int age ;
    fichier = fopen("test.txt", "w");
    if(fichier==NULL){
        printf("Impossible d'ouvrir le fichier\n");
        exit(-1);
    }
    /* on demande votre age */
    printf("Quel age avez-vous ?\n");
    scanf("%d", &age);

    /* On l'écrit dans le fichier */
    fprintf(fichier, "Le Monsieur qui utilise le programme, il a %d ans", age);

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

Quelques mots sur printf/fprintf et scanf/fscanf

- `scanf` et `printf` utilisent le flot standard (`stdin`, `stdout`)
- `fscanf` et `fprintf` permettent préciser le flot (c'est pourquoi on peut les utiliser sur un `FILE *`)

⇒ `scanf("%d",&x);` ⇔ `fscanf(stdin,"%d",&x);`

⇒ `printf("salut !");` ⇔ `fprintf(stdout,"salut!",&x);`

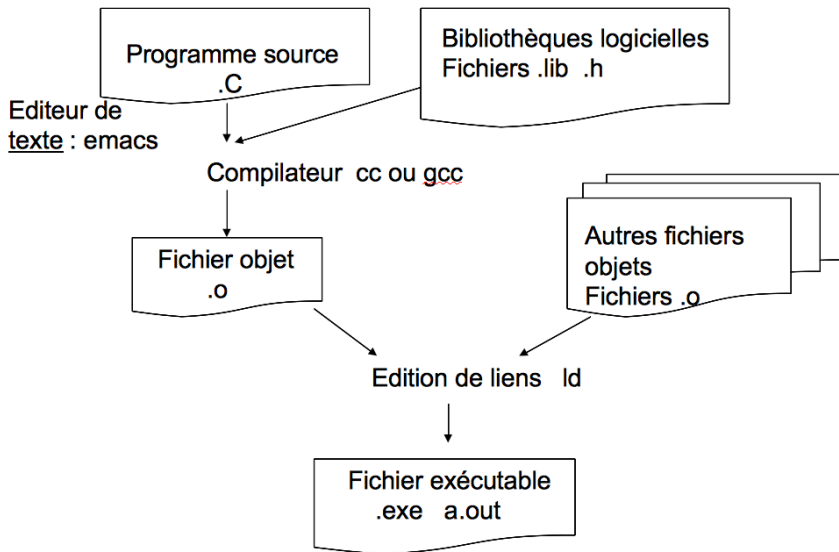
- Utile pour différencier les affichages d'erreurs des affichages "normaux"

⇒ On peut afficher nos messages d'erreurs sur `stderr` :

`fprintf(stderr,"ceci est un message d'erreur\n");`

Programmation modulaire

Rappel : Schéma de production de logiciel en C



Retour sur la production de programmes

1. Pré-processeur : $.c \rightarrow$ un $.c$ pour chaque fichier C
2. Compilateur : $.c \rightarrow$ un $.o$ (langage machine) pour chaque fichier $.c$
3. Edition de liens : $\sum .o \rightarrow$ executable (langage machine)

Abstraction de constantes littérales

Les constantes doivent être définies par des constantes à l'aide de la directive `#define`, les utilisations de ce type sont à éviter :

```
perimetre = 2 * 3.14 * rayon;
```

Factorisation du code

- Eviter la duplication du code
- Ne pas hésiter à définir une multitude de de fonctions de petite taille

Fragmentation du code

- Découper le programme en plusieurs fichiers
- Réutilisation facile d'une partie du code pour d'autres applications

Pré-processeur

- `#include` : inclusion textuelle de fichiers
 - Bibliothèque C (ex : `stdio.h`, `math.h`,...)
 - Fichier perso de déclarations de fonctions
- `#define` : remplacement textuel
 - ex : `#define PI 3.1415926`
- Option de gcc : `-E` pour visualiser le texte C modifié après passage du pré-processeur

Compilateur

- Traduction en langage machine
- Contrôle des arguments et des types des valeurs de retour pour toutes les fonctions
- Il ne connaît pas le code des fonctions seulement déclarées (option `-c`)

Édition des liens

- Crée l'exécutable à partir des fichiers objets
- "Résolution" des noms de fonctions et de variables indéfinis
- Arrêt dès qu'il y a un défaut

Compilation des modules

- Compilation séparée de chaque module indépendamment
- Permet
 - lisibilité
 - partage du travail
 - maintenance
 - réutilisabilité

Compilation des modules — Un exemple

1. Fichier `pluriel.c`

```
char pluriel_simple (int n) {  
    if (n ==1)  
        return ' ';  
    else  
        return 's';  
}
```

2. Compilation : `gcc -c pluriel.c` → `pluriel.o`

3. Utilisation dans un programme `prog.c`

- Déclarer la fonction : `char pluriel_simple(int n);`
- Lier sa définition : `gcc prog.c pluriel.o -o prog`

Découpage d'un programme en modules

Un module

- Propose un service via une interface de programmation
- Masque l'implantation (la réalisation)
- Un module = 3 fichiers : .h, .c, .o

- Le fichier **interface de programmation** : .h

↪ Déclarations des fonctions du module

- Le fichier **réalisation** : .c

↪ Définitions des fonctions et fonctions cachées (non présentes dans le .h)

- Le fichier **objet** : .o

↪ Code machine des fonctions du module

Exemple : main.c

```

/*****
/****  fichier: main.c                                     ****/
/****  saisit 2 entiers et affiche leur produit          ****/
/*****/
#include <stdlib.h>
#include <stdio.h>
#include "produit.h"
int main(void)
{
    int a, b, c;
    scanf("%d",&a);
    scanf("%d",&b);
    c = produit(a,b);
    printf("\nle produit vaut %d\n",c);
    return EXIT_SUCCESS;
}
```

Exemple : produit.h

```

/*****
/**  fichier: produit.h                                     ***/
/**  produit de 2 entiers                                   ***/
*****/
int produit(int a, int b);

```

Exemple : produit.c

```

/*****
/**  fichier: produit.c
/**  produit de 2 entiers
*****/
#include <stdio.h>
#include "produit.h"

int produit(int a, int b)
{
    return(a * b);
}
```

Compilation

```
>gcc -Wall -c produit.c  
>gcc -Wall -c main.c  
>gcc -Wall main.o produit.o -o mon_prog
```

Execution

```
>./mon_prog
```

Eviter les double inclusions

```
/******  
/**  fichier: produit.h                                **/  
/**  en-tete de produit.c                            **/  
/******  
#ifndef PRODUIT_H_  
#define PRODUIT_H_  
int produit(int a, int b);  
#endif /* PRODUIT_H_ */
```

Note : parfois on trouve le mot clé `extern` indiquant que la fonction/variable est définie dans un autre fichier

Ex : `extern int x;`

variable globale partagée déclarée dans un autre fichier - **Mais les variables globales sont à éviter !**

A tout fichier `.c` on associe un fichier `.h` qui définit son interface

Fichier `.h`

- directives du pré-processeur (inclusion, compilation conditionnelle)
- déclaration des constantes symboliques et macros
- déclarations des fonctions (utilisées dans d'autres fichiers)

Fichier `.c`

- variables permanentes utilisées que dans le fichier `.c`
- définition des fonctions d'interface présentes dans le fichier `.h`
- définition de fonctions locales au fichier `.c`
- le fichier `.h` doit être inclus dans le fichier `.c` et dans tous les `.c` qui ont besoin des fonctions du fichier

⇒ Fastidieux de gérer la compilation « à la main »

Structure d'un fichier Makefile

Un Makefile se compose différentes sections

- **déclarations de variables** sous la forme : `<nom> = <valeur>`
- **cible** : un nom d'exécutable ou de fichier objet
- **dépendances** : les éléments ou le code source nécessaires pour créer une cible
- **règles** : les commandes nécessaires pour créer la cible

`<déclarations de variables>`

```
<cible> : <dépendances>
    <TABULATION><regle1>
    ...
    <TABULATION><regle n>
```


Définition de la liste des dépendances

cible: liste de dépendances

<TAB> commandes UNIX

Exemple

```
prod: produit.o main.o
    gcc -o prod produit.o main.o
main.o: main.c produit.h
    gcc -c -O2 main.c
produit.o: produit.c produit.h
    gcc -c -O2 produit.c
```

```
>make
gcc -c -O2 produit.c
gcc -c -O2 main.c
gcc -o prod produit.o main.o
>make
make: 'prod' is up to date.
```

Si on modifie produit.c, main.o est à jour et n'est pas recompilé

```
>make
gcc -c -O2 produit.c
gcc -o prod produit.o main.o
```

Si on recommence :

```
>make
'prod' is up to date
```

Exemple avec debug

```
# Fichier executable prod
prod: produit.o main.o
    gcc -o prod produit.o main.o
main.o: main.c produit.h
    gcc -c -O2 main.c
produit.o: produit.c produit.h
    gcc -c -O2 produit.c

# Fichier executable pour le debuggage prod.db
prod.db: produit.do main.do
    gcc -o prod.db produit.do main.do
main.do: main.c produit.h
    gcc -o main.do -c -g -O2 main.c
produit.do: produit.c produit.h
    gcc -o produit.do -c -g -O2 produit.c

#cible permettant de nettoyer
clean:
    rm -f *.o *.do
```

```
>make prod.db
gcc -o main.do -c -g -O2 main.c
gcc -o produit.do -c -g -O2 produit.c
gcc -o prod.db produit.do main.do
```

```
>make clean
rm -f *.o *.do
```

⇒ cf TP 3

Note sur les options d'optimisation

Les différentes options -O

- -O : Premier niveau d'optimisation. Le compilateur tente de réduire la taille du code et le temps d'exécution tout en limitant le temps de compilation.
- -O2 : Deuxième niveau d'optimisation, le compilateur essaie d'effectuer une tâche d'optimisation plus importante. Par comparaison avec -O, cette option augmente à la fois le temps de compilation et les performances du code généré en termes de temps d'exécution. La taille du code peut être plus importante.
- -O3 : Troisième niveau d'optimisation, le compilateur essaie d'effectuer une tâche d'optimisation encore plus importante. -O3 active toutes les options d'optimisation spécifiées par -O2 plus d'autres. La taille du code produit peut être encore plus importante et le temps mis pour compiler est supérieur, cette option implique une utilisation mémoire bien plus importante.
- -Os : Optimisation concernant la taille du code produit. -Os active toutes les optimisations de -O2 qui n'ont pas d'impact négatif sur la taille du code, ainsi que d'autres optimisations conçues pour réduire la taille du code.
- -O0 : Aucune optimisation effectuée. C'est le comportement par défaut de GCC.
- -Og : autorise les options de compilation qui n'interfèrent pas avec le débogage.

Il n'y a pas de garantie que l'option -O3 produise un code plus efficace que -O2. La « science de l'optimisation » de programmes n'est pas toujours exacte et dépend de l'environnement d'exécution.

Note : en cas de code « pas forcément très propre », les options d'optimisation peuvent induire des bugs inexistantes sans leur activation. **Ces options sont donc à utiliser avec prudence.**

Pour spécifier une norme : l'option `-std/-ansi`

```
>gcc -o hello1 hello.c -ansi  
>gcc -o hello2 hello.c -std=c99
```

`-ansi` est équivalent à `-std=c89`.

`-std=c11` est une version récente de la norme

L'option `-pedantic` affiche tous les warnings requis par le standard ISO C (ou `-pedantic-errors` si vous préférez avoir des erreurs plutôt que des warnings).

Plus d'information sur les standards :

<https://gcc.gnu.org/onlinedocs/gcc/Standards.html>

Macros et abréviation : une meilleure généralisation

```
# definition du compilateur
CC = gcc

# definition des options de compilation pour obtenir un fichier .o
PRODUCTFLAGS = -c -std=c11 -O2

# definition des options de compilation pour obtenir un fichier .do
DEBUGFLAGS = -c -g -Og

# Fichier executable prod
prod: produit.o main.o
    $(CC) -o prod produit.o main.o
main.o: main.c produit.h
    $(CC) $(PRODUCTFLAGS) main.c
produit.o: produit.c produit.h
    $(CC) $(PRODUCTFLAGS) produit.c

# Fichier executable pour le debuggage prod.db
prod.db: produit.do main.do
    $(CC) -o prod.db produit.do main.do
main.do: main.c produit.h
    $(CC) -o main.do $(DEBUGFLAGS) main.c
produit.do: produit.c produit.h

$(CC) -o produit.do $(DEBUGFLAGS) produit.c
```

Un certain nombre de macros sont prédéfinies. En particulier,

- `$@` désigne le fichier cible courant
- `$*` désigne le fichier cible courant privé de son suffixe
- `$<` désigne le fichier qui a provoqué l'action.

Dans le Makefile précédent, la partie concernant la production de `main.do` peut s'écrire par exemple :

```
main.do: main.c produit.h
    $(CC) -o $@ $(DEBUGFLAGS) $<
```

Exemple avec macros

```
# definition du compilateur
CC = gcc
# definition des options de compilation pour obtenir un fichier .o
PRODUCTFLAGS = -c -std=c11 -O2
# definition des options de compilation pour obtenir un fichier .do
DEBUGFLAGS = -c -g -Og

# suffixes correspondant a des regles generales
.SUFFIXES: .c .o .do
# regle de production d'un fichier .o
.c.o;; $(CC) -o $@ $(PRODUCTFLAGS) $<
# regle de production d'un fichier .do
.c.do;; $(CC) -o $@ $(DEBUGFLAGS) $<

# Fichier executable prod
prod: produit.o main.o
    $(CC) -o prod produit.o main.o
produit.o: produit.c produit.h
main.o: main.c produit.h
# Fichier executable pour le debuggage prod.db
prod.db: produit.do main.do
    $(CC) -o prod.db produit.do main.do
produit.do: produit.c produit.h
main.do: main.c produit.h

clean:
    rm -f prod prod.db *.o *.do
```

Cf exemples vus en cours et présents sur claroline

Créer une librairie (statique)

Imaginons que l'on souhaite créer une librairie `libmesmath` contenant la fonction produit du fichier `produit.c` et la fonction somme du fichier `somme.c`

Procédures

- 1 Compiler chaque source :
`gcc -Wall -c produit.c somme.c`
- 2 Créer la bibliothèque `libmesmath.a` (option `r` pour concatener avec l'existant) :
`ar -r libmesmath.a produit.o somme.o`
- 3 Générer l'index de la librairie :
`ranlib libmesmat.a`
- 4 Créer l'en-tête `libmesmath.h`
(peut inclure `somme.h` et `produit.h` ou on indique les en-têtes directement)
- 5 On place la bibliothèque dans un répertoire (ex. `mylib/`) et l'en-tête dans un autre répertoire (ex. `myinclude/`)
- 6 Pour compiler un fichier source `essaifichier.c` avec cette bibliothèque, on utilise :
`gcc -I myinclude -L mylib essaifichier.c -o essaifichier -lmesmath`

On peut bien sûr automatiser tout ça avec un Makefile.

Portée des variables et des fonctions

Portée des déclarations dans une fonction

À l'intérieur d'une fonction, on a accès à

- les **fonctions** déclarées **avant**
- les **types** et les **variables** déclarés
 - **avant** et **dans la fonction**
 - **avant dans le fichier source** et **hors d'un bloc** (ex : hors d'une fonction)

À l'intérieur d'une fonction, on **ne voit pas**

- les **variables masquées** (par une déclaration locale de même nom)
- les **variables** déclarées **dans un bloc** (ex : dans une autre fonction)

Portée d'une fonction

Une fonction est **utilisable**

- dans **son module** de définition : après la définition ou après la déclaration
- dans un **autre module** : après la déclaration

Rq : Une fonction définie **static** **n'est pas accessible** depuis un **autre module**

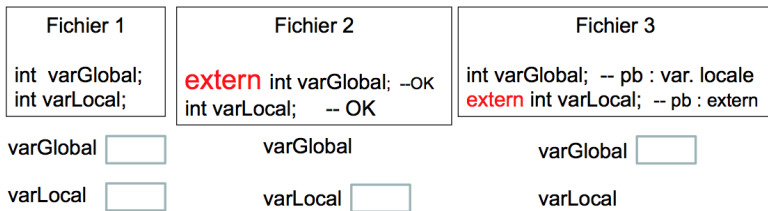
```
static int maFonctionLocale(){ ... }
```

Cette fonction ne sera utilisable **QUE** dans son module

Portée des variables

Une variable **définie en dehors d'un bloc (ou fonction)** est **accessible**

- dans **son module** de définition : après la déclaration
- dans un **autre module** : après une déclaration **extern**
(déclaration sans réserve de zone mémoire)



Portée des types

Un nouveau type (défini par `typedef`) a une portée **limitée à son module**

Pour pouvoir l'utiliser ailleurs

↪ Il faut créer et inclure un fichier de définition de types

Quelques mots sur `static` et `extern`

Attributs `static` et `extern` dans une programmation par modules

- `static` : pour une variable et une fonction

⇒ Implique que la définition reste locale au module

- `extern` : pour une variable

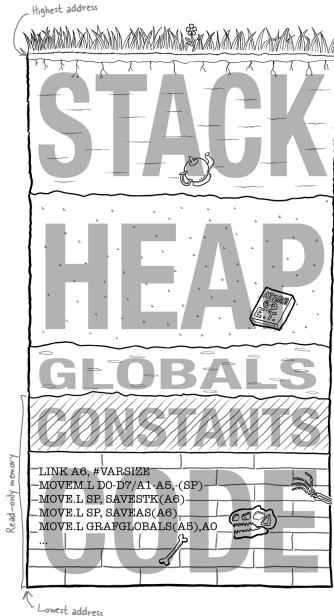
⇒ déclaration sans réserve de zone mémoire

⇒ variable dite “globale” qui doit être déclarée sans `static` dans un autre module

- `extern` : pour une fonction (implicite devant la déclaration)

Les différentes parties de la Mémoire

Les différentes parties en mémoire



La pile - stack

Section utilisée pour le stockage des **variables locales**. A chaque appel de fonction, toutes ses variables locales seront créées sur la pile. pile est en haut !

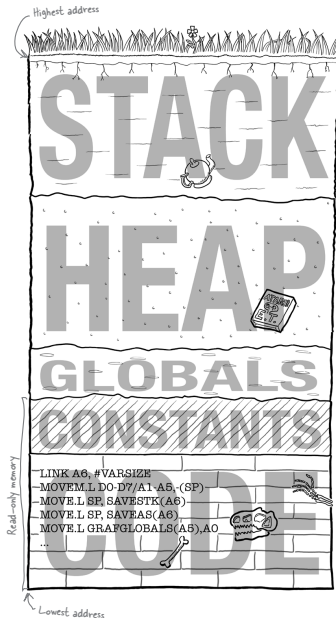
Les différentes parties en mémoire



Le tas - Heap

Le tas sert à l'allocation de mémoire **dynamique** que nous allons voir. Les variables allouées dynamiquement sont supposées rester longtemps en mémoire.

Les différentes parties en mémoire



Variables Globales

Une variable globale est créée et vit en dehors de toute fonction et est visible pour toutes les fonctions. Elles sont créées au début du programme et peuvent être modifiées quand vous le voulez. Mais ce n'est pas conseillé.

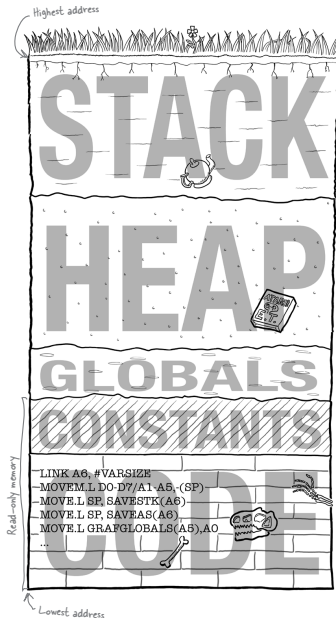
Les différentes parties en mémoire



Constantes

Les constantes sont aussi créées au début du programme mais elles sont stockées dans une zone de la mémoire en **lecture seule**. Ces constantes ne peuvent pas être modifiées.

Les différentes parties en mémoire



Le code

Pour terminer, le segment de code.
Beaucoup de systèmes d'exploitation placent le code à la plus petite adresse mémoire possible. Cette partie de code est en lecture seule. C'est la partie de la mémoire où le code assembleur est chargé.

Pointeurs

Quelques définitions

Lvalue

tout objet pouvant être placé à gauche d'un opérateur d'affectation, il est caractérisé par une adresse mémoire et une valeur.

Pointeur

Un objet (Lvalue) dont la valeur est égale à l'adresse d'un autre objet. Déclaration :

```
type * nom_pointeur
```

Exemple

```
int i = 3;  
int *p=NULL;  
  
p=&i;
```

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

Pointeurs et affectation

L'opérateur * permet d'accéder directement à la valeur de l'objet pointé.

Exemple

```
int main()
{
    int i = 3;
    int *p=NULL;

    p=&i;
    printf("*p = %d \n",*p);
    return EXIT_SUCCESS;
}
```

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Le programme affiche 3 et toute modification de *p modifie i, i et *p sont identiques dans ce programme.

Autre exemple

Exemple

```
int i = 3, j = 6;  
int *p1, *p2;  
p1=&i;  
p2=&j;
```

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

2 cas

```
*p1=*p2;
```

objet	adresse	valeur
i	4831836000	6
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

```
p1=p2;
```

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836004
p2	4831835992	4831836004

Opérations

- Addition d'un entier à un pointeur : le résultat est un pointeur de même type que le pointeur de départ ;
- Soustraction d'un entier à un pointeur : le résultat est un pointeur de même type que le pointeur de départ ;
- Différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.

La somme de deux pointeurs n'est pas autorisée.

Si i est un entier et p est un pointeur sur un objet de type `type`, l'expression $p + i$ désigne un pointeur sur un objet de type `type` dont la valeur est égale à la valeur de p incrémentée de $i * \text{sizeof}(\text{type})$. Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrément et de décrémentation `++` et `--`.

```
int i = 3;
int *p1, *p2;
p1 = &i;
p2 = p1 + 1;
printf("p1 = %ld \t p2 = %ld.\n", p1, p2);
```

Affiche :

p1 = 4831835984 p2 = 4831835988.

Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type.

Exemple 2

Avec des doubles

```
double i = 3;
double *p1, *p2;
p1 = &i;
p2 = p1 + 1;
printf("p1 = %ld \t p2 = %ld\n",p1,p2);
```

Affiche :

p1 = 4831835984 p2 = 4831835992.

Tableaux

```
#define N 5
int tab[N] = {1, 2, 6, 0, 7};
int main()
{
    int *p;
    printf("\n ordre croissant:\n");
    for (p = &tab[0]; p <= &tab[N-1]; p++)
        printf(" %d \n",*p);
    printf("\n ordre decroissant:\n");
    for (p = &tab[N-1]; p >= &tab[0]; p--)
        printf(" %d \n",*p);
}
```

Imprime les éléments du tableau tab dans l'ordre croissant puis décroissant des indices.

Si p et q sont deux pointeurs sur des objets de type type, l'expression $p - q$ désigne un entier dont la valeur est égale à $(p - q)/\text{sizeof}(\text{type})$.

Initialisation

- Avant de manipuler un pointeur (*c-à-d* avant d'appliquer l'opérateur `*`) il faut l'initialiser.
- On peut utiliser la constante `NULL` définie dans `<stdio.h>` (elle vaut en général 0). Le test `p == NULL` permet de savoir si un pointeur `p` pointe vers un objet.
- Pour initialiser un pointeur `p`, on peut lui affecter l'adresse d'une variable `&i`, l'autre solution est d'affecter une valeur directement par *allocation dynamique* de manière à réserver un espace mémoire de taille adéquate :
on réserve à `*p` un espace mémoire de taille suffisante, cet espace sera la valeur `p`.
⇒ la fonction `malloc` de `stdlib.h`

La fonction `malloc`

- `void* malloc (size_t size)` où `size_t size` est un nombre d'octets - on utilisera en général la fonction `sizeof()` pour obtenir ce nombre de manière portable
- Elle retourne un pointeur de type non spécifié, elle permet les conversions implicites mais on prendra l'habitude de convertir explicitement les types - attention dans l'indication dans le poly correspond à d'anciens compilateurs

Malloc - exemple 1

On définit un pointeur p de type int, et affecte à *p la valeur de la variable i.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i = 3;
    int *p=NULL;

    printf("valeur de p avant initialisation = %ld\n",p);
    p = (int *) malloc(sizeof(int));
    printf("valeur de p apres initialisation = %ld\n",p);
    *p = i;
    printf("valeur de *p = %d\n",*p);

    return EXIT_SUCCESS;
}
```

Résultat à l'écran :

```
valeur de p avant initialisation = 0
valeur de p apres initialisation = 5368711424
valeur de *p = 3
```

Analyse - exemple 1

Avant l'allocation

objet	adresse	valeur
i	4831836000	3
p	4831836004	0

A ce stade `*p` n'a aucun sens et toute manipulation provoquerait une violation d'accès mémoire `Segmentation fault`.

L'allocation dynamique

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	? (int)

A partir de `*p`, on a réservé 4 octets pour représenter un entier, non initialisé.

A la fin du programme

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	3

L'instruction `*p = i` permet d'affecter la valeur de `i` à `*p`.

Comparaison avec le programme précédent

```
int main()
{
    int i = 3;
    int *p=NULL;

    p=&i;
    printf("*p = %d \n",*p);
    return EXIT_SUCCESS;
}
```

Il correspond à la situation

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

- Ici i et *p sont identiques : elles ont la même adresse → toute modification de l'une modifie l'autre.
- Ce n'est pas vrai dans le programme précédent car *p et i ont la même valeur mais **des adresses différentes**
- Le programme ci-dessus ne nécessite pas d'allocation dynamique puisque l'espace-mémoire à l'adresse &i est déjà réservé pour un entier.

Allocation d'espace pour plusieurs objets contigus (tableaux)

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i = 3;
    int j = 6;
    int *p;
    p = (int *) malloc(2 * sizeof(int));
    *p = i;
    *(p + 1) = j;
    printf("p = %ld, *p = %d, p+1 = %ld, *(p+1) = %d, p[1] = %d.\n",
p,*p,p+1,*(p+1), p[1]);
    return EXIT_SUCCESS;
}
```

- On a ainsi réservé, à l'adresse donnée par la valeur de p, 8 octets en mémoire, qui permettent de stocker 2 objets de type int, soit un tableau de 2 entiers.
- Le programme affiche donc
p = 5368711424, *p = 3, p+1 = 5368711428, *(p+1) = 6, p[1] = 6.
- → on vient de voir comment allouer des tableaux dynamiquement !

- `void * calloc(int nb_objets, size_t size)`
Même rôle que `malloc` mais elle initialise en plus l'objet pointé (comprendre l'ensemble de l'espace mémoire) *p à zéro (NULL)
`p = (int*)calloc(N,sizeof(int));` est strictement équivalent à

```
p = (int*)malloc(N * sizeof(int));  
for (i = 0; i < N; i++)  
    *(p + i) = 0;
```

L'emploi de `calloc` est simplement plus rapide.

- `void *realloc(void *pcourrant, size_t size);`
Réallocation la mémoire à *pcourrant de taille size, s'il n'y a pas assez de place, `realloc` essaie une nouvelle allocation de taille size et renvoie le nouveau pointeur, copie le contenu initialement pointé par p et libère cette zone mémoire, si l'allocation n'est toujours pas possible il renvoie NULL comme `malloc`.
`p = realloc(p, 8*sizeof(int));`

- `void free(void * pointeur)`

Permet de libérer l'espace mémoire pointé par *p

A toute instruction de type malloc ou calloc, on doit associer une instruction de type free.

- **Important** : l'allocation peut ne pas avoir marché (plus de mémoire disponible), il est donc capital de tester le pointeur alloué dynamiquement et de générer une erreur si l'allocation n'a pas fonctionné.

```
p = (int *) malloc (4 * sizeof(int));  
/* on verifie si l'allocation a marche*/  
if (p==NULL)  
{  
    fprintf(stderr,"Impossible d'allouer la memoire %d, %d\n",4,  
sizeof(int));  
    exit(EXIT_FAILURE);  
}
```

⇒ Module allocation.c

Module allocation.c (1/2)

allocation_mem

```
void * allocation_mem(size_t nbjets,size_t taille)
{
    void * pt;
    pt = malloc (nbjets * taille);/*allocation*/
    /* on verifie si l'allocation a marche*/
    if (pt==NULL)
        mon_erreur("Impossible d'allouer la memoire %d %d\n",nbjets,taille);
    return(pt);
}
```

allocation_mem_init0 - version avec calloc

```
void * allocation_mem_init0(size_t nbjets,size_t taille)
{
    void * pt;
    pt = calloc (nbjets,taille);/*allocation avec calloc*/
    /* on verifie si l'allocation a marche*/
    if (pt==NULL)
        mon_erreur("Impossible d'allouer la memoire %d * %d\n",nbjets
,taille);
    return(pt);
}
```

Module allocation.c (2/2)

reallocation_mem - version avec realloc

```
void* reallocation_mem(void **pt, size_t nobjets, size_t taille)
{
    void *pt_realloc = realloc(*pt, nobjets*taille);
    if (pt_realloc != NULL)
        *pt = pt_realloc;
    else
        mon_erreur("Impossible de reallouer la memoire %d * %d a
l'adresse %p\n", nobjets, taille, *pt);

    return pt_realloc;
}
```

libere_mem - fonction de libération (version pédagogique)

```
void libere_mem_peda(void * *pt)
{
    if((*pt)!=NULL)
        free(*pt); /*liberation de *pt */

    *pt=NULL; /* *pt pointe maintenant sur NULL, cad rien*/
}
```

Le fichier allocation.h

```
#ifndef _ALLOCATION_H_
#define _ALLOCATION_H_

/*- ... */
void * allocation_mem(size_t nbjets,size_t taille);

/*- ... */
void * allocation_mem_init0(size_t nbjets,size_t taille);

/*- ... */
void* reallocation_mem(void **pt, size_t nbjets,size_t taille);

/*- ... */
void libere_mem(void *pt);

/*- ... */
void libere_mem_peda(void * *pt);

#endif
```

Une note sur void *

- Le type void * fait référence à tout type de pointeur, nous l'avons vu avec la fonctions malloc : int **, char *, float ***, ...
- Il faut donc généralement utiliser une conversion de type (cast) pour pouvoir utiliser un élément de type void * de sorte que le bon type soit associé à la variable.
- La fonction libere_mem_peda est un peu bizarre dans sa formulation en demandant un type void ** (générateur de warning). Une version plus générique peut être définie de la manière suivante :

```
void libere_mem(void *pt)
{
    void ** adr_pt=(void **) pt;
    if((*adr_pt)!=NULL)
        free(*adr_pt); /*liberation de *pt */

    *adr_pt=NULL; /* *pt pointe maintenant sur NULL, cad rien*/
}
```

- ❶ Il faut par contre bien appeler la fonction avec l'adresse du pointeur sur l'adresse mémoire que l'on souhaite libérer (en utilisant l'opérateur &).
- ❷ **On utilisera plutôt cette version de libere_mem.**
La fonction précédente peut être vue comme un outil pédagogique.

Note : tab a pour valeur &tab[0]

```
#define N 5
int tab[N] = {1, 2, 6, 0, 7};
main()
{
    int i;
    int *p;
    p = tab;
    for (i = 0; i < N; i++)
    {
        printf(" %d \n",*p);
        p++;
    }
    return EXIT_SUCCESS;
}
```

Tableaux à une dimension

Autre version - `*(p+i)=p[i]`

```
#define N 5
int tab[N] = {1, 2, 6, 0, 7};
int main()
{
    int i;
    int *p;
    p = tab;
    for (i = 0; i < N; i++)
        printf(" %d \n", p[i]);

    return EXIT_SUCCESS;
}
```

La manipulation de tableaux a des inconvénients (un tableau est un pointeur constant)

- On ne peut pas créer de tableaux dont la taille est donnée par une variable
- on ne peut pas créer de tableaux bi(multi)-dimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.

⇒ Mais c'est possible par allocation dynamique.

Création dynamique d'un tableau de taille variable

tableau à n éléments où n est une variable du programme

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

int main()
{
    int n;
    int *tab=NULL;
    ...
    tab = (int*)malloc(n * sizeof(int)); //(int *)allocation_mem(n,sizeof(int))
    ...
    free(tab); // libere_mem(&tab);
    ...
    return EXIT_SUCCESS;
}
```

Si on veut en plus que tous les éléments du tableau `tab` soient initialisés à zéro, on utilise : `tab = (int*)calloc(n, sizeof(int));`
Les éléments de `tab` peuvent être manipulés avec l'opérateur d'indexation `[]`, exactement comme pour les tableaux, ou avec l'opérateur `*`.

Deux différences principales entre tableau et pointeur

- un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse, par exemple `p = &i`;
- un tableau n'est pas une `Lvalue`, il ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut pas écrire `tab++`), par contre les pointeurs oui - **mais attention à ce que vous faites !**.
- Implication importante pour l'appel de fonctions avec tableaux à plusieurs dimensions ! (cf plus tard)

On reviendra sur ce sujet plus tard.

Pointeurs et tableaux à plusieurs dimensions

- Tableau à deux dimensions : lorsqu'ils sont manipulés par des pointeurs ils correspondent à un tableau de tableaux et donc un pointeur vers un pointeur
- Exemple avec un tableau de taille fixée : `int tab[M][N];` :
 - `tab` peut être associé à un pointeur, qui pointe vers un objet lui-même de type pointeur d'entier.
 - `tab` a une valeur constante égale à l'adresse du premier élément du tableau, `&tab[0][0]`.
 - `tab[i]`, pour `i` entre 0 et `M-1`, peut être vu comme un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice `i`. `tab[i]` a donc une valeur constante qui est égale à `&tab[i][0]`.

Pointeur de pointeurs

- Objet à 2 dimensions de type `type` : `type **nom-du-pointeur;`
- Objet à 3 dimensions de type `type` : `type ***nom-du-pointeur;` et ainsi de suite.

Exemple matrice d'entiers à k lignes et n colonnes à partir de pointeur de pointeur

Allocation en 2 temps : les lignes puis les colonnes de chaque ligne

```
int main() {
    int k, n;
    int **tab=NULL;

    tab = (int**)malloc(k * sizeof(int*));
    for (i = 0; i < k; i++)
        tab[i] = (int*)malloc(n * sizeof(int)); //(int *)allocation_mem(n,sizeof(int))
        ....

    for (i = 0; i < k; i++)
        free(tab[i]); // libere_mem(&tab[i])

    free(tab); // libere_mem(&tab)
    return EXIT_SUCCESS;
}
```

La première allocation de `tab` crée k pointeurs sur des entiers, ces pointeurs correspondent aux lignes de la matrice. On réserve ensuite pour chaque `tab[i]` les n entiers pour avoir les colonnes.

Exemple - Matrices - suite

Si on veut que les éléments du tableau soient initialisés à 0

```
tab[i] = (int *) calloc(n,sizeof(int));
```

ou avec le module `allocation.c` :

```
tab[i] = allocation_mem_init0(n,sizeof(int));
```

Des tailles de colonnes différentes pour chaque ligne

Si on veut que `tab[i]` contienne exactement `i+1` éléments, on peut écrire :

```
for(i = 0; i < k; i++)  
    tab[i] = (int *) malloc((i+1)*sizeof(int));
```

ou avec le module `allocation.c` :

```
for(i = 0; i < k; i++)  
    tab[i] = (int *) allocation_mem(i+1,sizeof(int));
```

Attention il ne faut pas oublier de vérifier que l'allocation a marché !

Pointeurs et chaînes de caractères

On sait qu'une chaîne de caractères est un tableau à une dimension d'objets de type `char`, se terminant par le caractère nul `'\0'`. On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type `char`.

- `char * chaine = NULL;`
- `chaine = "ceci est une chaine";` //l'allocation est automatique

Exemple : affichage du nb de caractères d'une chaîne

```
#include <stdio.h>
int main()
{
    int i;
    char *chaine;
    chaine = "chaine de caracteres";
    for (i = 0; *chaine != '\0'; i++)
        chaine++;
    printf("nombre de caracteres = %d\n",i);
    return EXIT_SUCCESS;
}
```

Note : à cause de l'instruction `chaine++`; la chaîne est perdue ici.

Exemple : concaténation de 2 chaînes (avec une erreur)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    int i;
    char *chaine1=NULL, *chaine2=NULL, *res=NULL, *p=NULL;
    chaine1 = "chaine ";
    chaine2 = "de caracteres";

    res = (char*)allocation_mem((strlen(chaine1) + strlen(chaine2)),
sizeof(char));
    p = res;
    for (i = 0; i < strlen(chaine1); i++)
        { *p = chaine1[i]; p++; }
    for (i = 0; i < strlen(chaine2); i++)
        { *p = chaine2[i]; p++; }
    printf("%s\n",res);

    return EXIT_SUCCESS;
}
```

Exemple : concaténation de 2 chaînes - NE PAS OUBLIER LE '\0'

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    int i;
    char *chaine1=NULL, *chaine2=NULL, *res=NULL, *p=NULL;
    chaine1 = "chaine ";
    chaine2 = "de caracteres";

    res = (char*)allocation_mem((strlen(chaine1) + strlen(chaine2)+1),
sizeof(char));
    p = res;
    for (i = 0; i < strlen(chaine1); i++)
        { *p = chaine1[i]; p++; }
    for (i = 0; i < strlen(chaine2)+1; i++)
        { *p = chaine2[i]; p++; }
    printf("%s\n",res);

    return EXIT_SUCCESS;
}
```


Exemple suite - attention si l'on se passe d'un pointeur ...

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    int i;
    char *chaine1=NULL, *chaine2=NULL, *res=NULL;
    chaine1 = "chaine ";
    chaine2 = "de caracteres";

    res = (char*)allocation_mem((strlen(chaine1) + strlen(chaine2)+1),
sizeof(char));
    for (i = 0; i < strlen(chaine1); i++)
        { *res = chaine1[i]; res++;}
    for (i = 0; i < strlen(chaine2)+1; i++)
        *res++ = chaine2[i];
    printf("\nnombre de caracteres de res = %d\n",strlen(res));
    return EXIT_SUCCESS;
}
```

⇒ imprime la valeur 0, puisque res a été modifié au cours du programme et pointe maintenant sur le caractère nul.

Exemple d'affichage inversé de chaîne avec pointeur

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

void affiche_inv_recu(char * chaine)
{
    if(*chaine!='\0')
    {
        affiche_inv_recu(chaine+1);
        printf("%c",*chaine);
    }
}

int main(int argc,char * argv[])
{
    if(argc==2)
    {
        char * pt_chaine=argv[1];
        affiche_inv_recu(pt_chaine);
        printf("\n");
    }
    return EXIT_SUCCESS;
}
```

Note sur scanf et const

scanf

- Hormis pour écrire des petits programmes simples d'illustration, il est en général déconseillé d'utiliser `scanf`
- Si vous demandez un entier `%d` et que l'utilisateur commence par écrire, le `scanf` risque de se "bloquer", ceci peut provoquer des dépassements de "*tampon*" (buffer)
- On préférera utiliser `fgets` ;

```
char buffer[256];
```

```
...
```

```
fgets(buffer, 256*sizeof(char), stdin);
```

⇒ lecture au maximum des 256 premiers caractères sur l'entrée standard puis copie dans `buffer`. Grâce au maximum on évite tout dépassement.

const

- sert à indiquer une constante ne pouvant pas être modifiée

```
const int x = 10;
```

```
const double t[3] = {10,11,12};
```

```
...
```

```
int f(const int * t, const char c, int a);
```

⇒ Ecrire `t[1]=3`; ou `c=4`; provoquera une erreur (souvent `bus error`)

Note sur sizeof

- S'applique sur des types (int, double, ...), une expression, ou un tableau :

```
double t[10];  
sizeof(t) // Taille de t en octets  
10 * sizeof(double) // 10 fois taille double  
sizeof(double[10]) // taille tableau de double  
10 * sizeof(t[0]) // 10 fois taille 1er element
```

- **Attention** : dans le contexte d'une fonction, ce n'est plus vrai !

```
#include <stdio.h>  
void ma_fonction(int * t) {  
    int nb = sizeof(t) / sizeof(t[0]);  
    printf("nb=%i\n", nb);  
}  
int main() {  
    double t[10];  
    int nb = sizeof(t) / sizeof(t[0]);  
    printf("nb=%i\n", nb);  
    ma_fonction(t);  
    ...  
}
```

⇒ affiche nb=10 puis nb=1 !

Tableaux à plusieurs dimensions et passage d'arguments

Tableaux « constants »

```
int tab[2][3];
```

Accès à l'élément i, j : `tab+i*3*sizeof(int)+j`

Il faut connaître la dimension du dernier argument

Pointeurs

```
int **p; //il faudra rajouter une allocation
```

Accès à l'élément i, j : `*(*(p+i)+j)`

Pas besoin de connaître la dimension du 2eme argument, on passe pour les adresses.

Il faut prendre en compte cet aspect dans les appels de fonction (cf correction TD3).

Pour un tableau « constant » à N dimensions, il faut indiquer les $N-1$ dernières dimensions lors d'un passage d'arguments.

Différence entre tableaux et pointeurs

Ce qui dit la norme c90

Sauf quand elle est l'opérande de l'opérateur `sizeof` ou de l'opérateur unaire `&`, ou est une chaîne de caractères littérale utilisée pour initialiser un tableau, une expression de type « *tableau de type* » est convertie en une expression de type « *pointeur de type* » qui pointe sur l'élément initial de l'objet tableau et n'est pas une lvalue. Si le tableau est d'une classe de stockage registre (mot clé `register`, le comportement est indéterminé.

Conséquence 1

```
int tab[3];  
int * p;  
p=&tab;
```

L'affectation ci-dessus est correcte : `p` pointe sur un tableau de 3 entiers

Conséquence 2

```
char *p = "hello";  
chat t1[] = "world";  
chat t2[] = p; /* Faux */  
chat t3[] = (char *) "titi"; /* Faux */
```

Les deux dernières lignes sont fausses car un tableau ne peut pas être initialisé avec une valeur scalaire.

t1 par contre est un tableau de 6 caractères initialisé avec la chaîne *world*.

Le pointeur p récupère l'adresse de la chaîne constante, on ne peut donc plus modifier son contenu, pour être plus propre on pourrait écrire

```
const char *p= "hello";
```

Pour le problème de passage de tableau à plusieurs dimensions, cf le TD.

Les points liés au cas `register` ne seront pas traités dans ce cours - il signifie que la variable doit être stockée dans le registre du processeur et dans la mémoire vive - dans le contexte de tableau le comportement est indéterminé par rapport à la relation avec les pointeurs (ça n'a pas forcément de sens déterminé ici).

Structure de données

Définition

Assemblage de données de types éventuellement distincts

Motivation : Regrouper des informations de types différents

Exemple

Un étudiant est défini par :

- un nom
- un (ou des) prénom(s)
- une adresse
- une date de naissance
- l'année d'inscription
- ...

Définition

Suite finie d'objets de types différents. Déclaration (notez le ; à la fin) :

```
struct modele
{
    type_1 membre_1;
    type_2 membre_2;
    ...
    type_n membre_n;
};
```

On peut ensuite utiliser la structure comme nouveau type :

```
struct modele nom_variable;
```

Accès

On accède aux différents membres d'une structure grâce à l'opérateur . :

```
nom_variable.membre_i;
```

Exemple classique

```
#include <math.h>
struct complexe
{
    double reelle;
    double imaginaire;
};
int main() {
    struct complexe z;
    double norme;
    ...
    norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
    printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme);
    return EXIT_SUCCESS;
}
```

Option d'écritures

- Déclaration + initialisation : `struct complexe z = {2., 2.};`
- On peut appliquer l'opérateur d'affectation à des structures (implique une recopie des champs)

```
struct complexe z1, z2;
...
z2 = z1;
```

Les types composés avec typedef

Déclaration

```
typedef type nom;
```

Exemple avec 2 solutions possibles

```
struct complexe  
{  
    double reelle;  
    double imaginaire;  
};
```

```
typedef struct complexe complexe;
```

```
int main() {  
    complexe z;  
    ...  
}
```

```
struct struct_complexe  
{  
    double reelle;  
    double imaginaire;  
};
```

```
typedef struct struct_complexe complexe;
```

```
int main() {  
    complexe z;  
    ...  
}
```

Un autre exemple

```
struct date {
    int an;
    short mois, jour;
} ;
typedef struct date date;
struct personne {
    char nom[20], prenom[20];
    date naissance;
} ;
typedef struct personne etudiant;
```

Il existe une écriture simplifiée :

```
typedef struct {
    int an;
    short mois, jour;
} date ;
typedef struct {
    char nom[20], prenom[20];
    date naissance;
} etudiant;
```

Pour accéder aux différents champs d'une variable de type étudiant :

```
etudiant etu; /* declaration d'une variable de type etudiant */
etu.nom="Dupond";
etu.prenom="Marcel";
etu.naissance.annee=1995;
etu.naissance.mois=3;
etu.naissance.jour=21;
```

Autre type : les unions (1/2)

Définition

Une union désigne un ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire. Une union permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types. Si les membres d'une union sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

Exemple/déclaration

```
union jour {  
    char lettre;  
    int numero;  
};  
main() {  
    union jour hier, demain;  
    hier.lettre = 'J';  
    printf("hier = %c\n",hier.lettre);  
    hier.numero = 4;  
    demain.numero = (hier.numero + 2) % 7;  
    printf("demain = %d\n",demain.numero);  
}
```

Autre type : les unions (2/2)

Autre exemple

Les unions peuvent être utiles lorsqu'on a besoin de voir un objet sous des types différents (mais en général de même taille). Par exemple, le programme suivant permet de manipuler en même temps les deux champs de type `unsigned int` d'une structure en les identifiant à un objet de type `unsigned long` (en supposant que la taille d'un entier long est deux fois celle d'un `int`).

```
struct coordonnees
{
    unsigned int x;
    unsigned int y;
};

union point {
    struct coordonnees coord;
    unsigned long mot;
};

int main() {
    union point p1, p2, p3;
    p1.coord.x = 0xf;
    p1.coord.y = 0x1;
    p2.coord.x = 0x8;
    p2.coord.y = 0x8;
    p3.mot = p1.mot ^ p2.mot;
    printf("p3.coord.x = %x \t p3.coord.y = %x\n", p3.coord.x, p3.coord.y);
}
```

Autre type : les énumérations

Définition

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clef `enum` et un identificateur, suivis de la liste des valeurs que peut prendre cet objet :

```
enum modele {constante-1, constante-2,...,constante-n};
```

Exemple

En pratique les objets de type `enum` sont souvent représentés comme des `int` (les valeurs possibles étant codées par des entiers). Par exemple, dans le programme suivant le type `booléen` associe l'entier 0 à `faux` et l'entier 1 à `vrai`.

```
int main(){
    enum booléen {faux, vrai};
    enum booléen b;
    b = vrai;
    printf("b = %d\n",b);
}
```

On peut aussi modifier le codage par défaut des valeurs lors de la déclaration :

```
enum booléen {faux = 12, vrai = 23};
```


Créations de types, structures et pointeurs

Pointeur et structure : on peut assigner des pointeurs sur des structures

```
#include <stdio.h>

struct eleve{
    char nom[20];
    int date;
};

typedef struct eleve * classe;

int main() {
    int n, i;  classe tab;
    printf("nombre d'eleves de la classe = ");
    scanf("%d",&n);
    tab = (classe)malloc(n * sizeof(struct eleve));
    for (i =0 ; i < n; i++){
        printf("\n saisie de l'eleve numero %d\n",i);
        printf("nom de l'eleve = ");
        scanf("%s",tab[i].nom);
        printf("\n date de naissance JJMMAA = ");
        scanf("%d",&tab[i].date);
    }
    printf("\n Entrez un numero  ");
    scanf("%d",&i);
    printf("\n Eleve numero %d:",i);
    printf("\n nom = %s",tab[i].nom);
    printf("\n date de naissance = %d\n",tab[i].date);
    free(tab);
}
```

Accès à un membre d'une structure pointée par un pointeur p

`(*p).membre`

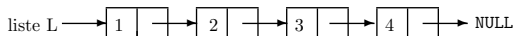
L'usage de parenthèses est indispensable. Cette notation peut être simplifiée grâce à l'opérateur *pointeur de membre de structure*, noté `->`, l'expression précédente est ainsi strictement équivalente à :

`p->membre`

Dans le programme précédent, on peut ainsi remplacer `tab[i].nom` et `tab[i].date` respectivement par `(tab + i)->nom` et `(tab + i)->date`.

Le type liste : structure auto-référencée

- Besoin de modèles de structure dont un des membres est un pointeur vers une structure de même modèle.
- Cette représentation permet en particulier de construire des listes chaînées. En effet, il est possible de représenter une liste d'éléments de même type par un tableau (ou un pointeur).
- Pour éviter de fixer une taille a priori, on utilise une représentation chaînée : l'élément de base de la chaîne est une structure appelée cellule qui contient la valeur d'un élément de la liste et un pointeur sur l'élément suivant. Le dernier élément pointe sur la liste vide NULL.
- La liste est alors définie comme un pointeur sur le premier élément de la chaîne.



Définition d'une cellule

```
typedef struct cellule
{
    int valeur;
    struct cellule * suivant;
}struct_cellule;
typedef struct_cellule * liste;
```

Insertion d'un élément en tête de liste

```
liste insere(int element, liste l)
{
    liste newcell=liste_vide();
    newcell=(liste)allocation_mem(1,sizeof(struct_cellule));
    newcell->valeur = element;
    newcell->suivant = l;
    return newcell;
}
```

Fonction de création d'une liste vide

```
liste liste_vide()
{
    return NULL;
}
```

Exemple d'utilisation dans un main

```
int main() {
    liste l, p;

    l = insere(1,insere(2,insere(3,insere(4,liste_vide()))));

    printf("\n impression de la liste:\n");
    p = l;
    while (p != liste_vide())
    {
        printf("%d \t",p->valeur);
        p = p->suivant;
    }
    printf("\n");

    return EXIT_SUCCESS;
}
```

TAD (Type Abstrait de Données) - ex : listes

TAD

Définition d'un type (représentation) et de l'ensemble des fonctionnalités pour le manipuler

Les objets

- La notion de cellule
- La notion de liste

Les fonctionnalités

- liste vide : renvoie une liste
- savoir si une liste est vide : prend une liste et renvoie un booléen (int)
- ajouter un élément à une liste : prend une liste et un élément et renvoie la nouvelle liste
- extraire le premier élément de la liste : prend une liste et renvoie un élément
- supprimer une cellule d'une liste : prend une liste et renvoie la nouvelle liste

(on peut en ajouter d'autre : rechercher, parcourir, détruire, insérer au début/au milieu/à la fin, trier, ...)

TAD - Le fichier liste.h (rajouter les #ifndef et #endif)

```
#include <stdio.h>
#include "allocation.h"

typedef int element;

typedef struct cellule{
    element objet;
    struct cellule * suivant;
}struct_cellule;
typedef struct_cellule * liste;

/* renvoie la liste vide */
liste liste_vide();
/* teste si une liste est vide */
int est_liste_vide(liste l);
/*ajoute un element elem a la liste l */
liste inserer_element_liste(liste l, element elem);
/* renvoie le premier element de la liste l */
element renvoie_premier_liste(liste l);
/* supprime la premiere cellule de la liste l */
liste supprimer_premier_liste(liste l);
```


Les fonctions du TAD (liste.c 1/3)

liste vide

```
liste liste_vide()
{
    return NULL;
}
```

savoir si une liste est vide

```
int est_liste_vide(liste l)
{
    if(l==liste_vide())
        return 1;
    return 0;
}
```

Les fonctions du TAD (liste.c 2/3)

insérer un élément (au début de la liste)

```
liste inserer_element_liste(liste l,element elem)
{
    liste lnew=(liste)allocation_mem(1,sizeof(struct_cellule));
    lnew->objet=elem;
    lnew->suivant=l;
    return lnew;
}
```

(cf schéma au tableau)

extraire le premier élément de la liste

```
element renvoie_premier_liste(liste l)
{
    if(est_liste_vide(l))
        mon_erreur("Erreur la liste est vide dans la fonction
renvoie_premier\n");
    return l->objet;
}
```

supprimer une cellule (au début de la liste)

```
liste supprimer_premier_liste(liste l)
{
    liste lsuivant=l->suivant;
    libere_mem(&l);
    return lsuivant;
}
```

Exercices :

- 1 écrire une fonction de suppression qui peut supprimer un élément (donné en argument) au milieu de la liste
- 2 ré-écrire les fonctions d'insertion et suppression de manière à modifier la liste donnée en argument de manière adéquate.

Objet de type LIFO : Last In First Out.

TAD Pile

- Créer une pile vide
- Empiler un nouvel élément
- Récupérer l'élément au sommet de la pile
- Dépiler le premier élément
- Connaître le nombre d'éléments de la pile

pile.h - On utilise le type Liste pour implémenter une pile

```
#ifndef _PILE_H_
#define _PILE_H_

#include "liste.h"

typedef liste pile;

pile pile_vide();

int est_pile_vide(pile p);

pile empiler(pile p,element e);

element sommet(pile p);

pile depiler(pile p);

int taille_pile(pile p);
#endif
```

```
#include <stdio.h>
#include "pile.h"

pile pile_vide()
{
    return liste_vide();
}

int est_pile_vide(pile p)
{
    return est_liste_vide(p);
}

pile empiler(pile p,element e)
{
    return inserer_element_liste(p,e);
}
```

```
element sommet_pile(pile p)
{
    return renvoie_premier_liste(p);
}

pile depiler(pile p)
{
    return supprimer_premier_liste(p);
}

int taille_pile(pile p)
{
    int nb=0;
    liste lcourant=p;
    while(lcourant)
    {
        nb++;
        lcourant=lcourant->suivant;
    }
    return nb;
}
```


- 1 ré-écrire les fonctions d'insertion et suppression de manière à modifier la liste donnée en argument de manière adéquate

Objet de type FIFO : First In First Out.

TAD File

- Créer une file vide
- enfiler un nouvel élément
- Récupérer l'élément au début de la file
- défiler le premier élément
- Connaître le nombre d'éléments de la file

file.h - on utilise encore le type Liste pour implémenter une file

```
#ifndef _FILE_H_
#define _FILE_H_

#include "liste.h"

typedef liste file;

file file_vide();

int est_file_vide(file f);

file enfiler(file f,element e);

element debut_file(file f);

file defiler(file f);

int taille_file(file f);
#endif
```

```
#include <stdio.h>
#include "file.h"

file file_vide()
{
    return liste_vide();
}

int est_file_vide(file f)
{
    return est_liste_vide(f);
}

file defiler(file f)
{
    return supprimer_premier_liste(f);
}
```

```
element debut_file(file f)
{
    return renvoie_premier_liste(f);
}
```

```
int taille_file(file f)
{
    int nb=0;
    liste lcourant=f;

    while(lcourant)
    {
        nb++;
        lcourant=lcourant->suivant;
    }

    return nb;
}
```

```
file enfiler(file f,element e)
{
    liste lnew=inserer_element_liste(liste_vide(),e);
    liste lcourant=f;

    if(est_file_vide(f))
        return lnew;

    while(lcourant->suivant!=liste_vide())
    {
        lcourant=lcourant->suivant;
    }

    lcourant->suivant=new;

    return f;
}
```

- ❶ Ré-écrire les fonctions d'insertion et suppression de manière à modifier la liste donnée en argument de manière adéquate (facile, c'est comme pour les piles).
- ❷ Implémenter une file à l'aide de 2 pointeurs : un sur le début de la file et un sur le dernier élément de la file (permet de faciliter l'écriture de la fonction enfiler)

File avec 2 pointeurs de liste - Fichier h

```
#ifndef _FILE_H_
#define _FILE_H_
#include "liste.h"

typedef struct cell_file{
    liste debut;
    liste fin;
}cell_file;
typedef cell_file * file;

file file_vide();

int est_file_vide(file f);

file enfiler(file f,element e);

element debut_file(file f);

defiler defiler(file f);

int taille_file(file f);
#endif
```


file.c (1/4)

```
#include <stdio.h>
#include "file.h"
#include "allocation.h"

file file_vide()
{
    file f=(file)allocation_mem(1,sizeof(cell_file));
    f->debut=liste_vide();
    f->fin=liste_vide();
    return f;
}

int est_file_vide(file f)
{
    return (est_liste_vide(f->debut) && est_liste_vide(f->fin));
}

element debut_file(file f)
{
    return renvoie_premier_liste(f->debut);
}
```

```
int taille_file(file f)
{
    int nb=0;
    liste lcourant=f->debut;

    while(lcourant)
    {
        nb++;
        lcourant=lcourant->suivant;
    }

    return nb;
}
```

```
/*supprimer l'element en tete de file */
file defiler(file f)
{
    if(est_file_vide(f))
        return f;

    f->debut=supprimer_premier_liste(f->debut);

    if(est_liste_vide(f->debut))
        f->fin=liste_vide();

    return f;
}
```

```
file enfiler(file f,element e)
{
    liste lnew=insere_element_liste(liste_vide(),e);

    if(est_file_vide(f))
        f->debut=lnew;
    else f->fin->suivant=lnew;

    f->fin=lnew;

    return f;
}
```

file.c (annexe) - autre prototype de suppression

Si on change le prototype de defiler de manière à renvoyer l'élément à supprimer, on doit ajouter le prototype suivant dans le fichier d'en-tête :

```
element defiler(file f);
```

La fonction peut alors être définie de la manière suivante, nous devons par contre gérer un cas particulier ne cas de file vide, on propose de renvoyer une constante appelée ELEMENT_VIDE à définir dans le fichier d'en-tête (par exemple -1).

```
element defiler_av_element(file f)
{
    element e;

    if(est_file_vide(f))
        return ELEMENT_VIDE;

    e=f->debut->objet;

    f->debut=supprimer_premier_liste(f->debut);

    if(est_liste_vide(f->debut))
        f->fin=liste_vide();

    return e;
}
```