Programmation C

J.-F. Lalande

15 novembre 2012

Résumé

Ce cours tente de couvrir tous les aspects liés au développement en langage C. Le cours débute par l'étude du langage C en lui même et se poursuit par l'étude des aspects spécifiques des tableaux et pointeurs. Ensuite, l'introduction des structures combinée à la notion de pointeurs permet d'étudier l'implémentation de structures de données complexes. Le cours termine par quelques éléments périphériques (compilateur, ncurses, ...).

Livre de référence

Disponible à la bibliothèque :

– Langage C, Claude DELANNOY, Eyrolles, 2005.

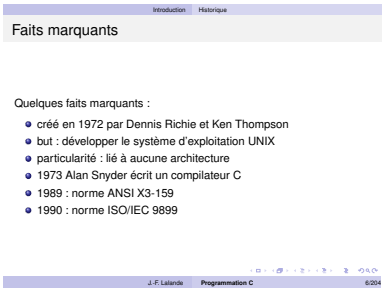
Table des matières

1 Introduction

1.1 Historique

Faits marquants

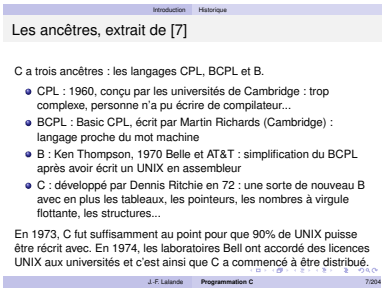
Quelques faits marquants :
- créé en 1972 par Dennis Richie et Ken Thompson
- but : développer le système d'exploitation UNIX
- particularité : lié à aucune architecture
- 1973 Alan Snyder écrit un compilateur C
- 1989 : norme ANSI X3-159
- 1990 : norme ISO/IEC 9899



Les ancêtres, extrait de [?]

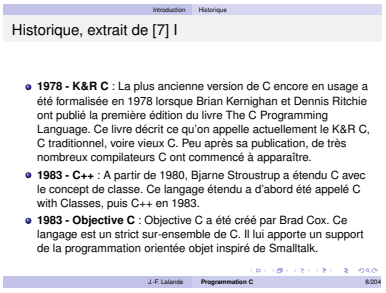
C a trois ancêtres : les langages CPL, BCPL et B.
- CPL : 1960, conçu par les universités de Cambridge : trop complexe, personne n'a pu écrire de compilateur...
- BCPL : Basic CPL, écrit par Martin Richards (Cambridge) : langage proche du mot machine
- B : Ken Thompson, 1970 Belle et AT&T : simplification du BCPL après avoir écrit un UNIX en assembleur
- C : développé par Dennis Ritchie en 72 : une sorte de nouveau B avec en plus les tableaux, les pointeurs, les nombres à virgule flottante, les structures...

En 1973, C fut suffisamment au point pour que 90% de UNIX puisse être récrit avec. En 1974, les laboratoires Bell ont accordé des licences UNIX aux universités et c'est ainsi que C a commencé à être distribué.



Historique, extrait de [?]

- **1978 - K&R C** : La plus ancienne version de C encore en usage a été formalisée en 1978 lorsque Brian Kernighan et Dennis Ritchie ont publié la première édition du livre The C Programming Language. Ce livre décrit ce qu'on appelle actuellement le K&R C, C traditionnel, voire vieux C. Peu après sa publication, de très nombreux compilateurs C ont commencé à apparaître.
- **1983 - C++** : A partir de 1980, Bjarne Stroustrup a étendu C avec le concept de classe. Ce langage étendu a d'abord été appelé C with Classes, puis C++ en 1983.
- **1983 - Objective C** : Objective C a été créé par Brad Cox. Ce langage est un strict sur-ensemble de C. Il lui apporte un support de la programmation orientée objet inspiré de Smalltalk.
- **1989 - ANSI C** : Un comité de standardisation a été créé en 1983 pour éviter que les quelques ambiguïtés et insuffisances du K&R C ne conduisent à des divergences importantes entre les compilateurs. Il a publié en 1989 le standard appelé ANSI C.
- **1998 - Standard C++** : C++ a évolué très longtemps. Ce n'est qu'en 1998, 8 ans après la création d'un comité, que le standard ISO C++ a été officiellement publié. Ce standard est tellement complexe (et légèrement incohérent), qu'en 2003, le compilateur GCC ne le met pas complètement en oeuvre, et ce n'est pas le seul.
- **1999 - C99** : Le dernier né de l'histoire est C99 (standard ISO de 1999) qui est une petite évolution de l'ANSI C de 1989. Les évolutions ne sont pas compatibles avec C++ et n'ont pas attiré beaucoup d'intérêt.



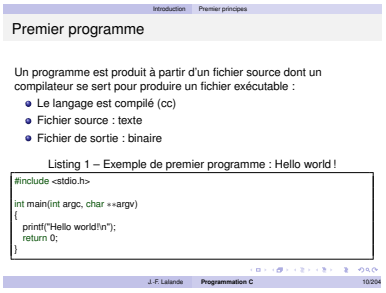
1.2 Premier principes

Premier programme

Un programme est produit à partir d'un fichier source dont un compilateur se sert pour produire un fichier exécutable :
- Le langage est compilé (cc)
- Fichier source : texte
- Fichier de sortie : binaire

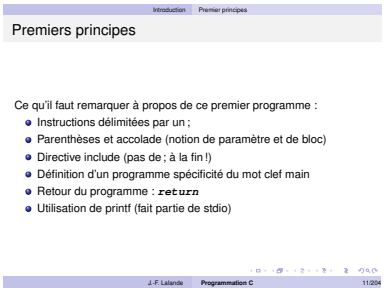
Listing 1 – Exemple de premier programme : Hello world !

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  printf("Hello world!\n");
  return 0;
}
```



Premiers principes

Ce qu'il faut remarquer à propos de ce premier programme :
- Instructions délimitées par un ;
- Parenthèses et accolade (notion de paramètre et de bloc)
- Directive include (pas de ; à la fin !)
- Définition d'un programme spécificité du mot clef main
- Retour du programme : ***return***
- Utilisation de printf (fait partie de stdio)

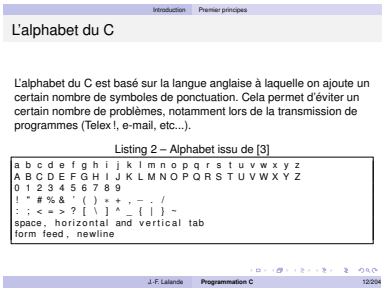


L'alphabet du C

L'alphabet du C est basé sur la langue anglaise à laquelle on ajoute un certain nombre de symboles de ponctuation. Cela permet d'éviter un certain nombre de problèmes, notamment lors de la transmission de programmes (Telex !, e-mail, etc...).

Listing 2 – Alphabet issu de [**?**]

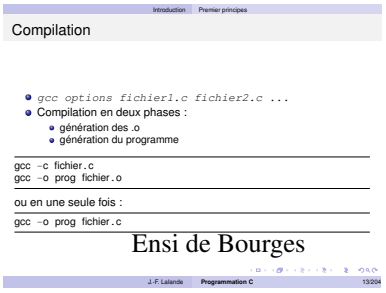
```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , – . /
: ; < = > ? [ \ ] ^ _ { | } ~
space, horizontal and vertical tab
form feed, newline
```



Compilation

- *gcc options fichier1.c fichier2.c ...*
- Compilation en deux phases :
  - génération des .o
  - génération du programme

```
gcc -c fichier.c
gcc -o prog fichier.o
```



ou en une seule fois :

```
gcc -o prog fichier.c
```

Compilation (2)

Attention, les espaces importent !

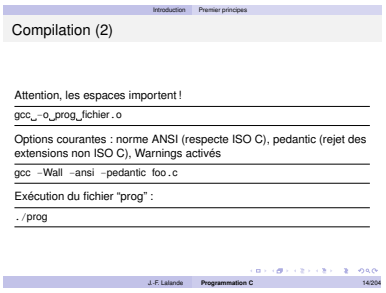
```
gcc␣-o␣prog␣fichier.o
```

Options courantes : norme ANSI (respecte ISO C), pedantic (rejet des extensions non ISO C), Warnings activés

```
gcc -Wall -ansi -pedantic foo.c
```

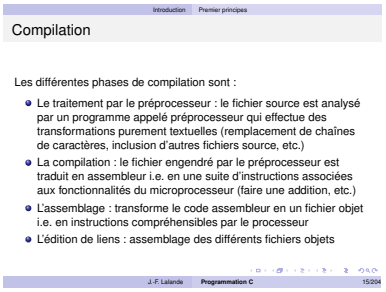
Exécution du fichier “prog” :

```
./prog
```



Compilation

Les différentes phases de compilation sont :
- Le traitement par le préprocesseur : le fichier source est analysé par un programme appelé préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d’autres fichiers source, etc.)
- 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



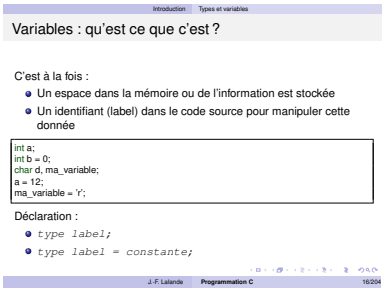
1.3 Types et variables

Variables : qu’est ce que c’est ?

C’est à la fois :
- Un espace dans la mémoire ou de l’information est stockée
- Un identifiant (label) dans le code source pour manipuler cette donnée

```
1 int a;
2 int b = 0;
3 char d, ma_variable;
4 a = 12;
5 ma_variable = 'r';
```

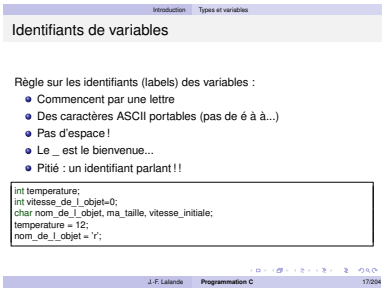
Déclaration :
- *type label;*
- *type label = constante;*



Identifiants de variables

Règle sur les identifiants (labels) des variables :
- Commencent par une lettre
- Des caractères ASCII portables (pas de é à à...)
- Pas d'espace !
- Le _ est le bienvenue...
- Pitié : un identifiant parlant ! !

```
int temperature;
int vitesse_de_l_objet=0;
char nom_de_l_objet, ma_taille, vitesse_initiale;
temperature = 12;
nom_de_l_objet = 'r';
```



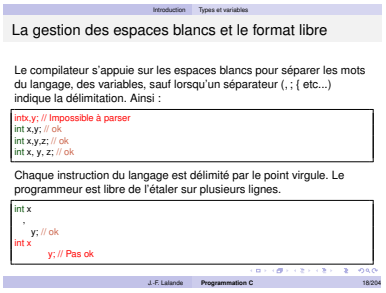
La gestion des espaces blancs et le format libre

Le compilateur s'appuie sur les espaces blancs pour séparer les mots du langage, des variables, sauf lorsqu'un séparateur (, ; { etc...) indique la délimitation. Ainsi :

```
intx,y; // Impossible à parser
int x,y; // ok
int x,y,z; // ok
int x, y, z; // ok
```

Chaque instruction du langage est délimité par le point virgule. Le programmeur est libre de l'étaler sur plusieurs lignes.

```
int x
  ,
    y; // ok
int x
        y; // Pas ok
```



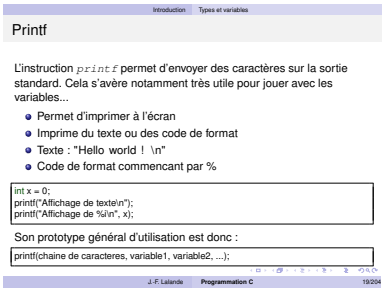
Printf

L'instruction `printf` permet d'envoyer des caractères sur la sortie standard. Cela s'avère notamment très utile pour jouer avec les variables...
- Permet d'imprimer à l'écran
- Imprime du texte ou des code de format
- Texte : "Hello world ! \n"
- Code de format commencant par %

```
int x = 0;
printf("Affichage de texte\n");
printf("Affichage de %i\n", x);
```

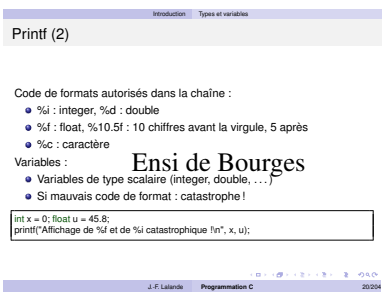
Son prototype général d'utilisation est donc :

```
printf(chaine de caracteres, variable1, variable2, ...);
```



Printf (2)

Code de formats autorisés dans la chaîne :
- %i : integer, %d : double
- %f : float, %10.5f : 10 chiffres avant la virgule, 5 après
- %c : caractère



Variables :
- Variables de type scalaire (integer, double, . . .)
- Si mauvais code de format : catastrophe !

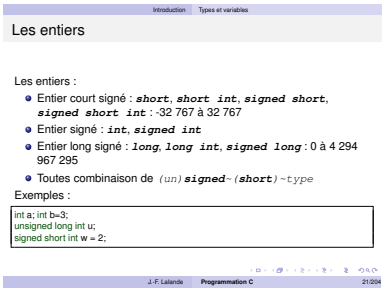
```
int x = 0; float u = 45.8;
printf("Affichage de %f et de %i catastrophique !\n", x, u);
```

Les entiers

Les entiers :
- Entier court signé : **`short`**, **`short int`**, **`signed short`**, **`signed short int`** : -32 767 à 32 767
- Entier signé : **`int`**, **`signed int`**
- Entier long signé : **`long`**, **`long int`**, **`signed long`** : 0 à 4 294 967 295
- Toutes combinaison de `(un)signed~(short)~type`

Exemples :

```
int a; int b=3;
unsigned long int u;
signed short int w = 2;
```

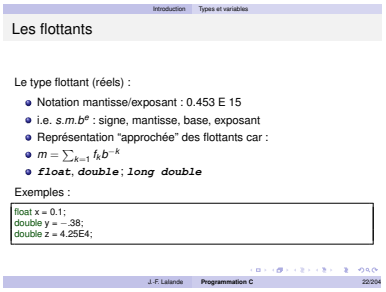


Les flottants

Le type flottant (réels) :
- Notation mantisse/exposant : 0.453 E 15
- i.e. $s.m.b^e$: signe, mantisse, base, exposant
- Représentation “approchée” des flottants car :
- $m = \sum_{k=1} f_k b^{-k}$
- **`float`**, **`double`**; **`long double`**

Exemples :

```
float x = 0.1;
double y = -.38;
double z = 4.25E4;
```



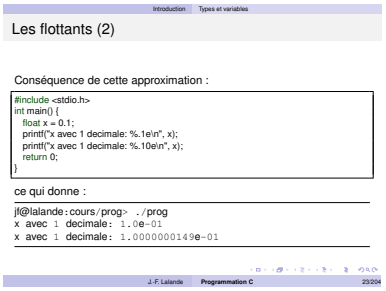
Les flottants (2)

Conséquence de cette approximation :

```
#include <stdio.h>
int main() {
  float x = 0.1;
  printf("x avec 1 decimale: %.1e\n", x);
  printf("x avec 1 decimale: %.10e\n", x);
  return 0;
}
```

ce qui donne :

```
jf@lalande:cours/prog> ./prog
x avec 1 decimale: 1.0e-01
x avec 1 decimale: 1.00000001490e-01
```



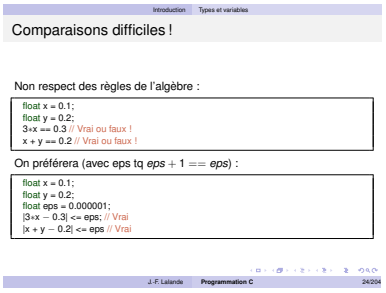
Comparaisons difficiles !

Non respect des règles de l'algèbre :

```
float x = 0.1;
float y = 0.2;
3*x == 0.3 // Vrai ou faux !
x + y == 0.2 // Vrai ou faux !
```

On préférera (avec eps tq $eps + 1 == eps$) :

```
float x = 0.1;
float y = 0.2;
float eps = 0.000001;
|3*x - 0.3| <= eps; // Vrai
|x + y - 0.2| <= eps // Vrai
```



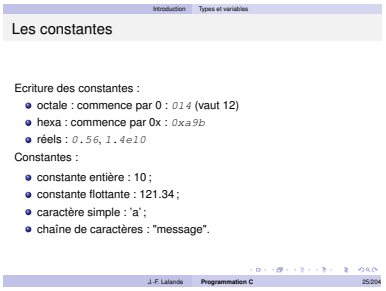
Les constantes

Ecriture des constantes :
- octale : commence par 0 : `014` (vaut 12)
- hexa : commence par 0x : `0xa9b`
- réels : `0.56`, `1.4e10`

Constantes :
- constante entière : 10 ;
- constante flottante : 121.34 ;
- caractère simple : 'a' ;
- chaîne de caractères : "message".

Exercice 1 (tiré de [?]) Définir les variables :
- i entier
- f flottant
- l long
- c caractère
- tc tableau de caractères.

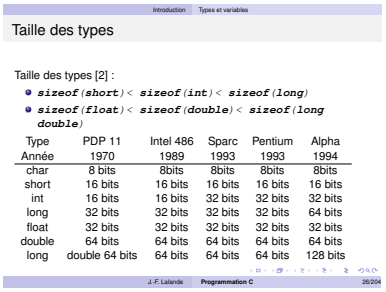
en les initialisant de la manière suivante :
- i à la valeur hexadécimale 50 ;
- f à 3.14 ;
- l à la valeur octale 40 ;
- c à z ;
- tc à qwertyuiop.



Taille des types

Taille des types [?] :
- **`sizeof(short)` < `sizeof(int)` < `sizeof(long)`**
- **`sizeof(float)` < `sizeof(double)` < `sizeof(long double)`**

Type	PDP 11	Intel 486	Sparc	Pentium	Alpha
Année	1970	1989	1993	1993	1994
char	8 bits	8bits	8bits	8bits	8bits
short	16 bits	16 bits	16 bits	16 bits	16 bits
int	16 bits	16 bits	32 bits	32 bits	32 bits
long	32 bits	32 bits	32 bits	32 bits	64 bits
float	32 bits	32 bits	32 bits	32 bits	32 bits
double	64 bits	64 bits	64 bits	64 bits	64 bits
long	double 64 bits	64 bits	64 bits	64 bits	128 bits



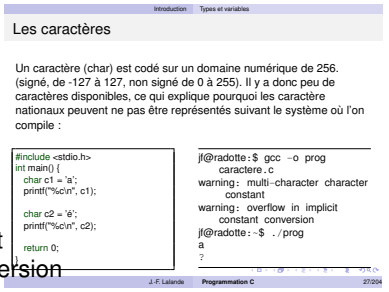
Exercice 2 A l'aide d'un printf et de l'opérateur sizeof, imprimez les tailles des différents types de base ci-dessus.

Les caractères

Un caractère (char) est codé sur un domaine numérique de 256. (signé, de -127 à 127, non signé de 0 à 255). Il y a donc peu de caractères disponibles, ce qui explique pourquoi les caractère nationaux peuvent ne pas être représentés suivant le système où l'on compile :

```c
#include <stdio.h>
int main() {
  char c1 = 'a';
  printf("%c\n", c1);

  char c2 = 'é';
  printf("%c\n", c2);

  return 0;
}
```

```
jf@radotte:$ gcc -o prog caractere.c
warning: multi-character character constant
warning: overflow in implicit constant conversion
jf@radotte:~$ ./prog
a
?
```



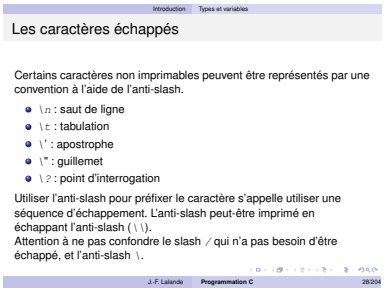
Les caractères échappés

Certains caractères non imprimables peuvent être représentés par une convention à l'aide de l'anti-slash.

- \n : saut de ligne
- \t : tabulation
- \' : apostrophe
- \" : guillemet
- \? : point d'interrogation

Utiliser l'anti-slash pour préfixer le caractère s'appelle utiliser une séquence d'échappement. L'anti-slash peut-être imprimé en échappant l'anti-slash (\ \).

Attention à ne pas confondre le slash / qui n'a pas besoin d'être échappé, et l'anti-slash \.

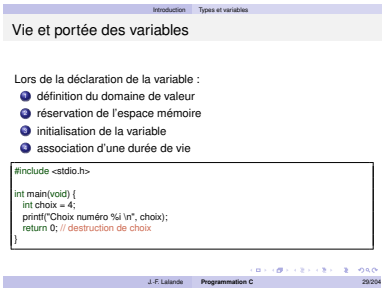


Vie et portée des variables

Lors de la déclaration de la variable :

1. définition du domaine de valeur
2. réservation de l'espace mémoire
3. initialisation de la variable
4. association d'une durée de vie

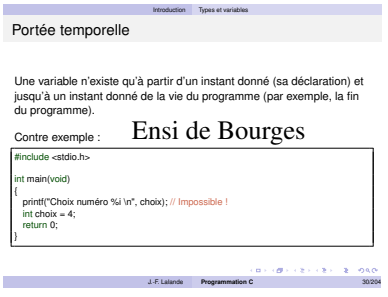
```c
#include <stdio.h>

int main(void) {
  int choix = 4;
  printf("Choix numéro %i \n", choix);
  return 0; // destruction de choix
}
```



Portée temporelle

Une variable n'existe qu'à partir d'un instant donné (sa déclaration) et jusqu'à un instant donné de la vie du programme (par exemple, la fin du programme).

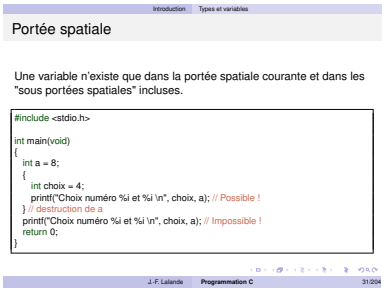
Contre exemple :



```c
#include <stdio.h>

int main(void)
{
  printf("Choix numéro %i \n", choix); // Impossible !
  int choix = 4;
  return 0;
}
```

Portée spatiale

Une variable n'existe que dans la portée spatiale courante et dans les "sous portées spatiales" incluses.

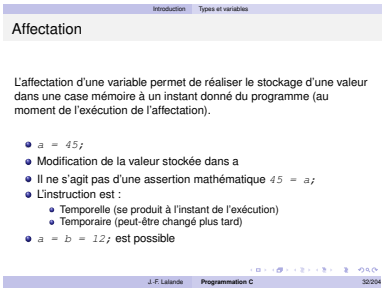



```c
#include <stdio.h>

int main(void)
{
  int a = 8;
  {
    int choix = 4;
    printf("Choix numéro %i et %i \n", choix, a); // Possible !
  } // destruction de a
  printf("Choix numéro %i et %i \n", choix, a); // Impossible !
  return 0;
}
```

Exercice 3 A tester soi-même chez soi. Quel affichage ce programme produit-il ?

Affectation

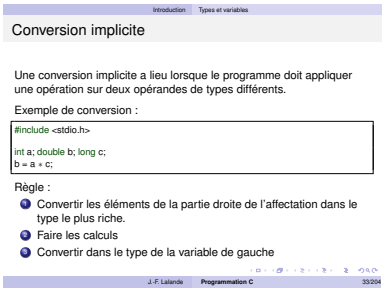
L'affectation d'une variable permet de réaliser le stockage d'une valeur dans une case mémoire à un instant donné du programme (au moment de l'exécution de l'affectation).



- `a = 45;`
- Modification de la valeur stockée dans a
- Il ne s'agit pas d'une assertion mathématique `45 = a;`
- L'instruction est :
  - Temporelle (se produit à l'instant de l'exécution)
  - Temporaire (peut-être changé plus tard)
- `a = b = 12;` est possible

Conversion implicite

Une conversion implicite a lieu lorsque le programme doit appliquer une opération sur deux opérandes de types différents.


Introduction Types et variables
Conversion implicite
Une conversion implicite a lieu lorsque le programme doit appliquer une opération sur deux opérandes de types différents.
Exemple de conversion :
#include <stdio.h>
int a; double b; long c;
b = a * c;
Règle :
1. Convertir les éléments de la partie droite de l'affectation dans le type le plus riche.
2. Faire les calculs
3. Convertir dans le type de la variable de gauche
J.-F. Lalande Programmation C 33/204

Exemple de conversion :

```
#include <stdio.h>

int a; double b; long c;
b = a * c;
```

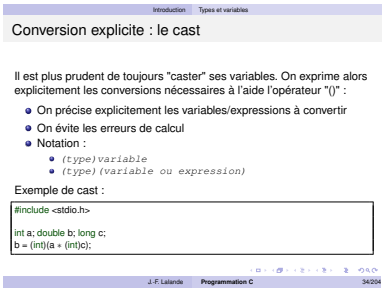
Règle :

1. Convertir les éléments de la partie droite de l'affectation dans le type le plus riche.
2. Faire les calculs
3. Convertir dans le type de la variable de gauche

Exercice 4 Testez la conversion implicite d'un float en int.

Conversion explicite : le cast

Il est plus prudent de toujours "caster" ses variables. On exprime alors explicitement les conversions nécessaires à l'aide l'opérateur "()" :
- On précise explicitement les variables/expressions à convertir
- On évite les erreurs de calcul
- Notation :
  - `(type)variable`
  - `(type)(variable ou expression)`


Introduction Types et variables
Conversion explicite : le cast
Il est plus prudent de toujours "caster" ses variables. On exprime alors explicitement les conversions nécessaires à l'aide l'opérateur "()" :
- On précise explicitement les variables/expressions à convertir
- On évite les erreurs de calcul
- Notation :
  - (type)variable
  - (type)(variable ou expression)

Exemple de cast :
#include <stdio.h>
int a; double b; long c;
b = (int)(a * (int)c);
J.-F. Lalande Programmation C 34/204

Exemple de cast :

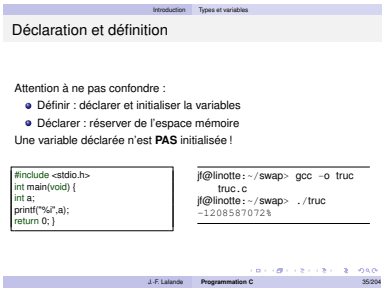
```
#include <stdio.h>

int a; double b; long c;
b = (int)(a * (int)c);
```

Exercice 5 Expliquer comment se comporte ce calcul.

Déclaration et définition

Attention à ne pas confondre :
- Définir : déclarer et initialiser la variables
- Déclarer : réserver de l'espace mémoire

Une variable déclarée n'est **PAS** initialisée !


Introduction Types et variables
Déclaration et définition
Attention à ne pas confondre :
- Définir : déclarer et initialiser la variables
- Déclarer : réserver de l'espace mémoire

Une variable déclarée n'est **PAS** initialisée !
#include <stdio.h>
int main(void) {
int a;
printf("%i",a);
return 0; }
jf@linotte:~/swap> gcc -o truc truc.c
jf@linotte:~/swap> ./truc
-1208587072%
J.-F. Lalande Programmation C 35/204

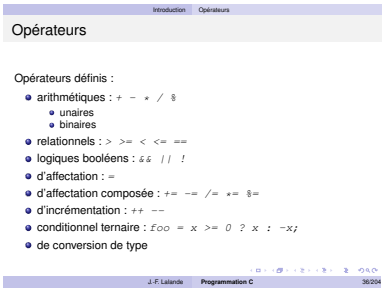
```
#include <stdio.h>
int main(void) {
int a;
printf("%i",a);
return 0; }
```

```
jf@linotte:~/swap> gcc -o truc truc.c
jf@linotte:~/swap> ./truc
-1208587072%
```

1.4 Opérateurs

Opérateurs

Opérateurs définis :
- arithmétiques : `+ - * / %`
  - unaires
  - binaires
- relationnels : `> >= < <= ==`
- logiques booléens : `&& || !`
- d'affectation : `=`
- d'affectation composée : `+= -= /= *= %=`
- d'incrémentation : `++ --`
- conditionnel ternaire : `foo = x >= 0 ? x : -x;`
- de conversion type



Exercice 6 Pourquoi `x *= y + 1` n'est pas équivalent à `x = x * y + 1` ?

Exercice 7 Écrire la ligne qui permet d'affecter 45 dans a et de stocker dans b le résultat de cette affectation.

Exercice 8 Écrire le programme qui : affecte 45 dans a et dans b, l'incrémente, puis compare a et b en enregistrant le résultat dans c.

Exercice 9 Écrire le programme qui initialise a et b, enregistre dans q le quotient de la division entière et dans r le reste.

Exercice 10 Écrire en C le test "a est plus grand que b ou c est égal à d et d est inférieur à e". Le placement de parenthèse peut il influencer le sens de cette proposition logique ?

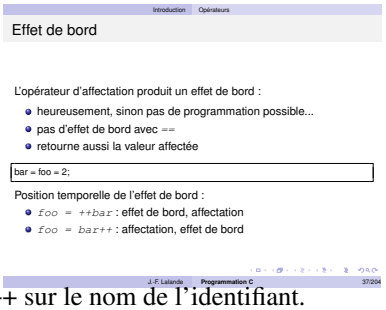
Exercice 11 Écrire le programme qui permet d'enregistrer dans res le fait que a est plus grand que b, puis à l'aide d'une condition ternaire de stocker x la valeur -1 si a est inférieur a b, sinon 12. Peut-on encore compresser ce programme écrit en deux étapes ?

Exercice 12 Écrire un programme qui permet de convertir un double en entier. Stocker dans `non_arrondie` la valeur VRAIE si cette variable n'a pas été arrondie.

Effet de bord

L'opérateur d'affectation produit un effet de bord :
- heureusement, sinon pas de programmation possible...
- pas d'effet de bord avec `==`
- retourne aussi la valeur affectée

```
bar = foo = 2;
```
1



Position temporelle de l'effet de bord :
- `foo = ++bar` : effet de bord, affectation
- `foo = bar++` : affectation, effet de bord

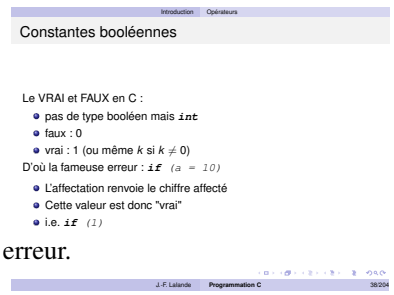
Exercice 13 Vérifiez dans un programme test l'influence de la position de l'opérateur ++ sur le nom de l'identifiant.

Exercice 14 Quand l'instruction conditionnelle **`if`** aura été vue, écrivez un test qui vérifie l'égalité entre une variable et une valeur. Insérez dans ce test un `++` et vérifiez l'influence de sa place. Que pensez-vous de cela ?

Constantes booléennes

Le VRAI et FAUX en C :
- pas de type booléen mais **`int`**
- faux : 0
- vrai : 1 (ou même k si $k \neq 0$)

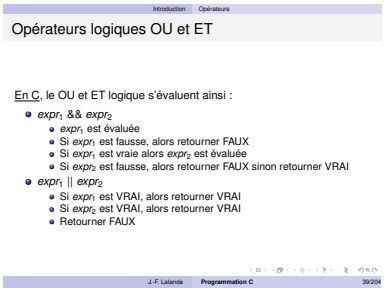
D'où la fameuse erreur : **`if`** `(a = 10)`
- L'affectation renvoie le chiffre affecté
- Cette valeur est donc "vrai"
- i.e. **`if`** `(1)`



Exercice 15 Promettez d'offrir un café à votre voisin à chaque fois où vous ferez cette erreur.

Opérateurs logiques OU et ET

En C, le OU et ET logique s'évaluent ainsi :
- $expr_1$ && $expr_2$
  - $expr_1$ est évaluée
  - Si $expr_1$ est fausse, alors retourner FAUX
  - Si $expr_1$ est vraie alors $expr_2$ est évaluée
  - Si $expr_2$ est fausse, alors retourner FAUX sinon retourner VRAI
- $expr_1$ || $expr_2$
  - Si $expr_1$ est VRAI, alors retourner VRAI
  - Si $expr_2$ est VRAI, alors retourner VRAI
  - Retourner FAUX



Exercice 16 Lorsque vous aurez vu en algorithmique les structures conditionnelles et itératives, écrire en pseudo-langage algorithmique les algorithmes qui permettent d'évaluer :
- $expr_1$ && $expr_2$ && ... && $expr_n$
- $expr_1$ || $expr_2$ || ... || $expr_n$

2 Langage

2.1 Entrées/Sorties de base

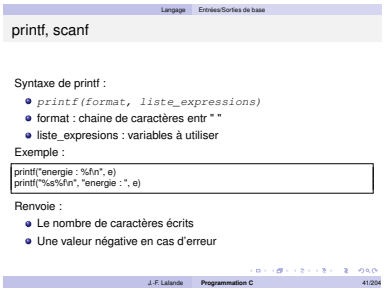
printf, scanf

Syntaxe de printf :
- *printf(format, liste_expressions)*
- format : chaine de caractères entr " "
- liste_expresions : variables à utiliser

Exemple :

```
printf("energie : %f\n", e)
printf("%s%f\n", "energie : ", e)
```

Renvoie :
- Le nombre de caractères écrits
- Une valeur négative en cas d'erreur



Exercice 17 A l'aide de deux variables entières a=1 et b=12, affichez "a=1 et b=12".

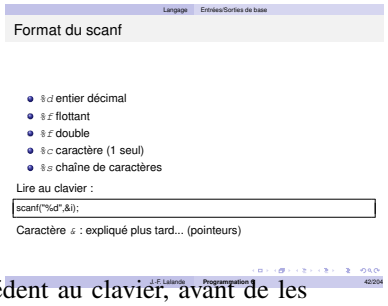
Format du scanf

- %d entier décimal
- %f flottant
- %f double
- %c caractère (1 seul)
- %s chaîne de caractères

Lire au clavier :

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

Caractère & : expliqué plus tard... (pointeurs)



Exercice 18 Demandez à l'utilisateur de rentrer les valeurs a et b de l'exercice précédent au clavier, avant de les afficher.

2.2 Instructions de contrôle

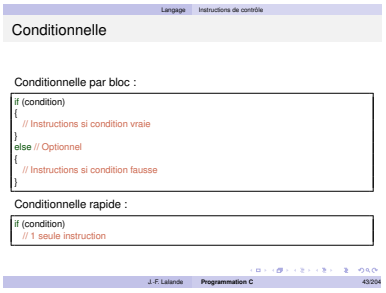
Conditionnelle

Conditionnelle par bloc :

```
if (condition)
{
  // Instructions si condition vraie
}
else // Optionnel
{
  // Instructions si condition fausse
}
```

Conditionnelle rapide :

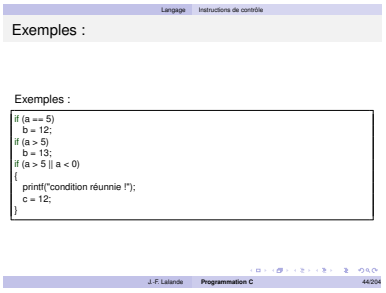
```
if (condition)
  // 1 seule instruction
```



Exemples :

Exemples :

```
if (a == 5)
   b = 12;
if (a > 5)
   b = 13;
if (a > 5 || a < 0)
{
   printf("condition réunnie !");
   c = 12;
}
```



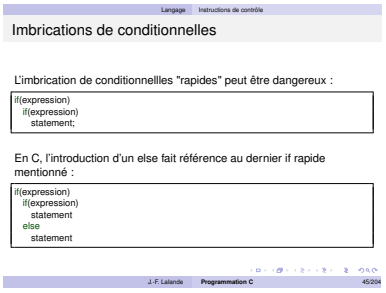
Exercice 19 Écrire un programme qui permet de déterminer si un nombre entre 1 et 10 est pair, puis s'il est premier (à l'aide de tests).

Exercice 20 Écrire un programme qui permet d'afficher à l'écran, dans l'ordre, trois entiers a, b et c.

Imbrications de conditionnelles

L'imbrication de conditionnellles "rapides" peut être dangereux :

```
if(expression)
   if(expression)
      statement;
```



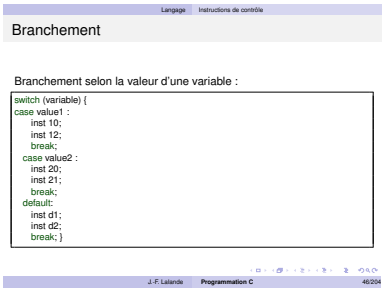
En C, l'introduction d'un else fait référence au dernier if rapide mentionné :

```
if(expression)
   if(expression)
      statement
   else
      statement
```

Branchement

Branchement selon la valeur d'une variable :

```
switch (variable) {
   case value1 :
      inst 10;
      inst 12;
      break;
   case value2 :
      inst 20;
      inst 21;
      break;
   default:
      inst d1;
      inst d2;
      break; }
```



Exercice 21 Écrire un programme qui imprime le type de lettre (voyelle ou consonne) entrée au clavier.

Itérations : POUR

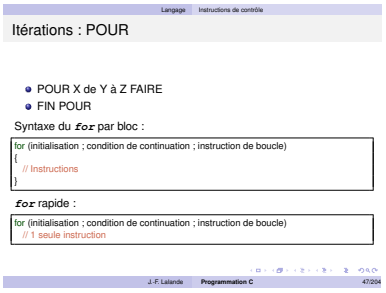
– POUR X de Y à Z FAIRE
– FIN POUR

Syntaxe du ***for*** par bloc :

```
for (initialisation ; condition de continuation ; instruction de boucle)
{
  // Instructions
}
```

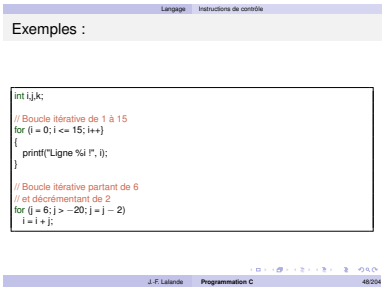
for rapide :

```
for (initialisation ; condition de continuation ; instruction de boucle)
  // 1 seule instruction
```



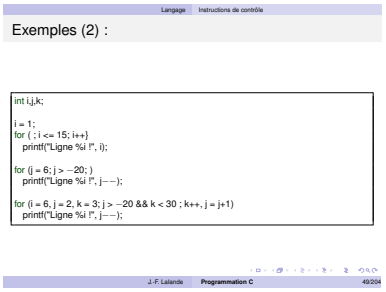
Exemples :

```
int i,j,k;

// Boucle itérative de 1 à 15
for (i = 0; i <= 15; i++}
{
  printf("Ligne %i !", i);
}

// Boucle itérative partant de 6
// et décrémentant de 2
for (j = 6; j > −20; j = j − 2)
  i = i + j;
```



Exemples (2) :

```
int i,j,k;

i = 1;
for ( ; i <= 15; i++}
  printf("Ligne %i !", i);

for (j = 6; j > −20; )
  printf("Ligne %i !", j−−);

for (i = 6, j = 2, k = 3; j > −20 && k < 30 ; k++, j = j+1)
  printf("Ligne %i !", j−−);
```

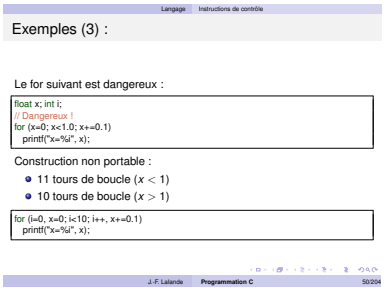


Exemples (3) :

Le for suivant est dangereux :

```
float x; int i;
// Dangereux !
for (x=0; x<1.0; x+=0.1)
  printf("x=%i", x);
```

Construction non portable :
– 11 tours de boucle ($x < 1$)
– 10 tours de boucle ($x > 1$)



```
for (i=0, x=0; i<10; i++, x+=0.1)
  printf("x=%i", x);
```

Exercice 22 Parcourez tous les nombres pairs et affichez les multiples de 3.

Itérations : TANT QUE

- TANT QUE ... FAIRE
- FIN TANT QUE

Syntaxe du **`while`** par bloc :

```
while (condition)
{
  // Instructions
}
```

`while` rapide :

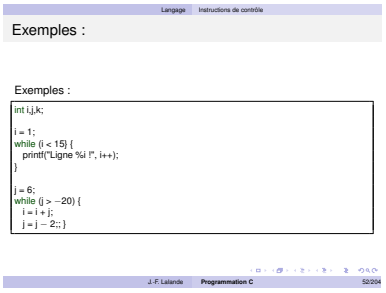
```
while (condition)
  // 1 seule instruction
```

Exercice 23 Affichez la suite $U_n = U_{n-2} + 2U_{n-1}$ jusqu'à $U_n > 1000$ en choisissant U_0 et U_1.

Exemples :

Exemples :

```
int i,j,k;

i = 1;
while (i < 15} {
  printf("Ligne %i !", i++);
}

j = 6;
while (j > −20) {
  i = i + j;
  j = j − 2;; }
```



Itérations : FAIRE

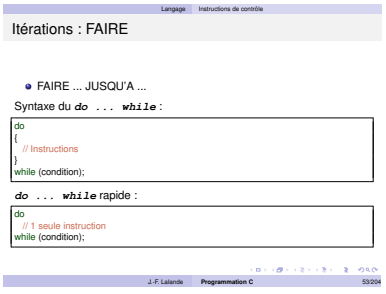
- FAIRE ... JUSQU'A ...

Syntaxe du **`do ... while`** :

```
do
{
  // Instructions
}
while (condition);
```

`do ... while` rapide :

```
do
  // 1 seule instruction
while (condition);
```



Exercice 24 Écrire un programme qui somme les 150 premiers entiers.

Exercice 25 Écrire un programme qui somme les 150 premiers entiers pairs.

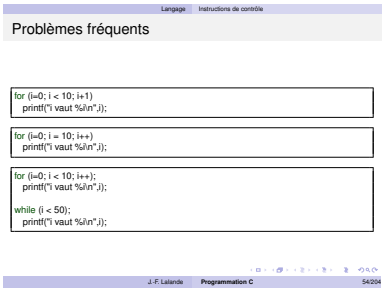
Exercice 26 Écrire un programme qui somme les premiers entiers jusqu'à ce que le carré de l'entier courant soit plus grand que 10 fois la somme.

Exercice 27 Écrire une boucle for à l'aide d'une boucle while.

Exercice 28 Écrire une boucle while à l'aide d'une boucle for.

Problèmes fréquents

```
for (i=0; i < 10; i+1)
  printf("i vaut %i\n",i);
```




```
for (i=0; i = 10; i++)
  printf("i vaut %i\n",i);
```

```
for (i=0; i < 10; i++);
  printf("i vaut %i\n",i);

while (i < 50);
  printf("i vaut %i\n",i);
```

Exercice 29 Testez ces codes et expliquez ce qui se produit.

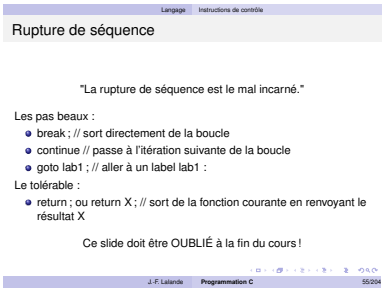
Rupture de séquence

"La rupture de séquence est le mal incarné."

Les pas beaux :
- break ; // sort directement de la boucle
- continue // passe à l'itération suivante de la boucle
- goto lab1 ; // aller à un label lab1 :

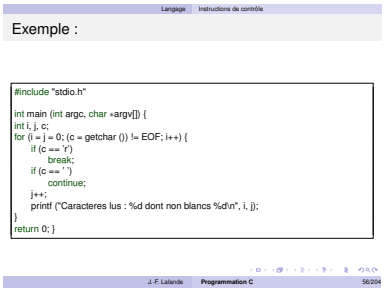
Le tolérable :
- return ; ou return X ; // sort de la fonction courante en renvoyant le résultat X

Ce slide doit être OUBLIÉ à la fin du cours !



Exemple :

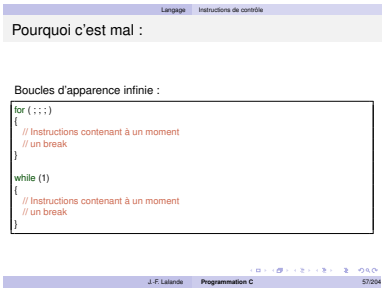
```
#include "stdio.h"

int main (int argc, char *argv[]) {
int i, j, c;
for (i = j = 0; (c = getchar ()) != EOF; i++) {
      if (c == 'r')
            break;
      if (c == ' ')
            continue;
      j++;
      printf ("Caracteres lus : %d dont non blancs %d\n", i, j);
}
return 0; }
```



Pourquoi c'est mal :

Boucles d'apparence infinie :

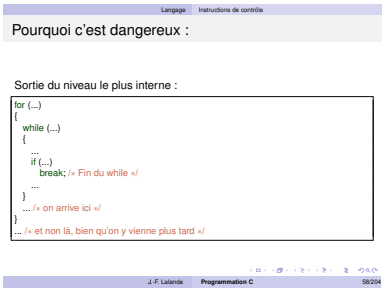
```
for ( ; ; )
{
  // Instructions contenant à un moment
  // un break
}

while (1)
{
  // Instructions contenant à un moment
  // un break
}
```



Pourquoi c'est dangereux :

Sortie du niveau le plus interne :

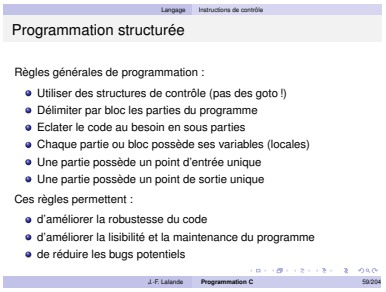
```
for (...)
{
  while (...)
  {
    ...
    if (...)
      break; /* Fin du while */
    ...
  }
  ... /* on arrive ici */
}
... /* et non là, bien qu'on y vienne plus tard */
```



Programmation structurée

Règles générales de programmation :
- Utiliser des structures de contrôle (pas des goto !)
- Délimiter par bloc les parties du programme
- Eclater le code au besoin en sous parties
- Chaque partie ou bloc possède ses variables (locales)
- Une partie possède un point d'entrée unique
- Une partie possède un point de sortie unique

Ces règles permettent :
- d'améliorer la robustesse du code
- d'améliorer la lisibilité et la maintenance du programme
- de réduire les bugs potentiels

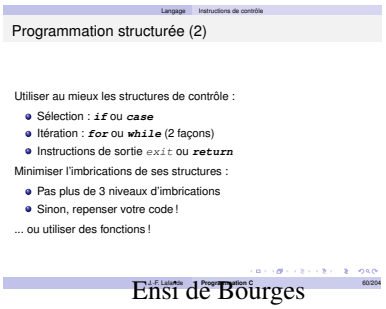


Programmation structurée (2)

Utiliser au mieux les structures de contrôle :
- Sélection : ***if*** ou ***case***
- Itération : ***for*** ou ***while*** (2 façons)
- Instructions de sortie `exit` ou ***return***

Minimiser l'imbrications de ses structures :
- Pas plus de 3 niveaux d'imbrications
- Sinon, repenser votre code !

... ou utiliser des fonctions !

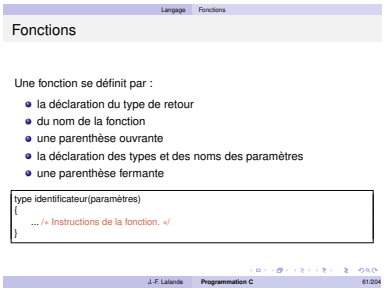


2.3 Fonctions

Fonctions

Une fonction se définit par :
- la déclaration du type de retour
- du nom de la fonction
- une parenthèse ouvrante
- la déclaration des types et des noms des paramètres
- une parenthèse fermante

```
type identificateur(paramètres)
{
    ... /* Instructions de la fonction. */
}
```

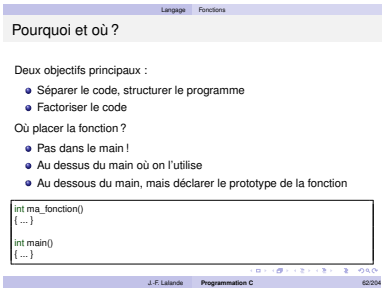


Pourquoi et où ?

Deux objectifs principaux :
- Séparer le code, structurer le programme
- Factoriser le code

Où placer la fonction ?
- Pas dans le main !
- Au dessus du main où on l'utilise
- Au dessous du main, mais déclarer le prototype de la fonction

```
int ma_fonction()
{ ... }

int main()
{ ... }
```

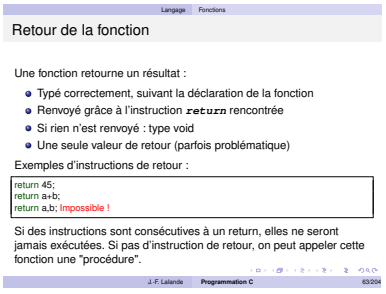


Retour de la fonction

Une fonction retourne un résultat :
- Typé correctement, suivant la déclaration de la fonction
- Renvoyé grâce à l'instruction ***return*** rencontrée
- Si rien n'est renvoyé : type void
- Une seule valeur de retour (parfois problématique)

Exemples d'instructions de retour :

```
return 45;
return a+b;
return a,b; Impossible !
```



Si des instructions sont consécutives à un return, elles ne seront jamais exécutées. Si pas d'instruction de retour, on peut appeler cette fonction une "procédure".

Exercice 30 Écrire une fonction prenant un entier et un double en paramètre. Cette fonction fait la somme et renvoie le résultat.

Exercice 31 Écrire une fonction qui prend trois entiers en paramètres et affiche à l'écran le plus grand et le plus petit.

Exercice 32 Ecrire une fonction qui calcule $n!$ et renvoie le résultat.

Exercice 33 Ecrire une fonction qui cacule k tel que $\sum_{i<k} i$ et renvoie le résultat.

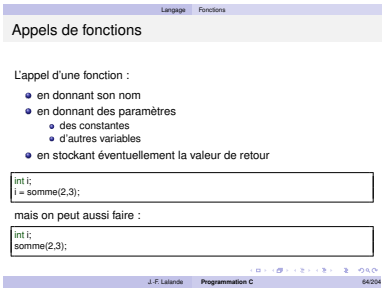
Appels de fonctions

L'appel d'une fonction :
- en donnant son nom
- en donnant des paramètres
  - des constantes
  - d'autres variables
- en stockant éventuellement la valeur de retour

```
int i;
i = somme(2,3);
```

mais on peut aussi faire :

```
int i;
somme(2,3);
```



Exemples

Une fonction "procédure" qui ne renvoie rien :

```
void affiche(int a) {
   printf("a vaut %i\n", a);
}
```

```
void main() {
   int i = 45;
   affiche(i); }
```

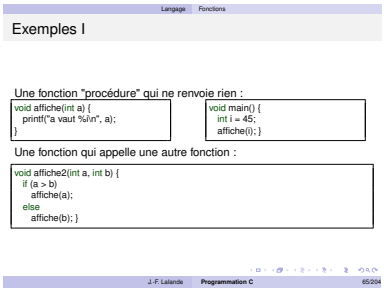
Une fonction qui appelle une autre fonction :

```
void affiche2(int a, int b) {
  if (a > b)
     affiche(a);
  else
     affiche(b); }
```

```
int plus(int a, int b)
{
   a = a + b ;
   return a
}
```

```
void main() {
   int resultat; int var = 10;
   resultat = calcul(var, 4, 3); }
```

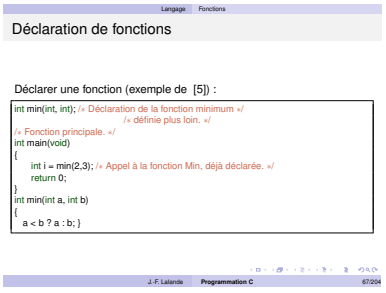
```
int calcul(int z, int x, int y)
{
   int u;
   u = z;
   u *= plus(x,y) ;
   return u;
}
```



Déclaration de fonctions

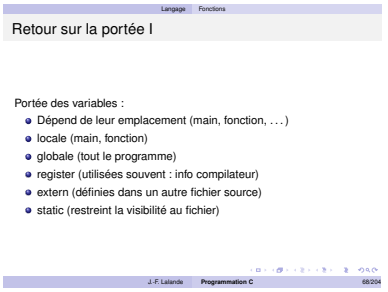
Déclarer une fonction (exemple de [**?**]) :

```
int min(int, int); /* Déclaration de la fonction minimum */
                   /* définie plus loin. */
/* Fonction principale. */
int main(void)
{
     int i = min(2,3); /* Appel à la fonction Min, déjà déclarée. */
     return 0;
}
int min(int a, int b)
{
   a < b ? a : b; }
```



Retour sur la portée

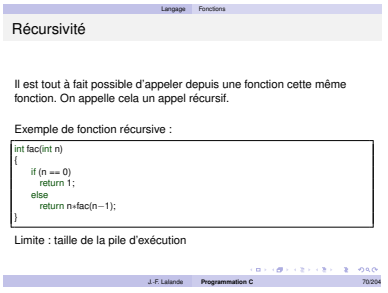
Portée des variables :
- Dépend de leur emplacement (main, fonction, . . .)
- locale (main, fonction)
- globale (tout le programme)
- register (utilisées souvent : info compilateur)
- extern (définies dans un autre fichier source)
- static (restreint la visibilité au fichier)




```c
#include <stdio.h>
int choix = 12;

int main(void) {
   int choix = 4;
   printf("Choix numéro %i \n", choix);
   ma_fonction();
   return 0;
}

void ma_fonction() {
   printf("Choix numéro %i\n", choix);
}
```

Récursivité

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



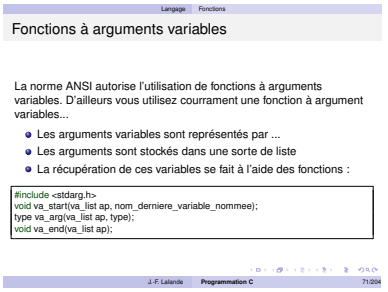
Exemple de fonction récursive :

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

Limite : taille de la pile d'exécution

Fonctions à arguments variables

La norme ANSI autorise l'utilisation de fonctions à arguments variables. D'ailleurs vous utilisez courrament une fonction à argument variables...
- Les arguments variables sont représentés par ...
- Les arguments sont stockés dans une sorte de liste
- La récupération de ces variables se fait à l'aide des fonctions :




```c
#include <stdarg.h>
void va_start(va_list ap, nom_derniere_variable_nommee);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

Fonctions à arguments variables (2)

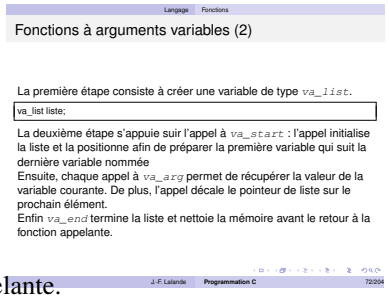
La première étape consiste à créer une variable de type `va_list`.

```
va_list liste;
```

La deuxième étape s'appuie suir l'appel à `va_start` : l'appel initialise la liste et la positionne afin de préparer la première variable qui suit la dernière variable nommée

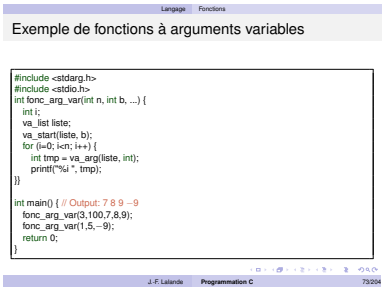
Ensuite, chaque appel à `va_arg` permet de récupérer la valeur de la variable courante. De plus, l'appel décale le pointeur de liste sur le prochain élément.

Enfin `va_end` termine la liste et nettoie la mémoire avant le retour à la fonction appelante.



Exemple de fonctions à arguments variables

```c
#include <stdarg.h>
#include <stdio.h>
int fonc_arg_var(int n, int b, ...) {
   int i;
   va_list liste;
   va_start(liste, b);
   for (i=0; i<n; i++) {
      int tmp = va_arg(liste, int);
      printf("%i ", tmp);
}}

int main() { // Output: 7 8 9 -9
   fonc_arg_var(3,100,7,8,9);
   fonc_arg_var(1,5,-9);
   return 0;
}
```



3 Les tableaux et pointeurs

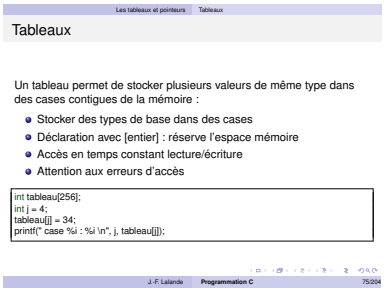
3.1 Tableaux

Tableaux

Un tableau permet de stocker plusieurs valeurs de même type dans des cases contigues de la mémoire :

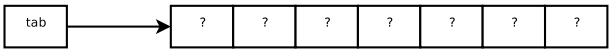
- Stocker des types de base dans des cases
- Déclaration avec [entier] : réserve l'espace mémoire
- Accès en temps constant lecture/écriture
- Attention aux erreurs d'accès

```
int tableau[256];
int j = 4;
tableau[j] = 34;
printf(" case %i : %i \n", j, tableau[j]);
```

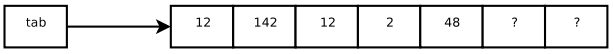


Initialisation des tableaux

Initialisation du tableau (***int*** *tab[7];*) :

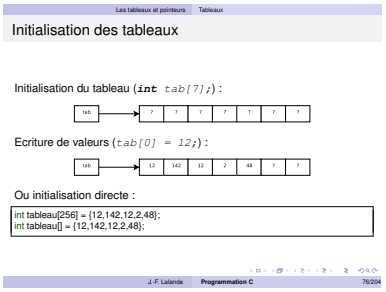


Ecriture de valeurs (*tab[0] = 12;*) :



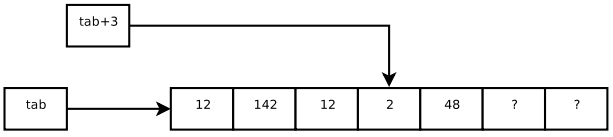
Ou initialisation directe :

```
int tableau[256] = {12,142,12,2,48};
int tableau[] = {12,142,12,2,48};
```

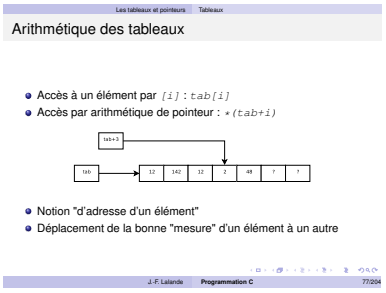


Arithmétique des tableaux

- Accès à un élément par *[i]* : *tab[i]*
- Accès par arithmétique de pointeur : **(tab+i)*



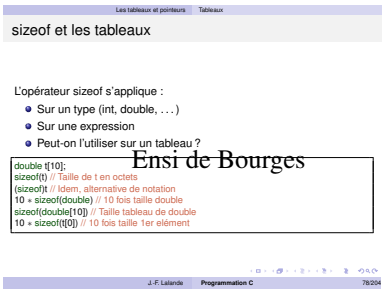
- Notion "d'adresse d'un élément"
- Déplacement de la bonne "mesure" d'un élément à un autre



sizeof et les tableaux

L'opérateur sizeof s'applique :

- Sur un type (int, double, . . .)
- Sur une expression
- Peut-on l'utiliser sur un tableau ?



```
double t[10];
sizeof(t) // Taille de t en octets
(sizeof)t // Idem, alternative de notation
10 * sizeof(double) // 10 fois taille double
sizeof(double[10]) // Taille tableau de double
10 * sizeof(t[0]) // 10 fois taille 1er elément
```

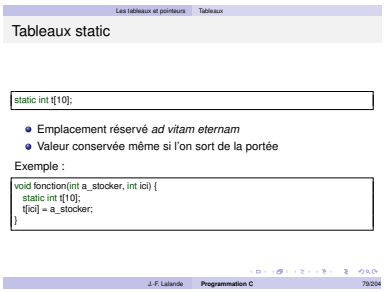
Tableaux static

```
static int t[10];
```

- Emplacement réservé *ad vitam eternam*
- Valeur conservée même si l'on sort de la portée

Exemple :

```
void fonction(int a_stocker, int ici) {
  static int t[10];
  t[ici] = a_stocker;
}
```



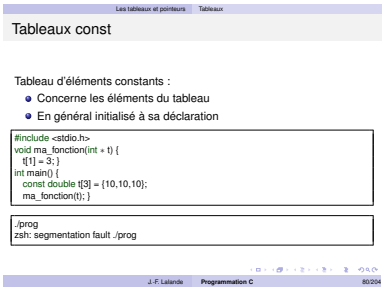
Tableaux const

Tableau d'éléments constants :

- Concerne les éléments du tableau
- En général initialisé à sa déclaration

```
#include <stdio.h>
void ma_fonction(int * t) {
  t[1] = 3; }
int main() {
  const double t[3] = {10,10,10};
  ma_fonction(t); }
```

```
./prog
zsh: segmentation fault ./prog
```



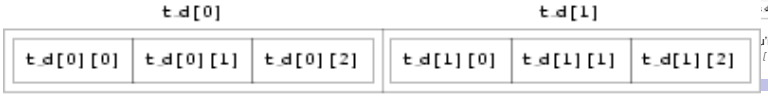
Tableaux multidimensionels

Il est possible de déclarer des tableaux à plusieurs dimensions

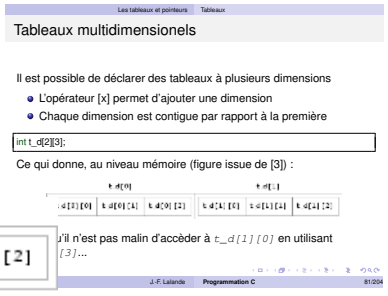
- L'opérateur [x] permet d'ajouter une dimension
- Chaque dimension est contigue par rapport à la première

```
int t_d[2][3];
```

Ce qui donne, au niveau mémoire (figure issue de [**?**]) :



Notez qu'il n'est pas malin d'accèder à `t_d[1][0]` en utilisant `t_d[0][3]`...



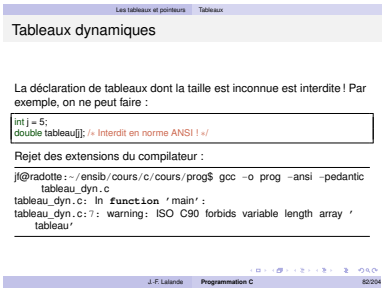
Tableaux dynamiques

La déclaration de tableaux dont la taille est inconnue est interdite ! Par exemple, on ne peut faire :

```
int j = 5;
double tableau[j]; /* Interdit en norme ANSI ! */
```

Rejet des extensions du compilateur :

```
jf@radotte
:~/ensib/cours/c/cours/prog$ gcc -o prog -ansi -pedantic tableau_dyn.c
tableau_dyn.c: In function 'main':
tableau_dyn.c:7: warning: ISO C90 forbids variable length array 'tableau'
```

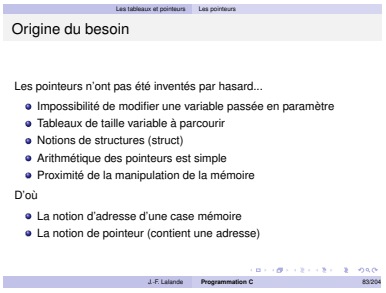


3.2 Les pointeurs

Origine du besoin

Les pointeurs n'ont pas été inventés par hasard...
- Impossibilité de modifier une variable passée en paramètre
- Tableaux de taille variable à parcourir
- Notions de structures (struct)
- Arithmétique des pointeurs est simple
- Proximité de la manipulation de la mémoire

D'où
- La notion d'adresse d'une case mémoire
- La notion de pointeur (contient une adresse)



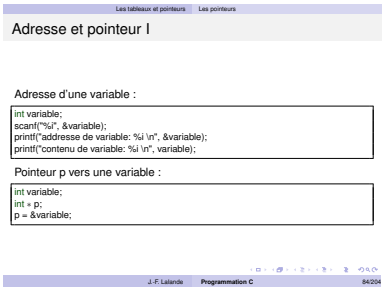
Adresse et pointeur

Adresse d'une variable :

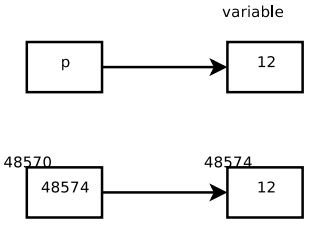
```
int variable;
scanf("%i", &variable);
printf("addresse de variable: %i \n", &variable);
printf("contenu de variable: %i \n", variable);
```

Pointeur p vers une variable :

```
int variable;
int * p;
p = &variable;
```



Exemple d'un pointeur sur un entier :



- Le pointeur p pointe vers la variable
- Le pointeur p contient l'adresse de la variable
- d'où l'homogénéité de `p = &variable;`.

Opérateur `*` donne la valeur d'un pointeur :

```c
int variable;
int * p;
p = &variable;
printf("addresse de variable: %i \n", p);
printf("contenu de variable: %i \n", *p);
```

Ne pas confondre :
- La signification du `&`
- La signification du `*`
- Et s'aider de petits dessins à l'occasion !

Exercice 34 Créez un pointeur p sur un float et un float f. Enregistrez 12.3 dans f et faites pointer p sur ce flottant.

Exercice 35 Créez un pointeur q sur un float. Initialisez directement la valeur pointée par q avec 45.5. Créez un pointeur r et faites le pointer sur la même valeur que q.

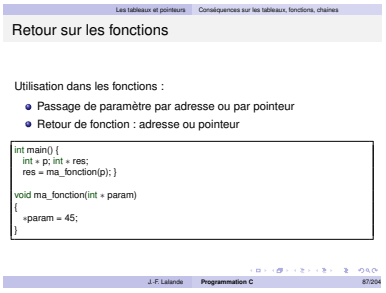
Exercice 36 Créez deux pointeurs et faites les pointer sur deux entiers. Echangez alors les valeurs pointés.

3.3 Conséquences sur les tableaux, fonctions, chaines

Retour sur les fonctions

Utilisation dans les fonctions :
- Passage de paramètre par adresse ou par pointeur
- Retour de fonction : adresse ou pointeur

```c
int main() {
  int * p; int * res;
  res = ma_fonction(p); }

void ma_fonction(int * param)
{
  *param = 45;
}
```



Exercice 37 Codez une fonction `au_carre` qui prend un pointeur sur un entier en paramètre et le passe au carré.

Exercice 38 Codez une fonction `echange` qui permet de prendre deux pointeurs en paramètre et qui échange le pointage vers ces deux entiers.

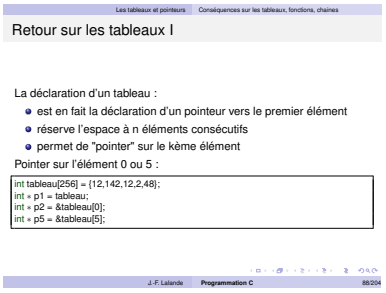
Exercice 39 Codez une fonction `initialise` ne prenant pas de paramètre et renvoyant un pointeur sur un entier.

Retour sur les tableaux

La déclaration d'un tableau :
- est en fait la déclaration d'un pointeur vers le premier élément
- réserve l'espace à n éléments consécutifs
- permet de "pointer" sur le kème élément

Pointer sur l'élément 0 ou 5 :

```c
int tableau[256] = {12,142,12,2,48};
int * p1 = tableau;
int * p2 = &tableau[0];
int * p5 = &tableau[5];
```



Opérations possibles sur un pointeur :
- affectation d'une adresse au pointeur
- opérateur "*" : accès à la donnée
- addition d'un entier k : pointe sur la kème case
- soustraction de deux pointeurs : nombre d'éléments

Exemple d'addition / de soustraction :

```
long *px, *px2;
px2 = px + i; // qui équivaut en fait a:
px2 = (long *) ((int) px + i * sizeof (long));
i = px2 - px;
```

Le parcours d'un tableau :
- se fait en utilisant un pointeur sur les éléments
- utilise l'arithmétique des pointeurs (très pratique)

Exemple :

```
int tableau[256] = {12,142,12,2,48};
int * p1 = tableau;
for (i= 0; i < 5; i++)
  printf("tableau[%i]=%i\n", *(p1+i));
for (i= 0; i < 5; i++) // Est aussi possible:
  printf("tableau[%i]=%i\n", *(tableau+i));
while (p1 != tableau + 5)
  printf("tableau[%i]=%i\n", *(p1++));
```

Exercice 40 Quel est l'erreur dans le while précédent ? Comment la corriger ?

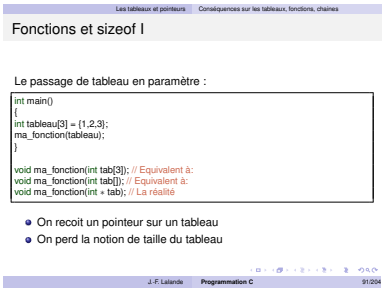
Exercice 41 Codez deux pointeurs qui pointent sur le début et la fin d'un tableau. Puis faites parcourir à ces deux pointeurs le tableau l'un de gauche à droite et l'autre de droite à gauche.

Exercice 42 Ecrire une fonction qui prend en paramètre un tableau et renvoie un pointeur sur l'élément minimum.

Fonctions et sizeof

Le passage de tableau en paramètre :

```
int main()
{
int tableau[3] = {1,2,3};
ma_fonction(tableau);
}

void ma_fonction(int tab[3]); // Equivalent à:
void ma_fonction(int tab[]); // Equivalent à:
void ma_fonction(int * tab); // La réalité
```



- On recoit un pointeur sur un tableau
- On perd la notion de taille du tableau

Taille d'un tableau :

```
double t[10];
sizeof(t) // Taille du tableau
sizeof(t) / sizeof(t[0]) // Nombre d'éléments
```

Par un appel de fonction, cela ne marche plus :

```
void ma_fonction(int * t)
{
  int nb = sizeof(t) / sizeof(t[0]);
  printf("nb=%i\n", nb); // nb vaut 1 ici
}
```

Preuve :

```
#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); }
```

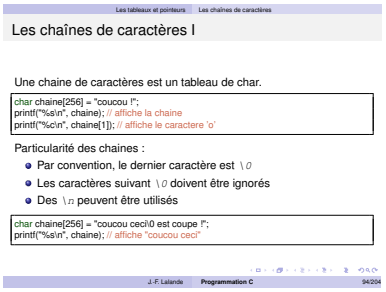
```
./prog
nb=10
nb=1
```

3.4 Les chaînes de caractères

Les chaînes de caractères

Une chaine de caractères est un tableau de char.

```
1 char chaine[256] = "coucou !";
2 printf("%s\n", chaine); // affiche la chaine
3 printf("%c\n", chaine[1]); // affiche le caractere 'o'
```



Particularité des chaines :
- Par convention, le dernier caractère est *\0*
- Les caractères suivant *\0* doivent être ignorés
- Des *\n* peuvent être utilisés

```
1 char chaine[256] = "coucou ceci\0 est coupe !";
2 printf("%s\n", chaine); // affiche "coucou ceci"
```

Récupérer une chaine de caractère :

```
1 char chaine[256];
2 scanf("%s", chaine);
```

Comparer deux chaines de caractères :

```
1 char chaine[256];
2 char chaine2[256];
3 int res;
4 res = strcmp(chaine, chaine2);
```

Autres fonctions : *strcasecmp*, *strlen*, *strcat*, *strcpy*,... plus d'informations : man string !

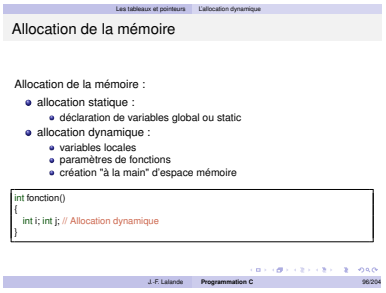
Exercice 43 Ecrire une fonction qui prend en paramètre une chaine et renvoie le nombre de caractères.

3.5 L'allocation dynamique

Allocation de la mémoire

Allocation de la mémoire :
- allocation statique :
  - déclaration de variables global ou static
- allocation dynamique :
  - variables locales
  - paramètres de fonctions
  - création "à la main" d'espace mémoire

```
1 int fonction()
2 {
3   int i; int j; // Allocation dynamique
4 }
```



Allocation des tableaux

Allocation dynamique "à la main" :

```
void* calloc(size_t nb_blocs, size_t taille_bloc)
```

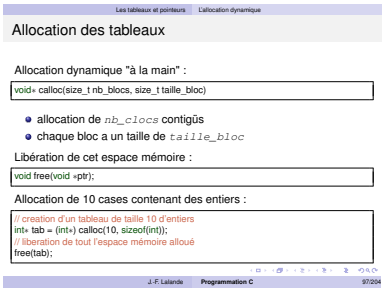
- allocation de `nb_clocs` contigüs
- chaque bloc a un taille de `taille_bloc`

Libération de cet espace mémoire :

```
void free(void *ptr);
```

Allocation de 10 cases contenant des entiers :

```
// creation d'un tableau de taille 10 d'entiers
int* tab = (int*) calloc(10, sizeof(int));
// liberation de tout l'espace mémoire alloué
free(tab);
```

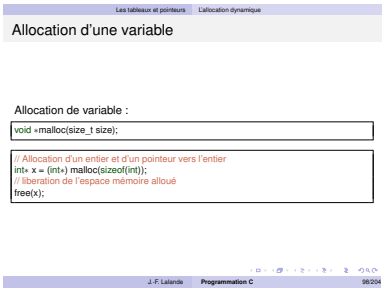


Allocation d'une variable

Allocation de variable :

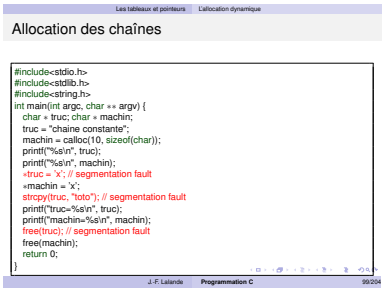
```
void *malloc(size_t size);
```

```
// Allocation d'un entier et d'un pointeur vers l'entier
int* x = (int*) malloc(sizeof(int));
// liberation de l'espace mémoire alloué
free(x);
```



Allocation des chaînes

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main(int argc, char ** argv) {
  char * truc; char * machin;
  truc = "chaine constante";
  machin = calloc(10, sizeof(char));
  printf("%s\n", truc);
  printf("%s\n", machin);
  *truc = 'x'; // segmentation fault
  *machin = 'x';
  strcpy(truc, "toto"); // segmentation fault
  printf("truc=%s\n", truc);
  printf("machin=%s\n", machin);
  free(truc); // segmentation fault
  free(machin);
  return 0;
}
```



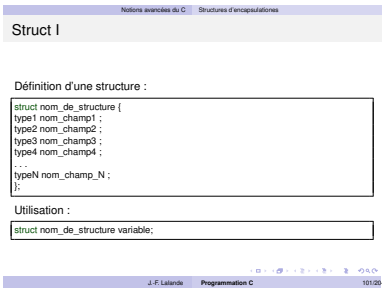
4 Notions avancées du C

4.1 Structures d'encapsulationes

Struct

Définition d'une structure :

```
struct nom_de_structure {
  type1 nom_champ1 ;
  type2 nom_champ2 ;
  type3 nom_champ3 ;
  type4 nom_champ4 ;
  . . .
  typeN nom_champ_N ;
};
```



Utilisation :

```
struct nom_de_structure variable;
```

Exemple :

```
struct animal {
  char nom[50];
  int age;
  float taille; };
```

Instanciation d'une structure :

```
struct animal bar={"Bar",12,7.5};
```

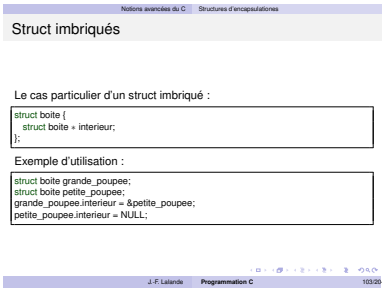
Accès aux attributs :

```
bar.age = 13;
bar.taille = bar.taille + 5.0;
```

Struct imbriqués

Le cas particulier d'un struct imbriqué :

```
struct boite {
  struct boite * interieur;
};
```



Exemple d'utilisation :

```
struct boite grande_poupee;
struct boite petite_poupee;
grande_poupee.interieur = &petite_poupee;
petite_poupee.interieur = NULL;
```

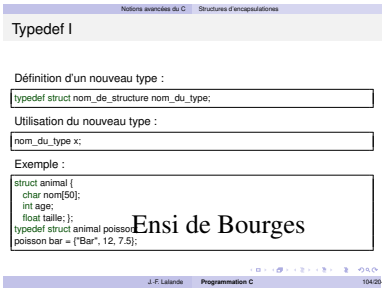
Typedef

Définition d'un nouveau type :

```
typedef struct nom_de_structure nom_du_type;
```

Utilisation du nouveau type :

```
nom_du_type x;
```



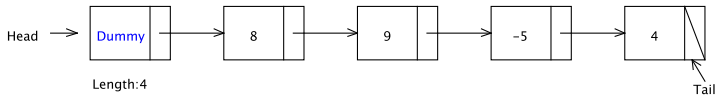


FIGURE 1 – Liste avec un élément "dummy" en tête

Exemple :

```c
struct animal {
  char nom[50];
  int age;
  float taille; };
typedef struct animal poisson;
poisson bar = {"Bar", 12, 7.5};
```

Le struct imbriqué avec un typedef :

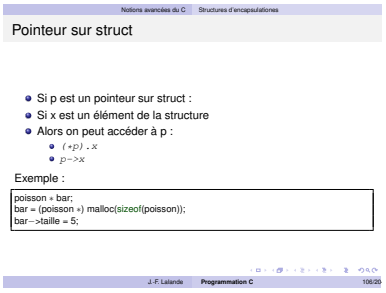
```c
typedef struct BOITE {
  struct BOITE * interieur;
} boite;
boite ma_boite;
ma_boite.interieur = (boite *)malloc(sizeof(boite));
```

Pointeur sur struct

- Si p est un pointeur sur struct :
- Si x est un élément de la structure
- Alors on peut accéder à p :
  - `(*p).x`
  - `p->x`

Exemple :

```c
poisson * bar;
bar = (poisson *) malloc(sizeof(poisson));
bar->taille = 5;
```

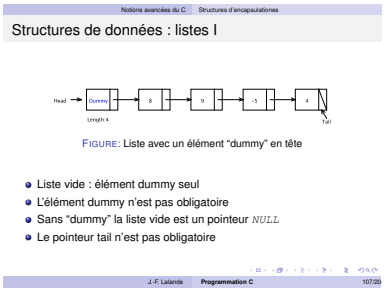


Structures de données : listes

- Liste vide : élément dummy seul
- L'élément dummy n'est pas obligatoire
- Sans "dummy" la liste vide est un pointeur `NULL`
- Le pointeur tail n'est pas obligatoire

La liste de poissons :

```c
#include <stdlib.h>
// Définitions
struct animal {
  char nom[50];
  int age; };
typedef struct animal poisson;
typedef struct CELLULE {
  struct CELLULE * suivante;
  poisson * p;
} cellule;

// Instanciation
int main() {
cellule * dummy, * premiere, * deuxieme;
poisson * bar = (poisson *) malloc(sizeof(poisson));
bar->age = 5;
```



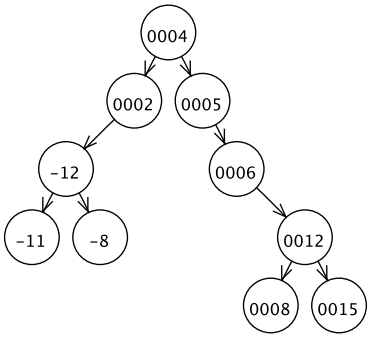


FIGURE 2 – Arbre binaire de recherche d'entiers

```
dummy = (cellule *) malloc(sizeof(cellule));
premiere = (cellule *) malloc(sizeof(cellule));
deuxieme = (cellule *) malloc(sizeof(cellule));
dummy->p = NULL;
dummy->suivante = premiere;
premiere->p = bar;
premiere->suivante = deuxieme;
deuxieme->suivante = NULL;
}
```

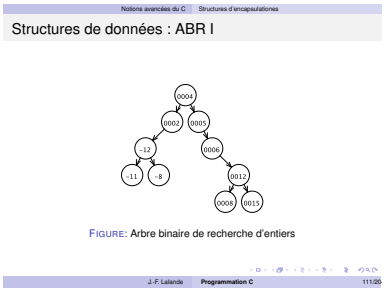
La liste doublement chaînée de poissons :
- cf liste de poissons avec en plus :
- pointeur poisson précédent : `cellule * prec`
- prec est NULL pour le début et la fin

La liste chaînée circulaire de poissons :
- cf liste doublement chaînée de poissons
- prec de cellule début pointe sur la fin

Structures de données : ABR

L'arbre binaire de poissons :

```
#include <stdlib.h>
// Définitions
struct animal {
   char nom[50];
   int age; };
typedef struct animal poisson;
typedef struct NOEUD {
   struct NOEUD * gauche;
   struct NOEUD * droite;
   poisson * p;
} noeud;

// Instanciation
int main() {
noeud * premier, * deuxieme, * troisieme;
poisson * bar = (poisson *) malloc(sizeof(poisson));
bar->age = 5;
premier = (noeud *) malloc(sizeof(noeud));
deuxieme = (noeud *) malloc(sizeof(noeud));
troisieme = (noeud *) malloc(sizeof(noeud));
premier->p = bar; // A faire dans les autres noeuds
premier->gauche = deuxieme;
//premier->droite = deuxieme; // hérésie !!
premier->droite = troisieme;
}
```



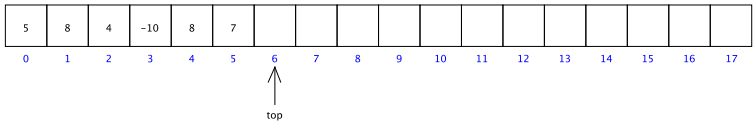
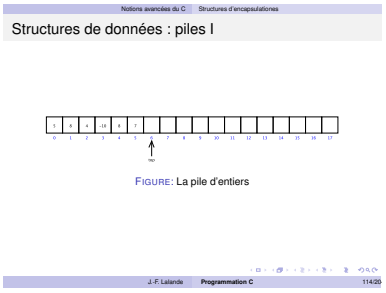


FIGURE 3 – La pile d'entiers

Structures de données : piles

La pile de poissons (version 1) :

```c
#include <stdlib.h>
// Définitions
struct animal {
   char nom[50];
   int age; };
typedef struct animal poisson;
typedef struct PILE {
   int taille_max; // Taille de la pile
   int courant; // Index de l'élément courant
   poisson * p; // Tableau de poissons
} pile;

// Instanciation
int main() {
pile * ma_pile = (pile *)malloc(sizeof(pile));
ma_pile->taille_max = 256;
ma_pile->p = (poisson *)calloc(256, sizeof(poisson));
poisson * bar = (poisson *) malloc(sizeof(poisson));
bar->age = 5;
//*(ma_pile->p+ma_pile->courant) = bar; // Ne marche pas
*(ma_pile->p+ma_pile->courant++) = *bar; // Copie du bar dans le tableau
}
```



La pile de poissons (version 2) :

```c
#include <stdlib.h>
// Définitions
struct animal {
   char nom[50];
   int age; };
typedef struct animal poisson;
typedef struct PILE {
   int taille_max; // Taille de la pile
   int courant; // Index de l'élément courant
   poisson ** p; // Tableau de pointeur sur poissons
} pile;

// Instanciation
int main() {
pile * ma_pile = (pile *)malloc(sizeof(pile));
ma_pile->taille_max = 256;
ma_pile->p = (poisson **)calloc(256, sizeof(poisson *));
poisson * bar = (poisson *) malloc(sizeof(poisson));
bar->age = 5;
*(ma_pile->p+ma_pile->courant++) = bar; // Marche !
}
```

Structures de données : files

Structures de données : files

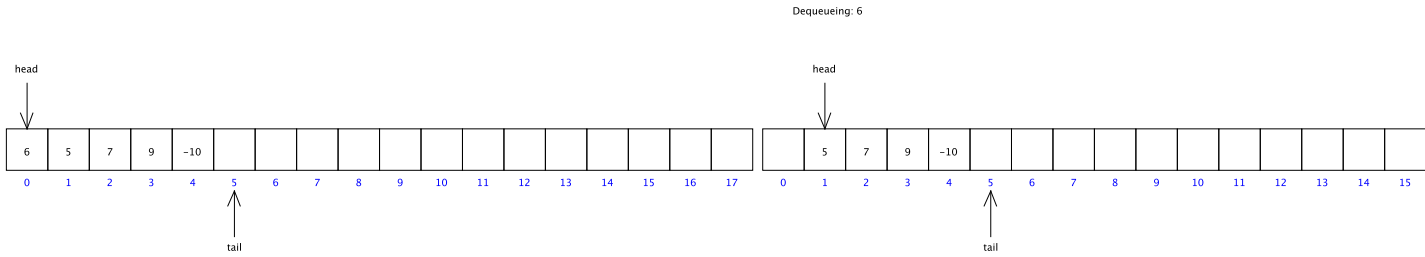


FIGURE 4 – File d'entiers

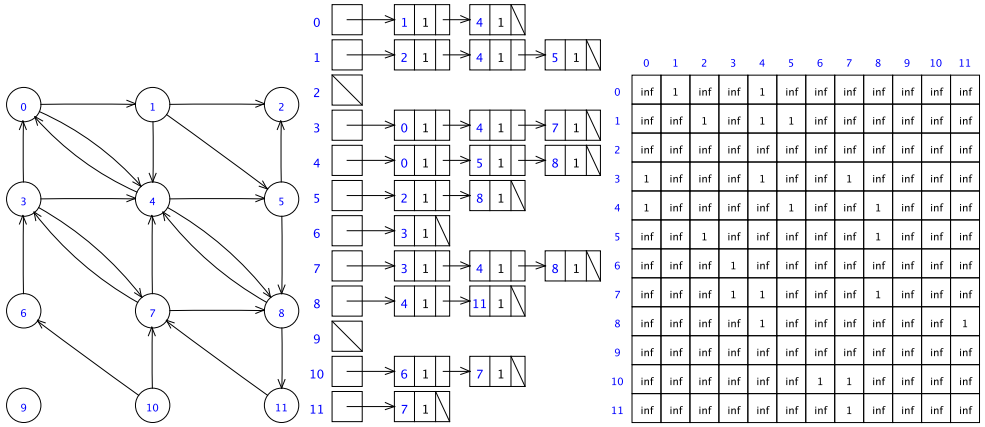


FIGURE 5 – Graphes : différentes représentations

La file de poissons :
- cf file de poisson (version 2) avec en plus :
- index courant empiler (endroit pour empiler)
- index courant depiler (endroit pour dépiler)
- petite difficulté : gérer de façon circulaire

Structures de données : graphes

Structures de données : graphes

Le graphe de poissons :
- Graphe sous sa première forme :
  - Définir un noeud
  - Définir un tableau de pointeurs sur noeuds
  - Chaque noeuds pointe sur un poisson
- Graphe sous sa deuxième forme :
  - Tableau de poissons
  - Tableau de listes d'adjacences
- Graphe sous sa troisième forme
  - Tableau de poissons
  - Matrice d'entiers

Union

Définition d'une union :

Notions avancées du C Structures d'encapsulationes

Structures de données : files

La file de poissons :
- cf file de poisson (version 2) avec en plus :
- index courant empiler (endroit pour empiler)
- index courant depiler (endroit pour dépiler)
- petite difficulté : gérer de façon circulaire

J.-F. Lalande Programmation C 120/20

Notions avancées du C Structures d'encapsulationes

Structures de données : graphes

Le graphe de poissons :
- Graphe sous sa première forme :
  - Définir un noeud
  - Définir un tableau de pointeurs sur noeuds
  - Chaque noeuds pointe sur un poisson
- Graphe sous sa deuxième forme :
  - Tableau de poissons
  - Tableau de listes d'adjacences
- Graphe sous sa troisième forme
  - Tableau de poissons
  - Matrice d'entiers

J.-F. Lalande Programmation C 122/20

Notions avancées du C Structures d'encapsulationes

Union I

Définition d'une union :
- Peut contenir différents types
- Mais une seule valeur à la fois

```
union nom_de_union {
type1 nom_champ1 ;
type2 nom_champ2 ;
. . .
typeN nom_champ_N ;
};
```

Utilisation :

– Peut contenir différents types
– Mais une seule valeur à la fois

```
union  nom_de_union  {
  type1 nom_champ1 ;
  type2 nom_champ2 ;
  . . .
  typeN nom_champ_N ;
};
```

Utilisation :

```
union nom_de_structure variable;
```

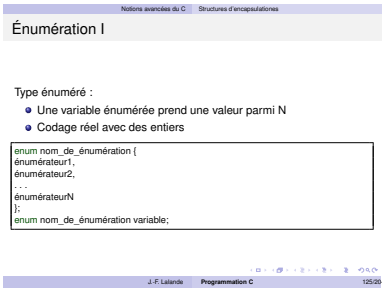
Exemple d'union :

```
union number // Peut contenir 3 types
{
  short shortnumber;
  long longnumber;
  float floatnumber;
} anumber;
anumber.longnumer = 45; // considère anumber comme long
anumber.floatnumer = 45.4; // considère anumber comme float
```

Énumération

Type énuméré :
– Une variable énumérée prend une valeur parmi N
– Codage réel avec des entiers

```
enum  nom_de_énumération {
  énumérateur1,
  énumérateur2,
  . . .
  énumérateurN
};
enum nom_de_énumération variable;
```



Exemple d'utilisation :

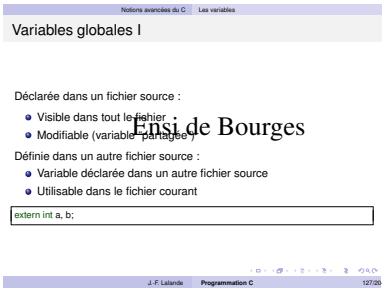
```
#include <stdio.h>
enum days {mon, tues, sun};

int main() {
  enum days d = mon;
  printf("d=%i\n", d);
  d= tues;
  printf("d=%i\n", d); }
```

```
./prog
d=0
d=1
```

4.2 Les variables

Variables globales

Déclarée dans un fichier source :
– Visible dans tout le fichier
– Modifiable (variable "partagée")



Définie dans un autre fichier source :
- Variable déclarée dans un autre fichier source
- Utilisable dans le fichier courant

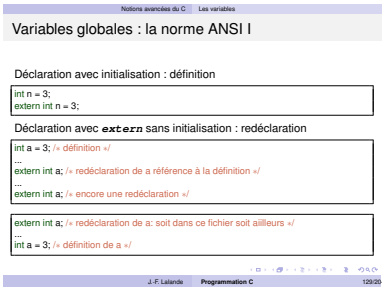
```
extern int a, b;
```

Variable globale “cachée” dans un fichier source :
- Mot clef static
- Visible globalement dans le fichier
- Invisible en externe depuis un autre fichier
- Aucune collision avec une autre variable de même nom

Variables globales : la norme ANSI

Déclaration avec initialisation : définition

```
int n = 3;
extern int n = 3;
```



Déclaration avec ***extern*** sans initialisation : redéclaration

```
int a = 3; /* définition */
...
extern int a; /* redéclaration de a référence à la définition */
...
extern int a; /* encore une redéclaration */
```

```
extern int a; /* redéclaration de a: soit dans ce fichier soit aiilleurs */
...
int a = 3; /* définition de a */
```

```
static int a; /* définition de a */
...
extern int a; /* redéclaration de a se référant à la définition */
```

Déclaration sans ***extern*** sans initialisation :
- définition potentielle
- ne devient valide que si aucune définition n’est trouvée
- si une définition est trouvée c’est une redéclaration

```
int a; /* définition potentielle de a */
...
int a; /* définition potentielle de a */
...
/* pas d’autre définition: a est définie en tant que int a = 0; */
```

```
extern int a; /* redéclaration de a: soit dans ce fichier soit ailleurs */
...
int a; /* définition potentielle de a */
...
int a = 2; /* définition de a */
```

```
void stat(); /* prototype fn */

main()
{ int i;
   for (i=0;i<5;++i)
      stat();
}

stat() {
   int auto_var = 0;
   static int static_var = 0;
   printf( "auto = %d, static = %d\n", auto_var, static_var);
   ++auto_var;
   ++static_var;
}

/Output is:

auto_var = 0, static_var= 0
```

```
auto_var = 0, static_var = 1
auto_var = 0, static_var = 2
```

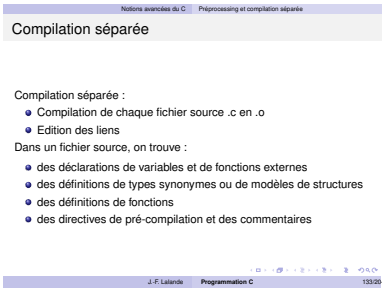
```
auto_var = 0, static_var = 3
auto_var = 0, static_var = 4
```

4.3 Préprocessing et compilation séparée

Compilation séparée

Compilation séparée :
- Compilation de chaque fichier source .c en .o
- Edition des liens

Dans un fichier source, on trouve :
- des déclarations de variables et de fonctions externes
- des définitions de types synonymes ou de modèles de structures
- des définitions de fonctions
- des directives de pré-compilation et des commentaires

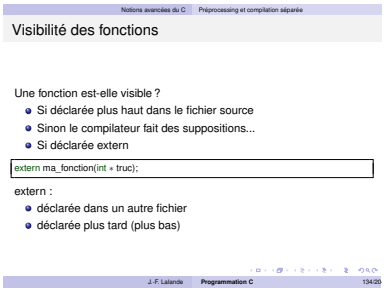


Visibilité des fonctions

Une fonction est-elle visible ?
- Si déclarée plus haut dans le fichier source
- Sinon le compilateur fait des suppositions...
- Si déclarée extern

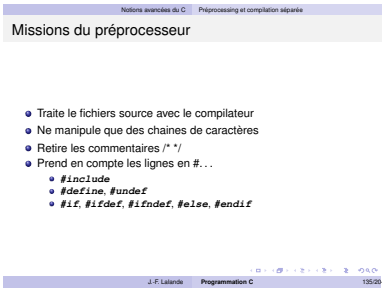
```
extern ma_fonction(int * truc);
```

extern :
- déclarée dans un autre fichier
- déclarée plus tard (plus bas)



Missions du préprocesseur

- Traite le fichiers source avec le compilateur
- Ne manipule que des chaines de caractères
- Retire les commentaires /* */
- Prend en compte les lignes en #...
  - **`#include`**
  - **`#define`**, **`#undef`**
  - **`#if`**, **`#ifdef`**, **`#ifndef`**, **`#else`**, **`#endif`**



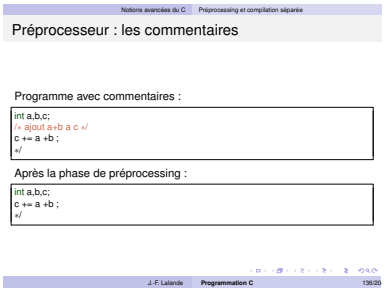
Préprocesseur : les commentaires

Programme avec commentaires :

```
int a,b,c;
/* ajout a+b a c */
c += a +b ;
*/
```

Après la phase de préprocessing :

```
int a,b,c;
c += a +b ;
*/
```



Préprocesseur : directive include

Directive **`#include`** sert à définir :
- types non prédéfinis,
- modèles et noms de structures,
- types et noms de variables,
- prototypes de fonctions.

Utilisation :
- Premier avantage : agréger les informations
- Deuxième avantage : déclaration des prototypages

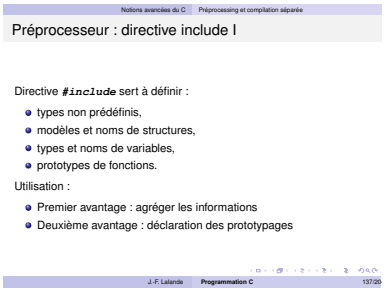
Différentes façons d'inclure :

```
#include <stdio.h>                          1
#include "/users/chris/essai/header.h"      2
#include "header.h"                         3
```

- Chemins standards ou définis à la compilation (catalogues)
- Chemin absolu, relatif au répertoire de compilation

```
gcc -Icatalogue                             1
```

- "" : catalogue courant, puis -I
- <> : -I, puis catalogues par défaut du compilateur



Préprocesseur : variables de pré-compilation

Définition de constantes :

```
#define n 20                                1
```

Utile pour remplacer :

```
int tab[20];                                1
for ( i = 0 ; i < 20 ; i++ )                2
```

Par :

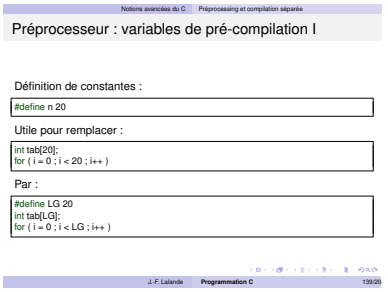
```
#define LG 20                               1
int tab[LG];                                2
for ( i = 0 ; i < LG ; i++ )                3
```

Utile aussi pour des expressions compliquées :

```
#define MAXxy (x>y?x:y)                     1

int z;                                      2
z = MAXxy                                   3
                                            4
```

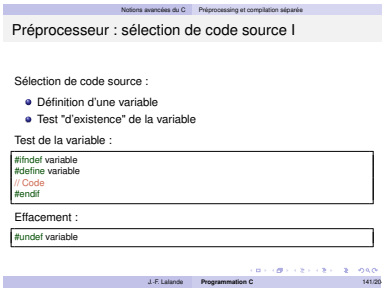
Attention cependant :

```
#define XMY x-y                             1

if (XMY * 2 < z}                            2
                                            3
```



Préprocesseur : sélection de code source

Sélection de code source :
– Définition d'une variable
– Test "d'existence" de la variable
Test de la variable :




```
#ifndef variable
#define variable
// Code
#endif
```

Effacement :

```
#undef variable
```

```
#define vrai 1
#define faux 0
#if vrai
// Code
#endif
```

```
#define DEBUG
#if defined DEBUG
  code passe
#endif
#if defined(DEBUG)
  code passe
#endif
```

Test plus complexe :

```
#define TRUC 1
#define CHOSE 0
#if TRUC && !CHOSE
...
#endif
```

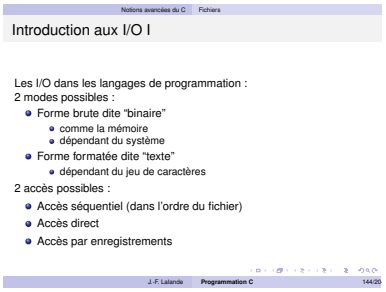
Option à la compilation *gcc -Dvariable=valeur* :

```
gcc -DCHOSE=1 -o truc truc.c
```

4.4 Fichiers

Introduction aux I/O

Les I/O dans les langages de programmation :
2 modes possibles :
– Forme brute dite "binaire"
  – comme la mémoire
  – dépendant du système
– Forme formatée dite "texte"
  – dépendant du jeu de caractères
2 accès possibles :
– Accès séquentiel (dans l'ordre du fichier)
– Accès direct
– Accès par enregistrements
En C, les choses sont différentes :
– Enregistrements : cela n'existe pas
– Fichier vu comme collection d'octets
– Distinction fichier binaire/formaté :
  – faite parfois à l'ouverture



  - dépend des implémentations
- Types d'opérations :
  - binaire : `fread`, `fwrite`
  - formaté : `fscanf`, `fprintf`, `fgets`, `fputs`
  - mixte : `fgetc`, `getc`, `fputc`, `puc`
- Accès séquentiel/direct :
  - Pas de fonctions spécifiques
  - Manipulation du pointeur de fichier (octet)

Ouverture/Fermeture

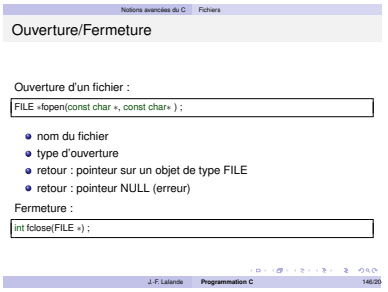
Ouverture d'un fichier :

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

- nom du fichier
- type d'ouverture
- retour : pointeur sur un objet de type FILE
- retour : pointeur NULL (erreur)

Fermeture :

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



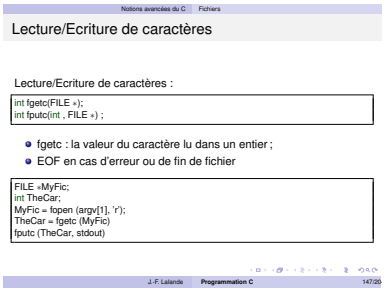
Lecture/Ecriture de caractères

Lecture/Ecriture de caractères :

```
int fgetc(FILE *);
int fputc(int , FILE *) ;
```

- fgetc : la valeur du caractère lu dans un entier ;
- EOF en cas d'erreur ou de fin de fichier

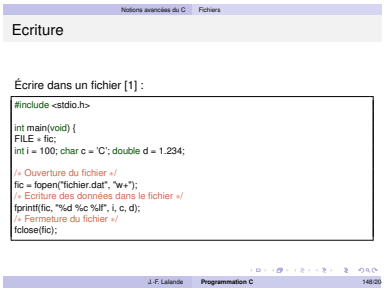
```
FILE *MyFic;
int TheCar;
MyFic = fopen (argv[1], 'r');
TheCar = fgetc (MyFic)
fputc (TheCar, stdout)
```



Ecriture

Écrire dans un fichier [**?**] :

```
#include <stdio.h>

int main(void) {
FILE * fic;
int i = 100; char c = 'C'; double d = 1.234;

/* Ouverture du fichier */
fic = fopen("fichier.dat", "w+");
/* Ecriture des données dans le fichier */
fprintf(fic, "%d %c %lf", i, c, d);
/* Fermeture du fichier */
fclose(fic);
```



Lecture/Ecriture de string

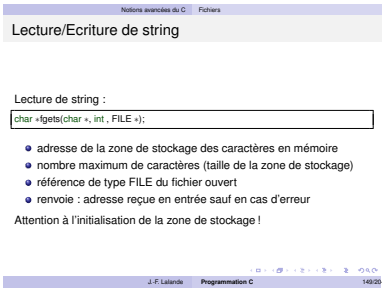
Lecture de string :

```
char *fgets(char *, int , FILE *);
```

1

- adresse de la zone de stockage des caractères en mémoire
- nombre maximum de caractères (taille de la zone de stockage)
- référence de type FILE du fichier ouvert
- renvoie : adresse reçue en entrée sauf en cas d'erreur

Attention à l'initialisation de la zone de stockage !



Lecture/Ecriture de string

De manière similaire :

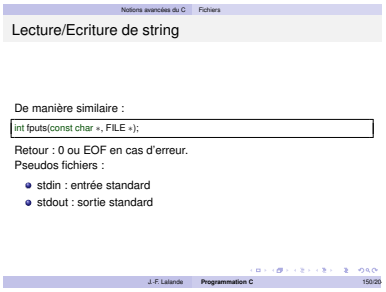
```
int fputs(const char *, FILE *);
```

1

Retour : 0 ou EOF en cas d'erreur.

Pseudos fichiers :

- stdin : entrée standard
- stdout : sortie standard



Fin de fichier / Erreurs

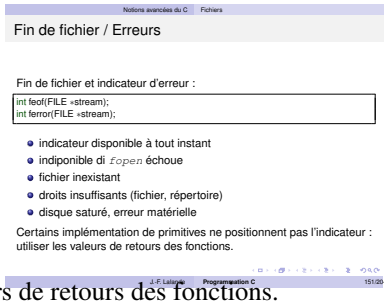
Fin de fichier et indicateur d'erreur :

```
int feof(FILE *stream);
int ferror(FILE *stream);
```

1
2

- indicateur disponible à tout instant
- indiponible di `fopen` échoue
- fichier inexistant
- droits insuffisants (fichier, répertoire)
- disque saturé, erreur matérielle

Certains implémentation de primitives ne positionnent pas l'indicateur : utiliser les valeurs de retours des fonctions.



Accès par enregistrement

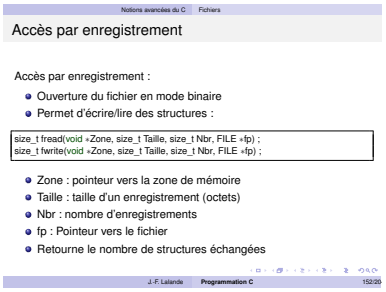
Accès par enregistrement :

- Ouverture du fichier en mode binaire
- Permet d'écrire/lire des structures :

```
size_t fread(void *Zone, size_t Taille, size_t Nbr, FILE *fp) ;
size_t fwrite(void *Zone, size_t Taille, size_t Nbr, FILE *fp) ;
```

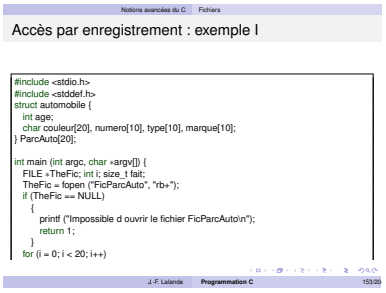
1
2

- Zone : pointeur vers la zone de mémoire
- Taille : taille d'un enregistrement (octets)
- Nbr : nombre d'enregistrements
- fp : Pointeur vers le fichier
- Retourne le nombre de structures échangées



Accès par enregistrement : exemple

```c
#include <stdio.h>
#include <stddef.h>
struct automobile {
  int age;
  char couleur[20], numero[10], type[10], marque[10];
} ParcAuto[20];

int main (int argc, char *argv[]) {
  FILE *TheFic; int i; size_t fait;
  TheFic = fopen ("FicParcAuto", "rb+");
  if (TheFic == NULL)
    {
      printf ("Impossible d ouvrir le fichier FicParcAuto\n");
      return 1;
    }
  for (i = 0; i < 20; i++)
    {
      fait = fread (&ParcAuto[i], sizeof (struct automobile), 1, TheFic);
      if (fait != 1)
  {
    printf ("Erreur lecture fichier parcauto \n");
    return 2;
  }
    }
  fclose (TheFic);
  return 0; }
```

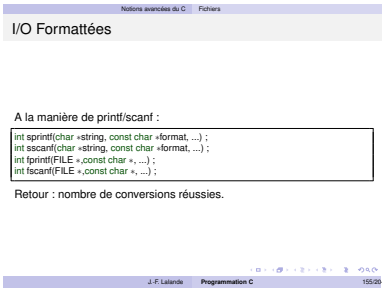


I/O Formattées

A la manière de printf/scanf :

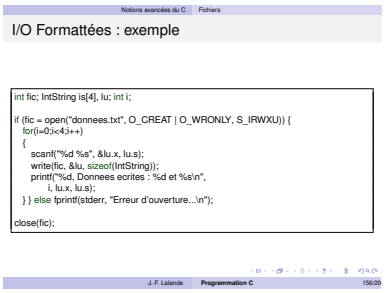
```c
int sprintf(char *string, const char *format, ...) ;
int sscanf(char *string, const char *format, ...) ;
int fprintf(FILE *,const char *, ...) ;
int fscanf(FILE *,const char *, ...) ;
```

Retour : nombre de conversions réussies.



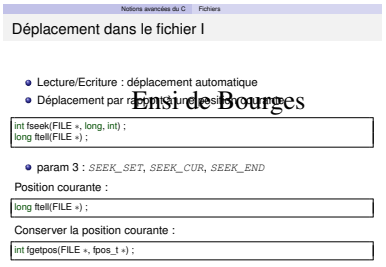
I/O Formattées : exemple

```c
int fic; IntString is[4], lu; int i;

if (fic = open("donnees.txt", O_CREAT | O_WRONLY, S_IRWXU)) {
  for(i=0;i<4;i++)
  {
    scanf("%d %s", &lu.x, lu.s);
    write(fic, &lu, sizeof(IntString));
    printf("%d, Donnees ecrites : %d et %s\n",
        i, lu.x, lu.s);
  } } else fprintf(stderr, "Erreur d'ouverture...\n");

close(fic);
```



Déplacement dans le fichier

- Lecture/Ecriture : déplacement automatique
- Déplacement par rapport à une position courante



```
int fseek(FILE *, long, int) ;
long ftell(FILE *) ;
```

– param 3 : *SEEK_SET*, *SEEK_CUR*, *SEEK_END*

Position courante :

```
long ftell(FILE *) ;
```

Conserver la position courante :

```
int fgetpos(FILE *, fpos_t *) ;
```

Modifie la position courante :

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

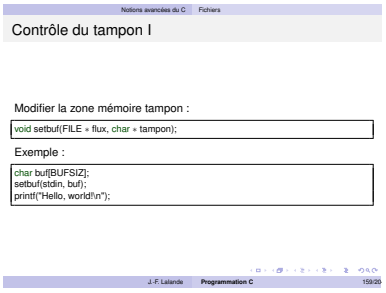
Comme pour les K7 :

```
void rewind(FILE *) ;
```

Contrôle du tampon

Modifier la zone mémoire tampon :

```
void setbuf(FILE * flux, char * tampon);
```



Exemple :

```
char buf[BUFSIZ];
setbuf(stdin, buf);
printf("Hello, world!\n");
```

Modifier la zone mémoire tampon, avec un mode :

– *_IONBF* pas de tampon
– *_IOLBF* tampon de ligne
– *_IOFBF* tampon complet

```
int setvbuf (FILE *stream, char *buf, int mode , size_t size);
```

Vider le tampon :

```
int fflush(FILE *);
```

man setbuf Les trois types de tampons disponibles sont les suivants : pas de tampons, tampons de blocs, et tampons de lignes. Quand un flux de sortie n'a pas de tampon, les données apparaissent dans le fichier destination, ou sur le terminal, dès qu'elles sont écrites. Avec les tampons par blocs, une certaine quantité de données est conservée avant d'être écrite en tant que bloc. Avec les tampons de lignes, les caractères sont conservés jusqu'à ce qu'un saut de ligne soit transmis, ou que l'on réclame une lecture sur un flux attaché au terminal (typiquement stdin). La fonction fflush(3) peut être utilisée pour forcer l'écriture à n'importe quel moment (voir fclose(3)). Normalement, tous les fichiers utilisent des tampons de blocs. Quand une première opération d'entrée-sortie se déroule sur un fichier, malloc(3) est appelée, et un tampon est créé. Si le flux se rapporte à un terminal (comme stdout habituellement) il s'agit d'un tampon de ligne. Le flux standard de sortie d'erreur stderr n'a jamais de tampon par défaut.

4.5 Pointeurs de fonctions

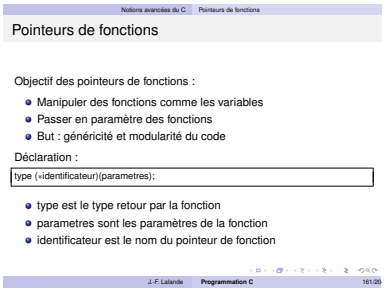
Pointeurs de fonctions

Objectif des pointeurs de fonctions :
- Manipuler des fonctions comme les variables
- Passer en paramètre des fonctions
- But : généricité et modularité du code

Déclaration :

```
type (*identificateur)(parametres);
```

- type est le type retour par la fonction
- parametres sont les paramètres de la fonction
- identificateur est le nom du pointeur de fonction



Pointeurs de fonctions : exemple

Exemple : pf pointeur de fonction (param : 2 int, retour : int)

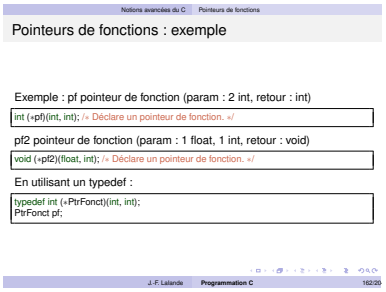
```
int (*pf)(int, int); /* Déclare un pointeur de fonction. */
```

pf2 pointeur de fonction (param : 1 float, 1 int, retour : void)

```
void (*pf2)(float, int); /* Déclare un pointeur de fonction. */
```

En utilisant un typedef :

```
typedef int (*PtrFonct)(int, int);
PtrFonct pf;
```

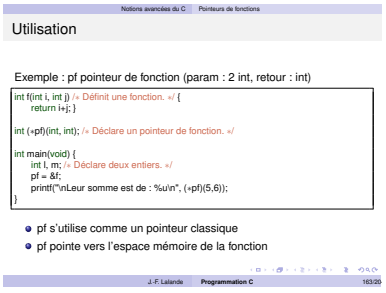


Utilisation

Exemple : pf pointeur de fonction (param : 2 int, retour : int)

```
int f(int i, int j) /* Définit une fonction. */ {
    return i+j; }

int (*pf)(int, int); /* Déclare un pointeur de fonction. */

int main(void) {
    int l, m; /* Déclare deux entiers. */
    pf = &f;
    printf("\nLeur somme est de : %u\n", (*pf)(5,6));
}
```

- pf s'utilise comme un pointeur classique
- pf pointe vers l'espace mémoire de la fonction



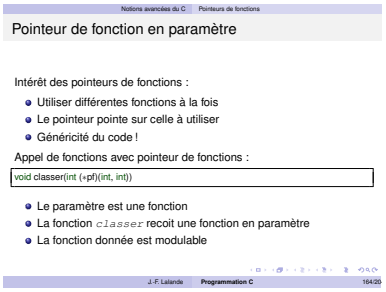
Pointeur de fonction en paramètre

Intérêt des pointeurs de fonctions :
- Utiliser différentes fonctions à la fois
- Le pointeur pointe sur celle à utiliser
- Généricité du code !

Appel de fonctions avec pointeur de fonctions :

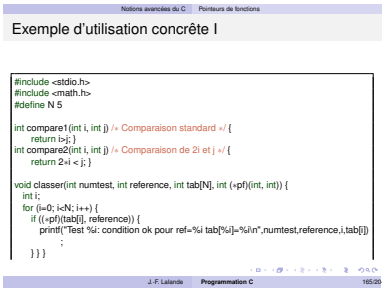
```
void classer(int (*pf)(int, int))
```

- Le paramètre est une fonction
- La fonction `classer` recoit une fonction en paramètre
- La fonction donnée est modulable



Exemple d'utilisation concrête

```c
#include <stdio.h>
#include <math.h>
#define N 5

int compare1(int i, int j) /* Comparaison standard */ {
      return i>j; }
int compare2(int i, int j) /* Comparaison de 2i et j */ {
      return 2*i < j; }

void classer(int numtest, int reference, int tab[N], int (*pf)(int, int)) {
   int i;
   for (i=0; i<N; i++) {
      if ((*pf)(tab[i], reference)) {
         printf("Test %i: condition ok pour ref=%i tab[%i]=%i\n",numtest,reference,i,tab[i]);
      } } }

int main(void) {
      int i; int tab[N];
      int (*pf)(int, int); /* Déclare un pointeur de fonction. */
      srand(time(NULL));
      for (i=0; i<N; i++)
         tab[i] = rand() % 800;
      pf = &compare1;
      classer(1,500,tab,pf);
      pf = &compare2;
      classer(2,500,tab,pf); }
```



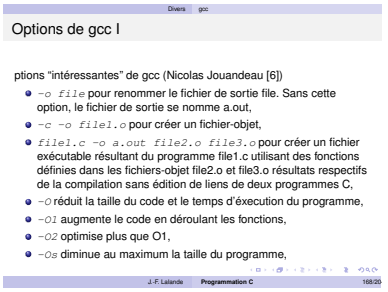
5 Divers

5.1 gcc

Options de gcc

ptions “intéressantes” de gcc (Nicolas Jouandeau [**?**])

- `-o file` pour renommer le fichier de sortie file. Sans cette option, le fichier de sortie se nomme a.out,
- `-c -o file1.o` pour créer un fichier-objet,
- `file1.c -o a.out file2.o file3.o` pour créer un fichier exécutable résultant du programme file1.c utilisant des fonctions définies dans les fichiers-objet file2.o et file3.o résultats respectifs de la compilation sans édition de liens de deux programmes C,
- `-O` réduit la taille du code et le temps d'éxecution du programme,
- `-O1` augmente le code en déroulant les fonctions,
- `-O2` optimise plus que O1,
- `-Os` diminue au maximum la taille du programme,
- `-I` répertoire ajoute le répertoire rep dans la tête de liste des répertoires contenant hypothétiquement les fichiers d'entête inclus,
- `-l` répertoire ajoute un librairie dans la phase d'édition de liens,
- `-L` répertoire ajoute le répertoire rep dans la tête de liste des répertoires contenant hypothétiquement les librairies utilisées,
- `-Dnom` réalise l'équivalent d'un **`#define`** nom,
- `-Dnom=valeur` réalise l'équivalent d'un **`#define`** nom valeur,
- `-include file` réalise l'équivalent d'un **`#include`** `file`,
- `-g` produit des informations de déboguage (il annule l'option -O),
- `-Wall` combine toute les options -W,
- `-Werror` transforme tous les warnings en erreurs,
- `-pedantic` assure la norme des warnings au strict C et C++ ISO,
- `-ansi` impose au code d'être conforme à la norme ANSI (du langage correspondant).



Bibliothèque partagée

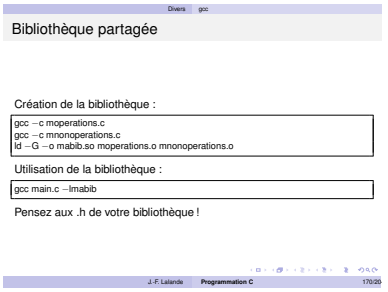
Création de la bibliothèque :

```
gcc −c moperations.c                                          1
gcc −c mnonoperations.c                                       2
ld −G −o mabib.so moperations.o mnonoperations.o              3
```

Utilisation de la bibliothèque :

```
gcc main.c −lmabib                                            1
```

Pensez aux .h de votre bibliothèque !

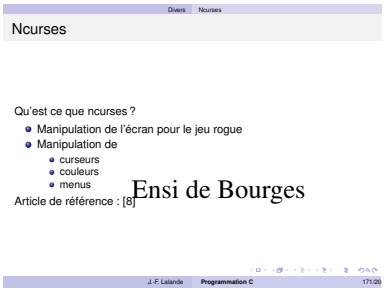


5.2 Ncurses

Ncurses

Qu'est ce que ncurses ?

- Manipulation de l'écran pour le jeu rogue
- Manipulation de
  - curseurs
  - couleurs



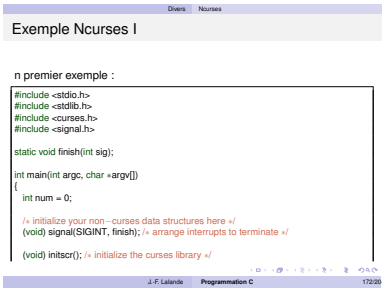
– menus

Article de référence : [**?**]

Exemple Ncurses

n premier exemple :

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

static void finish(int sig);

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

  /* initialize your non-curses data structures here */
  (void) signal(SIGINT, finish); /* arrange interrupts to terminate */

  (void) initscr(); /* initialize the curses library */
  keypad(stdscr, TRUE); /* enable keyboard mapping */
  (void) nonl(); /* tell curses not to do NL->CR/NL on output */
  (void) cbreak(); /* take input chars one at a time, no wait for \n */
  (void) echo(); /* echo input - in color */

  if (has_colors())
  {
    start_color();

    /*
     * Simple color assignment, often all we need. Color pair 0 cannot
     * be redefined. This example uses the same value for the color
     * pair as for the foreground color, though of course that is not
     * necessary:
     */
    init_pair(1, COLOR_RED, COLOR_BLACK);
    init_pair(2, COLOR_GREEN, COLOR_BLACK);
    init_pair(3, COLOR_YELLOW, COLOR_BLACK);
    init_pair(4, COLOR_BLUE, COLOR_BLACK);
    init_pair(5, COLOR_CYAN, COLOR_BLACK);
    init_pair(6, COLOR_MAGENTA, COLOR_BLACK);
    init_pair(7, COLOR_WHITE, COLOR_BLACK);
  }

  for (;;)
  {
    int c = getch(); /* refresh, accept single keystroke of input */
    attrset(COLOR_PAIR(num % 8));
    num++;
    /* process the command keystroke */
  }

  finish(0); /* we're done */
}

static void finish(int sig)
{
  endwin();
  /* do your non-curses wrapup here */
  exit(0);
}
```

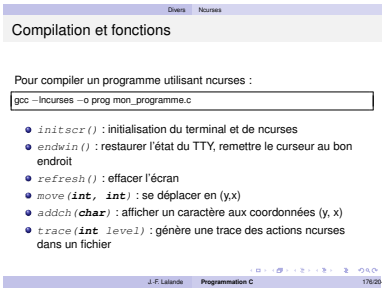


Compilation et fonctions

Pour compiler un programme utilisant ncurses :

```
gcc -lncurses -o prog mon_programme.c
```

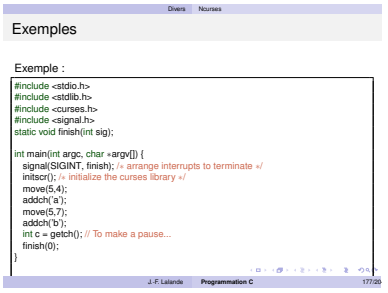
- `initscr()` : initialisation du terminal et de ncurses
- `endwin()` : restaurer l'état du TTY, remettre le curseur au bon endroit
- `refresh()` : effacer l'écran
- `move(int, int)` : se déplacer en (y,x)
- `addch(char)` : afficher un caractère aux coordonnées (y, x)
- `trace(int level)` : génère une trace des actions ncurses dans un fichier



Exemples

Exemple :

```c
#include <stdio.h>
#include <stdlib.h>
#include <curses.h>
#include <signal.h>
static void finish(int sig);

int main(int argc, char *argv[]) {
  signal(SIGINT, finish); /* arrange interrupts to terminate */
  initscr(); /* initialize the curses library */
  move(5,4);
  addch('a');
  move(5,7);
  addch('b');
  int c = getch(); // To make a pause...
  finish(0);
}

static void finish(int sig) {
  endwin();
  exit(0);
}
```



Sortir/Restaurer du mode ncurses :

```c
  addstr("Shelling out...");
  def_prog_mode(); /* save current tty modes */
  endwin(); /* restore original tty modes */
  system("sh"); /* run shell */
  addstr("returned.\n"); /* prepare return message */
  refresh(); /* restore save modes, repaint screen */
```

Ncurses : menus

Création d'un menu :

1. Initiliser ncurses menu
2. Créer les items du menu avec `new_item`
3. Créer le menu avec `new_menu`
4. "Poster" le menu avec `post_menu`
5. Rafraichir l'écran
6. Traiter les demandes de l'utilisateur dans une boucle
7. "Déposter" le menu
8. Libérer le menu avec `free_menu`

9. Libérer les items avec *free_item*
10. Terminer ncurses

```
ITEM *new_item(const char *name, const char *description);
int free_item(ITEM *item);
MENU *new_menu(ITEM **items);
int free_menu(MENU *menu);
int post_menu(MENU *menu);
int unpost_menu(MENU *menu);
```

Agir sur le menu :

- Déplacer la selection : *set_current_item*
- Récupérer la selection : *get_current_item*
- Vers le bas/haut : *menu_driver*

```
int set_current_item(MENU *menu, const ITEM *item);
ITEM *current_item(const MENU *menu);
int menu_driver(MENU *menu, int c);
```

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
#include <curses.h>
#include <menu.h>
#include <signal.h>
static void finish(int sig);

int main(int argc, char *argv[]) {
  int c;
  signal(SIGINT, finish);
  initscr();
  noecho();
  ITEM * myitem[4];
  myitem[0] = new_item("Menu", "Description 1");
  myitem[1] = new_item("Menu2", "Description 2");
  myitem[2] = new_item("Menu3", "Description 3");
  myitem[3] = NULL;
  MENU * menu = new_menu(myitem);
  post_menu(menu);
  refresh();
  c = getch(); // Pause
  set_current_item(menu, myitem[1]); // Déplacer la sélection
  refresh();
  c = getch(); // Pause
  move(5,5);
  refresh();
  printf("menu: %s", current_item(menu)->description);
  while((c = getch()) != 'x') {
    switch(c) {
      case 's':
        menu_driver(menu, REQ_DOWN_ITEM);
        break;
      case 'z':
        menu_driver(menu, REQ_UP_ITEM);
        break;
      case 10: /* Enter */
        move(20, 0);
        clrtoeol();
        mvprintw(20, 0, "Item selected is : %s",
            item_name(current_item(menu)));
        pos_menu_cursor(menu);
        break;
    }
    refresh();
  }
  unpost_menu(menu);
  free_menu(menu);
  int i;
  for(i = 0; i < 4; i++)
```

```
    free_item(myitem[i]);
  finish(0);
}

static void finish(int sig) {
  endwin();
  exit(0);
}
```

Pour compiler les menus :

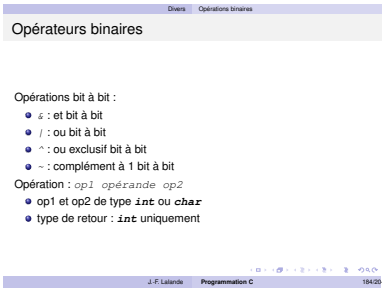
```
#include <menu.h>
gcc -lncurses -lmenu -o prog mon_programme.c
```

5.3 Opérations binaires

Opérateurs binaires

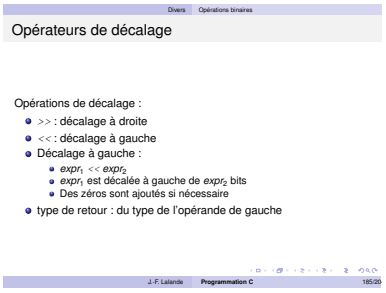
Opérations bit à bit :
- `&` : et bit à bit
- `|` : ou bit à bit
- `^` : ou exclusif bit à bit
- `~` : complément à 1 bit à bit

Opération : `op1 opérande op2`
- op1 et op2 de type **`int`** ou **`char`**
- type de retour : **`int`** uniquement



Opérateurs de décalage

Opérations de décalage :
- `>>` : décalage à droite
- `<<` : décalage à gauche
- Décalage à gauche :
  - $expr_1$ << $expr_2$
  - $expr_1$ est décalée à gauche de $expr_2$ bits
  - Des zéros sont ajoutés si nécessaire
- type de retour : du type de l'opérande de gauche

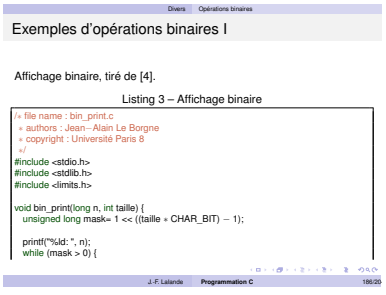


Exemples d'opérations binaires

Affichage binaire, tiré de [**?**].

Listing 3 – Affichage binaire

```
/* file name : bin_print.c
 * authors : Jean-Alain Le Borgne
 * copyright : Université Paris 8
 */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

void bin_print(long n, int taille) {
  unsigned long mask= 1 << ((taille * CHAR_BIT) - 1);

  printf("%ld: ", n);
  while (mask > 0) {
    putchar(n & mask ? '1' : '0');
    mask>>= 1;
```



```
  }
  putchar('\n');
}

void bin_print_char(char n) {
  bin_print(n, sizeof(n));
}

void bin_print_int(int n) {
  bin_print(n, sizeof(n));
}
```

Listing 4 – Affichage -1 255 char

```
/* file name : bin_print.c
 * authors : Jean-Alain Le Borgne
 * copyright : Université Paris 8
 *
 * modifications:
 * - separated in two files
 */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

extern void bin_print(long n, int taille);
extern void bin_print_char(char n);
extern void bin_print_int(int n);

int main() {
  bin_print_int(-1);
  bin_print_int(255);

  bin_print_char(-1);
  bin_print_char(255);

  return EXIT_SUCCESS;
}
```

Listing 5 – Exécution Affichage -1 255 char

```
-1: 11111111111111111111111111111111
255: 00000000000000000000000011111111
-1: 11111111
-1: 11111111
```

Listing 6 – Non, ou, et

```
/* file name : bin_print.c
 * authors : J.-F. Lalande
 * copyright : Ensi de Bourges
 *
 * modifications:
 * - separated in two files
 */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

extern void bin_print(long n, int taille);
extern void bin_print_char(char n);
extern void bin_print_int(int n);

int main() {
  int c = 17;
  int d = 204;
  printf("c:");
  bin_print_int(c);
  printf("d:");
```

```
  bin_print_int(d);
  printf("~c:");
  bin_print_int(~c);
  printf("c|d:");
  bin_print_int(c | d);
  printf("c&d:");
  bin_print_int(c & d);
  printf("c << 2:");
  bin_print_int(c << 2);
  printf("d >> 2:");
  bin_print_int(d >> 2);

  return EXIT_SUCCESS;
}
```

Listing 7 – Exécution Non, ou, et

```
c:17: 00000000000000000000000000010001
d:204: 00000000000000000000000011001100
~c:-18: 11111111111111111111111111101110
c|d:221: 00000000000000000000000011011101
c&d:0: 00000000000000000000000000000000
c << 2:68: 00000000000000000000000001000100
d >> 2:51: 00000000000000000000000000110011
```

Le masquage

Forcer à 1 les 9 bits de poids faible :

```
n = n | 0x1FFu; /* u: masque non signé /*
n |= 0x1FFu; /* écriture ramassée /*
```

Forcer à 0 les 9 bits de poids faible :

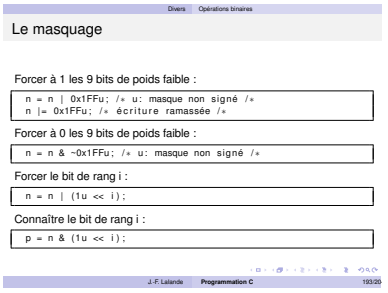
```
n = n & ~0x1FFu; /* u: masque non signé /*
```

Forcer le bit de rang i :

```
n = n | (1u << i);
```

Connaître le bit de rang i :

```
p = n & (1u << i);
```



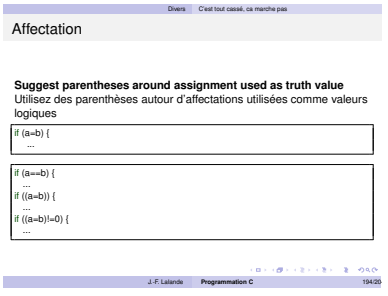
5.4 C'est tout cassé, ca marche pas

Affectation

Suggest parentheses around assignment used as truth value
Utilisez des parenthèses autour d'affectations utilisées comme valeurs logiques

```
if (a=b) {
   ...
```

```
if (a==b) {
   ...
if ((a=b)) {
   ...
if ((a=b)!=0) {
   ...
```



Format du printf

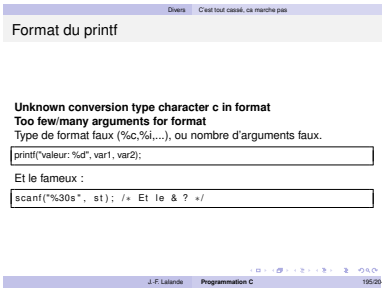
Unknown conversion type character c in format
Too few/many arguments for format

Type de format faux (%c,%i,...), ou nombre d'arguments faux.

```
printf("valeur: %d", var1, var2);
```

Et le fameux :

```
scanf("%30s", st); /* Et le & ? */
```



Return

Control reaches end of non void function

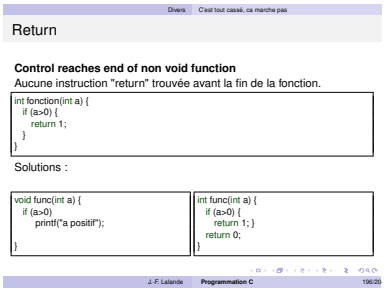
Aucune instruction "return" trouvée avant la fin de la fonction.

```
int fonction(int a) {
  if (a>0) {
    return 1;
  }
}
```

Solutions :

```
void func(int a) {
  if (a>0)
     printf("a positif");

}
```

```
int func(int a) {
 if (a>0) {
    return 1; }
 return 0;
}
```



Déclarations

Implicit declaration of function func

La fonction utilisée n'a pas été déclarée. Par défaut, ce sera :

```
int func();
```

too few/many arguments to function func

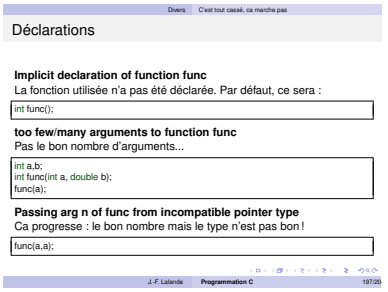
Pas le bon nombre d'arguments...

```
int a,b;
int func(int a, double b);
func(a);
```

Passing arg n of func from incompatible pointer type

Ca progresse : le bon nombre mais le type n'est pas bon !

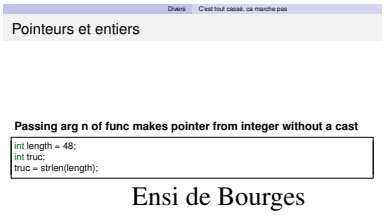
```
func(a,a);
```



Pointeurs et entiers

Passing arg n of func makes pointer from integer without a cast

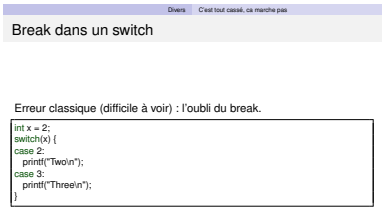
```
int length = 48;
int truc;
truc = strlen(length);
```



Break dans un switch

Erreur classique (difficile à voir) : l'oubli du break.

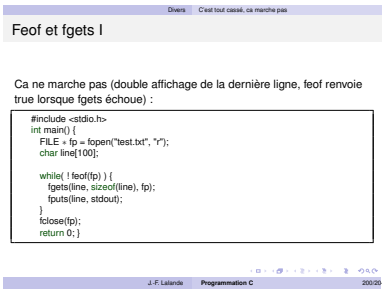
```
int x = 2;
switch(x) {
case 2:
  printf("Two\n");
case 3:
  printf("Three\n");
}
```



Feof et fgets

Ca ne marche pas (double affichage de la dernière ligne, feof renvoie true lorsque fgets échoue) :

```
#include <stdio.h>
int main() {
  FILE * fp = fopen("test.txt", "r");
  char line[100];

  while( ! feof(fp) ) {
    fgets(line, sizeof(line), fp);
    fputs(line, stdout);
  }
  fclose(fp);
  return 0; }
```



C'est mieux (même si c'est moche) :

```
#include <stdio.h>
int main() {
  FILE * fp = fopen("test.txt", "r");
  char line[100];
  while( 1 ) {
    fgets(line, sizeof(line), fp);
    if ( feof(fp) ) /* check for EOF right after fgets() */
      break;
    fputs(line, stdout); }
  fclose(fp);
  return 0; }
```

Carrément plus mieux (fgets renvoie NULL le dernier coup) :

```
#include <stdio.h>
int main()
{
  FILE * fp = fopen("test.txt", "r");
  char line[100];

  while( fgets(line, sizeof(line), fp) != NULL )
    fputs(line, stdout);
  fclose(fp);
  return 0;
}
```

Je boucle à l'infini (balèze)

```
int main(int argc, char ** argv){
int i, tab[9];
```


```
int main(int argc, char ** argv){
int i, tab[9];
int *p;
p=&tab[1];
// Remplire le tableau de 4
for(i=0;i<9;i++){
*(p+i)=4;
printf("%d ", i);
}}
```

```
0 1 2 3 4 5 6 7 4 5 6 7 4 5 6 7 4 5 6 7 4 5 6 7 4 5 6 7 4
  5 6 7 4 5 6 7 4 ...
```

```
int *p;
p=&tab[1];
// Remplire le tableau de 4
for(i=0;i<9;i++){
*(p+i)=4;
printf("%i ", i);
}}
```

```
0 1 2 3 4 5 6 7
 4 5 6 7 4 5 6 7 4 5 6 7 4 5 6 7 4 5 6 7 4 5 6 7 4 ...
```