

# 3-IF / 3-IFA Programmation C

Frédéric Prost  
`frederic.prost@insa-lyon.fr`

INSA Lyon

2025-2026

- Cours: Frédéric Prost ; e-mail: [frederic.prost@insa-lyon.fr](mailto:frederic.prost@insa-lyon.fr)
- Pages du cours : Moodle INSA.
- Structure du cours :
  - Cours : 4 x 1.5 h de C et 2 x 1.5 h d'Introduction à la sécurité informatique.
  - TP : 4 x 4h. (Non notés, vérification des acquis)
- Evaluation du cours : Un examen le 20/11/2025.

# Plan

- 1 Introduction
- 2 Concepts de base du langage C
- 3 C - mémoire, tableaux, pointeurs et dépendances
- 4 Intermède : process en cours d'exécution
- 5 Structuration des données et des programmes
  - Structuration des données
  - Structuration des programmes
- 6 Aspects systèmes

# Histoire du C

- Inventé fin des années 60 (69/73) par Ken Thompson et Dennis Ritchie pour programmer un système d'exploitation: Unix.
- C est un macro assembleur portable. Un langage très proche de la machine.
  - Avantages :
    - Très efficace.
    - Rien n'est mystérieux.
  - Inconvénients :
    - Il faut tout faire (ou presque) à la main.
    - Il est possible d'écrire n'importe quoi...
- Toujours un des langages les plus utilisé au monde.
  - C++, C# sont des descendants directs (objet).
  - Java, Javascript adoptent la syntaxe "style C".

# The Internation Obfuscated C Contest

- IOCC :
    - To write the most Obscure/Obfuscated C program within the rules.
    - To show the importance of programming style, in an ironic way.
    - To stress C compilers with unusual code.
    - To illustrate some of the subtleties of the C language.
  - Le but était de faire un langage très compact : les limitations en terme de mémoire n'existent plus.
  - Il n'y a même pas l'excuse de l'efficacité : le code produit par le compilateur ne change pas c'est juste moins lisible.  
Une expression qui tient moins de place n'est pas différente d'une plus longue et humainement compréhensible.
- ⇒ Programmer en C est une école de self-contrôle et de politesse (enver vous même la plupart du temps).

[illegible]

# Plan du cours

- ❶ Concepts de base (compilation, types, structures de contrôle,...).
- ❷ Gestion des pointeurs et mémoire.
- ❸ Structuration des données et des programmes.
- ❹ Entrées-Sorties, Interface système.

Utilisation de gdb (debugger), Makefile, ...

# Plan

- 1 Introduction
- 2 Concepts de base du langage C**
- 3 C - mémoire, tableaux, pointeurs et dépendances
- 4 Intermède : process en cours d'exécution
- 5 Structuration des données et des programmes
  - Structuration des données
  - Structuration des programmes
- 6 Aspects systèmes

# Philosophie du paradigme impératif

- L'action de base est la modification de la mémoire : affectation de variables.  
 $\implies x = 15$
- Il faut indiquer à chaque instant ce que doit faire l'ordinateur.  
 $\implies$  Structures de contrôles 'if', 'while', 'for', ';;', ...
- Le programme, écrit en C, dans un fichier est compilé en langage machine (assembleur) qui peut être exécuté dans un environnement logiciel (le système d'exploitation).

# Un premier programme

- Un programme qui ne fait rien :

```
int main()
{
    return 0;
}
```

- la fonction `main()` est le point d'entrée du programme.
- le `return 0;` indique que le programme, à la fin de son exécution, renvoie un code de retour de 0, indiquant que tout se passe bien (on verra ça plus tard dans le chapitre sur le système).

# Conversion entre Fahrenheit et Celsius

```
#include <stdio.h>

int main(void) {

    double fahr;
    double celsius;
    int    lower = 0;    /******//
    int    upper = 300;  /*    Déclarations des variables    */
    int    step  = 20;   /******//

    printf("Fahrenheit\tCelsius\n");
    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0 / 9.0) * (fahr - 32.0);
        printf("%3.0f\t\t%6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
    return 0;
}
```

# Retour sur le programme de conversion

- Notion de variable, assignation (lvalue et rvalue), types, coercion.
- Structures de contrôle : `';`, `while`
- Types `int`, `float`.  
le test (`fahr <= upper`) est un entier avec la sémantique:
  - "faux" est équivalent à 0.
  - tout autre entier est interprété comme "vrai".
- Commentaires : `/* blabla */`.

## Conversion (2) boucle for et constantes

```
#include <stdio.h>

#define LOWER 0
#define UPPER 300
#define STEP 20

int main(){
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr += STEP)
        printf ("%3d\_%6.1f\n",fahr,(5.0/9.0) * (fahr-32.0));

    return 0;
}
```

# Compter le nombre de mots

```
#include <stdio.h>
#define IN 1 // dans un mot
#define OUT 0 // en dehors d'un mot.
int main(void) {
    char c; // caractère courant
    int inorout = OUT; // est on dedans ou en dehors d'un mot
    int words = 0; // nombre de mots

    while ((c = getchar()) != EOF) {
        if (((c >= 'a') && (c <= 'z')) || ((c >= 'A') && (c <= 'Z'))) {
            if (inorout == OUT) {
                inorout = IN;
                words++;
            }
        } else if ((c == ' ') || (c == '\n')) {
            inorout = OUT;
        } else {} // rien car on ignore le reste
    }
    printf("%d\n", words);
    return 0;
}
```

# Compter le nombre de caractères, mots et lignes (unix wc)

```
#include <stdio.h>
#define IN 1 // inside a word
#define OUT 0 // outside a word
int main(void) {
    int c, nl, nw, nc, state;
    state = OUT;
    nl = nw = nc = 0; /* affectations multiples */
    while ((c = getchar()) != EOF) {
        nc++;
        if (c == '\n')
            nl++;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT){
            state = IN; nw++;
        }
    } printf("%d_%d_%d\n", nl, nw, nc);
    return 0;}
```

# Structures de Contrôles

- Affectation : `x=expression`, ATTENTION ne pas confondre avec le test d'égalité `exp1 == exp2`.
- Séquence : enchaîner les expressions (';' est terminateur d'expression non séparateur).
- Bloc d'expressions : utiliser les '{' et '}' pour transformer plusieurs commandes en une commande.
- Choix :

```
if (expression) /* expression est évaluée a 0 ou non 0 */  
    commande_non_0;  
else  
    commande_0;
```

- Boucle 'tant que' :

```
while (expression)  
    commande_du_while
```

# while et dépendances

- Sémantique informelle du while :

```
while (e)
    c1
c2
```

est équivalent à

```
if (e){
    c1;
    while (e)
        c1;
}
c2;
```

- Le for est un while particulier :

```
for (c1;e;c2)
    c3;
```

est équivalent à

```
c1;
while (e){
    c3;
    c2;
}
```

# Fonctions

```
#include <stdio.h>

int power(int m, int n); /* déclaration de la fonction */

int main (){
    int i;
    for (i = 0 ; i < 10 ; i++)
        printf("%d_□%d_□\n",i,power(2,i)); /*utilisation de la fonction*/
    return 0;
}

int  power(int base, int n) /* implantation de la fonction */
{
    int i,p=1;
    for (i = 1;i <= n;i++)
        p = p*base;
    return p;
}
```

# Fonctions

- Syntaxe d'une définition de fonction:

```
return-type function-name(parameter declarations)
{  déclarations locales;
    commandes;}
```

- L'instruction de valeur retournée est `return e`, ou `e` est une expression du type attendu pour le retour de la fonction.
- Subtilités (à relire plus tard): les fonctions sont en appel par valeur. Les paramètres formels peuvent être utilisés comme variables locales...

# Programme mystère

```
#include<stdio.h>

int f(int y,int w,int z)
    {return y+w+z;}

int g(int y)
    {return y+3;}

int h(int y)
    {return 2*y;}

int main(){
    int x=4;

    printf("%d_\n",f(g(x),h(x),x));
    printf("%d_\n",f(g(++x),h(x),x));
    printf("%d_\n",f(x,g(++x),h(x)));
    printf("%d_\n",f(x,g(x++),h(x)));
    printf("%d_\n",f(x,g(x),h(x++)));
}
```

# Utiliser le debugger gdb

- gdb est un utilitaire qui permet d'exécuter un programme petit à petit en inspectant la valeur des variables. Cela permet de comprendre d'où proviennent les erreurs quand un programme s'arrête sur un Segmentation Fault ou autre.
- Il faut ajouter l'option `-g` : `gcc -Wall -g prog.c -o prog`
- Le lancement de gdb se fait par : `gdb prog`
- Il existe toute une liste de commande permettant d'exécuter le code "à la demande" et d'imprimer le contenu des variables et diverses informations sur l'exécution en cours:  
`break`, `run`, `print`, `watch`, `clear`, `delete`...
- Manuel en ligne (ou IA) :  
<https://www.gnu.org/software/gdb/documentation/>

# Plan

- 1 Introduction
- 2 Concepts de base du langage C
- 3 C - mémoire, tableaux, pointeurs et dépendances**
- 4 Intermède : process en cours d'exécution
- 5 Structuration des données et des programmes
  - Structuration des données
  - Structuration des programmes
- 6 Aspects systèmes

# La chose et le nom de la chose

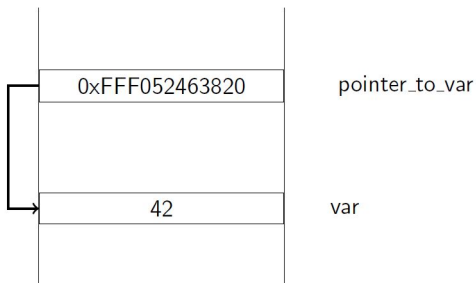
- Toute information est enregistrée dans la mémoire de l'ordinateur à une certaine adresse.  
⇒ `int x=4`: définit une variable `x` qui a une valeur 4 et qui est enregistrée dans la cellule mémoire `0xFE34H`. . .
- Les adresses sont des entiers... sur lesquels on peut faire des calculs.
- Une variable qui représente une adresse est appelée un *pointeur*.
- La manipulation de la mémoire en C est de très bas niveau. Il faut tout faire soi même :
  - Avantages : gestion très précise.
  - Inconvénients : très compliqué, possibilité d'erreurs, aliasing  
Segmentation fault
- Les pointeurs sont utilisés pour :
  - Définir les tableaux.
  - Permettre le passage par adresse.
- Référencement, déréférencement et arithmétique des pointeurs.

# Référencement

- Une variable : `int var = 42;`
- Opérateur de récupération d'adresse : `&`
  - On *référence* cette variable. `& var` est l'adresse où est enregistrée `var`.
  - `&` est un opérateur (unaire)
- Opérateur de récupération du contenu de l'adresse : `*`
  - On *déréférence* cette variable. Si `p` est un pointeur d'entier alors `*p` est le contenu de l'adresse `p`.
  - `*` est un opérateur (unaire)
- Des *types* pour les adresses : les *pointeurs* de type `t`.
  - Exemple: `int* pointer_to_var = & var;`
  - Dans l'exemple ci-dessus, le type de `pointer_to_var` est `int*`.

# Déréférencement - 1

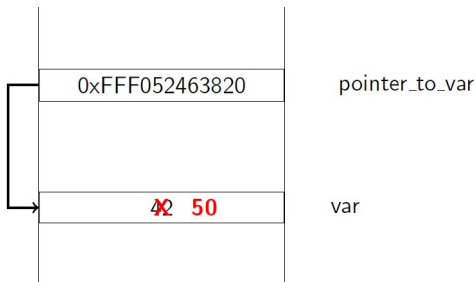
```
int var = 42;  
int* pointer_to_var = & var;
```



## Déréférencement - 2

- Déréférencement (récupérer la valeur à l'adresse) avec l'opérateur \*:

```
int var = 42;  
int* pointer_to_var = & var;  
  
*pointer_to_var = 50;
```



```
#include<stdio.h>
void main()
{
    int entier=42;
    int* adresse_entier;
    long int pourvoir;

    adresse_entier= & entier;
    pourvoir = (long int) adresse_entier;
    printf("adresse_entier_␣pointeur_␣%p\n",adresse_entier);
    *adresse_entier=43;
    printf("alias_␣entier_␣%d\n",entier);
    printf("adresse_entier_␣%li\n", (long int) adresse_entier);
    printf("pourvoir_␣%li\n\n\n",pourvoir);
    pourvoir++;
    adresse_entier++;
    printf("adresse_entier_␣2_␣%li\n", (long int) adresse_entier);
    printf("pourvoir_␣2_␣%li\n",pourvoir);
}
```

# Tableaux

- La déclaration d'un tableau `int tab[10]` fait deux choses :
  - Déclare une variable `tab` de type pointeur (d'entier ici) vers la première case du tableau.
  - Réserve en mémoire la place pour 10 int.
- Dans le programme `t[x]` désigne la  $x + 1$ ème case du tableau. En fait `t[x]` est exactement équivalent à `*(t+x)`.  
⇒ Arithmétique des pointeurs !!  
C'est le point le plus délicat (de loin) du C.

# Tableaux : exemple de programme

```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
#define SIZE 100
int main(){
    bool prem[SIZE]; /* nombres premiers plus petits que SIZE */
    int i,dernier_trouve=2;

    for(i=0;i<SIZE;i++) prem[i]=true;
    while (dernier_trouve <= (int) sqrt(SIZE))
        { if (prem[dernier_trouve]){
            for (i=2*dernier_trouve;i<SIZE;i=i+dernier_trouve)
                if (i<SIZE) prem[i]=false;i
            dernier_trouve=dernier_trouve+1;}

    for(i=2;i<SIZE;i++) /* le plus petit nombre premier est 2 */
        if (prem[i]) printf("%d,\u00a0",i);

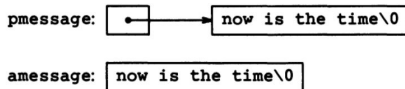
    return 0;}
```

## Cas particulier des tableaux de caractères

- Le type "string" en C c'est implanté par un tableaux de char (ou une variable de type char \*) qui a une structure particulière.
- L'expression "I am a string" est représentée en interne comme un tableau de caractères se terminant par le char '\0' (taille!!).
- char phrase[]="test" est équivalent à  
char phrase[] = {'t','e','s','t','\0'}.
- Subtilité dans les définitions :

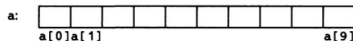
```
char amessage[] = "now is the time";
char *pmessage = "now is the time";
```

ont une interprétation différente : amessage n'est pas une variable (c'est le nom d'un tableau) mais pmessage est une variable.

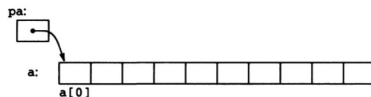


# Arithmétique des pointeurs types et tableaux

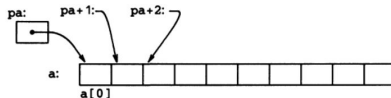
- La déclaration `int a[10];` produit :



- La déclaration `int *pa; pa = a;` produit :



- Le contenu de la case `a[1]` est le même que `*(pa+1)`.



- la signification de `a+x` avec `x` une expression de type entier, est le pointeur vers `a` plus `x` cases de la taille du type de `a`.

# Allocation dynamique de mémoire

- Il est possible de réserver de manière dynamique de la mémoire : `malloc`. Il faut lui donner la taille en octet à libérer pour laquelle on peut utiliser `sizeof` qui prend en entrée un type (ou une expression qu'il transforme en type du résultat de l'expression).

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int* p;
    int i,z=12;

    p=malloc(sizeof(int)*z); /* taille en octet de mémoire libérée */
    for(i=0;i<z;i++) p[i]=2*i+1;
    for(i=0;i<z;i++) printf("P[%d]=%d,\t",i,p[i]);

    return 0;}
```

- On peut libérer la mémoire grace à `free`.

# Passage par adresse - 1

- En C ce sont les valeurs qui sont passées aux fonctions.

```
#include<stdio.h>

int test(int x){
    x=x+1; return x;
}

int main(){
    int avant,apres;
    avant=10; apres=test(avant);
    printf("%d et %d\n",avant, apres);

    return 0
}
```

## Passage par adresse - 2

- Pour modifier les valeurs il faut les passer par adresse.

```
#include<stdio.h>

int test(int* x){
    *x=*x+1; return *x;
}

int main(){
    int avant,apres;
    avant=10; apres=test(&avant);
    printf("%d\et %d\n",avant, apres);

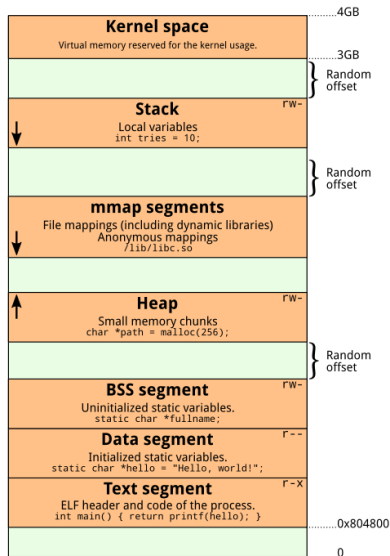
    return 0;}
```

- Exemple du quicksort.

# Plan

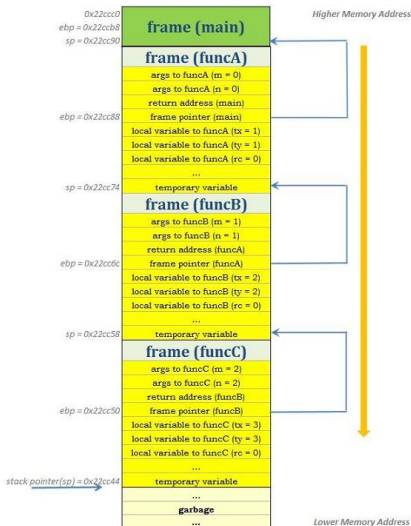
- 1 Introduction
- 2 Concepts de base du langage C
- 3 C - mémoire, tableaux, pointeurs et dépendances
- 4 Intermède : process en cours d'exécution**
- 5 Structuration des données et des programmes
  - Structuration des données
  - Structuration des programmes
- 6 Aspects systèmes

# Execution d'un programme en mémoire



# Pile d'exécution

## Stack frames a.k.a Activation records



```

Perry: ~/stack_frames.c
1 int funcA(int, int);
2 int funcB(int, int);
3 int funcC(int, int);
4
5 int main() {
6     int tx = 0;
7     int ty = 0;
8     int rc = funcA(tx, ty);
9     return rc;
10 }
11
12 int funcC(int n, int n) {
13     int tx = 1;
14     int ty = 1;
15     int rc = 0;
16     rc = tx + ty;
17     return rc;
18 }
19
20 int funcB(int n, int n) {
21     int tx = 2;
22     int ty = 2;
23     int rc = 0;
24     funcC(tx, ty);
25     rc = tx + ty;
26     return rc;
27 }
28
29 int funcA(int n, int n) {
30     int tx = 1;
31     int ty = 1;
32     int rc = 0;
33     funcB(tx, ty);
34     rc = tx + ty;
35     return rc;
36 }

```

# Implication

Différence fondamentale entre les deux fonctions suivantes (du fait de l'utilisation de variable locale).

```
int *f_oublie(){
    int t[10],i;
    for (i=0;i<10;i++)t[i]=i;
    return t;
}
```

```
int *f_souvient(){
    int *t,i;
    t=malloc(10*sizeof(int));
    for (i=0;i<10;i++)t[i]=i;
    return t;
}
```

# Plan

- 1 Introduction
- 2 Concepts de base du langage C
- 3 C - mémoire, tableaux, pointeurs et dépendances
- 4 Intermède : process en cours d'exécution
- 5 Structuration des données et des programmes**
  - Structuration des données
  - Structuration des programmes
- 6 Aspects systèmes

# Plan

- 1 Introduction
- 2 Concepts de base du langage C
- 3 C - mémoire, tableaux, pointeurs et dépendances
- 4 Intermède : process en cours d'exécution
- 5 Structuration des données et des programmes
  - Structuration des données
  - Structuration des programmes
- 6 Aspects systèmes

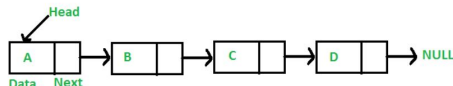
# Structuration des données

- Les types de base insuffisants pour organiser des données complexes.  
typedef permet de renommer un type existant exemple :  
typedef int age;
- Pour combiner les informations on peut combiner des types existants (comme le produit cartésien en mathématiques) au moyen de struct.

```
#include<stdio.h>
int main(){
    struct point{ /* déclare un type structure point de deux */
        int x;    /* coordonnées une x et une y */
        int y;
    };
    struct point pt;
    pt.x = 1 ;    /* accès aux champs du struct par '.' */
    pt.y = 2 ;
    printf("coordonées de pt\n");
    printf("\t-abscisse: %d\n",pt.x); printf("\t-ordonnée: %d",pt.y);
    return 0;}
```

# Structs récursifs

- Listes chaînées. Graphiquement :



- En C :

```
struct Node {
    int data;
    struct Node* next;}

struct Node* Head=NULL,inter;

inter = malloc (sizeof(struct Node));
inter -> data = A; inter -> next = Head; Head = inter;
inter = malloc (sizeof(struct Node));
inter -> data = B; inter -> next = Head; Head = inter;
...
```

# Plan

- 1 Introduction
- 2 Concepts de base du langage C
- 3 C - mémoire, tableaux, pointeurs et dépendances
- 4 Intermède : process en cours d'exécution
- 5 Structuration des données et des programmes
  - Structuration des données
  - Structuration des programmes
- 6 Aspects systèmes

# Structuration des programmes

- Les programmes réalistes sont trop importants pour tenir dans un seul fichier.
- Comment réutiliser des morceaux de programmes pour ne pas tout réécrire ?  
⇒ COMPILE ET LIER.
- On peut séparer les sources dans plusieurs fichiers, les compiler de manière symbolique.  
Construction de bibliothèque de programmes (sans `main()`).
- Notion de fichier interface (fichier `.h`) pour déclarer les fonctions (ancien équivalent des API).

# Structuration des programmes

- Deux fichiers '.c' : `exemple.c`, `somme.c`.
- Un fichier d'en-tête (header) '.h' : `somme.h`.
  - Précompilation des deux fichiers séparément (dans n'importe quel ordre):

```
gcc -c exemple.c
gcc -c somme.c
```

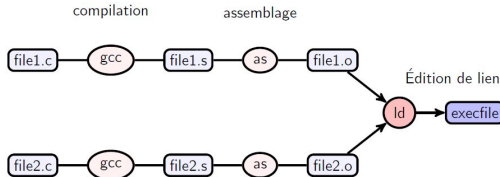
 $\Rightarrow$  création de `exemple.o` et `somme.o`
  - édition de liens pour terminer la compilation :

```
gcc -o exemple somme.o exemple.o
```

 $\Rightarrow$  création de l'exécutable `exemple`
- On peut modifier `somme.c` le recompiler sans toucher à `exemple.c`.

# Structuration des programmes

- Schématiquement :



- Notion primitive d'API : on a un fichier par fonctionnalité, qui est connu à travers son interface. La manière dont le fichier est écrit est abstraite vis-à-vis du programme principal.
  - ⇒ Que mettre dans un fichier '.h' ?
- Le problème est que dans de vrais projets il peut y avoir plusieurs niveaux d'intégrations.
  - ⇒ Makefile et la commande `make`

# Que mettre dans un fichier '.h' ?

- Chaque fichier '.c' doit inclure `#include "<file>.h"` pour pouvoir utiliser les fonctions déclarées dans `<file>.h` (et définies dans `<file>.c`).
- Similaire à l'utilisation de bibliothèques standard...  
`#include <stdio.h>.`
- Contenu des '.h' :
  - 1 Chaque module ('.h' et '.c' correspondants) implante une fonctionnalité claire.
  - 2 Utiliser des gardes `#ifndef`, `#define`, `#endif`.
  - 3 Toutes les déclarations d'un module doivent être dans le '.h' et l'utilisation de ce module doit passer par son `#include`.
- Tutorial en ligne sur les fichiers header:  
[https://www.tutorialspoint.com/cprogramming/c\\_header\\_files.htm](https://www.tutorialspoint.com/cprogramming/c_header_files.htm)

# Dépendances des fichiers et Makefile

- Pour éviter de tout refaire à la main à chaque fois : Makefile.
- Le fichier Makefile (UTILISEZ DES TABULATIONS) :

```
exemple: somme.o exemple.o
    gcc -o exemple exemple.o somme.o

somme.o: somme.c
    gcc -o somme.o -c somme.c

exemple.o: exemple.c
    gcc -o exemple.o -c exemple.c
```

- forme du Makefile :

```
cible: dépendences
    commandes
```

- utilisation par make cible

# Plan

- 1 Introduction
- 2 Concepts de base du langage C
- 3 C - mémoire, tableaux, pointeurs et dépendances
- 4 Intermède : process en cours d'exécution
- 5 Structuration des données et des programmes
  - Structuration des données
  - Structuration des programmes
- 6 Aspects systèmes

# Aspects systèmes

- Le C est très proche de la machine : différences concernant un même programme sur différents systèmes.
- Un programme est généralement exécuté dans un contexte : le système d'exploitation qui permet à plusieurs process de s'exécuter simultanément.
- Unix system interface : permet d'accéder aux fichiers et commandes du système.
- Il suffit de savoir manipuler les fichiers : produire un son correspond en fait à l'écriture d'un fichier dans `/dev/audio` en Unix.

# Les arguments de la ligne de commandes

- Un exécutable prog peut avoir des options. Par exemple :  
`prog -l -s -f fichier.txt`
- L'accès à ces paramètres se fait par l'ajout de paramètres à la fonction main
  - `int main(int argc, char* argv[])`
  - `argc` est le nombre d'arguments.
  - `argv` est un tableau de `argc` strings.
  - `argv[0]` est le nom du program "prog".

# Les arguments de la ligne de commandes - exemple

- Programme qui imprime la liste de ses arguments dans l'ordre.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int index = 0;

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

# Accès aux fichiers

- La manière la plus simple : `putchar`, `getchar` et l'utilisation des opérateurs unix `<` et `>`  
`monprog < fichier_in > fichier_out`  
fonctionne également pour les fonctions d'impression formatées `printf`, `scanf`.
- Ca reste limité : que faire quand il y a plusieurs fichiers ? Quand on crée le nom du fichier de sortie durant l'exécution du programme etc. ?

# Les flots

- Diverses fonctions de la librairie C permettent d'accéder aux fichiers.
- Ces fonctions d'accès aux fichiers ont besoin de diverses informations :
  - le mode d'accès (lecture seule, écriture seule...) ;
  - la position courante de la "tête de lecture" virtuelle ;
  - l'adresse d'une mémoire-tampon permettant de ne pas accéder systématiquement au matériel, ce qui serait inefficace ;
  - ...
- Ces informations sont rassemblées dans une structure `FILE`
- Cette structure est définie dans `stdio.h`
- Un objet de type `FILE *` est appelé *flot de données* (en anglais, *stream*).

# Manipulation de fichiers (1)

- Demander l'initialisation d'une structure de données FILE : fonction `fopen()`.

```
FILE *fopen(const char *pathname, const char *mode);
```

mode :

- "r" (read) lecture seule (s'il existe)
- "w" (write) : le fichier est écrasé s'il existe, créé sinon
- "a" (append) : écriture à la fin d'un fichier existant
- Ce flot sera ensuite utilisé lors de l'écriture ou de la lecture (via diverses fonctions)
- Après les manipulation sur un fichier, on annule la liaison entre le fichier et le flot de données grâce à la fonction `int fclose(FILE *fichier);`

# Manipulation de fichiers (2)

- Pour utiliser toutes les fonctions d'entrées-sorties, vous devez inclure `<stdio.h>`
- Pour connaître toutes les fonctions de cette librairie stdio :  
`man stdio`
- Pour de l'aide sur une fonction : `man fopen`.
- Les accès à l'entrée standard peuvent se faire de la même manière qu'à un fichier, via le *stream* `stdin`
- Idem pour la sortie avec `stdout`.
- La taille minimale d'un accès est l'octet (8 bits).

# Lecture de fichier

- Plusieurs fonctions permettent de lire dans un fichier. 3 exemples :

```
int fgetc(FILE *stream);

char *fgets(char *s, int size, FILE *stream);

size_t fread(void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

# Écrire dans un fichier

- Plusieurs fonctions permettent d'écrire dans un fichier. 3 exemples :

```
int fputc(int c, FILE *stream);

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

size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

# Position dans un flot

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);

long ftell(FILE *stream);

void rewind(FILE *stream);

int fgetpos(FILE *stream, fpos_t *pos);

int fsetpos(FILE *stream, const fpos_t *pos);
```