Programmation Procédurale en langage C

Cours de 1re Année

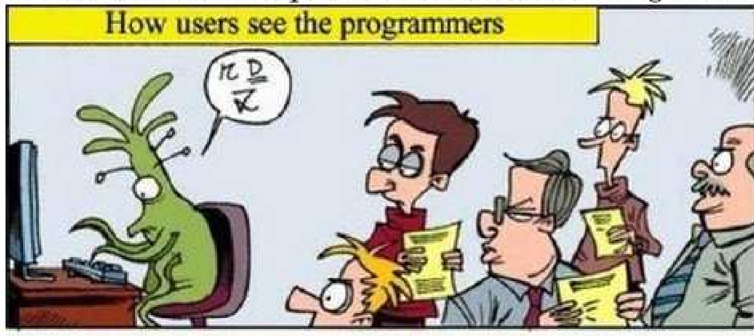
Dernière révision : mars 2020

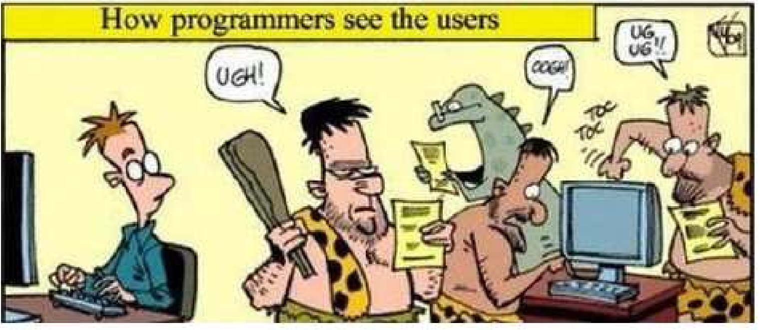
Guillaume Rivière

ESTIA est membre de la Conférence des Grandes Écoles.

LaTeX

Le monde (informatique) se divise en deux catégories…





Pourquoi l'ingénieur ESTIA apprend-t-il à programmer ?

1. Pour comprendre le monde de l'informatique
2. Pour écrire de petits programmes (10 - 100 lignes)
3. Pour écrire de gros programmes (1.000 - 10.000 lignes)
4. Pour savoir concevoir, rédiger, passer commande de projets (impossible sans connaître la réalité de la programmation)
5. Parce que l'informatique est incontournable en entreprise

Principaux champs d'application qui vous concernent :

- Électronique numérique
- Robotique / Automatique
- Commande numérique
- Objets connectés
- Système d'Information de l'Entreprise

Microcontrôleur PIC1655A (Environnement MikroC)

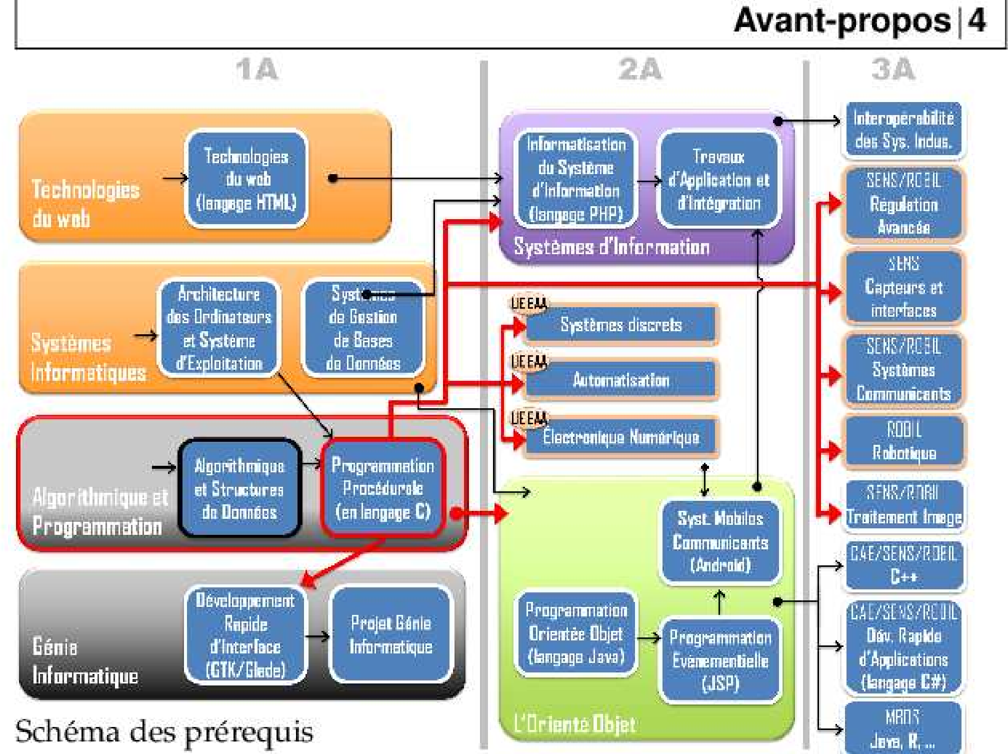


Schéma des prérequis

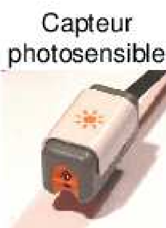
Programmation de robots LEGO MINDSTORMS NXT

- 1 micro-ordinateur intelligent
- 4 ports d'entrée (lecture)
- 4 ports de sortie (écriture)
- Différents langages possibles :
  C, C sharp, Ada, Java, MATLAB, .Net, ...

Capteur d'ultrasons

Capteur tactile

Capteur photosensible

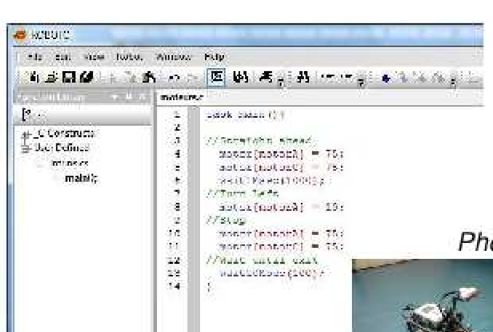


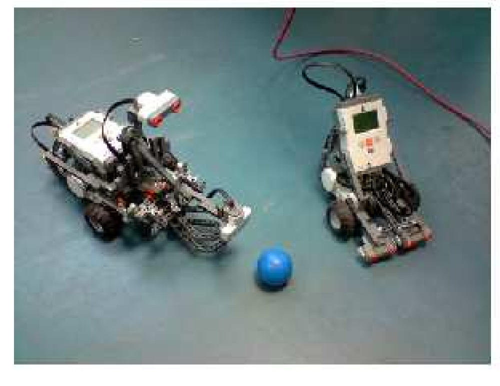
Capteur sonore

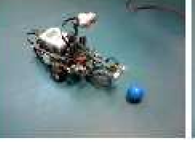
Servo-moteur

Programmation de robots LEGO MINDSTORMS NXT

- Faire coopérer 2 robots
- 1 robot explorateur
- 1 robot ramasseur



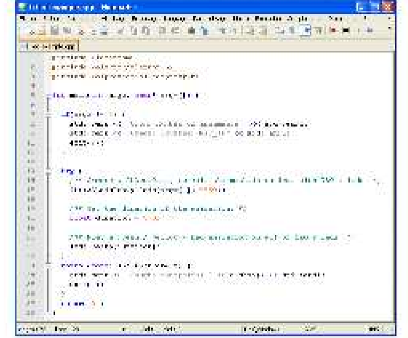




Photos projet de 2 élèves ESTIA

Programmation du robot NAO

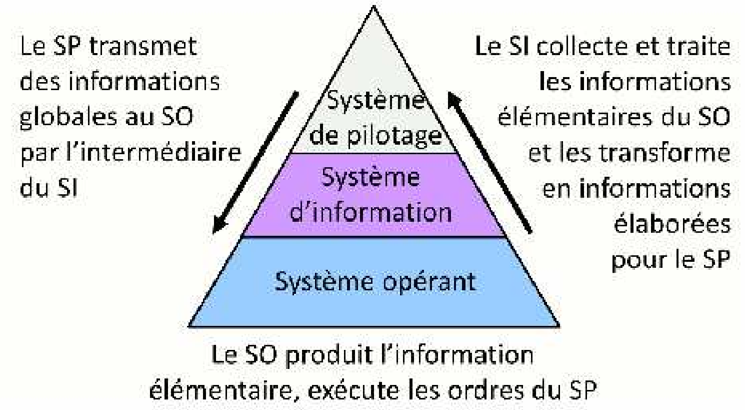
- 25 degrés de liberté
- 58 cm de hauteur
- Vision par ordinateur
- Reconnaissance vocale
- Synthèse vocale
- Mains préhensiles
- Ordinateur embarqué
- Système Linux
- Différents langages possibles : C++, Python, Java, MATLAB, .Net, ...
- SDK sur `http://www.aldebaran-robotics.com`



Programmation du robot KUKA

- KR6-ARC
- 6 axes de rotation
- Différents langages possibles : KRL (C et C++ à venir)

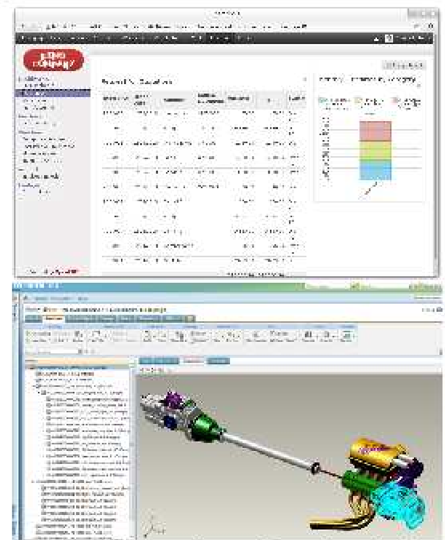
Analyse Systémique de l'Entreprise



Mots-clés du **Système d'Information**

- Intranet, Site web, Formulaires
- Réseaux, Serveurs, ...
- Bases de données, ERP (PGI)
- GPAO, SCM, GRC, SGDT, ...

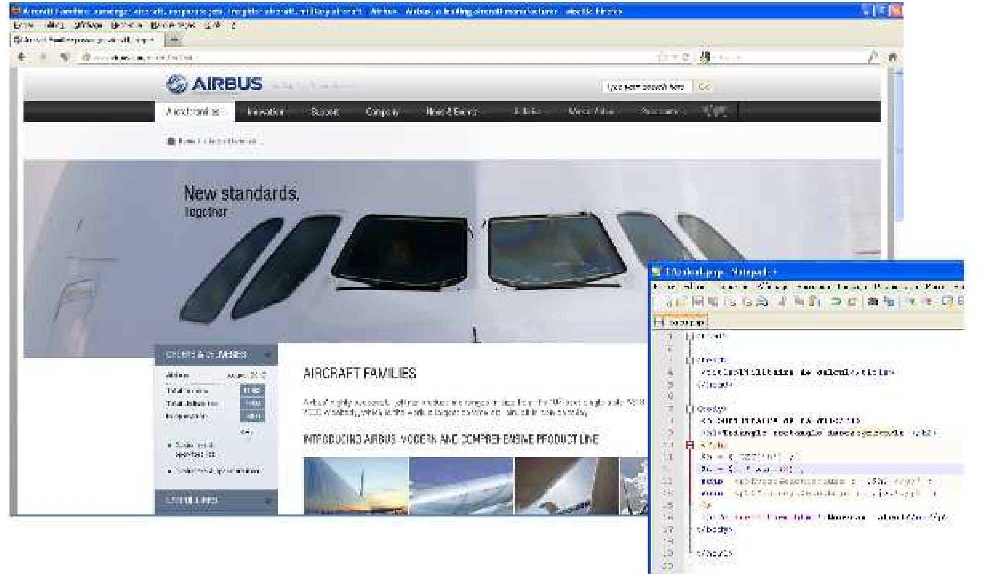
Intégration d'Applications d'Entreprise



- Progiciel de Gestion Intégré (Enterprise Resource Planning)
  - SAP, Oracle E-Business Suite, Sage X3, Compiere, OpenBravo, OpenERP (Odoo)...
- Gestion du Cycle de vie du Produit (Product Lyfecycle Management)
  - PTC Windchill
- Chaîne d'approvisionnement, Gestion de la relation client, Gestion de la Production Assistée par Ordinateur, Conception Assitée par Ordinateur, ...

Programmation web

- PHP, Javascript, Applets Java, ASP, JSP, Ajax, ...



PROGRAMMATION PROCÉDURALE EN LANGAGE C

PROGRAMME

Où trouve-t-on des programmes ?
↪ Dans les ordinateurs : oui... mais pas seulement !

- Systèmes d'exploitation, Bureautique, Navigateur web, ...
- Bases de données, Sites web, eMails, S.I. des entreprises, ...
- Téléphones, GPS, Avions, Voitures, Fusées, électroménager, ...
- Opérateurs mobiles, Banques, Assurances, ...
- Jeux vidéos, Films animés (Pixar), Météo, Géophysique, ...
- CAO (Catia, Pro-eng), Bras robotisés, Chaînes de montage, ...
- Reconnaissance vocale, MP3, ...
- ...

Programmation Procédurale en Langage C

Langage de Programmation

- Un **langage machine** est l'ensemble des mots machines compréhensibles par un processeur (jeu d'instructions).
- Un programme écrit en **assembleur** décrit (quasiment) unes à unes les opérations que doit faire le processeur (`JMP, CALL, ADD, SUB, MUL, AND, OR, NOT, CMP, MOV, PUSH, POP, JS, JI, JE, ...`). L'écriture d'un programme en assembleur reste un travail laborieux et chronophage (aussi les compétences se raréfient).
- Un **langage de programmation** permet de décrire un programme à un plus haut niveau de logique, en s'astreignant (d'une grande part) des contraintes machines. La traduction, dans un langage machine, du code ainsi écrit, est assurée par un compilateur.

Différents paradigmes de programmation existent :

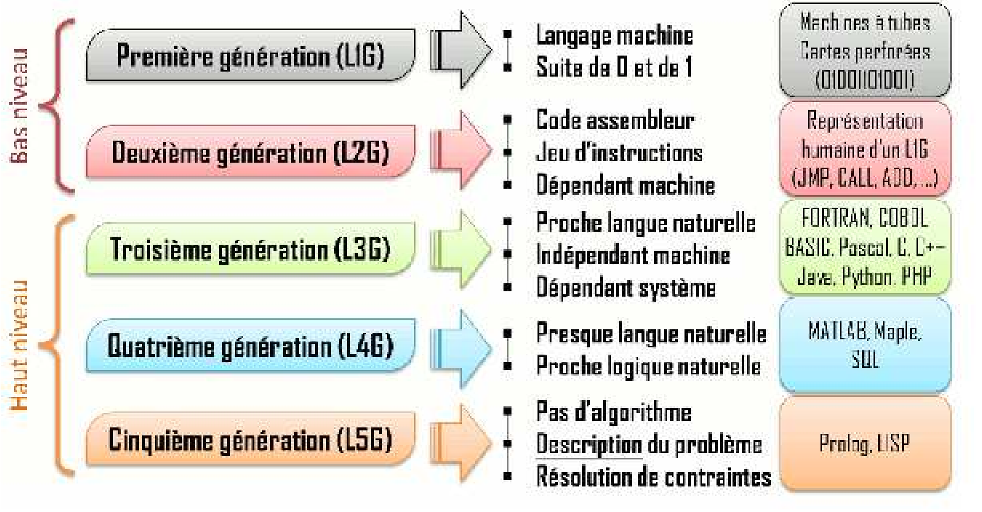
Paradigmes de programmation

- Programmation Impérative
  - Programmation Structurée
    - Programmation Procédurale : Fortran, COBOL, Pascal, C, PHP4
    - Programmation Orientée Objet : Smalltalk, C++, Java, PHP5
- Programmation Déclarative (requêtes) : SQL
  - Programmation Fonctionnelle (λ-calcul) : ML, LISP, Scheme, OCaml
  - Programmation Logique (inférence, induction) : Prolog

Encore d'autres caractéristiques :

- Multi-paradigme : Python, Javascript...

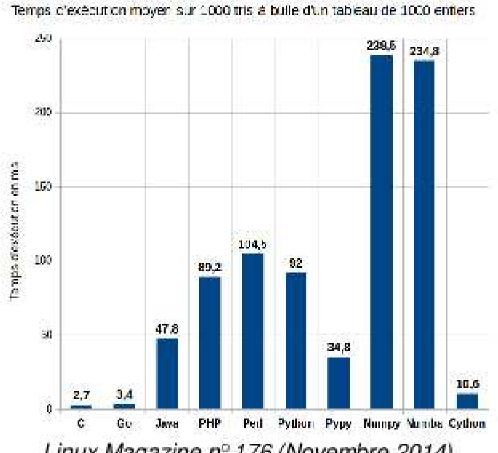
Cinq générations de langages :



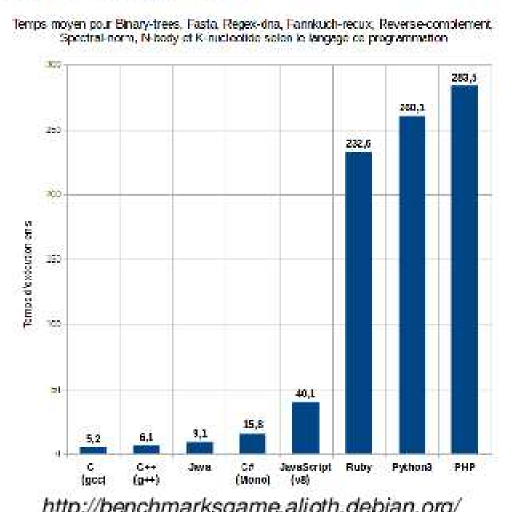
Mais que diable avoir créé autant de langages de programmation !!!

↪ Chaque langage répond à des caractéristiques propres

⇒ Exemple, la vitesse d'exécution :



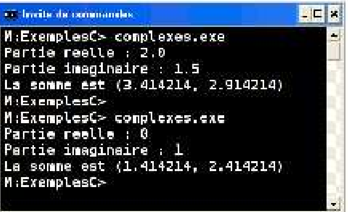
Linux Magazine n° 176 (Novembre 2014)

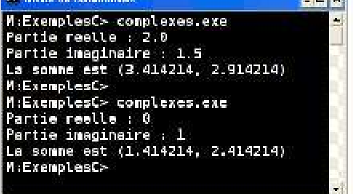


http://benchmarksgame.alioth.debian.org/

↬ **Rmq** : Plus rapide que le C : le Fortran ou l'Assembleur

à la fin de ce cours, vous saurez :





- Conduire les itérations nécessaires pour écrire un programme qui fonctionne
  - utiliser un compilateur et un IDE
  - corriger les erreurs
  - tester exécution ⇒ aboutir programme qui réponde au problème posé
- Connaître le langage C (norme ANSI 89)
  - très grande efficacité pour tout ce qui concerne le développement système
  - reste un des langages les plus utilisés en informatique industrielle
- Construire rapidement une interface graphique et la connecter pour appeler des fonctions écrites précédemment.
- Vous serez aussi capable de comprendre et de coder en Pascal, MatLab, Mupad, Basic, Fortran, Perl, . . . et bien sûr en C, langage d'application choisi car c'est un prérequis pour l'EEA

Plan

1. Introduction
2. Types de base
3. Opérateurs
4. Entrées/Sorties
5. Structures de Contrôle & Boucles
6. Fonctions
7. Pointeurs
8. Structures
9. Le préprocesseur
10. Bibliothèque standard
11. Génie logiciel et C

Plan

1. Introduction
2. Types de base
3. Opérateurs
4. Entrées/Sorties
5. Structures de Contrôle & Boucles
6. Fonctions
7. Pointeurs
8. Structures
9. Le préprocesseur
10. Bibliothèque standard
11. Génie logiciel et C

Bibliothèque en ligne de l'ESTIA

SCHOLARVox sciences BY CYBERLIBRIS

`https://univ.scholarvox.com` (liens depuis Moodle)

- **"Programmer en langage C"**, Delannoy (Ed. Eyrolles) 267 pages
- **"Mini manuel d'algorithmique et de programmation en langage C"**, Granet (Ed. Dunod) 175 pages
- **"Initiation à l'algorithmique et à la programmation en C"**, Malgouyres, Zrour & Feschet (Ed. Dunod) 333 pages
- **"Exercices et problèmes d'algorithmique"**, Flasque, Kassel & Lepoivre (Ed. Dunod) 218 pages
- **"Le guide complet du langage C"**, Delannoy (Ed. Eyrolles) 844 p.

Autres livres de référence

- **"Le langage C"**, Kernighan & Ritchie (Ed. Dunod) 280 pages : *à lire !*
- **"Méthodologie de la programmation en langage C"**, Braquelaire (Ed. Dunod) 672 pages

Polycopié disponible en ligne

- **"Le langage C"**, Garreta `http://c.developpez.com/cours/poly-c/`

→ Vous êtes issus de parcours différents

1. Jamais programmé ?
2. Déjà programmé, mais dans d'autres langages ?
   (Pascal, Fortran, Ada, OCaml, Python, VB, VBA, . . .)
3. Déjà programmé en C ?
4. Déjà programmé en C++, Java, C# ou PHP ???

↪ La programmation reste d'une manière générale un **apprentissage long et difficile** (de l'ordre de plusieurs années. . .)

↪ Comment apprendre ?
Commencer par de petits programmes **basiques** pour évoluer **plus tard** vers des plus gros *(Interfaces Graphiques, Scènes 3D, Échanges réseaux, Temps réel, Son, Image, Vidéo, Webcam, Tactile, Smartphones, GPS, SMS, emails, DB, Sécurité, Cryptage, . . .).*

⇒ Cet apprentissage commence **dès maintenant**

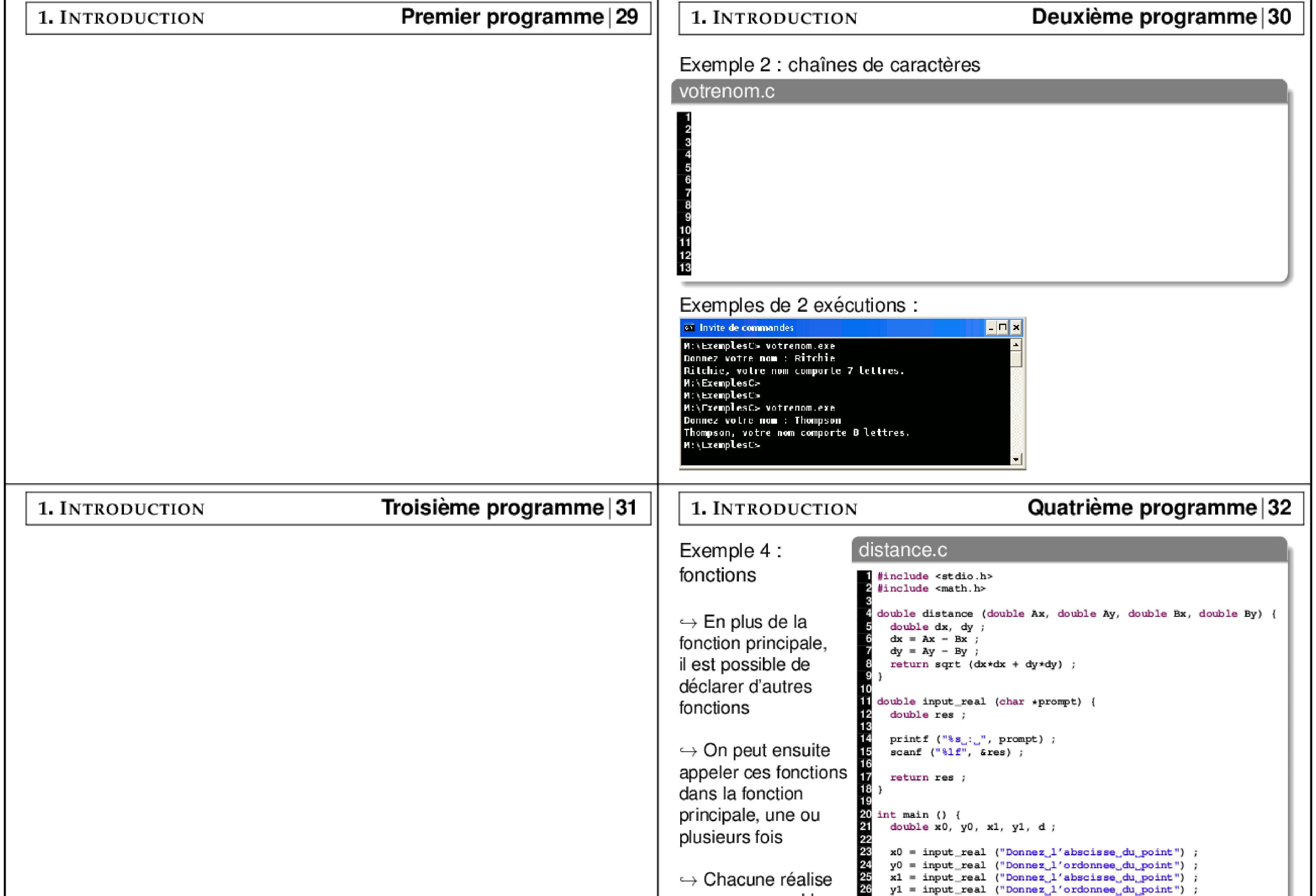
- Programmation Procédurale
  - 6h Cours + 4h TD + 12h TP
  - Contrôle continu par QCM (questions de cours)
  - Rendu de TP
  - Examen de 3 h (module A&P)
- Développement Rapide d'Interface
  - Rendu de TP

Cours :
★ Étudiants : Gómez
★ Apprentis : Rivière

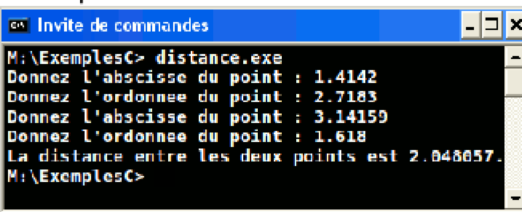
TD :
★ G1, G2 : Gómez
★ G3, G4 : Delamare
★ G5, G6 : Rivière

TP :
★ G1, G2 : Gómez, Maiza
★ G3, G4 : Delamare, Daniel
★ G5, G6 : Rivière, Camara

- Projet Génie Informatique
  - 6h TP + 10h Travail Personnel
  - Rendu : rapport, code source, vidéo de démonstration


1. INTRODUCTION
Premier programme | 29
1. INTRODUCTION
Deuxième programme | 30
Exemple 2 : chaînes de caractères
votrenom.c
Exemples de 2 exécutions :
Invite de commandes
M:\ExemplesC> votrenom.exe
Donnez votre nom : Ritchie
Ritchie, votre nom comporte 7 lettres.
M:\ExemplesC>
M:\ExemplesC>
M:\ExemplesC> votrenom.exe
Donnez votre nom : Thompson
Thompson, votre nom comporte 8 lettres.
M:\ExemplesC>
1. INTRODUCTION
Troisième programme | 31
1. INTRODUCTION
Quatrième programme | 32
Exemple 4 : fonctions
↪ En plus de la fonction principale, il est possible de déclarer d'autres fonctions
↪ On peut ensuite appeler ces fonctions dans la fonction principale, une ou plusieurs fois
↪ Chacune réalise un sous-ensemble d'instructions
distance.c
#include <stdio.h>
#include <math.h>
double distance (double Ax, double Ay, double Bx, double By) {
double dx, dy ;
dx = Ax - Bx ;
dy = Ay - By ;
return sqrt (dx*dx + dy*dy) ;
}
double input_real (char *prompt) {
double res ;
printf ("%s : ", prompt) ;
scanf ("%lf", &res) ;
return res ;
}
int main () {
double x0, y0, x1, y1, d ;
x0 = input_real ("Donnez l'abscisse du point") ;
y0 = input_real ("Donnez l'ordonnee du point") ;
x1 = input_real ("Donnez l'abscisse du point") ;
y1 = input_real ("Donnez l'ordonnee du point") ;
d = distance (x0, y0, x1, y1) ;
printf ("La distance entre les deux points est %lf.\n", d) ;
return 0 ;
}

Exemple d'exécution :


Invite de commandes
```
M:\ExemplesC> distance.exe
Donnez l'abscisse du point : 1.4142
Donnez l'ordonnee du point : 2.7183
Donnez l'abscisse du point : 3.14159
Donnez l'ordonnee du point : 1.618
La distance entre les deux points est 2.048057.
M:\ExemplesC>
```

Ordre d'exécution des lignes :

```
20 main()
21
23
11 input_real()
12
14
15
17
23
24
11 input_real()
12
14
15
17
24
25
11 input_real()
12
14
15
17
25
26
11 input_real()
12
14
15
17
26
27
4 distance()
5
6
7
8
27
29
31
```

Remarque :

↪ La fonction principale reste compréhensible

↪ Sans utiliser les fonctions `input_real()` et `distance()`, la fonction principale serait 65% plus longue (et avec +80% de lignes pleines)

Longueur de `main()`	LOC	NBLOC
avec les deux fonctions	14	10
sans les deux fonctions	23	18

LOC : Lines Of Code / NBLOC : Non Blank Lines Of Code

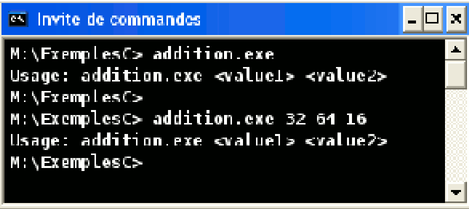
Exemple 5 : paramètres en ligne de commande

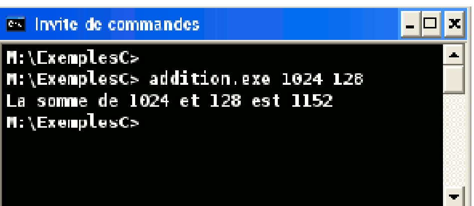
addition.c

```c
 1 #include <stdio.h>  /* Declaration de printf() et fprintf() */
 2 #include <stdlib.h> /* Declaration de exit () */
 3 
 4 int main (int argc, char *argv[]) {
 5   int a, b, c ;
 6 
 7   /* Verification du nombre d'arguments */
 8   if (argc != 3) {
 9     fprintf (stderr, "Usage: %s <value1> <value2>\n", argv[0]) ;
10     exit (1) ; /* Arret sur erreur */
11   }
12 
13   /* Recuperation des arguments */
14   a = atoi (argv[1]) ;
15   b = atoi (argv[2]) ;
16 
17   /* Calcul du resultat */
18   c = a + b ;
19 
20   /* Affichage du resultat */
21   printf ("La somme de %d et %d est %d\n", a, b, c) ;
22 
23   return 0 ; /* Sortie normale */
24 }
```

↬ **RMQ** : Commenter correctement le code est important

Exemples de 3 exécutions :


Invite de commandes
```
M:\ExemplesC> addition.exe
Usage: addition.exe <value1> <value2>
M:\ExemplesC>
M:\ExemplesC> addition.exe 32 64 16
Usage: addition.exe <value1> <value2>
M:\ExemplesC>
```


Invite de commandes
```
M:\ExemplesC>
M:\ExemplesC> addition.exe 1024 128
La somme de 1024 et 128 est 1152
M:\ExemplesC>
```

→ Le passage des arguments se fait au moment de l'appel du programme

→ Dans notre programme, nous exigeons que le nombre d'arguments soit exactement égal à 3

↬ **RMQ** : Le premier argument (`argv[0]`) est le nom du programme

↬ **RMQ** : Les arguments sont des chaînes de caractère

↪ La variable `argv[]` est un tableau de chaînes de caractères

⇒ La fonction `atoi()` calcule la valeur entière à partir d'une chaîne de caractères

Exemple 6 : une calculatrice à deux opérandes

calculatrice.c

```c
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 
 4 void usage (char *program) {
 5   fprintf (stderr, "Usage: %s <value1> <operator> <value2>\n", program) ;
 6   fprintf (stderr, "  with operator in {+,-,x,/}\n") ;
 7 }
 8 
 9 int main (int argc, char *argv[]) {
10   double v1, v2, res ;
11   char op ;
12 
13   /* Verification du nombre d'arguments */
14   if (argc != 4) {
15     usage (argv[0]) ;
16     exit (1) ; /* Arret sur erreur */
17   }
18 
19   /* Recuperation des arguments */
20   v1 = atof (argv[1]) ;
21   v2 = atof (argv[3]) ;
22   op = argv[2][0] ;
23 
24   /* Calcul du resultat */
25   switch (op) {
26   case '+':
27     res = v1 + v2 ;
28     break ;
29   case '-':
```

calculatrice.c (suite et fin)

```
30     res = v1 - v2 ;
31     break ;
32   case 'x':
33     res = v1 * v2 ;
34     break ;
35   case '/':
36     res = v1 / v2 ;
37     break ;
38   default:
39     usage (argv[0]) ;
40     exit (1) ; /* Arret sur erreur */
41   }
42
43   /* Affichage du resultat */
44   printf ("%.1f␣%c␣%.1f␣=␣%.1f\n", v1, op, v2, res) ;
45
46   return 0 ; /* Sortie normale */
47 }
```

Exemples de 4 exécutions :

Invite de commandes

```
N:\ExemplesC> calculatrice.exe 3
Usage: calculatrice.exe <value1> <operator> <value2>
  with operator in {+,-,x,/}
N:\ExemplesC> calculatrice.exe 2 plus 5
Usage: calculatrice.exe <value1> <operator> <value2>
  with operator in {+,-,x,/}
N:\ExemplesC>
```

Invite de commandes

```
N:\ExemplesC> calculatrice.exe 1024 x 32
1024.0 x 32.0 = 32768.0
N:\ExemplesC>
N:\ExemplesC> calculatrice.exe 2 / 4
2.0 / 4.0 = 0.5
N:\ExemplesC>
```


Exemple 7 : tableaux, répétitions, E/S

somme.c

```
1  #include <stdio.h>
2  #define TAILLE_MAX_TAB 20
3
4  int tab[TAILLE_MAX_TAB] ; /* Declaration d'un tableau global d'entiers */
5  int somme_tab (int n) ;   /* Declaration d'une fonction */
6
7  /**
8   * Fonction principale
9   */
10 int main () {
11   int i, n ; /* Declaration de variables locales */
12
13   /* Boucle d'entree : lecture taille du tableau tab */
14   do {
15     printf ("Donnez␣un␣entier␣entre␣2␣et␣%d␣:␣", TAILLE_MAX_TAB-1) ;
16     scanf ("%d", &n) ;
17   } while (n<2 || n>=TAILLE_MAX_TAB) ;
18
19   /* Boucle : lecture des n entiers */
20   for (i=0 ; i<n ; i++) {
21     printf ("T[%d]␣=␣", i) ;
22     scanf ("%d", tab+i) ;
23   }
24
25   /* Affichage de la somme des n entiers lus */
26   printf ("La␣somme␣des␣%d␣entiers␣est␣:␣%d\n", n, somme_tab(n)) ;
27
28   return 0 ; /* Sortie normale */
29 }
```

Ordre d'exécution des lignes de l'exemple ci-après :

```
10 main()
11
14 do
15
16 n reçoit 5
17 while
20 for i=0
21
22
20 for i=1
21
22
20 for i=2
21
22
20 for i=3
21
22
20 for i=4
21
22
26
34 somme_tab()
35
37 while n=5
38
37 while n=4
38
37 while n=3
38
```


somme.c (suite et fin)

```
30
31 /**
32  * Fonction : somme des n premiers termes de tab
33  */
34 int somme_tab (int n) {
35   int res = 0 ;      /* Declaration d'une variable initialisee avec 0 */
36
37   while (n-- > 0)    /* Ajout des n entiers */
38     res += tab[n] ;
39
40   return res ;       /* Retour du resultat */
41 }
```

```
37 while n=2
38
37 while n=1
38
40
26
28
```

Exemple d'exécution :

Invite de commandes

```
N:\ExemplesC> somme.exe
Donnez un entier entre 2 et 19 : 5
T[0] = 8
T[1] = 2
T[2] = 4
T[3] = 512
T[4] = 64
La somme des 5 entiers est : 590
N:\ExemplesC>
```

Exercice :
Modifier le programme pour calculer la moyenne (en plus de la somme)
Pour ce faire, vous créerez une nouvelle fonction :

```
double moyenne_tab (int n) ;
```

Exemple 9 : accès fichiers

copier.c

```c
#include <stdio.h>  /* fprintf(), fopen(), getc(), putc(), fclose() */
#include <stdlib.h> /* exit() */
#include <string.h> /* strcmp() */

/* Programme equivalent a la commande copy (sans les arguments optionnels).
 * NB : copy permet la copie d'un fichier dans un autre. */
int main (int argc, char *argv[]) {
  FILE *fsrc, *fdst ;
  char c ;

  /* Validation des arguments en ligne de commande */
  if (argc != 3 || strcmp(argv[1], argv[2]) == 0) {
    fprintf (stderr, "Usage: %s <file_src> <file_dst>\n", argv[0]) ;
    exit (1) ;
  }

  /* Ouverture des 2 fichiers (source et destination) */
  fsrc = fopen (argv[1], "r") ;
  if (fsrc == NULL) {
    fprintf (stderr, "Error: cannot open %s in read mode\n", argv[1]) ;
    exit (-1) ;
  }

  fdst = fopen (argv[2], "w") ;
  if (fsrc == NULL) {
    fprintf (stderr, "Error: cannot open %s in write mode\n", argv[2]) ;
    fclose (fsrc) ;
    exit (-2) ;
  }
```


copier.c (suite et fin)

```c

  /* Copie caractere par caractere */
  while ((c = getc (fsrc)) != EOF)      /* EOF = fin de fichier */
    putc (c, fdst) ;                    /* fprintf (fdst, "%c", c) ; */

  /* Fermeture des deux fichiers */
  fclose (fsrc) ;
  fclose (fdst) ;

  printf ("Done.\n") ;
  return 0 ;
}
```

Exemple d'exécution :

```
M:ExemplesC> copier.exe helloworld.c helloworld2.c
Done.
M:ExemplesC>
```

Exercice :
Modifier le programme pour compter au passage le nombre de caractères recopiés et en informer l'utilisateur.

Les espaces seront ignorés dans ce comptage :

```c
int cpt = 0 ;
/* ... */
if (c != ' ')
   cpt = cpt + 1 ;
/* ... */
printf ("Caracteres : %d\n", cpt)
;
```

3 grandes étapes : coder, compiler, exécuter

TRÈS nombreuses itérations. . . . . .

1. **CODER**
   - Écriture en langage C par un programmeur
   - Le code source définit le comportement du programme

   ⇒ Fichiers sources lisibles par un humain (ASCII)
2. **COMPILER**
   - Lecture des fichiers sources par un compilateur
   - Analyse du code C, traduction en code machine
   - Détection d'erreur (langage, bibliothèques, . . .)

   ⇒ Un fichier "exécutable" est généré (binaire)
3. **EXÉCUTER**
   - Exécution des instructions sur le processeur ⇝ *processus*
   - Tests : le programme répond-il au besoin, au cahier des charges ?
   - Tests : le programme fonctionne-t-il correctement ? Fiabilité ?

     ↪ corriger le code source, debbugage, . . .
4. **UTILISER / EXPLOITER / DIFFUSER (?)**

Exemple : Écrire un programme qui permet d'évaluer la fonction $F(x) = \frac{\sqrt{x}}{x^2-2x-3}$ pour des valeurs de x données par l'utilisateur.

0. RÉFLÉCHIR ⇝ ASD
1. CODER

evaluer.c

```c
int main () {
  double x ;

  /* Demander la valeur de x */
  printf ("Donnez x : ") ;
  scanf ("%lf", &x) ;

  /* Calculer F(x) */
  F = sqrt (x) / (x*x - 2*x - 3) ;

  /* Afficher le resultat */
  printf ("F(%f)=%d\n", x, F) ;

  return 0 ;
}
```

2. COMPILER `gcc -Wall -ansi evaluer.c`

Invite de commandes

```
M:\ExemplesC> gcc -Wall -ansi evaluer.c
evaluer.c: In function 'main':
evaluer.c:5: warning: implicit declaration of function
'printf'
evaluer.c:5: warning: incompatible implicit declaration
 of built-in function 'printf'
evaluer.c:6: warning: implicit declaration of function
'scanf'
evaluer.c:6: warning: incompatible implicit declaration
 of built-in function 'scanf'
evaluer.c:9: error: 'F' undeclared (first use in this f
unction)
evaluer.c:9: error: (Each undeclared identifier is repo
rted only once
evaluer.c:9: error: for each function it appears in.)
evaluer.c:9: warning: implicit declaration of function
'sqrt'
evaluer.c:9: warning: incompatible implicit declaration
 of built-in function 'sqrt'
M:\ExemplesC>
```

- Warnings ⇒ Les fonctions `printf()`, `scanf()` et `sqrt()` n'ont pas été déclarées
- Error ⇒ La variable `F` n'a pas été déclarée

1. CODER

→ Inclusion du fichier d'entête `stdio.h` où les fonctions `printf()` et `scanf()` sont déclarées
→ Inclusion du fichier d'entête `math.h` où la fonction `sqrt()` est déclarée
→ Enfin, déclaration de la variable `F`

evaluer.c

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

int main () {
  double x ;
  double F ;

  /* Demander la valeur de x */
  printf ("Donnez x : ") ;
  scanf ("%lf", &x) ;

  /* Calculer F(x) */
  F = sqrt (x) / (x*x - 2*x - 3) ;

  /* Afficher le resultat */
  printf ("F(%f)=%d\n", x, F) ;

  return 0 ;
}
```

2. COMPILER

Invite de commandes

```
M:\ExemplesC> gcc -Wall -ansi evaluer.c
evaluer.c: In function 'main':
evaluer.c:16: warning: format '%d' expects type 'int',
but argument 3 has type 'double'
/tmp/ccymQd6N.o: In function `main':
evaluer.c:(.text+0x51): undefined reference to `sqrt'
collect2: ld returned 1 exit status
M:\ExemplesC>
```

- Warning ⇒ Ligne 16 : le type attendu pour le 3ᵉ argument de `printf()` est `int` mais `F` est de type `double`

1. CODER

→ Changement du `%d` en `%f` dans la chaîne de formatage de `printf()`

evaluer.c

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

int main () {
  double x ;
  double F ;

  /* Demander la valeur de x */
  printf ("Donnez x : ") ;
  scanf ("%lf", &x) ;

  /* Calculer F(x) */
  F = sqrt (x) / (x*x - 2*x - 3) ;

  /* Afficher le resultat */
  printf ("F(%f)=%f\n", x, F) ;

  return 0 ;
}
```

2. COMPILER

Invite de commandes

```
M:\ExemplesC> gcc -Wall -ansi evaluer.c
/tmp/ccmY3BKo.o: In function `main':
evaluer.c:(.text+0x51): undefined reference to `sqrt'
collect2: ld returned 1 exit status
M:\ExemplesC>
```

- Échec lors de l'édition des liens ⇒ Le code machine de la fonction `sqrt()` est introuvable

  ↪ Inclure `<math.h>` ne suffit pas

2. (RE)COMPILER

→ Ajout de l'option de compilation `-lm` pour lier la bibliothèque mathématique qui fournit, parmi d'autres, le code machine de la fonction `sqrt()`

↪ `gcc -Wall -ansi -lm evaluer.c`

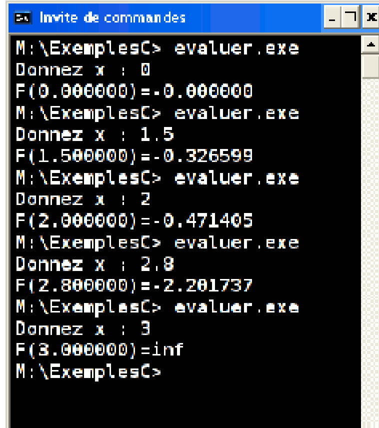
Invite de commandes

```
M:\ExemplesC> gcc -Wall -ansi -lm evaluer.c
M:\ExemplesC>
```

❸ EXÉCUTER

→ Le nom par défaut de l'exécutable généré par GCC est `a.exe`

↪ Soit renommer le fichier, soit ajouter l'option `-o` à la commande de compilation : `gcc -Wall -ansi -lm evaluer.c -o evaluer.exe`



- Problème lorsque l'utilisateur entre $x = 3$

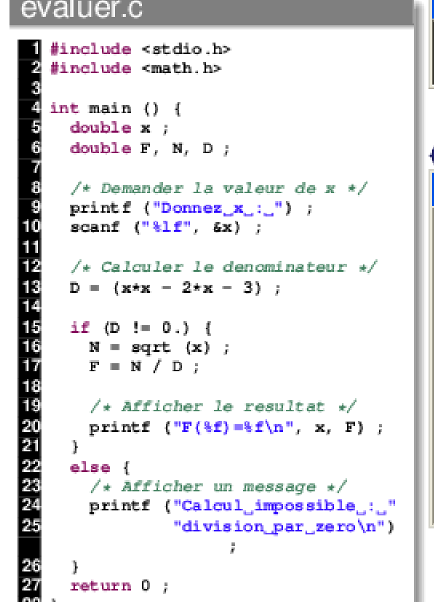
↪ En effet, 3 est une racine du polynôme $x^2 - 2x - 3$ ce qui entraîne une division par 0

⇒ Solution 1 :
Vérifier au préalable que x n'est pas une racine du polynôme
(programmation défensive)

⇒ Solution 2 :
Évaluer le dénominateur et vérifier qu'il est non nul avant de procéder à la division

❶ CODER

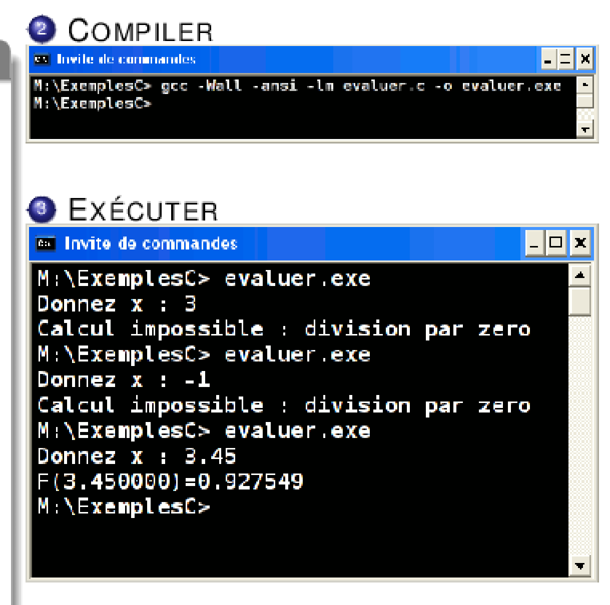
evaluer.c



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

int main () {
  double x ;
  double F, N, D ;

  /* Demander la valeur de x */
  printf ("Donnez␣x␣:␣") ;
  scanf ("%lf", &x) ;

  /* Calculer le denominateur */
  D = (x*x - 2*x - 3) ;

  if (D != 0.) {
    N = sqrt (x) ;
    F = N / D ;

    /* Afficher le resultat */
    printf ("F(%f)=%f\n", x, F) ;
  }
  else {
    /* Afficher un message */
    printf ("Calcul␣impossible␣:␣"
            "division␣par␣zero\n")
            ;
  }
  return 0 ;
}
```

❷ COMPILER

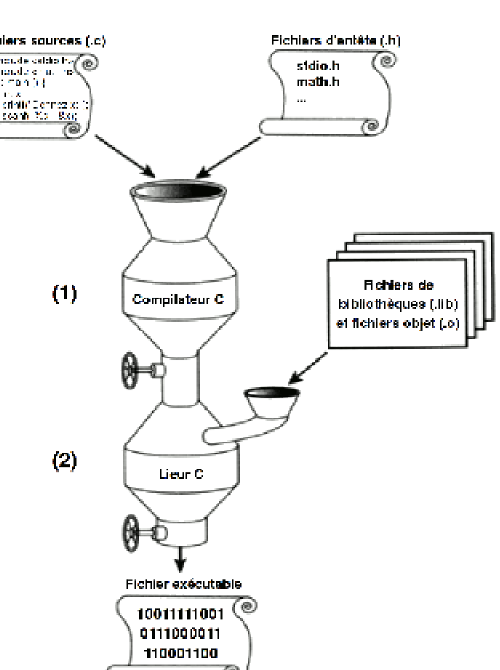
❸ EXÉCUTER



■fin

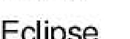
La compilation C comporte 3 phases principales

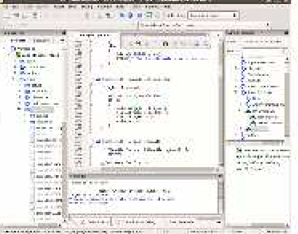
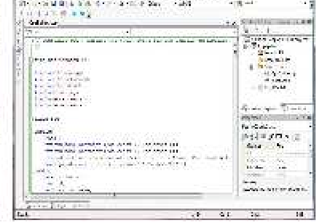
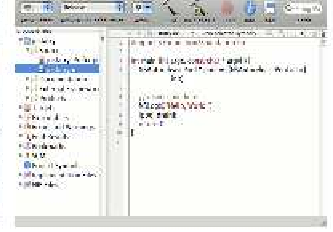
- Pour chaque fichier source
  - **(1a)** Traitement du fichier source par le préprocesseur C
  - **(1b)** Compilation et génération du code cible
- **(2)** Édition des liens
  - Réunit les fichiers cibles,
  - puis ajoute les fonctions issues des bibliothèques,
  - pour créer un seul fichier "exécutable" autonome



Code, Compilation, Exécution : tout dans la même fenêtre

Principaux IDE **gratuits** permettant de développer en C :
(Integrated Development Environment)

- Code::Blocks
- Visual C++ ← (version "Community" gratuite)
- wxDev-c++
- Xcode
- Eclipse  ← ESTIA

En savoir plus : `http://c.developpez.com/compilateurs/`

ECLIPSE

- Initié par IBM (Initialement dédié à Java)
- Passage en logiciel libre en 2001 (Eclipse Foundation)
- IDE extensible, universel et polyvalent
- Multi-plateforme (Windows, Linux, Mac, . . .)
- Langages : Java, C/C++, C#, Ada, Python, Pascal, Cobol. . .

Version	Date
1.0	nov. 2001
3.0	juin 2004
3.3 Europa	juin 2007
Ganymède	juin 2008
3.5 Galileo	juin 2009
3.6 Helios	juin 2010

Version	Date
3.7 Indigo	juin 2011
4.2 Juno	juin 2012
4.3 Kepler	juin 2013
4.4 Luna	juin 2014
4.5 Mars	juin 2015
4.6 Neon	juin 2016

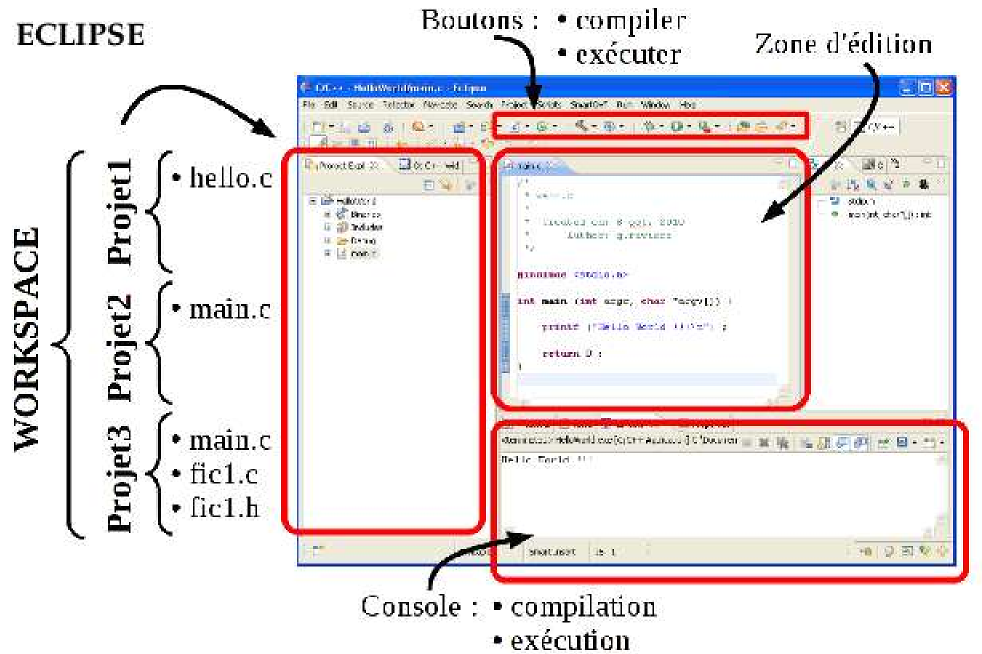
Version	Date
4.7 Oxygen	juin 2017
4.8 Photon	juin 2018
4.9	sept 2018
4.10	déc 2018
4.11	mars 2019
4.12	juin 2019

Eclipse propose une douzaine de paquetages :

- **CDT (C/C++ Development Tools)**
- JDT (Java Development Tools)
- ADT (Android Development Tools)
- . . .

`http://www.eclipse.org`

ECLIPSE



→ Ce n'est pas en regardant un forgeron que l'on apprend à forger (*canfere* un proverbe bien connu)

→ Il en va de même en programmation :

- Ce n'est pas parce que vous aurez **compris** le code d'un programme (exemples, corrections, . . .) que vous saurez **programmer**
- Ce n'est pas non plus en **recopiant** des programmes qu'on apprend à programmer (même si c'est un bon début)
- Vous saurez programmer lorsque vous serez capables d'écrire un programme "*from scratch*", c.à.d. en partant d'un **fichier vide**
    - Commencer par de **petits programmes** est le mieux pour apprendre (Par exemple réécrire `helloworld.c` depuis zéro)
    - Essayer de **modifier et/ou compléter** le comportement des programmes donnés en exemple est aussi un bon réflexe pour apprendre

Plan

1. Introduction
2. Types de base
3. Opérateurs
4. Entrées/Sorties
5. Structures de Contrôle & Boucles
6. Fonctions
7. Pointeurs
8. Structures
9. Le préprocesseur
10. Bibliothèque standard
11. Génie logiciel et C

Introduction

Variables

Nombres entiers

Proposition :
La représentation (en virgule flottante) des nombres réels (sur un espace discret) selon la norme IEEE754 est :

- plus dense dans l'intervale [0 ; 1] (densité très forte)
- que dans l'intervalle [1 ; 2] (densité moyenne)
- et que dans l'intervalle [2 ; $+\infty$] (densité faible).

Lemme 1 :
$\forall$ m : $1 \leq 1{,}m < 2$

Lemme 2 :
$E < 127 \Leftrightarrow 2^{E-127} < 1$
Donc si $E < 127$ alors $\forall$ m : $1{,}m \times 2^{E-127} < 1{,}m$

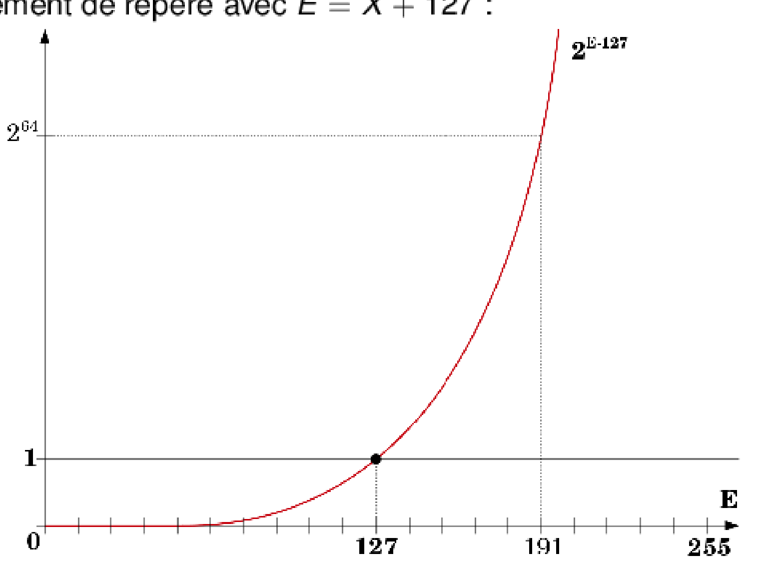
Lemme 3 :
$E > 127 \Leftrightarrow 2^{E-127} > 1$
Donc si $E > 127$ alors $\forall$ m : $1{,}m \times 2^{E-127} > 1{,}m$

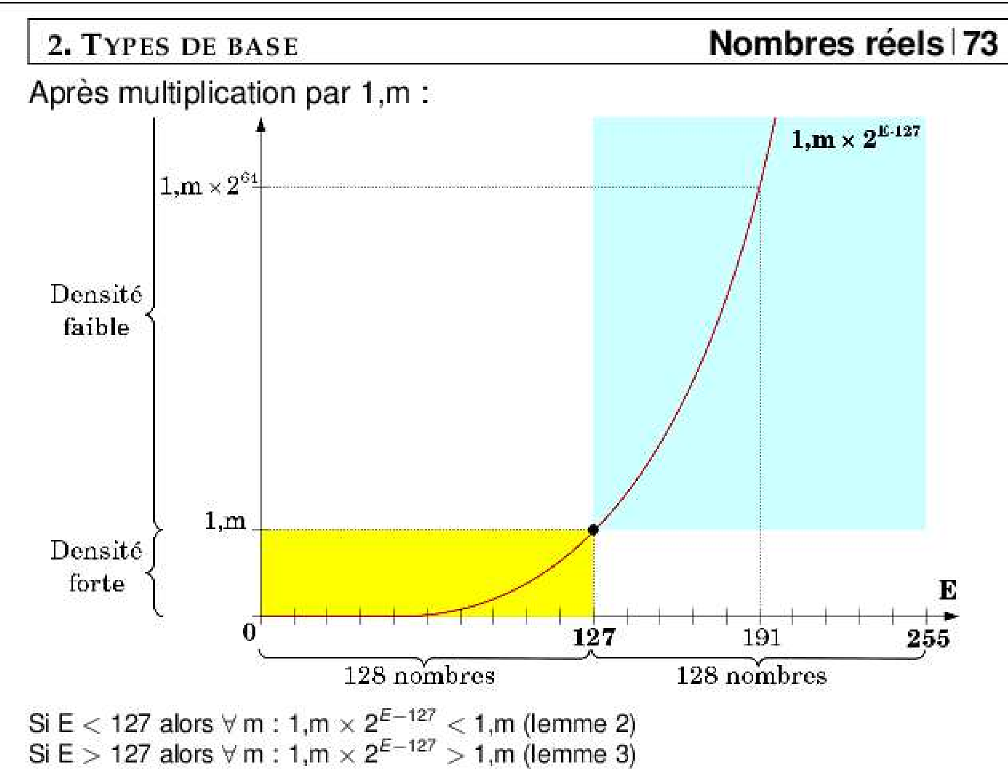
Démonstration :

Considérons la fonction 2^X :

Bien entendu, cette courbe n'est pas à l'échelle !
$2^{64} = 1,84467441 \times 10^{19} = 18\,446,7441$ millions de milliards
$2^{127} = 1,70141183 \times 10^{38}$

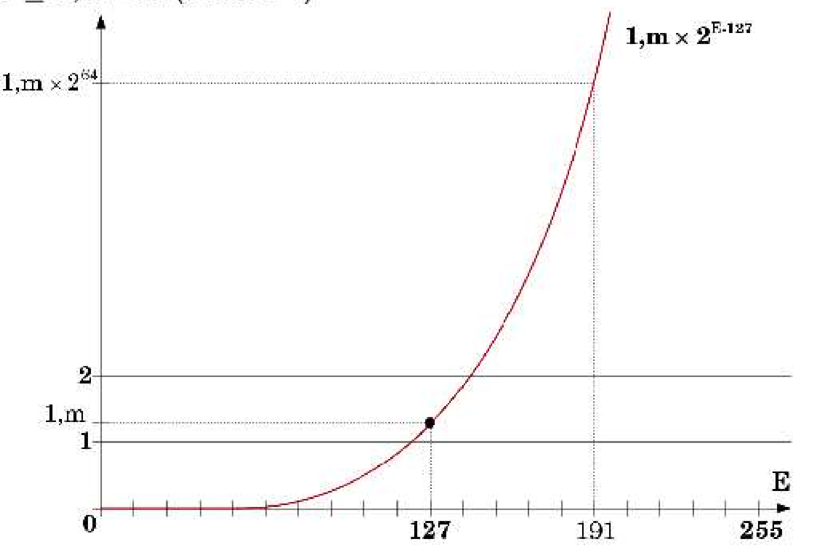
Après changement de repère avec $E = X + 127$:

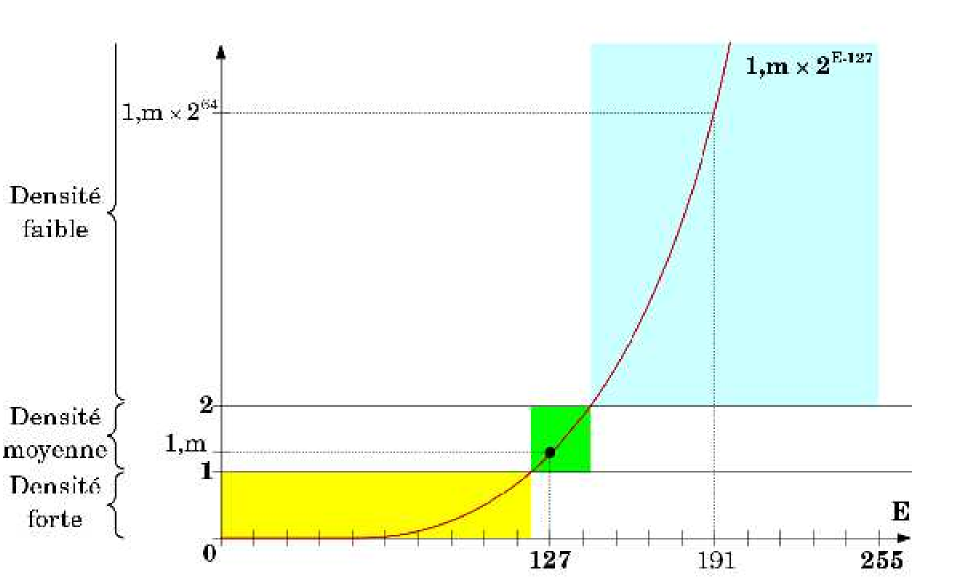




Si E $<$ 127 alors $\forall$ m : 1,m $\times$ 2^{E-127} $<$ 1,m (lemme 2)
Si E $>$ 127 alors $\forall$ m : 1,m $\times$ 2^{E-127} $>$ 1,m (lemme 3)

Or $\forall$ m : 1 $\leq$ 1,m $<$ 2 (lemme 1)





mantisse sur 23 bits : $2^{23} = 8\,388\,608 \approx 8,4 \times 10^6$ (8,4 millions)
mantisse sur 52 bits : $2^{52} \approx 4,5 \times 10^{15} \approx 4\,503\,599 \times 10^9$ (4,5 millions de milliards)

1. Introduction
2. Types de base
3. Opérateurs
4. Entrées/Sorties
5. Structures de Contrôle & Boucles
6. Fonctions
7. Pointeurs
8. Structures
9. Le préprocesseur
10. Bibliothèque standard
11. Génie logiciel et C

- À chaque variable est associé un espace mémoire (d'une taille déterminée par le type → `sizeof()`). Les affectations permettent tout simplement d'aller écrire dans cet espace mémoire.

	Pseudo-code	Code C	En mémoire
(1)	a : entier	`int a ;`	a: [ ]
(2)	b : entier	`int b ;`	a: [ ] b: [ ]
(3)	$b \leftarrow 4$	`b = 4 ;`	a: [ ] b: [4]
(4)	$a \leftarrow b$	`a = b ;`	a: [4] b: [4]
(5)	$a \leftarrow a + 2$	`a = a + 2 ;`	a: [6] b: [4]

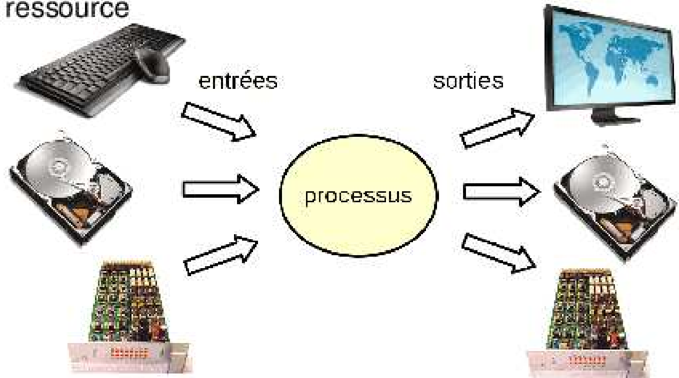
	Pseudo-code	Code C	En mémoire
(1)	a, b, tmp : entier	`int a, b, tmp ;`	a: [ ], b: [ ], tmp: [ ]
(2)	$a \leftarrow 1$	`a = 1 ;`	a: 1, b: [ ], tmp: [ ]
(3)	$b \leftarrow 6$	`b = 6 ;`	a: 1, b: 6, tmp: [ ]
(4)	$a \leftrightarrow b$	`/* Unknown ? */`	
(5)	$tmp \leftarrow a$	`tmp = a ;`	a: 1, b: 6, tmp: 1
(6)	$a \leftarrow b$	`a = b ;`	a: 6, b: 6, tmp: 1
(7)	$b \leftarrow tmp$	`b = tmp ;`	a: 6, b: 1, tmp: 1

Plan

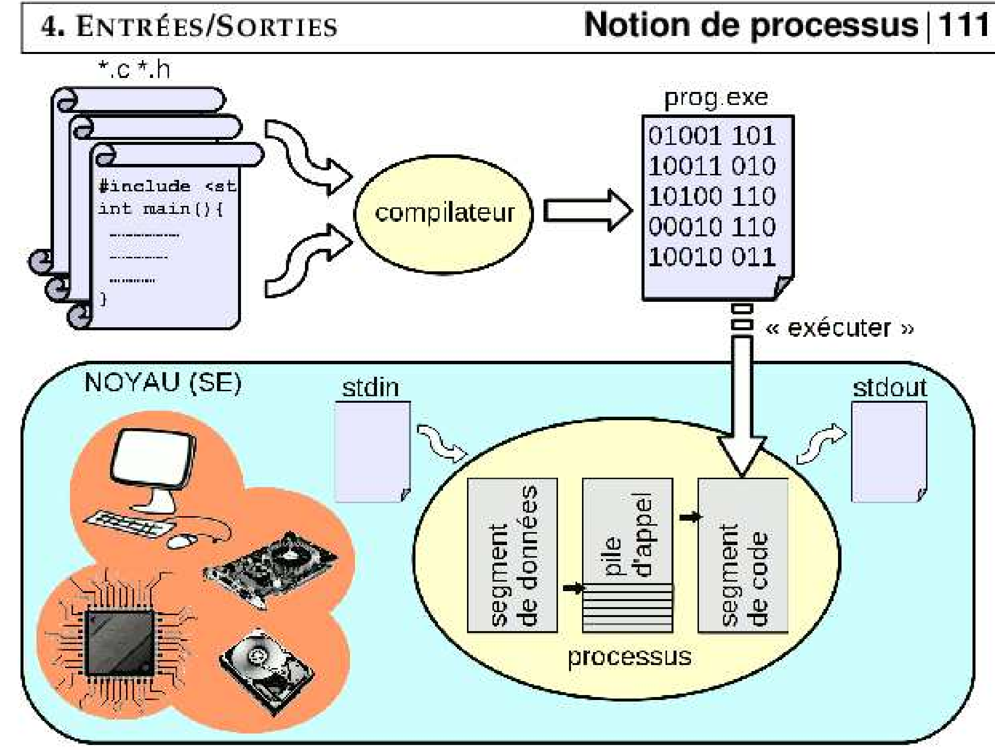
1. Introduction
2. Types de base
3. Opérateurs
4. Entrées/Sorties
5. Structures de Contrôle & Boucles
6. Fonctions
7. Pointeurs
8. Structures
9. Le préprocesseur
10. Bibliothèque standard
11. Génie logiciel et C

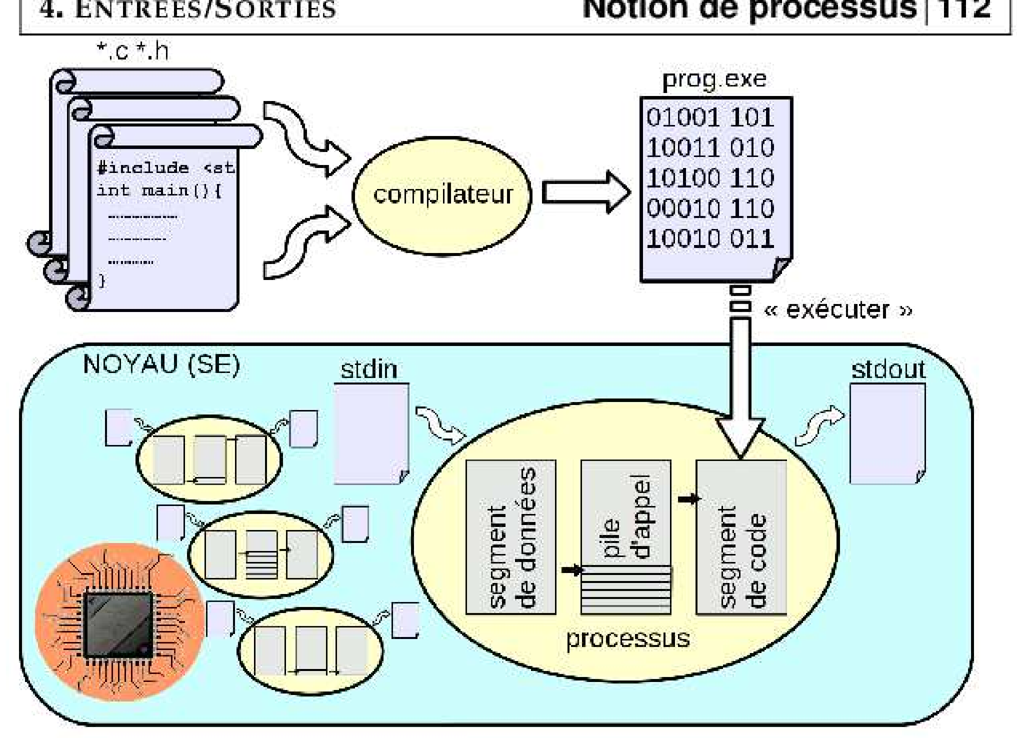
Les entrées/sorties (**Input/Ouput, I/O**) sont les échanges d'informations entre un **processus** et les ressources système (matérielles et logicielles)

- Les *entrées* sont les données envoyées par une ressource à destination d'un processus
- Les *sorties* sont les données envoyées par un processus à destination d'une ressource



Processus = un programme en cours d'exécution *(définition approximative)*





4.1 Un flux de sortie distinct pour les erreurs

En fait, par défaut, un deuxième flux de sortie est toujours créé pour chaque processus.

stdin
stdout
processus
stderr

→ Ce "canal" est destiné à recevoir les messages d'erreur, ou d'avertissement, pour les distinguer des messages dits "normaux"

→ Par défaut, le flux d'erreur est dirigé vers la console

→ Mais des mécanismes existent (nous en verrons un juste après) pour modifier la destination de ces flux et pouvoir les dissocier

4.2 Les redirections

Au moment de l'appel d'un programme, la destination habituelle des sorties standards (`stdout` et `stderr`) peut être changée

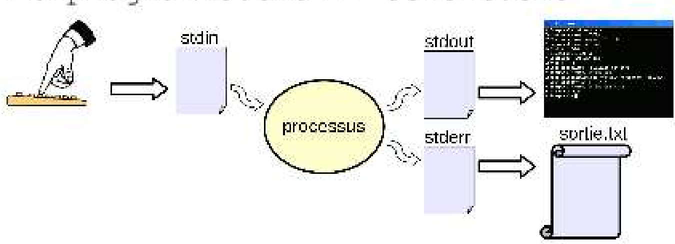
1. Récupérer la sortie standard en la redirigeant dans un fichier

Exemple : `monprogramme.exe > sortie.txt`

sortie.txt
stdin
stdout
processus
stderr

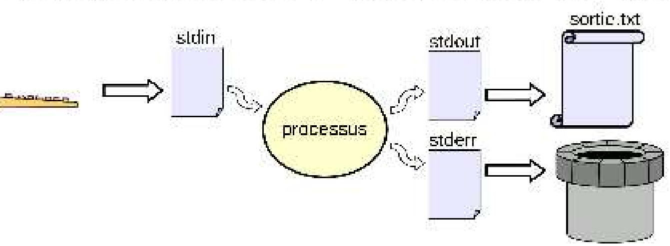
❷ Récupérer la sortie d'erreur en la redirigeant dans un fichier

Exemple : `monprogramme.exe 2> sortie.txt`

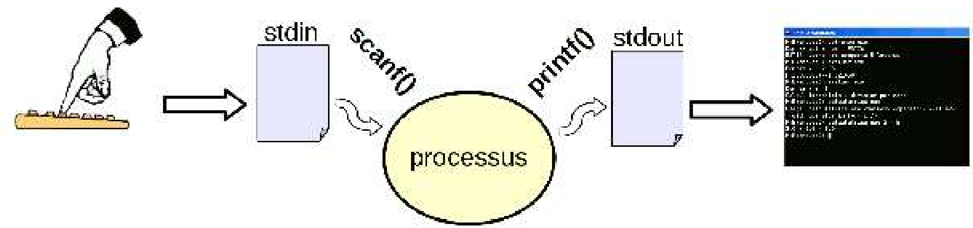


❸ Ignorer une des sorties en la redirigeant vers NUL

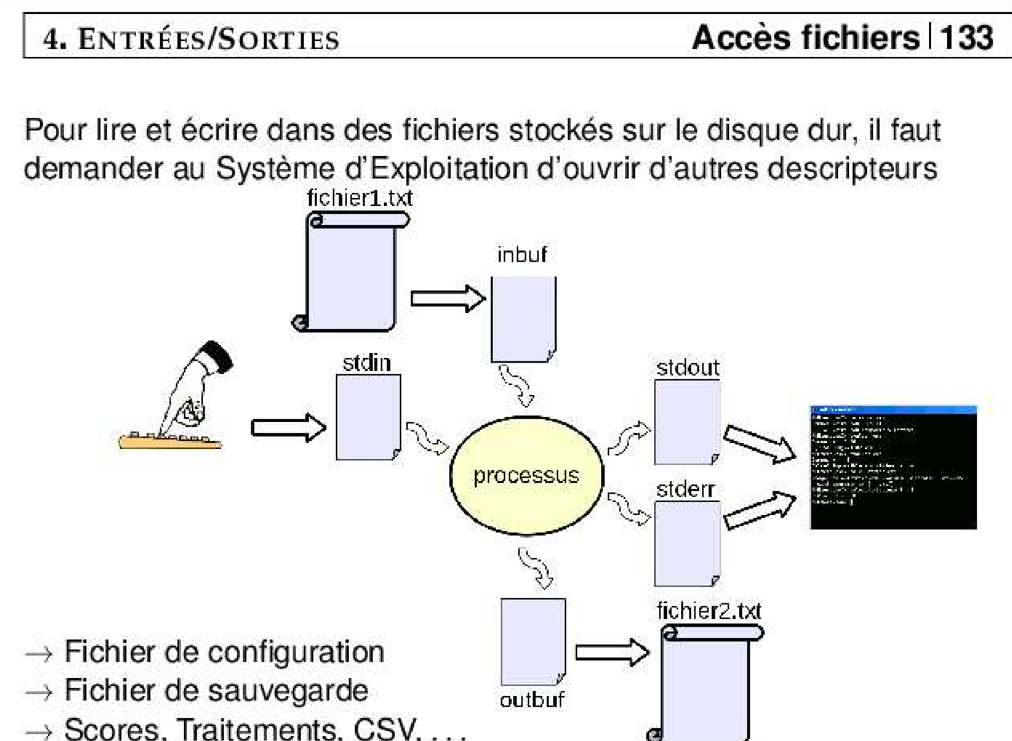
Exemple : `monprogramme.exe > sortie.txt 2> NUL`



La fonction `scanf()` lit des caractères en provenance de l'**entrée standard** (`stdin`). Sauf cas particuliers, l'entrée standard est le clavier de l'ordinateur.



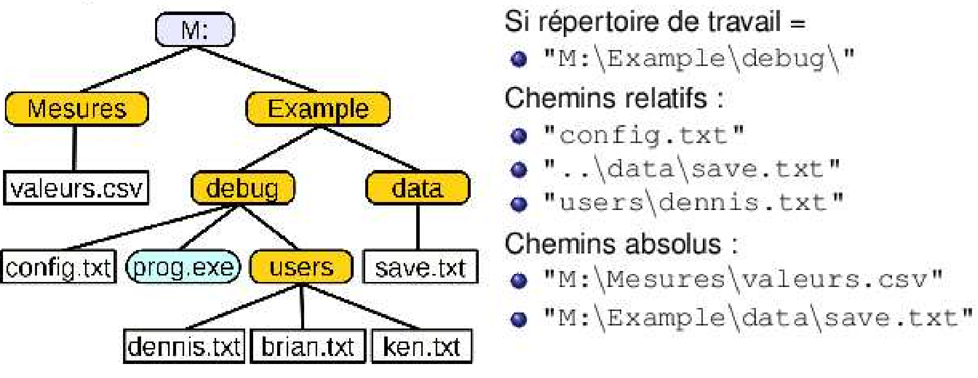
Pour lire et écrire dans des fichiers stockés sur le disque dur, il faut demander au Système d'Exploitation d'ouvrir d'autres descripteurs



→ Fichier de configuration
→ Fichier de sauvegarde
→ Scores, Traitements, CSV, . . .

→ Le paramètre `path` est une chaîne de caractères décrivant le nom du fichier à ouvrir (précédé du chemin d'accès sur le disque)

Exemple :



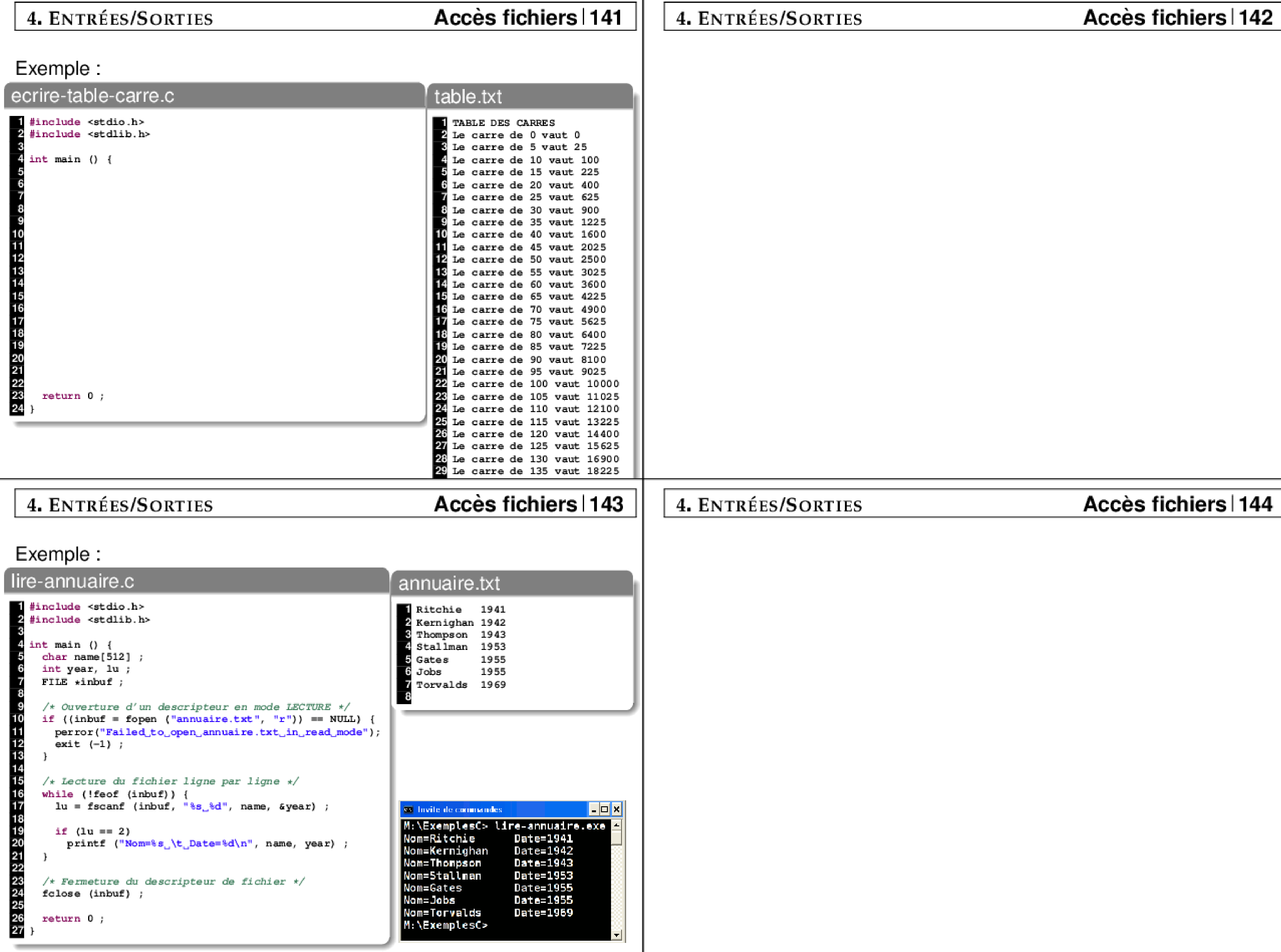
Si répertoire de travail =
- `"M:\Example\debug\"`

Chemins relatifs :
- `"config.txt"`
- `"..\data\save.txt"`
- `"users\dennis.txt"`

Chemins absolus :
- `"M:\Mesures\valeurs.csv"`
- `"M:\Example\data\save.txt"`

↬ **RMQ** : Le répertoire de travail sera le répertoire depuis lequel l'exécution du programme sera lancée

↬ **RMQ** : Le caractère de séparation est / sous Unix, Linux et Mac



4. Entrées/Sorties
Accès fichiers | 141
Exemple :
ecrire-table-carre.c
#include <stdio.h>
#include <stdlib.h>
int main () {
return 0 ;
}
table.txt
TABLE DES CARRES
Le carre de 0 vaut 0
Le carre de 5 vaut 25
Le carre de 10 vaut 100
Le carre de 15 vaut 225
Le carre de 20 vaut 400
Le carre de 25 vaut 625
Le carre de 30 vaut 900
Le carre de 35 vaut 1225
Le carre de 40 vaut 1600
Le carre de 45 vaut 2025
Le carre de 50 vaut 2500
Le carre de 55 vaut 3025
Le carre de 60 vaut 3600
Le carre de 65 vaut 4225
Le carre de 70 vaut 4900
Le carre de 75 vaut 5625
Le carre de 80 vaut 6400
Le carre de 85 vaut 7225
Le carre de 90 vaut 8100
Le carre de 95 vaut 9025
Le carre de 100 vaut 10000
Le carre de 105 vaut 11025
Le carre de 110 vaut 12100
Le carre de 115 vaut 13225
Le carre de 120 vaut 14400
Le carre de 125 vaut 15625
Le carre de 130 vaut 16900
Le carre de 135 vaut 18225
4. Entrées/Sorties
Accès fichiers | 142
4. Entrées/Sorties
Accès fichiers | 143
Exemple :
lire-annuaire.c
#include <stdio.h>
#include <stdlib.h>
int main () {
char name[512] ;
int year, lu ;
FILE *inbuf ;
/* Ouverture d'un descripteur en mode LECTURE */
if ((inbuf = fopen ("annuaire.txt", "r")) == NULL) {
perror("Failed to open annuaire.txt in read mode");
exit (-1) ;
}
/* Lecture du fichier ligne par ligne */
while (!feof (inbuf)) {
lu = fscanf (inbuf, "%s %d", name, &year) ;
if (lu == 2)
printf ("Nom=%s \t Date=%d\n", name, year) ;
}
/* Fermeture du descripteur de fichier */
fclose (inbuf) ;
return 0 ;
}
annuaire.txt
Ritchie 1941
Kernighan 1942
Thompson 1943
Stallman 1953
Gates 1955
Jobs 1955
Torvalds 1969
Invite de commandes
M:\ExemplesC> lire-annuaire.exe
Nom=Ritchie Date=1941
Nom=Kernighan Date=1942
Nom=Thompson Date=1943
Nom=Stallman Date=1953
Nom=Gates Date=1955
Nom=Jobs Date=1955
Nom=Torvalds Date=1969
M:\ExemplesC>
4. Entrées/Sorties
Accès fichiers | 144

1. Introduction
2. Types de base
3. Opérateurs
4. Entrées/Sorties
5. Structures de Contrôle & Boucles
6. Fonctions
7. Pointeurs
8. Structures
9. Le préprocesseur
10. Bibliothèque standard
11. Génie logiciel et C

Trois exemples de `if` simples :

```
3 int x ;
4 double y ;
5
6 /* ... */
7
8 if (x != 0)
9   y = 1 / x ;
```

```
3 int x, max ;
4
5 /* ... */
6
7 if (x > max) {
8   max = x ;
9 }
```

```
3 int x ;
4
5 /* ... */
6
7 if (x < 0) {
8   x = -x ;
9 }
```

Deux exemples de `if else` :

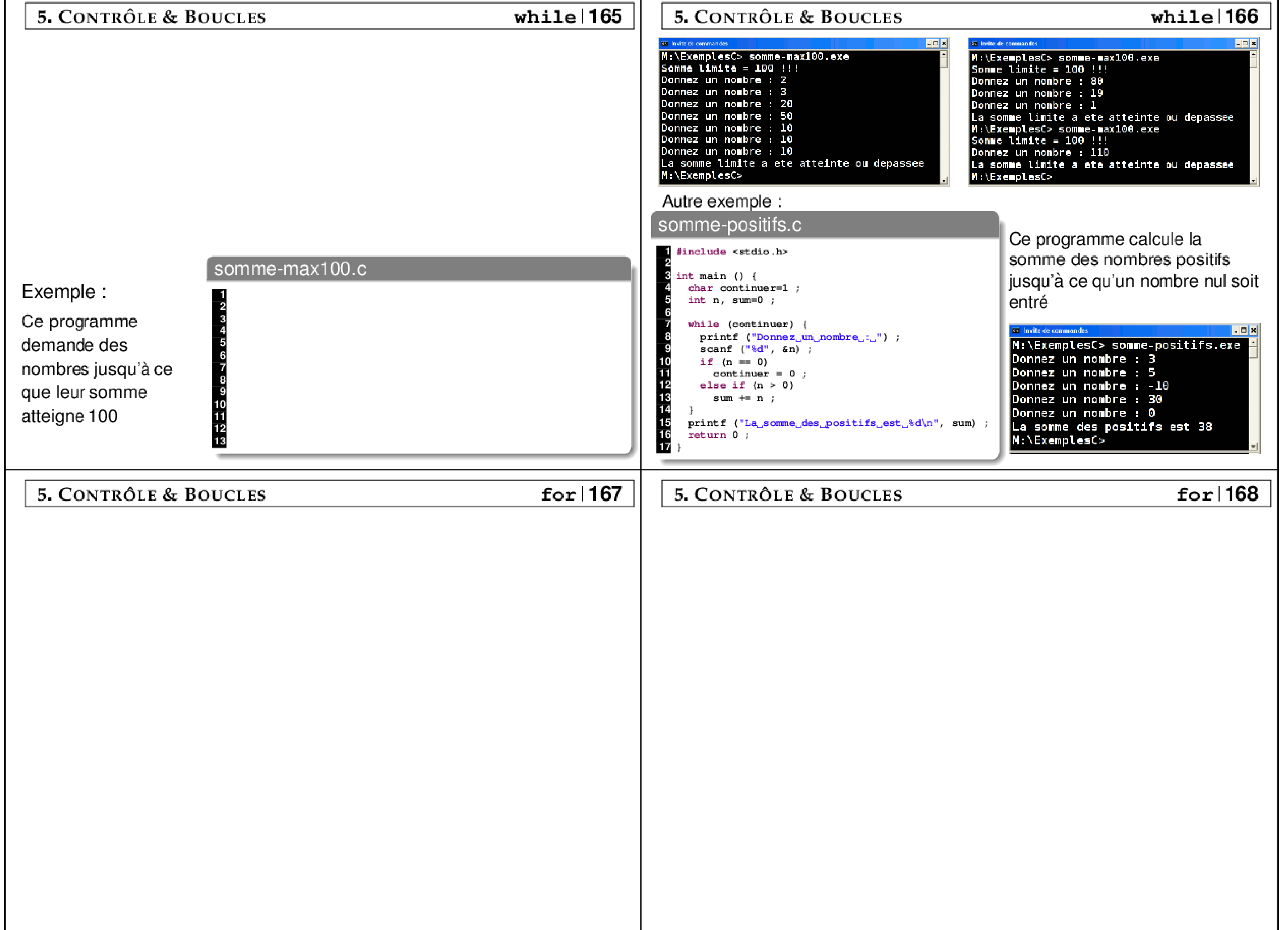
```
5 if (a > b) {
6   max = a ;
7   min = b ;
8 }
9 else {
10   max = b ;
11   min = a ;
12 }
```

```
7 int year, ndays ;
8
9 /* ... */
10
11 if (is_bissextile (year))
12   ndays = 366 ;
13 else
14   ndays = 365 ;
```


Encore deux exemples divers :

```
4 int i, N=10 ;
5 int a[N], val ;
6
7 /* ... */
8
9 if (i >= 0 && i < N)
10   a[i] = val ;
11 else
12   printf ("index out of range\n");
```

```
4 char found = 0 ;
5
6
7 /* ... */
8
9
10 if (!found) {
11   printf ("search failed\n") ;
12 }
```


5. Contrôle & Boucles
while | 165
Exemple :
Ce programme demande des nombres jusqu'à ce que leur somme atteigne 100
somme-max100.c
5. Contrôle & Boucles
while | 166
M:\ExemplesC> somme-max100.exe
Somme limite = 100 !!!
Donnez un nombre : 2
Donnez un nombre : 3
Donnez un nombre : 20
Donnez un nombre : 50
Donnez un nombre : 10
Donnez un nombre : 10
Donnez un nombre : 10
La somme limite a ete atteinte ou depassee
M:\ExemplesC>
M:\ExemplesC> somme-max100.exe
Somme limite = 100 !!!
Donnez un nombre : 80
Donnez un nombre : 19
Donnez un nombre : 1
La somme limite a ete atteinte ou depassee
M:\ExemplesC> somme-max100.exe
Somme limite = 100 !!!
Donnez un nombre : 110
La somme limite a ete atteinte ou depassee
M:\ExemplesC>
Autre exemple :
somme-positifs.c
#include <stdio.h>
int main () {
char continuer=1 ;
int n, sum=0 ;
while (continuer) {
printf ("Donnez un nombre : ") ;
scanf ("%d", &n) ;
if (n == 0)
continuer = 0 ;
else if (n > 0)
sum += n ;
}
printf ("La somme des positifs est %d\n", sum) ;
return 0 ;
}
Ce programme calcule la somme des nombres positifs jusqu'à ce qu'un nombre nul soit entré
M:\ExemplesC> somme-positifs.exe
Donnez un nombre : 3
Donnez un nombre : 5
Donnez un nombre : -10
Donnez un nombre : 30
Donnez un nombre : 0
La somme des positifs est 38
M:\ExemplesC>
5. Contrôle & Boucles
for | 167
5. Contrôle & Boucles
for | 168

Plan

1. Introduction
2. Types de base
3. Opérateurs
4. Entrées/Sorties
5. Structures de Contrôle & Boucles
6. Fonctions
7. Pointeurs
8. Structures
9. Le préprocesseur
10. Bibliothèque standard
11. Génie logiciel et C

→ En C toute instruction doit être **dans** une fonction...
↪ Il en faut donc au moins une dans tout programme, la fonction principale (`main()`)

⇒ De plus, plutôt que d'avoir une unique fonction regroupant l'ensemble des traitements d'un programme, il est préférable "découper" en plus *petits* morceaux

Quatre principaux avantages à cela :

1. La **lisibilité** : la clarté du code augmente, puisqu'une sémantique est donnée à un ensemble d'instructions
2. La **facilité** de développement, puisque le nombre de bugs diminue inévitablement une fois le découpage effectué ! (tests unitaires)
3. Le développement **coopératif** : plusieurs personnes peuvent travailler sur un même programme sans savoir comment les autres morceaux fonctionnent en interne. On parle de *modularité* ou de *paquetages*
4. La **maintenabilité** : il est plus facile d'intervenir sur un petit morceau de code que sur un gros !

Exemple :

version-1fct.c

```c
#include <stdio.h>

int main () {
  int a, b, c, x, y, w ;

  printf ("Donnez un nombre : ") ;
  scanf ("%d", &a) ;

  printf ("Donnez un nombre : ") ;
  scanf ("%d", &b) ;

  printf ("Donnez un nombre : ") ;
  scanf ("%d", &c) ;

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

  y = (a > b) ? a : b ;
  y = (c > y) ? c : y ;

  w = y - x ;

  /* ... */

  return 0 ;
}
```

version-4fct.c

```c
#include <stdio.h>

int lire_nombre () {
  int n ;
  printf ("Donnez un nombre : ") ;
  scanf ("%d", &n) ;
  return n ;
}

int max (int n1, int n2, int n3) {
  int m ;
  return ((m = (n1 > n2 ? n1 : n2)) < n3) ? n3 : m ;
}

int min (int n1, int n2, int n3) {
  int m ;
  return ((m = (n1 < n2 ? n1 : n2)) > n3) ? n3 : m ;
}

int main () {
  int a, b, c, w ;

  a = lire_nombre () ;
  b = lire_nombre () ;
  c = lire_nombre () ;
  w = max(a,b,c) - min(a,b,c) ;

  /* ... */

  return 0 ;
}
```


Principales conséquences (corollaires) :

1. La **factorisation** du code : évite la duplication de certaines portions de code, en regroupant au sein d'une fonction les traitements identiques, et évite ainsi la redondance (dangereuse)
2. La **réutilisabilité** du code : les fonctions écrites peuvent être utilisées à différents endroits dans le programme et réutilisées dans d'autres programmes

Exemples :

```c
void trait (void) {
  printf ("-=-=-=-=-=-=-=-=-=-\n") ;
}

void afficher_nombre (int n) {
  printf ("Votre nombre : %d\n", n) ;
}

int saisir_nombre () {
  int n ;
  printf ("Donnez nombre : ") ;
  scanf ("%d", &n) ;
  return n ;
}
```

```c
void nl () {
  printf ("\n") ;
}

int carre (int n) {
  return n * n ;
}

char pair (int n) {
  return (n % 2) == 0 ;
}

int max (int n1, int n2) {
  return n1 > n2 ? n1 : n2 ;
}
```


- Le **nom**, le **type de retour** et les **types des arguments** spécifient comment la fonction doit être appelée

↪ on parle alors du **PROTOTYPE** de la fonction

Exemples :

```c
void trait (void) ;
void afficher_nombre (int n) ;
int saisir_nombre () ;
```

```c
void nl () ;
int carre (int n) ;
char pair (int n) ;
int max (int n1, int n2) ;
```


Un exemple :

version1-lineaire.c

```c
#include <stdio.h>

/** Codes des fonctions */

int saisir_nombre () {
  int n ;
  printf ("Donnez nombre : ") ;
  scanf ("%d", &n) ;
  return n ;
}

int carre (int n) {
  return n * n ;
}

int max (int n1, int n2) {
  return n1 > n2 ? n1 : n2 ;
}

/** Fonction principale */
int main () {
  int a, b, c ;
  a = saisir_nombre() ;
  b = saisir_nombre() ;
  c = carre (max (a, b)) ;
  /* ... */
  return 0 ;
}
```

version2-prototype.c

```c
#include <stdio.h>

/** Declarations des fonctions */
int saisir_nombre () ;
int carre (int n) ;
int max (int n1, int n2) ;

/** Fonction principale */
int main () {
  int a, b, c ;
  a = saisir_nombre() ;
  b = saisir_nombre() ;
  c = carre (max (a, b)) ;
  /* ... */
  return 0 ;
}

/** Codes des fonctions */

int saisir_nombre () {
  int n ;
  printf ("Donnez nombre : ") ;
  scanf ("%d", &n) ;
  return n ;
}

int carre (int n) {
  return n * n ;
}

int max (int n1, int n2) {
  return n1 > n2 ? n1 : n2 ;
}
```


Attention !!! Important !

Doit être bien clair (dans votre esprit) ! Ne pas confondre :

1. le moment où l'on **déclare/défini** une fonction
2. et le moment où l'on **appelle** une fonction

→ Le code déclaré dans une fonction ne déclenche **aucune** d'exécution

→ Le code déclaré dans une fonction ne fait que **définir** une suite d'actions à exécuter dans le cas où la fonction serait appelée

→ Le code d'une fonction ne s'exécutera que lorsque la fonction aura été **appelée** par la fonction `main()` ou (indirectement) par une autre fonction qui aura été appelée par la fonction `main()`

⇒ Une fonction peut être appelée **plusieurs fois**, mais n'est définie qu'**une seule fois**

6. Fonctions — Appel | 181

Exemple : le programme suivant... ne fait rien !

nothing.c

```c
#include <stdio.h>

int saisir_nombre () {
  int n ;
  printf ("Donnez nombre : ") ;
  scanf ("%d", &n) ;
  return n ;
}

int max (int n1, int n2) {
  return n1 > n2 ? n1 : n2 ;
}

int demander () {
  int a, b, c ;
  a = saisir_nombre () ;
  b = saisir_nombre () ;
  c = max (a, b) ;
  printf ("Le plus grand est %d\n", c) ;
  return c ;
}

int main () {

  return 0 ;
}
```

6. Fonctions — Appel | 182

Exemple : appels multiples de fonctions

calls.c

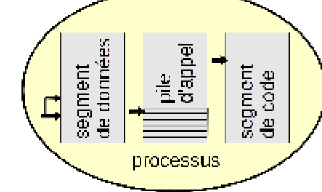
```c
#include <stdio.h>

int saisir_nombre () {
  int n ;
  printf ("Donnez nombre : ") ;
  scanf ("%d", &n) ;
  return n ;
}

int carre (int n) { return n * n ; }

char pair (int n) { return (n % 2) == 0 ; }

void afficher (int n) {
  printf ("%d est %s et son carre vaut %d\n", n, pair(n) ? "pair" : "impair", carre(n)) ;
}

int main () {
  int a, b, c ;
  a = saisir_nombre () ;
  b = saisir_nombre () ;
  afficher (a) ;
  afficher (b) ;
  afficher (4) ;
  c = carre (24) ;
  printf ("%d + %d + %d = %d\n", a, b, c, a+b+c) ;
  return 0 ;
}
```

Invite de commandes

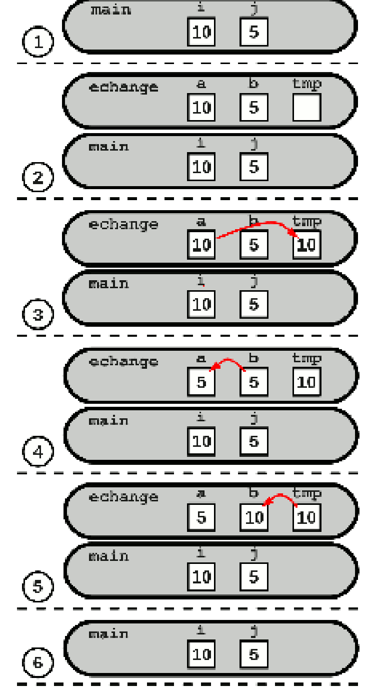
```
M:\ExemplesC> calls.exe
Donnez nombre : 2
Donnez nombre : 9
2 est pair et son carre vaut 4
9 est impair et son carre vaut 81
4 est pair et son carre vaut 16
2 + 9 + 576 = 587
```

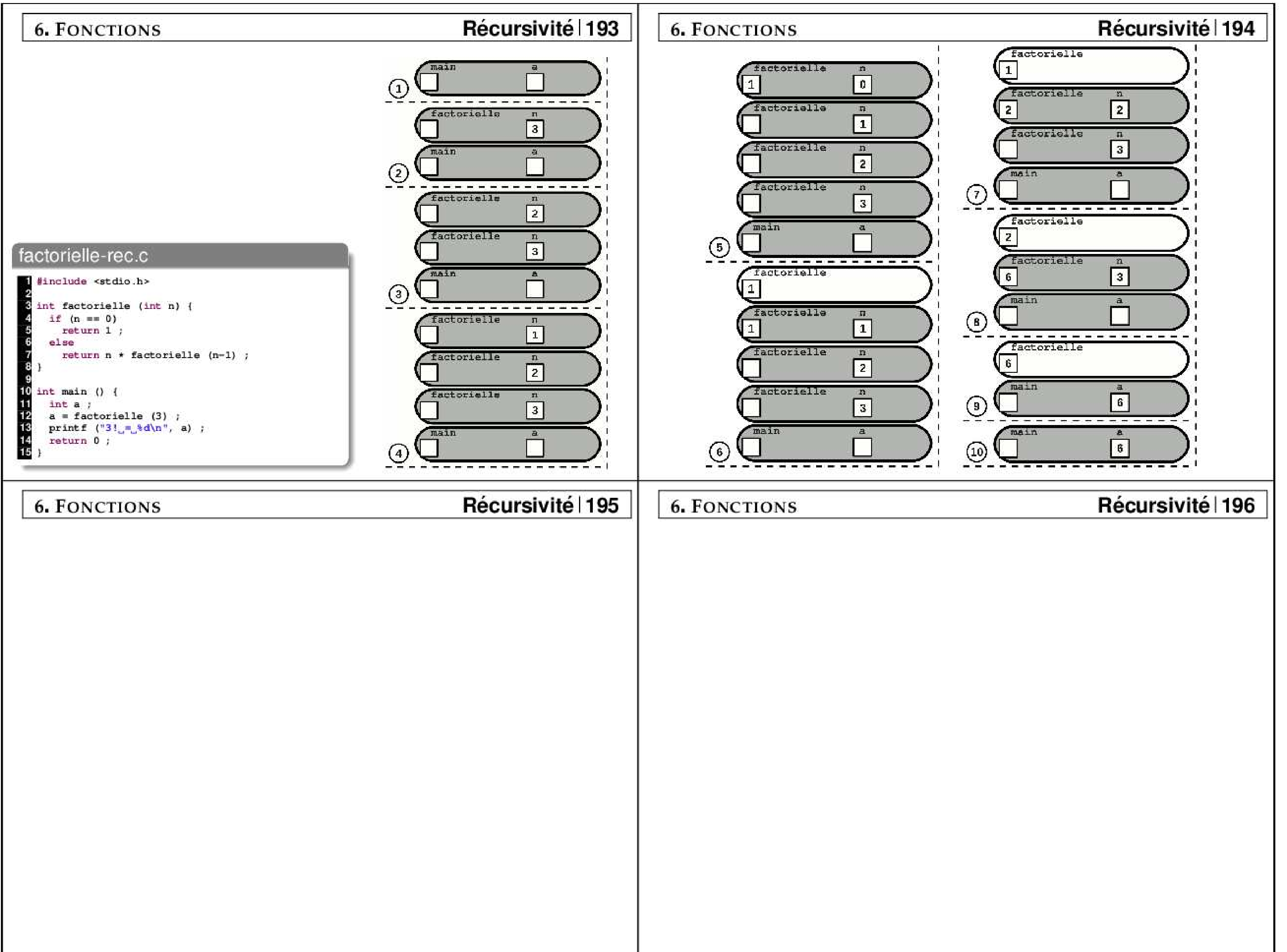
6. Fonctions — Passage des paramètres | 183

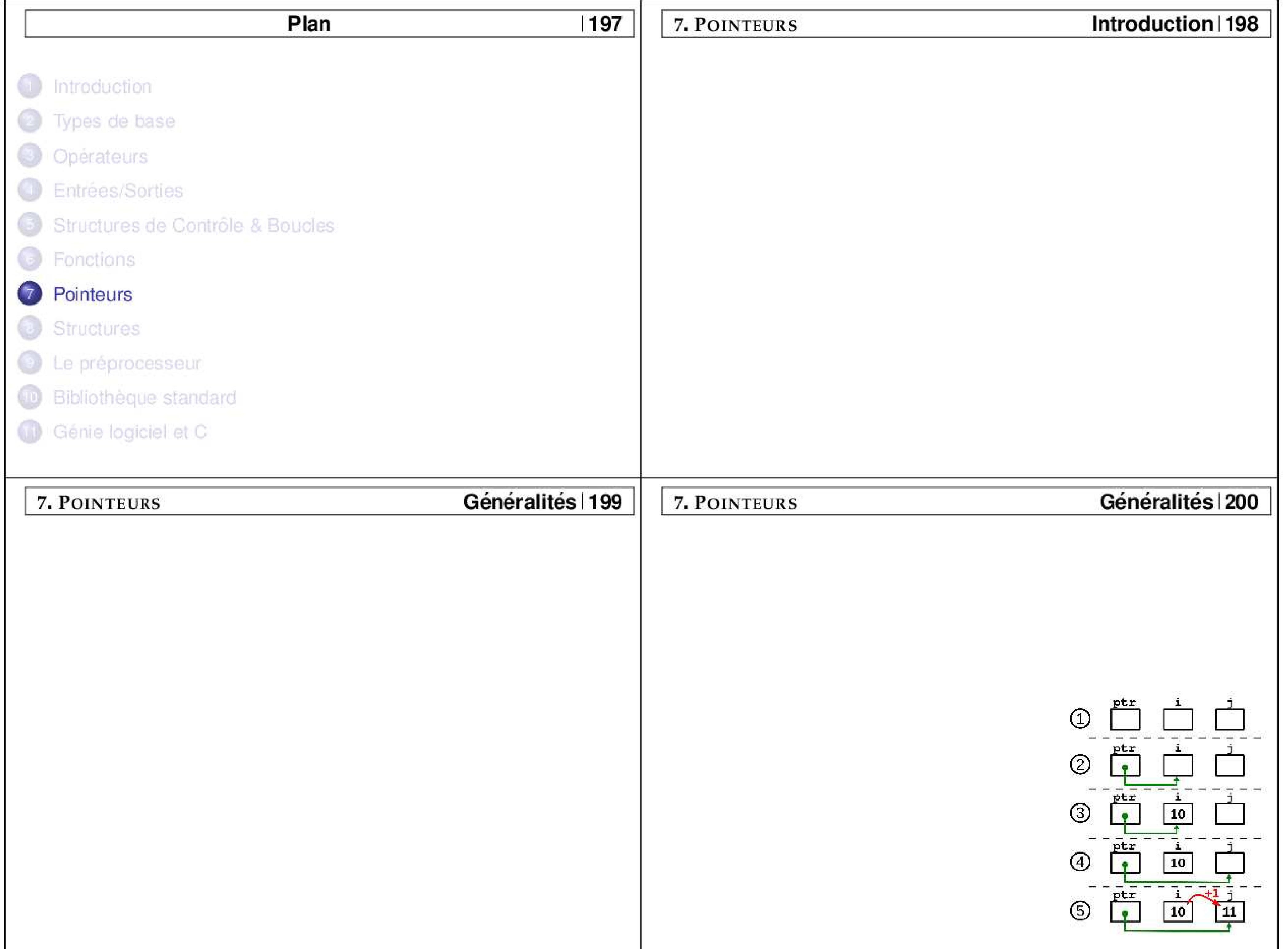
↬ **Rmq** : Lié au fonctionnement des ordinateurs
- ↪ Chaque paramètre d'une fct se voit *empiler* par valeur dans la *pile d'appel des fonctions*
- ↪ Le code de la fonction va ensuite utiliser des **variables locales**
- ↪ Changement zone d'adressage mémoire...



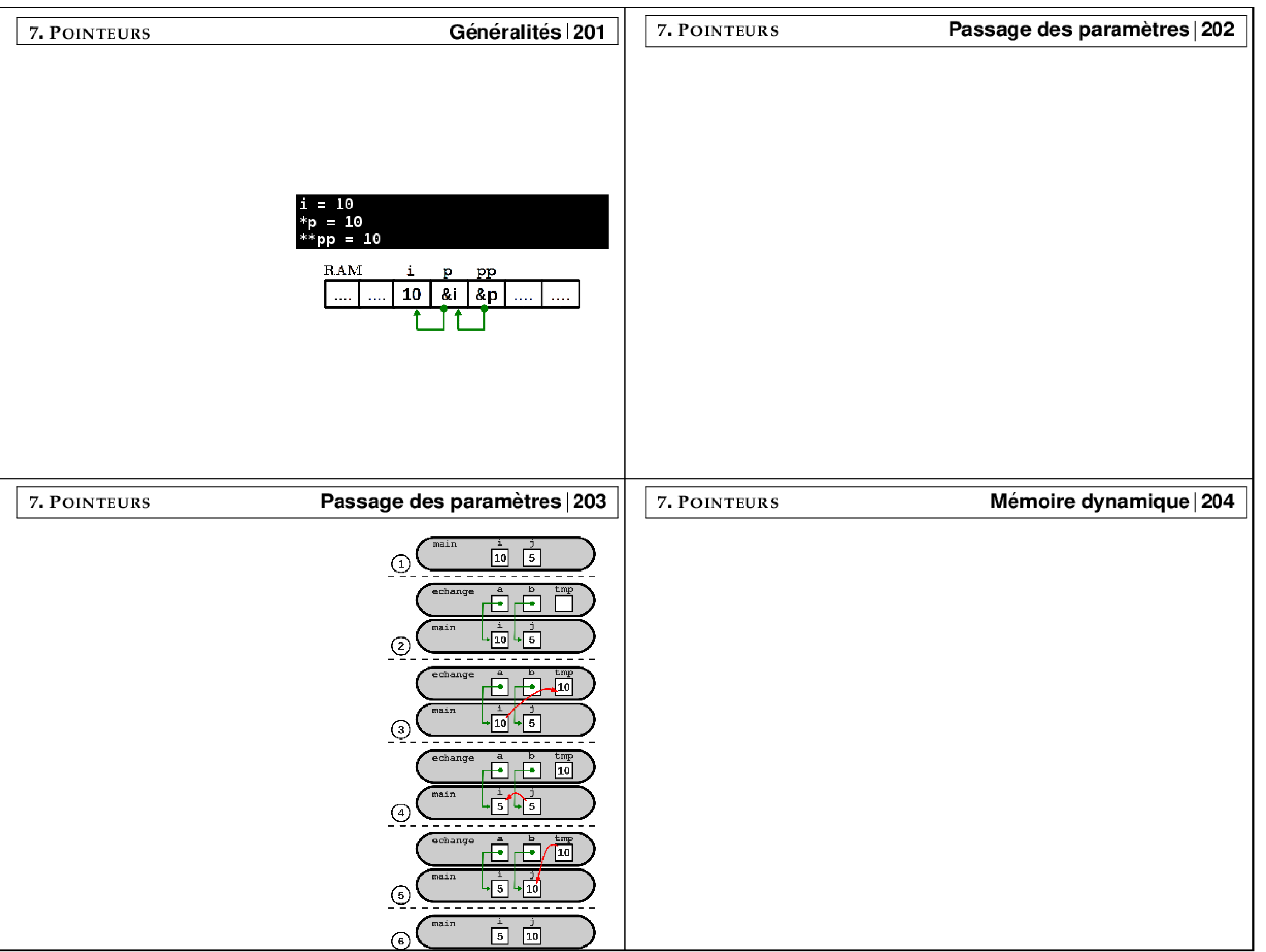
6. Fonctions — Passage des paramètres | 184

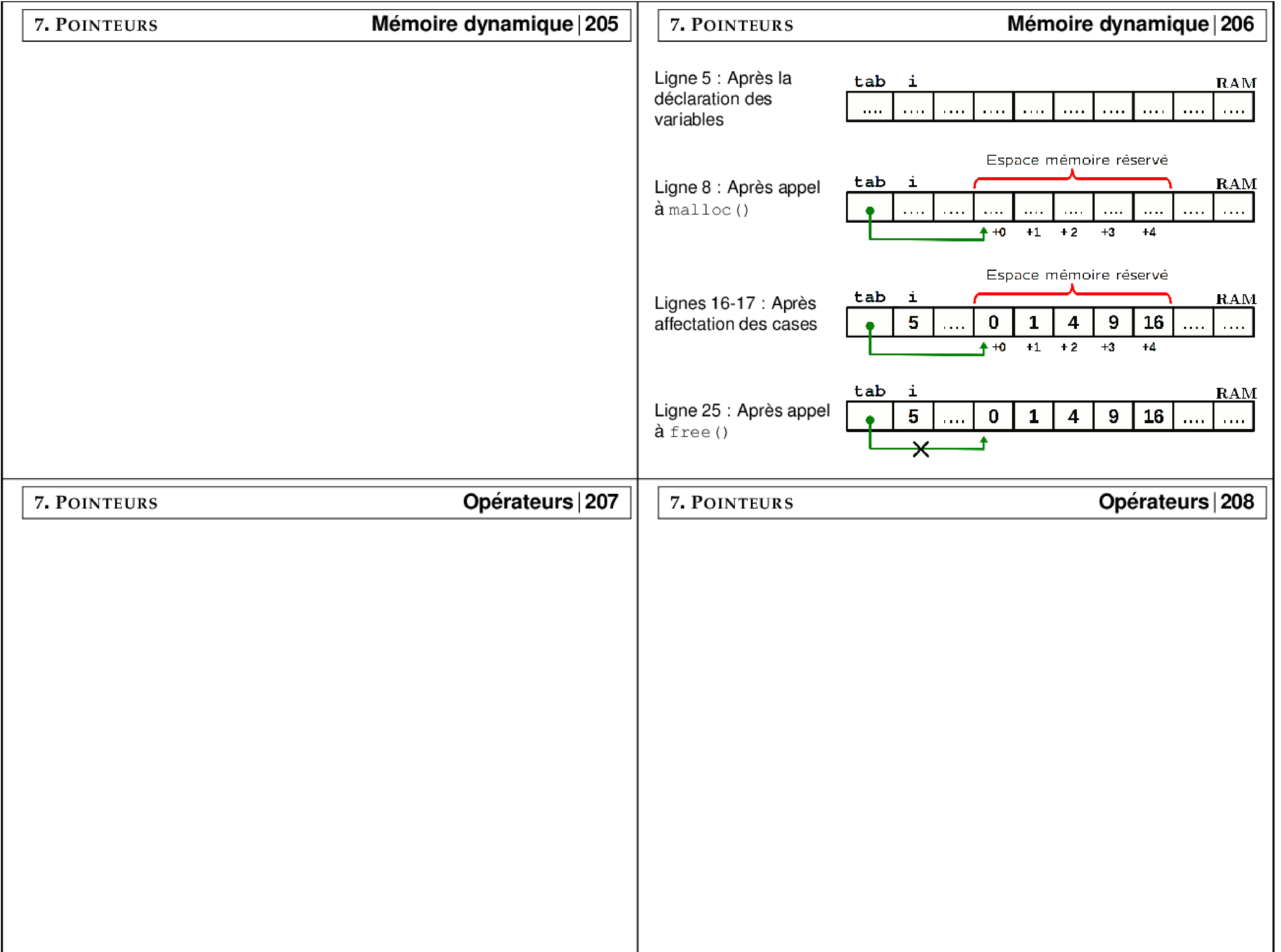



6. FONCTIONS
Récursivité | 193
factorielle-rec.c
#include <stdio.h>
int factorielle (int n) {
if (n == 0)
return 1 ;
else
return n * factorielle (n-1) ;
}
int main () {
int a ;
a = factorielle (3) ;
printf ("3!␣=␣%d\n", a) ;
return 0 ;
}
6. FONCTIONS
Récursivité | 194
6. FONCTIONS
Récursivité | 195
6. FONCTIONS
Récursivité | 196



1 Introduction
2 Types de base
3 Opérateurs
4 Entrées/Sorties
5 Structures de Contrôle & Boucles
6 Fonctions
7 Pointeurs
8 Structures
9 Le préprocesseur
10 Bibliothèque standard
11 Génie logiciel et C


7. POINTEURS
Généralités | 201
i = 10
*p = 10
**pp = 10
RAM
i
p
pp
10
&i
&p
7. POINTEURS
Passage des paramètres | 202
7. POINTEURS
Passage des paramètres | 203
main
echange
tmp
7. POINTEURS
Mémoire dynamique | 204


7. POINTEURS
Mémoire dynamique | 205
7. POINTEURS
Mémoire dynamique | 206
Ligne 5 : Après la déclaration des variables
tab
i
RAM
Ligne 8 : Après appel à malloc()
Espace mémoire réservé
+0
+1
+2
+3
+4
Lignes 16-17 : Après affectation des cases
5
0
1
4
9
16
Ligne 25 : Après appel à free()
7. POINTEURS
Opérateurs | 207
7. POINTEURS
Opérateurs | 208

1. Introduction
2. Types de base
3. Opérateurs
4. Entrées/Sorties
5. Structures de Contrôle & Boucles
6. Fonctions
7. Pointeurs
8. Structures
9. Le préprocesseur
10. Bibliothèque standard
11. Génie logiciel et C

- Autre exemple :

```c
#include <string.h>

enum GENDER { FEMALE, MALE } ;

struct date {
  unsigned char d ;
  unsigned char m ;
  unsigned int y ;
} ;

struct user {
  char *name ;
  char *password ;
  struct date birthday ;
  enum GENDER gender ;
} ;

int main () {
  struct user playerOne, playerTwo ;

  playerOne.name = strdup("Me");
  playerOne.birthday.d = 26 ;
  playerOne.birthday.m = 7 ;
  playerOne.birthday.y = 1982 ;
  playerOne.gender = MALE ;
  playerOne.name = strdup("You");
  /*...*/
```

```c
#include <string.h>

typedef enum GENDER { FEMALE, MALE } Gender ;

typedef struct date {
  unsigned char d ;
  unsigned char m ;
  unsigned int y ;
} Date ;

typedef struct user {
  char *name ;
  Date birthday ;
  Gender gender ;
} User ;

int main () {
  User playerOne, playerTwo ;

  playerOne.name = strdup("Me");
  playerOne.birthday.d = 26 ;
  playerOne.birthday.m = 7 ;
  playerOne.birthday.y = 1982 ;
  playerOne.gender = MALE ;
  playerOne.name = strdup("You");
  /*...*/
```

↪ **RMQ** : Utiliser `typedef` pour définir un nouveau type et éviter d'avoir à écrire `struct` partout

1 Introduction
2 Types de base
3 Opérateurs
4 Entrées/Sorties
5 Structures de Contrôle & Boucles
6 Fonctions
7 Pointeurs
8 Structures
9 Le préprocesseur
10 Bibliothèque standard
11 Génie logiciel et C

Fichiers sources (.c)
Fichiers d'entête (.h)
stdio.h
math.h
...
(1)
Compilateur C
Fichiers de bibliothèques (.lib) et fichiers objet (.o)
(2)
Lieur C
Fichier exécutable
10011111001
0111000011
110001100

boolean.h

geometrie.h

geometrie.c

main.c

main2.c

1. Introduction
2. Types de base
3. Opérateurs
4. Entrées/Sorties
5. Structures de Contrôle & Boucles
6. Fonctions
7. Pointeurs
8. Structures
9. Le préprocesseur
10. Bibliothèque standard
11. Génie logiciel et C

Plan

1. Introduction
2. Types de base
3. Opérateurs
4. Entrées/Sorties
5. Structures de Contrôle & Boucles
6. Fonctions
7. Pointeurs
8. Structures
9. Le préprocesseur
10. Bibliothèque standard
11. Génie logiciel et C

Ce que nous n'avons pas abordé, ou alors que faiblement, dans ce cours de 8h (non exhaustif) :

- Les pointeurs; Les pointeurs de fonction;
- Les listes variables d'arguments;
- Les champs de bits; Les unions;
- Les macro fonctions; La compilation conditionnelle;
- Les structures auto-référentielles (liste, pile, file, arbre, graphe);
- Les mots clés `continue` et `goto` (proscrits);
- Les E/S binaires (`open()`, `read()`, `write()`, `close()`);
- ...

Pour aller plus loin :

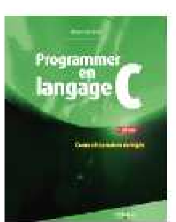
↪ **"Le langage C"** Henri Garreta
`http://c.developpez.com/cours/poly-c/`

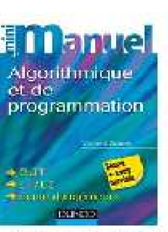
↪ **"Le langage C"** B. Kernighan & D. Ritchie (éditions Dunod)

↪ **Ressources en ligne de l'ESTIA** : voir diapo suivante

↪ Ressources

- Bibliothèque en ligne de l'ESTIA `https://univ.scholarvox.com`





Voir diapo 23 pour la liste des titres sélectionnés
(ou liens directs depuis Moodle)

- Techniques de l'ingénieur `https://www.techniques-ingenieur.fr`
  Accueil
  > Ressources documentaires
  > Technologies de l'information
  > Technologies logicielles Architectures des systèmes
  > Langages de programmation
  > Langage C

1. Introduction 19
2. Types de base 61
   1. Variables 63
   2. Variables entières (`char` et `int`) 64
   3. Variables réelles (`float` et `double`) 66
   4. Les tableaux 76
   5. Les caractères et la table `ASCII` 79
   6. Les variables constantes 81
   7. Constantes classiques 82
   8. Constantes énumérées 83
   9. Conversions de types 85
   10. Exemples d'erreurs classiques 88
3. Opérateurs 90
   1. L'opérateur d'affectation = ... 91
   2. Les opérateurs arithmétiques + − * / % 94
   3. Incrémentation et décrémentation ++ et −− 97
   4. Les opérateurs logiques == <= >= < > != && || 98

   5. Traitement des bits & | << >> 101
   6. Opérateurs et expressions d'affectations += −= /= *= %= 105
   7. Priorité et associativité des opérateurs 106
4. Entrées/Sorties 108
   1. Écriture d'une chaîne simple 113
   2. Écriture de valeurs 114
   3. Nombre minimum de caractères produits 117
   4. Table des formats possibles pour `printf()` 119
   5. Un flux de sortie distinct pour les erreurs : `stderr` 122
   6. Les redirections 124
   7. Comportement (asynchrone) de `scanf()` 126
   8. Utilisation de `scanf()` 127
   9. Table des formats possibles pour `scanf()` 130
   10. Ouvrir un descripteur de fichier avec `fopen()` 134
   11. Écrire dans un fichier avec `fprintf()` 140
   12. Lire dans un fichier avec `fscanf()` 142

Table des matières – III

- 13 Accès fichiers caractère par caractère `fgetc()` et `fputc()` 144

5 Structures de Contrôle & Boucles 146
- 1 La structure conditionnelle classique `if` ... 148
- 2 Le contrôle par `else if` ... 154
- 3 L'opérateur de conditionnelle `? :` ... 158
- 4 La structure de contrôle `switch` ... 160
- 5 La boucle `while` ... 165
- 6 La boucle `for` ... 167
- 7 La boucle `do...while` ... 171

6 Fonctions 172
- 1 La fonction principale ... 175
- 2 Définition d'une fonction ... 176
- 3 Passage des paramètres par valeur ... 183
- 4 Retour sur les variables ... 185
- 5 Variable locale ... 186
- 6 Variable globale ... 187
- 7 Variable statique ... 189

Table des matières – IV

- 8 Surcharge des noms de variables ... 190
- 9 Cas des tableaux ... 192
- 10 Récursivité ... 193

7 Pointeurs 197
- 1 Généralités ... 199
- 2 Passage des paramètres par adresse ... 202
- 3 Mémoire dynamique avec `malloc()` et `free()` ... 204
- 4 Opérateurs ... 207

8 Structures 209
- 1 Principes généraux ... 210
- 2 Fonctions et structures ... 214
- 3 Tableaux et structures ... 219
- 4 Les structures auto-référencielles ... 221

9 Le préprocesseur 222
- 1 Les macro-constantes avec `#define` ... 224
- 2 Les paquetages ... 226

Table des matières – V

10 Bibliothèque standard 233
- 1 Les entrées sorties : `<stdio.h>` ... 235
- 2 Tests sur les caractères : `<ctype.h>` ... 241
- 3 Manipulation de chaînes de caractères : `<string.h>` ... 242
- 4 Fonctions mathématiques : `<math.h>` ... 245
- 5 Fonctions utilitaires : `<stdlib.h>` ... 248
- 6 La date et l'heure : `<time.h>` ... 252

11 Génie logiciel et C 256
- 1 Du génie logiciel à la programmation ... 257
- 2 Caractéristiques d'un programme ... 259