Slide 1

Université de Nice-Sophia Antipolis

Le langage C

Norme ANSI

Licence Professionnelle
des Métiers Informatiques

Carine Fédèle, Olivier Ponsini
ponsini@i3s.unice.fr
2002–2003

Bibliographie

Slide 2

1. *The C Programming Language*
   Kernighan B.W., Ritchie D.M.
   `Prentice Hall, 1978`
2. *Le langage C - C ANSI*
   Kernighan B.W., Ritchie D.M.
   `Masson - Prentice Hall, 1994, 2e édition`
   Traduit par J.-F. Groff et E. Mottier
3. *Langage C - Manuel de référence*
   Harbison S.P., Steele Jr. G.L.
   `Masson, 1990`
   Traduit en français par J.C. Franchitti

Slide 3

4. *Méthodologie de la programmation en langage C - Principes et applications*
   Braquelaire J.-P.
   `Masson, 1995, 2e édition`
5. et la section 3 du manuel (`man`) d'UNIX...

1 Généralités

1.1 Historique

Slide 4

Langage procédural typé.

Conçu par Dennis Ritchie, aux laboratoires Bell aux environs de 1972, pour la programmation d'un système d'exploitation.

Descendant de

- Algol 60 (1960) ;
- BCPL (1967) ;
- B (1970) ;
- CPL (1973).

Slide 5

Deux versions utilisées aujourd'hui :

- C traditionnel de Kernighan et Ritchie, 1978 (obsolète)
- C normalisé : ANSI 1989, ISO 1990, POSIX[a]

Ce cours suit la norme ANSI.

[a] Portable Operating System Interface.

Slide 6

1.2 Évaluation du langage

Qualités

- Nombreux types de données.
- Riche ensemble d'opérateurs et de structures de contrôle.
- Bibliothèque d'exécution standard.
- Efficacité des programmes.
- Transportabilité des programmes (plus facile si on respecte une norme).
- Liberté du programmeur.
- Interface privilégiée avec UNIX.

Slide 7

Défauts

- Syntaxe : langage à deux niveaux.
- Liberté du programmeur.
- Peu d'aide pour la détection d'erreurs à l'exécution (message sibyllin : « `bus error-core dumped` »).
- Absence de typage fort.
- Rapidement complexe.
- Vieux langage.

Slide 8

2 Pour débuter

Un programme C est constitué d'un ou plusieurs fichiers sources suffixés par `.c` et `.h`, dans le(s)quel(s) une unique fonction `main` doit apparaître (ce sera le point d'entrée du programme).

On ne peut emboîter les sous-programmes : ils sont tous au niveau 0.

En C, seules des fonctions peuvent être définies. Pour définir une procédure, il faut déclarer une fonction renvoyant le type spécial **`void`**.

Pour renforcer le fait que la fonction n'a pas de paramètres, mettre **`void`** entre les parenthèses.

Slide 9

2.1 Un premier exemple

2.1.1 Calcul d'une puissance (1)

```
int main (void) {
  int i, x = 3,
    y = 4; /* y doit être positif */
  double z = 1.0;

  i = 0;
  while (i < y) {
    z = z * x; i = i + 1;
  }
  return 0;
}
```

Slide 10

On compile et on exécute.


```
$ gcc -Wall -pedantic -ansi foo.c
$ a.out
$
```

Ce *programme* C calcule x^y, `x` et `y` étant donnés (`x` vaut `3`, et `y` `4`). Il n'affiche cependant rien du tout...

Slide 11

2.1.2 Calcul d'une puissance (2)

```
#include <stdio.h>

int main (void) {
  int x = 3, y = 4;
  double z = 1.0;

  printf("x = %d, y = %d", x, y);
  while (y > 0) {
    z *= x;
    y -= 1;
  }
  fprintf(stdout, ", z = %.2f\n", z);
  return 0;
}
```

On compile et on exécute.

```
$  gcc -Wall -pedantic -ansi foo.c
$  a.out
x = 3, y = 4, z = 81.00
$
```

Slide 12

Le programme calcule 3^4 et affiche les valeurs de `x`, `y` et `z`.

En C, il n'y a pas d'instructions d'E/S. En revanche, il existe des fonctions de bibliothèque, dont le fichier de déclarations s'appelle `stdio.h`[a].

Les fonctions de bibliothèque d'E/S font partie de la bibliothèque C `libc`.

[a] `stdio` : standard input-output. `.h` pour « headers ».

2.1.3 Compilation et exécution

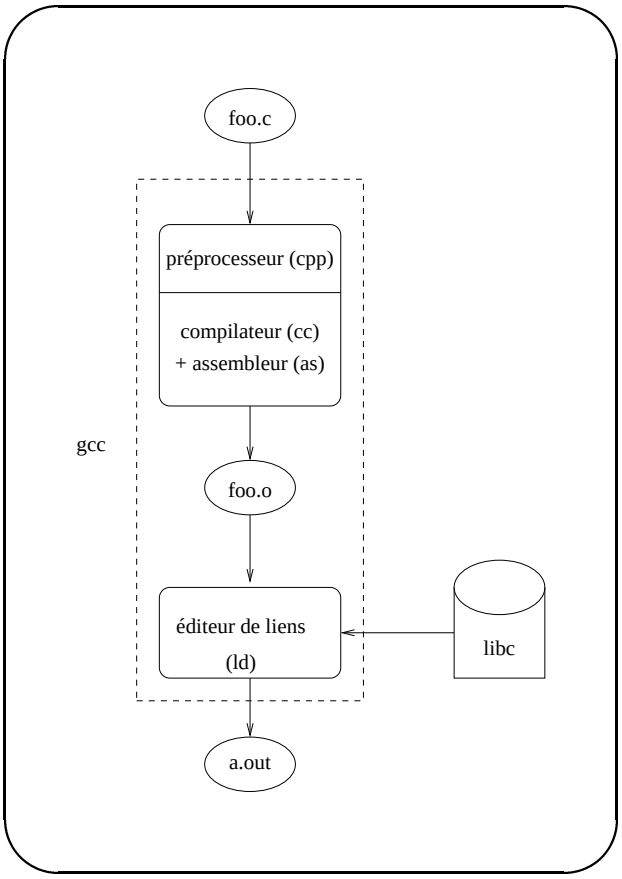
Le compilateur C utilisé est celui du projet GNU : GCC.

Du fichier source au fichier exécutable, différentes étapes sont effectuées :

Slide 13

- le préprocesseur `cpp` ;
- le compilateur C `cc` traduit le programme source en un programme équivalent en langage d'assemblage ;
- l'assembleur `as` construit un fichier appelé objet contenant le code machine correspondant ;
- l'éditeur de liens `ld` construit le programme exécutable à partir des fichiers objet et des bibliothèques.

Slide 14



Slide 15

Quelques options du compilateur

- `-c` : pour n'obtenir que le fichier objet (donc l'éditeur de liens n'est pas appelé).
- `-E` : pour voir le résultat du passage du préprocesseur.
- `-g` : pour le débogueur symbolique.
- `-o` : pour renommer la sortie.
- `-Wall` : pour obtenir tous les avertissements.
- `-lnom_de_bibliothèque` : pour inclure une bibliothèque précise.
- `-ansi` : pour obtenir des avertissements à certaines extensions non ANSI de GCC.
- `-pedantic` : pour obtenir les avertissements requis par le C standard strictement ANSI.

2.1.4 Calcul d'une puissance (3)

Slide 16

```c
#include <stdio.h>
#define UNITE 1.0

double puissance (int a, int b) {
  /* Rôle : retourne a^b (ou 1.0 si b < 0) */
  double z = UNITE;

  while (b > 0) {
    z *= a; b -= 1;
  }
  return z;
}

int main (void) {
  fprintf(stdout, "3^4 = %.2f\n", puissance(3, 4));
  fprintf(stdout, "3^0 = %.2f\n", puissance(3, 0));
  return 0;
}
```

Slide 17

En C, on peut définir des fonctions.

La fonction principale `main` appelle la fonction `puissance`, afin de calculer 3^4 et affiche le résultat. Elle appelle aussi la fonction `puissance` avec les valeurs `3` et `0` et affiche le résultat.

Remarque : Pour compiler, là encore, il n'y a aucun changement.

```
$  gcc -Wall -pedantic -ansi foo.c
$  a.out
3^4 = 81.00
3^0 = 1.00
$
```

Slide 18

2.1.5 Fonctions

Deux formes de *définition*

- À la K&R (le C originel)

```
double puissance (a, b)
  int a, b;
{
  /* corps de la fonction puissance
     déclarations
     instructions */
}
```

Slide 19

- Forme ANSI

```
double puissance (int a, int b) {
  /* corps de la fonction puissance
     déclarations
     instructions */
}
```

Slide 20

Deux formes de *déclaration*

- À la K&R

```
extern double puissance ();
```

- Forme ANSI (utilisant un *prototype*)

```
extern double puissance (int, int);
```

L'utilisation de prototypes permet une détection des erreurs sur le type et le nombre des paramètres lors de l'appel effectif de la fonction.

Slide 21

2.1.6 Calcul d'une puissance (4)

```
#include <stdio.h>

double puissance (int a, int b) {
  /* Rôle : retourne a^b (ou 1.0 si b < 0) */
  double z = 1.0;

  while (b > 0) {
    z *= a; b -= 1;
  }
  return z;
}

int main (void) {
  int x, y;

  fprintf(stdout, "x = ");
  scanf("%d", &x);
  fprintf(stdout, "y = ");
  fscanf(stdin, "%d", &y);
  fprintf(stdout, "x = %d, y = %d, x^y = %.2f\n",
          x, y, puissance(x, y));
  return 0;
}
```

Dans les précédentes versions, pour modifier les valeurs de `x` et de `y`, il fallait modifier le texte source du programme, le re-compiler et l'exécuter.

En C, on peut demander à l'utilisateur des valeurs sur l'entrée standard.

Slide 22

```
$  gcc -Wall -pedantic -ansi foo.c
$  a.out
x = 3
y = 4
x = 3, y = 4, x^y = 81.00
$
$
```

2.2 Un autre exemple

Écrire sur la sortie standard ce qui est lu sur l'entrée standard (l'entrée et la sortie standard sont ouvertes par défaut).

En pseudo-langage :

Slide 23

```
variable c : caractère
début
  lire(c)
  tantque non fdf(entrée) faire
    écrire(c)
    lire(c)
  fintantque
fin
```

Slide 24

Traduction littérale :

```
#include <stdio.h>

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

  c = fgetc(stdin);
  while (c != EOF) {
    fputc(c, stdout);
    c = fgetc(stdin);
  }
  return 0;
}
```

Slide 25

Mais comme C est un langage d'expressions...

```
#include <stdio.h>

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

  while ((c = fgetc(stdin)) != EOF)
    fputc(c, stdout);
  return 0;
}
```

2.3 Et un dernier exemple

Compter le nombre de caractères lus sur l'entrée standard et écrire le résultat sur la sortie standard.

En pseudo-langage :

Slide 26

```
variable nb : entier
début
  nb ← 0
  tantque non fdf(entrée) faire
    nb ← nb + 1
  fintantque
  afficher(nb)
fin
```

Une première version (les fonctions non déclarées avant leur utilisation sont considérées comme renvoyant un entier) :

Slide 27

```
#include <stdio.h>

extern long compte (void); /* déclaration de compte */
  /* Rôle : compte le nombre de caractères lus sur l'entrée
     standard jusqu'à une fin de fichier */

int main (int argc, char *argv[]) {
  fprintf(stdout, "nombre de caractères = %ld\n", compte());
  return 0;
}

long compte (void) { /* définition de compte */
  long nb;

  nb = 0;
  while (fgetc(stdin) != EOF)
    nb += 1;
  return nb;
}
```

Slide 28

À la C :

```
#include <stdio.h>

extern long compte (void);
  /* Rôle : compte le nombre de caractères lus sur l'entrée
     standard jusqu'à une fin de fichier */

int main (int argc, char *argv[]) {
  fprintf(stdout, "nombre de caractères = %ld\n", compte());
  return 0;
}

long compte (void) {
  long nb = 0;

  for (; fgetc(stdin) != EOF; nb++)
    /* Rien */;
  return nb;
}
```

Slide 29

2.4 Compilation séparée

Un programme C peut être constitué de plusieurs fichiers sources (compilation séparée « archaïque » reposant sur un préprocesseur).

- fichier `math.h` : fichier de « déclarations »

```
#ifndef _MATH_H
#define _MATH_H

extern double puissance (int /* a */, int /* b */);
  /* Rôle : retourne a^b (ou 1.0 si b < 0) */

#endif
```

Slide 30

- fichier `math.c` : fichier de « définitions »

```
#include "math.h"

double puissance (int a, int b) {
  double z = 1.0;

  while (b > 0) {
    z *= a;
    b--;
  }
  return z;
}
```

Slide 31

- fichier `essai.c` : fichier « principal »

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

int main (int argc, char *argv[]) {
  if (argc != 3) {
    fprintf(stderr, "usage: %s x y>=0 (x^y)\n", argv[0]);
    return 1;
  }
  {
    int x = atoi(argv[1]), y = atoi(argv[2]);

    if (y < 0) {
      fprintf(stderr, "usage: %s x y>=0 (x^y)\n", argv[0]);
      return 2;
    }
    fprintf(stdout, "x = %d, y = %d, z = %.2f\n",
            x, y, puissance(x, y));
  }
  return 0;
}
```

Slide 32

Compilation du programme composé des fichiers `essai.c` et `math.[hc]`

```
$  gcc -Wall -pedantic -ansi -o vasy \
      math.c essai.c
$
```

ou bien

```
$  gcc -c -Wall -pedantic -ansi math.c
$  gcc -c -Wall -pedantic -ansi essai.c
$  gcc -o vasy math.o essai.o
$
```

Slide 33

ou encore grâce à un `makefile` et l'outil `make` :

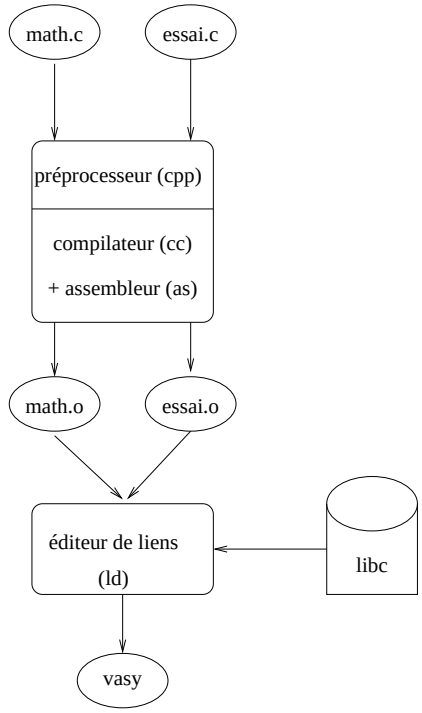
```
CC=gcc
CFLAGS=-Wall -pedantic -ansi
OBJECTS=math.o essai.o

vasy : $(OBJECTS)
        $(CC) -o vasy $(OBJECTS)
        @echo "La compilation est finie"
essai.o : essai.c math.h
        $(CC) -c $(CFLAGS) essai.c
math.o : math.c math.h
        $(CC) -c $(CFLAGS) math.c

clean :
        -rm -f $(OBJECTS) vasy *~
print :
        a2ps math.h math.c essai.c | lpr
```

Slide 34

Remarque : Si `math.o` n'est pas « lu » lors de l'édition de liens, il y aura une référence non résolue de la fonction `puissance`.

```
$  gcc essai.c
...: In function 'main':
...: undefined reference to 'puissance'
... ld returned 1 exit status
$
```

Slide 35



Slide 36

3 Éléments lexicaux

- Commentaires : /* */
- Identificateurs : suite de lettres[a], de chiffres ou de souligné, débutant par une lettre ou un souligné
- Mots réservés
  - classes de variables : **`auto`**, **`const`**, **`extern`**, **`register`**, **`static`**, **`volatile`**
  - instructions : **`break`**, **`case`**, **`continue`**, **`default`**, **`do`**, **`else`**, **`for`**, **`goto`**, **`if`**, **`return`**, **`switch`**, **`while`**
  - types : **`char`**, **`double`**, **`float`**, **`int`**, **`long`**, **`short`**, **`signed`**, **`unsigned`**, **`void`**
  - constructeurs de types : **`enum`**, **`struct`**, **`typedef`**, **`union`**

[a] non accentuées.

Slide 37

- Séquences d'échappement (utilisées dans les constantes de type caractère et chaîne de caractères)

`\a`	sonnerie
`\b`	retour en arrière
`\f`	saut de page
`\n`	fin de ligne
`\r`	retour chariot
`\t`	tabulation horizontale
`\v`	tabulation verticale
`\\`	barre à l'envers
`\?`	point d'interrogation
`\'`	apostrophe
`\"`	guillemet
`\o`, `\oo`, `\ooo`	nombre octal (`o` $\in$ [0-7])
`\xh`, `\xhh`	nombre hexadécimal (`h` $\in$ [0-9a-fA-F])

Slide 38

- Constantes de type entier en notation décimale, octale ou hexadécimale

```
123  12u   100L
0173 0123u
0x7b 0XAb3 0X12L
```

- Constantes de type réel

```
123e0 123.456e+78
12.3f 12.3L       12.3  12F
```

Slide 39

- Constantes de type caractère : un caractère entre apostrophes

```
'a'  '\141' '\x61'
'\n' '\0'   '\12'
```

  en C, un caractère est considéré comme un entier (conversion unaire).

- Constantes de type chaîne : chaîne placée entre guillemets

```
- ""
- "here␣we␣go"
- "une␣chaîne␣sur␣\
   deux␣lignes"
- "et␣"  "une␣autre"
- "autre␣chaîne␣sur"
  "␣deux␣lignes"
```

Slide 40

4 Variables

Une variable est un nom auquel on associe une valeur que le programme peut modifier pendant son exécution. Lors de sa déclaration, on indique son type.

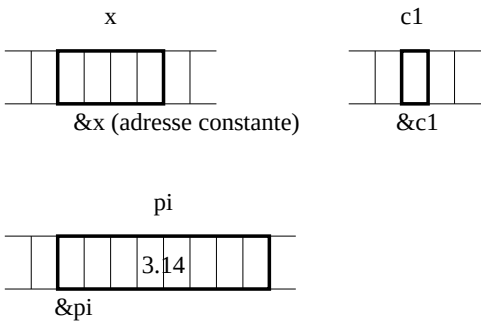
Il faut *déclarer* **toutes** les variables avant de les utiliser.

On peut initialiser une variable lors de sa déclaration.

On peut préfixer toutes les déclarations de variables par **`const`** (jamais modifiés).

Slide 41

```
{
  int x;
  unsigned long int v2 = 1234L;
  char c1, c2 = 'c';
  float z, zz, zzz;
  const double pi = 3.14;
}
```



5 Types

5.1 Types élémentaires

Slide 42

- Type spécial : **`void`**
  ne peut pas être considéré comme un type « normal »
- Type caractère : **`char`** (**`unsigned`**[a], **`signed`**)
- Type entier : **`short`**, **`int`**, **`long`** (**`unsigned`**, **`signed`**)
- Type réel : **`float`**, **`double`**, **`long double`**
- **PAS** de type booléen : 0 représente le faux, et une valeur différente de 0 le vrai.

[a] entier non signé obéissant aux règles de l'arithmétique modulo 2^n (n étant le nombre de bits de représentation), $0 .. 2^n - 1$

Slide 43

Le standard IEEE en format simple précision utilise 32 bits pour représenter un nombre réel.

Soit x un nombre réel,

$$x = (-1)^S \times 2^{E-127} \times 1, M$$

où

S est le bit de signe (1 bit),

E l'exposant (8 bits),

M la mantisse (23 bits).

S	E			M				
0	1	...	8	9	10	...	30	31

Pour la double et quadruple précision, seul le nombre de bits de l'exposant et de la mantisse diffèrent.

5.2 Type énuméré

Slide 44

```
#include <stdio.h>

enum Lights { green, yellow, red };
enum Cards { diamond=1, spade=-5, club=5, heart };
enum Operator { Plus = '+', Min = '−', Mult = '*', Div = '/' };

int main (int argc, char *argv[]) {
  enum Lights feux = red;
  enum Cards jeu = spade;
  enum Operator op = Min;

  fprintf(stdout, "L = %d %d %d\n", green, yellow, red);
  fprintf(stdout, "C = %d %d %d %d\n", diamond, spade, club, heart);
  fprintf(stdout, "O = %d %d %d %d\n", Plus, Min, Mult, Div);
  jeu = yellow;
  fprintf(stdout, "%d %d %c\n", feux, jeu, op);
  return 0;
}
```

Slide 45

On peut affecter ou comparer des variables de types énumérés.

Un type énuméré est considéré comme de type **`int`** : la numérotation commence à 0, mais on peut donner n'importe quelles valeurs entières.

Comme on a pu le remarquer, il n'y a pas de vérification.

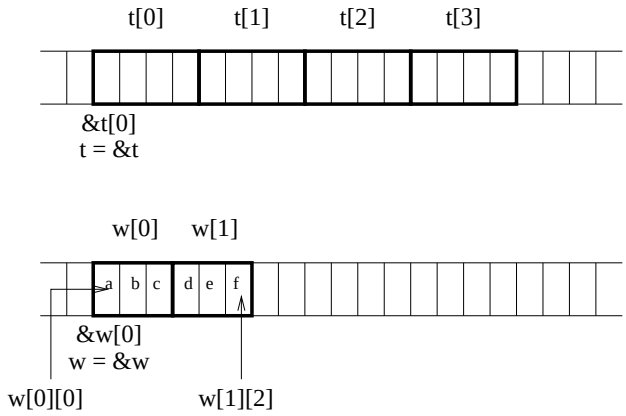
5.3 Types structurés

5.3.1 Type tableau

Slide 46

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  int t[4], i, j, u[] = {0, 1, 2, 3}; float x[3][10];
  char w[][3] = {{'a', 'b', 'c'}, {'d', 'e', 'f'}};

  for (i = 0; i < 4; i++) {
    t[i] = 0; fprintf(stdout, "t[%d]=%d␣", i, t[i]);
  }
  fputc('\n', stdout);
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      w[i][j] = 'a'; fprintf(stdout, "w[%d][%d]=%c␣", i, j, w[i][j]);
    }
    fputc('\n', stdout);
  }
  return 0;
}
```

Slide 47



Slide 48

Tableaux à une seule dimension : possibilité de tableau de tableau. Dans ce cas, la dernière dimension varie plus vite.

Indice entier uniquement (borne inférieure toujours égale à 0).

On peut initialiser un tableau lors de sa déclaration (par agrégat).

Dimensionnement automatique par agrégat (seule la première dimension peut ne pas être spécifiée).

Les opérations se font élément par élément. Aucune vérification sur le dépassement des bornes.

Slide 49

La dimension doit être connue statiquement.

```
{
   int n = 10;
   int t[n]; /* INTERDIT */
}
```

Ce qu'il faut plutôt faire :

```
#define N 10
...
{
   int t[N]; /* c'est le préprocesseur qui travaille */
}
```

Slide 50

Comment initialiser un tableau ?

```
#include <stdio.h>

void init (int t[], int n, int v) {
  /* Antécédent : n et v initialisés */
  /* Conséquent : ∀ i ∈ [0..n-1], t[i] = v */
  int i;
  for (i = 0; i < n; i++)
    t[i] = v;
}

void aff (int t[], int n) {
  /* Antécédent : n initialisé */
  /* Rôle : affiche les n premiers éléments de t
     séparés par des blancs et terminés par fdl */
  int i;
  for (i = 0; i < n; i++)
    fprintf(stdout, "%d ", t[i]);
  fputc('\n', stdout);
}

int main (int argc, char *argv[]) {
  int tab[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  aff(tab, 5);
  init(tab, 5, 0);
  aff(tab, sizeof tab / sizeof tab[0]);
  return 0;
}
```

Slide 51

5.3.2 Chaînes de caractères

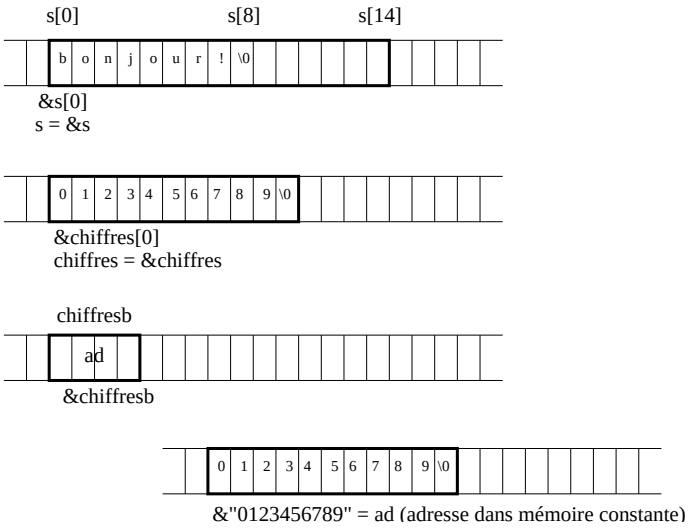
En fait des tableaux de caractères : ce n'est pas un type en soi.

Par convention, elles se terminent par le caractère nul '\0' (valeur 0).

Il n'y a pas d'opérations pré-définies (puisque ce n'est pas un type), mais il existe des fonctions de bibliothèque, dont le fichier de déclarations s'appelle `string.h` (toutes ces fonctions suivent la convention).

```
{
  char string1[100];
  char s[15] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '!', '\0'};
             /* "bonjour!" */
  char chiffres[] = "0123456789";
  char *chiffresb = "0123456789";
}
```

Slide 52



Slide 53

Comment initialiser une chaîne ? (utilisation des fonctions de bibliothèque)

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

int main (int argc, char *argv[]) {
  char chaine[] = "bonjour ";
  char chRes[256];

  fprintf(stdout, "%s\n", strcpy(chRes, chaine));
  fprintf(stdout, "%s\n", strcat(chRes, "tout le monde!"));
  fprintf(stdout, "%s\n", chRes);
  return 0;
}
```

Comment écrire une chaîne sur la sortie standard ?

```
fprintf(stdout, "%s", chaine);
fputs(s, stdout);
```

5.3.3 Structures

Slide 54

```
#include <stdio.h>
struct date {
  short jour, mois, annee;
};
struct personne {
  int num; /* numéro de carte */
  struct date neLe;
  short tn[3]; /* tableau de notes */
};
int main (int argc, char *argv[]) {
  struct personne p, etud[2];
  short e, n, somme;
  p.num = 15; p.neLe.jour = 5; p.neLe.mois = 10;
  p.neLe.annee = 1992; p.tn[0] = 10; p.tn[1] = 15;
  p.tn[2] = 20; etud[0] = p;
  p.num = 20; p.neLe.jour = 1; p.neLe.mois = 1;
  p.neLe.annee = 1995; p.tn[0] = 0; p.tn[1] = 5;
  p.tn[2] = 10; etud[1] = p;
  for (e = 0; e < 2; e++) {
    somme = 0;
    for (n = 0; n < 3; n++)
      somme = somme + etud[e].tn[n];
    fprintf(stdout, "moy␣de␣%d,␣ne␣en␣%d␣=␣%.2f\n",
            etud[e].num, etud[e].neLe.annee,
            somme/(double)3);
  }
  return 0;
}
```

Slide 55

En fait le type article de Pascal ou d'Ada (`record`).

Objet composite formé d'éléments de types quelconques.

La place réservée est la somme de la longueur des champs (au minimum, à cause de l'alignement).

On peut affecter des variables de types structures.

Lors de la déclaration de variables de types structures, on peut initialiser avec un agrégat.

5.3.4 Unions

Slide 56

```
#include <stdio.h>

enum forme { rectangle, carre, cercle };
struct point {
  short x, y;
};
struct dessin {
  struct point p;
  /* coin sup gauche si carré ou rectangle
     centre du cercle sinon */
  enum forme f;
  union {
    struct point p2;
    /* rectangle : coin inf droit */
    short lg;
    /* carré : cote en cm */
    short rayon;
    /* cercle */
  } autre;
};
```

Slide 57

```
int main (int argc, char *argv[]) {
  struct dessin c, r;

  c.p.x = 2; c.p.y = 3;
  c.f = carre;
  c.autre.lg = 10;
  fprintf(stdout, "aire␣du␣carré␣=␣%dcm2\n",
          c.autre.lg * c.autre.lg);
  r.p.x = 1; r.p.y = 4;
  r.f = rectangle;
  r.autre.p2.x = 7; r.autre.p2.y = 1;
  fprintf(stdout, "aire␣du␣rectangle␣=␣%dcm2\n",
          (r.autre.p2.x - r.p.x)
          * (r.p.y - r.autre.p2.y));
  return 0;
}
```

Un peu comme le type article avec variante de Pascal ou d'Ada.

La place réservée est le maximum de la longueur des champs.

Slide 58

Sert à partager la mémoire dans le cas où l'on a des objets dont l'accès est exclusif.

Sert à interpréter la représentation interne d'un objet comme s'il était d'un autre type.

5.3.5 Les champs de bits

Slide 59

```
#include <stdio.h>
typedef struct {
  unsigned rouge : 8;
  unsigned vert : 8;
  unsigned bleu : 8;
  unsigned : 8;
} couleur;
int main (int argc, char *argv[]) {
  couleur rouge = {0xFF, 0x00, 0x00}, vert = {0x33, 0xFF, 0x33},
          indef = {0xFF, 0xFF, 0xFF};
  fprintf(stdout, "rouge = %x\tvert = %x\tindef = %x\n",
          rouge, vert, indef);
  fprintf(stdout, "rouge = %u\tvert = %u\tindef = %u\n",
          rouge, vert, indef);
  return 0;
}
```

Pour les champs de bits, on donne la longueur du champ en bits; longueur spéciale 0 pour forcer l'alignement (champ sans nom pour le remplissage). Attention aux affectations (gauche à droite ou *vice versa*).

Utilisés dans les programmes proches de la machine : structure de données définie par le matériel.

Slide 60

Dépend fortement de la machine, donc difficilement transportable.

- Architectures droite à gauche (PDP-11, Vax, NS 32000) : **little endian**
  plus significatif → moins significatif
- Architectures gauche à droite (IBM, Intel 8086, Motorola 68000) : **big endian**
  moins significatif → plus significatif

5.4 Définition de types

Slide 61

```
#ifndef _TYPEDEF_H
#define _TYPEDEF_H

typedef int Longueur;
typedef char tab_car[30];
typedef struct people {
  char Name[20];
  short Age;
  long SSNumber;
} Human;
typedef struct node *Tree;
typedef struct node {
  char *word;
  Tree left, right;
} Node;
typedef int (*PFI) (char *, char *);

#endif
```

6 Instructions

6.1 Instruction vide

Slide 62

Dénotée par le point-virgule.

Le point-virgule sert de terminateur d'instruction ⇒ la liste des instructions est une suite d'instructions se terminant *toutes* par un ';'.

```
for (; (fgetc(stdin) != EOF); nb++)
  /* rien */ ;
```

Note : Ne pas mettre systématiquement un `;` après la parenthèse fermante du **`for`**...

6.2 Expression, affectation et instruction

Slide 63

En général { instruction = action ; expression = valeur

Or, en C, une expression peut avoir un effet de bord.

Une expression a un *type* statique (c'est-à-dire qui ne dépend pas de l'exécution du programme) et une *valeur*.

L'affectation en C est dénotée par le signe '=' et est à la fois une instruction et une expression : c'est un opérateur à effet de bord.

Slide 64

Affectation composée

`partie_gauche ?= expression`

où ? est l'un des opérateurs suivants :

`* / % - + << >> & ^ |`

```
int main (int argc, char *argv[]) {
  int a, b;

  a = b = 3;
  3; /* warning: statement with no effect */
  a += 3; /* a = a + 3 */
  a *= 3 - 5; /* a = a * (3 - 5) */
  a *= (3 - 5); /* a = a * (3 - 5) */
  return 0;
}
```

Slide 65

6.3 Bloc

Sert à grouper plusieurs instructions en une seule.

Sert à restreindre (localiser) la visibilité d'une variable.

Sert à marquer le corps d'une fonction.

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  int i = 3, j = 5;
  {
    int i = 4;
    fprintf(stdout, "i = %d, j = %d\n", i, j);
    i++; j++;
  }
  fprintf(stdout, "i = %d, j = %d\n", i, j);
  return 0;
}
```

6.4 Instruction conditionnelle

Slide 66

Il n'y a pas de mot-clé « alors » et la partie « sinon » est facultative.

La condition doit être parenthésée.

Rappel : en C, il n'y a pas d'expression booléenne ; une expression numérique est considérée comme faux si sa valeur est égale à 0, vrai sinon.

Slide 67

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

int main (int argc, char *argv[]) {
  int x, y = -3, z = -5;
  fprintf(stdout, "valeur de x? ");
  fscanf(stdin, "%d", &x);
  if (x == 0)
    x = 3;
  else if (x == 2) {
    x = 4; y = 5;
  }
  else
    z = x = 10;
  fprintf(stdout, "x = %d, y = %d, z = %d\n", x, y, z);
  if (0) x = 3; else x = 4; /* à éviter */
  return 0;
}
```

6.5 Instruction d'aiguillage

Slide 68

Là encore, l'expression doit être parenthésée.

Les étiquettes de branchement doivent avoir des valeurs calculables à la compilation et de type *discret*.

Pas d'erreur si aucune branche n'est sélectionnée.

Exécution des différentes branches en séquentiel ⇒ ne pas oublier une instruction de débranchement (**`break`** par exemple).

Slide 69

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  short i = 0, nbc = 0, nbb = 0, nba = 0;

  if (argc != 2) {
    fprintf(stderr, "usage: %s chaîne\n", argv[0]);
    return 1;
  }
  while (argv[1][i]) {
    switch (argv[1][i]) {
    case '0' : case '1' : case '2' : case '3' :
    case '4' : case '5' : case '6' : case '7' :
    case '8' : case '9' :
      nbc++; break;
    case ' ' : case '\t' : case '\n' :
      nbb++; break;
    default :
      nba++;
    }
    i++;
  }
  fprintf(stdout, "chiffres = %d, blancs = %d, "
          "autres = %d\n", nbc, nbb, nba);
  return 0;
}
```

6.6 Instructions de bouclage (while, for, do)

- Boucle while

while *(expression_entière)*

instruction

Slide 70

```
#include <stdio.h>
#define MAX 30

int main (int argc, char *argv[]) {
  char tab[MAX], c;
  int i;

  i = 0;
  while ((i < MAX - 1)
         && (c = fgetc(stdin)) != EOF)
    tab[i++] = c;
  tab[i] = '\0';
  fprintf(stdout, "\n *%s*\n", tab);
  return 0;
}
```

- Boucle do

do

instruction

while *(expression_entière);*

Slide 71

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

#define NB 3

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

  do {
    fprintf(stdout, "Voulez-vous sortir (Oo/Nn)? ");
    fgets(rep, NB, stdin); rep[0] = toupper(rep[0]);
  } while (strcmp(rep, "O\n"));
  return 0;
}
```

Slide 72

- Boucle for

for (init; cond_conti; rebouclage)
instruction

⇔

init;
while (cond_conti) {
instruction; rebouclage;
}

```
#include <stdio.h>
#define MAX 30

int main (int argc, char *argv[]) {
  char tab[MAX] = "toto";
  int i;

  fprintf(stdout, "*%s*\n", tab);
  for (i = 0; i < MAX; i++)
    tab[i] = '\O';
  fprintf(stdout, "*%s*\n", tab);
  return 0;
}
```

Slide 73

```
#include <stdio.h>

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

  i = 0;
  while (i < 10) {
    i++;
  }
  fprintf(stdout, "i = %d\n", i);

  i = 0;
  do {
    i++;
  } while (i < 10);
  fprintf(stdout, "i = %d\n", i);

  for (i = 0; i < 10; i++);
  fprintf(stdout, "i = %d\n", i);
  return 0;
}
```

Slide 74

6.7 Instructions de débranchement

- Instruction **`break`**
  - Utilisée dans un **`switch`** ou dans une boucle
  - Se débranche sur la première instruction qui suit le **`switch`** ou la boucle
- Instruction **`continue`**
  - Utilisée dans les boucles
  - Poursuit l'exécution de la boucle au test (ou au rebouclage)

Slide 75

- Instruction **`goto`** : l'étiquette est placée comme suit

```
étiquette :
```

- Instruction **`return`**
  - Provoque la sortie d'une fonction
  - La valeur de retour est placée derrière le mot-clé
- Fonction `exit`
  - Met fin à l'exécution d'un programme
  - 0 en paramètre indique l'arrêt normal

Slide 76

```
#include <stdio.h>

void f (int j) {
  int i = 0;

  while (i < j) {
    /* instruction? */
    i++;
  }
  fprintf(stdout, "fin de f, i = %d\n", i);
}

int main (int argc, char *argv[]) {
  int i = 100;

  f(i);
  fprintf(stdout, "fin de main, i = %d\n", i);
  return 0;
}
```

Slide 77

instruction?	résultat
break;	fin de f, i = 0 fin de main, i = 100
continue;	ça boucle
return;	fin de main, i = 100
exit(2);	
	fin de f, i = 100 fin de main, i = 100

7 Opérateurs

7.1 Affectation

Slide 78

C'est un opérateur (signe `=`) dont les opérandes sont de tout type (*attention* si de type tableau).

Cas particulier des opérateurs unaires (d'in-/dé-)crémentation : `++`, `--`.

Attention : l'ordre d'évaluation n'est pas garanti.

```
t[i++] = v[i++]; /* à éviter */
```

7.2 Opérateurs de calcul

Slide 79

Arithmétiques	`+ * - / %`
Relationnels	`< <= > >= == !=`
Logiques	`! || &&`
Opérateurs bit à bit	`~ & | ^ << >>`

```
3 + 4          3 * 4 + 5
3 / 4          3.0 / 4
3 % 4          !(n % 2)
(x = 1) || b
(x = 0) && b
```

Slide 80

```
#include <stdio.h>

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

  for (i = 0; i < 10; i++)
    fprintf(stdout, "i = %hx, ~i = %hx, !i = %hx,"
            " i << 1 = %hx, i >> 1 = %hx\n",
            i, ~i, !i, i << 1, i >> 1);
  return 0;
}
```

Slide 81

Résultat :

```
i = 0, ~i = ffff, !i = 1, i << 1 = 0, i >> 1 = 0
i = 1, ~i = fffe, !i = 0, i << 1 = 2, i >> 1 = 0
i = 2, ~i = fffd, !i = 0, i << 1 = 4, i >> 1 = 1
i = 3, ~i = fffc, !i = 0, i << 1 = 6, i >> 1 = 1
i = 4, ~i = fffb, !i = 0, i << 1 = 8, i >> 1 = 2
i = 5, ~i = fffa, !i = 0, i << 1 = a, i >> 1 = 2
i = 6, ~i = fff9, !i = 0, i << 1 = c, i >> 1 = 3
i = 7, ~i = fff8, !i = 0, i << 1 = e, i >> 1 = 3
i = 8, ~i = fff7, !i = 0, i << 1 = 10, i >> 1 = 4
i = 9, ~i = fff6, !i = 0, i << 1 = 12, i >> 1 = 4
```

Slide 82

7.3 Opérateurs sur les types

- Taille d'un objet (nombre d'octets nécessaires à la mémorisation d'un objet)

```
sizeof (nom_type)
sizeof expression
```

Renvoie une valeur de type `size_t` déclaré dans le fichier de déclarations `stdlib.h`.

Slide 83

```
#include <stdio.h>

#define imp(s, t) fprintf(stdout, \
                          "sizeof␣%s␣=␣%d\n", s, sizeof(t))

int main (int argc, char *argv[]) {
  int t1[10];
  float t2[20];

  imp("char", char);
  imp("short", short);
  imp("int", int);
  imp("long", long);
  imp("float", float);
  imp("double", double);
  imp("long␣double", long double);
  fprintf(stdout, "sizeof␣t1␣=␣%d,␣sizeof␣t2␣=␣%d\n",
          sizeof t1, sizeof t2);
  fprintf(stdout, "sizeof␣t1[0]␣=␣%d,␣sizeof␣t2[1]␣=␣%d\n",
          sizeof t1[0], sizeof t2[1]);
  return 0;
}
```

Slide 84

Exécution :

```
$  a.out
sizeof char = 1
sizeof short = 2
sizeof int = 4
sizeof long = 4
sizeof float = 4
sizeof double = 8
sizeof long double = 12
sizeof t1 = 40, sizeof t2 = 80
sizeof t1[0] = 4, sizeof t2[1] = 4
$
```

Slide 85

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

void f (int t[]) {
  fprintf(stdout, "f: sizeof t = %d, sizeof t[0] = %d\n",
          sizeof t, sizeof t[0]);
}
void g (char s[]) {
  fprintf(stdout, "g: sizeof s = %d, sizeof s[0] = %d\n",
          sizeof s, sizeof s[0]);
  fprintf(stdout, "g: lgr de s = %d\n", strlen(s));
}
int main (int argc, char *argv[]) {
  int t1[10];
  char s1[] = "12345";
  fprintf(stdout, "main: sizeof t1 = %d\n", sizeof t1);
  f(t1);
  fprintf(stdout, "main: sizeof s1 = %d, strlen(s1) = %d\n",
          sizeof s1, strlen(s1));
  g(s1);
  return 0;
}
```

Slide 86

Exécution :

```
$  a.out
main: sizeof t1 = 40
f: sizeof t = 4, sizeof t[0] = 4
main: sizeof s1 = 6, strlen(s1) = 5
g: sizeof s = 4, sizeof s[0] = 1
g: lgr de s = 5
$
```

Slide 87

- Conversion explicite (« casting » ou transtypage)

```
(<type>) expression
```

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  fprintf(stdout, "%.2f␣", 3/(double)4);
  fprintf(stdout, "%d␣",  (int)4.5);
  fprintf(stdout, "%d␣",  (int)4.6);
  fprintf(stdout, "%.2f",  (double)5);
  fputc('\n', stdout);
  return 0;
}
```

7.4 Opérateur de condition

condition ? expression$_1$: expression$_2$

Seul opérateur ternaire du langage.

Slide 88

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

int main (int argc, char *argv[]) {
  int x, y, n;
  if (argc != 2) {
    fprintf(stderr, "usage: %s nb\n", argv[0]); return 1;
  }
  n = atoi(argv[1]);
  x = (n % 2) ? 0 : 1;
  y = (n == 0) ? 43 : (n == -1) ? 52 : 100;
  fprintf(stdout, "x = %d, y = %d\n", x, y);
  return 0;
}
```

7.5 Opérateur virgule

Slide 89

Le résultat de

$$\text{expr}_1, \text{expr}_2, \ldots \text{expr}_n$$

est le résultat de expr$_n$.

Les expr$_1$, ..., expr$_{n-1}$ sont évaluées, mais leurs résultats oubliés (sauf si effet de bord).

Slide 90

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  int a, b, i, j, t[20];

  for (i = 0, j = 19; i < j; i++, j--)
    t[i] = t[j];
  fprintf(stdout, "%d\n", (a = 1, b = 2));
  fprintf(stdout, "%d\n", (a = 1, 2));
  return 0;
}
```

Slide 91

7.6 Tableau récapitulatif

Opérateurs rangés par ordre de priorité décroissante (15 niveaux)

Types	Symboles	Associativité
postfixé	`()`, `[]`, `.`, `->`, `++`, `--`	G → D
unaire	`&`, `*`, `+`, `-`, `~`, `!`, `++`, `--`, **`sizeof`**	D → G
casting	`(<type>)`	D → G
multiplicatif	`*`, `/`, `%`	G → D
additif	`+`, `-`	G → D
décalage	`<<`, `>>`	G → D
relationnel	`<`, `>`, `<=`, `>=`	G → D
(in)égalité	`==`, `!=`	G → D
et bit à bit	`&`	G → D
xor	`^`	G → D
ou bit à bit	`|`	G → D
et logique	`&&`	G → D
ou logique	`||`	G → D
condition	`? :`	D → G
affectation	`=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`	D → G
virgule	`,`	G → D

Slide 92

8 Pointeurs

Les pointeurs sont très utilisés en C.

En général, ils permettent de rendre les programmes

- plus compacts,
- plus efficaces,
- mais beaucoup moins lisibles.

L'accès à un objet se fait de façon indirecte.

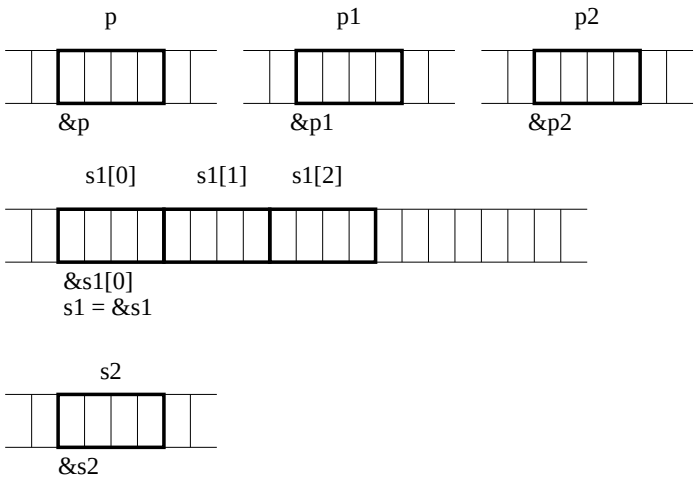
Slide 93

8.1 Comment déclarer un pointeur ?

*type *nom_pointeur;*

```
{
  char *p;
  int *p1, *p2;
  struct {int x, y;} *q;
  void *r;
  int *s1[3];
  int (*s2)[3];
  int (*fct)(void);
  int (*T[5])(void);
  double *f(void);
  int *(*fonc(void))(void);
}
```

Slide 94



Slide 95

8.2 Qu'est-ce qu'un pointeur ?

Un pointeur contient l'adresse d'un objet, l'adresse étant

- statique, si l'objet a été alloué de façon statique ;

```
{
  int i, *p, t[10];
  p = &i; *p = 3; p = t; *p = 6;
}
```

- dynamique, si l'objet a été alloué explicitement.

```
{
  int *p = malloc(sizeof(int) * 4);
  char *s = malloc(sizeof(char) * (strlen("123" + 1));
}
```

L'opérateur `*` permet de *référer* l'objet pointé. Si `p` est une variable de type `int *`, alors `*p` désigne l'entier pointé par `p`.

L'opérateur `&` permet de donner l'adresse d'un objet.

Pointeur « universel » : `void *`

Slide 96

Pointeur sur « rien du tout » : constante entière 0 ou `NULL` (macro définie dans le fichier de déclarations `stdlib.h`)

Simplification d'écriture : si `q` est un pointeur sur une structure contenant un champ `x`

```
q -> x ⇔ (*q).x
```

8.3 Comment allouer de la mémoire ?

En utilisant les fonctions standard suivantes, l'utilisateur a la garantie que la mémoire allouée est contigue et respecte les contraintes d'alignement.

Ces fonctions se trouvent dans le fichier de déclarations `stdlib.h`.

Slide 97

- `void *malloc (size_t taille);` permet d'allouer `taille` octets dans le tas.
- `void *calloc (size_t nb, size_t taille);` permet d'allouer `taille` $\times$ `nb` octets dans le tas, en les initialisant à `0`.
- `void *realloc (void *ptr, size_t taille);` permet de réallouer de la mémoire.
- `void free (void *ptr);` permet de libérer de la mémoire (ne met pas `ptr` à `NULL`).

Slide 98

Attention aux références « fantômes » ...

```
#include <stdlib.h>

int *f1 (void) {
  int a = 3;
  return &a;
  /* warning: function returns address of local variable */
}
int *f2 (void) {
  int *a = malloc(sizeof(int));
  *a = 3;
  return a;
}
int main (int argc, char *argv[]) {
  int *p1 = f1();
  int *p2 = f2();
  return 0;
}
```

Slide 99

8.4 Que se passe-t-il ? (1)

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  int i = 0;
  char c, *pc;
  int *pi;

  fprintf(stdout, "sizeof(char) = %d, "
          "sizeof(int) = %d,\n"
          "sizeof(void *) = %d\n\n", sizeof(char),
          sizeof(int), sizeof(void *));
  fprintf(stdout, "&i = %p, &c = %p,\n&pc = %p, &pi = %p\n",
          (void *)&i, (void *)&c, (void *)&pc, (void *)&pi);
  c = 'a';
  pi = &i; pc = &c;
  *pi = 50; *pc = 'B';
  fprintf(stdout, "i = %d, c = %c,\npc = %p, *pc = %c,\n"
          "pi = %p, *pi = %d\n", i, c, pc, *pc, pi, *pi);
  return 0;
}
```

Slide 100

Exécution :

```
$  a.out
sizeof(char) = 1, sizeof(int) = 4,
sizeof(void *) = 4

&i = 0xbffff7b4, &c = 0xbffff7b3,
&pc = 0xbffff7ac, &pi = 0xbffff7a8
i = 50, c = B,
pc = 0xbffff7b3, *pc = B,
pi = 0xbffff7b4, *pi = 50
$
```

Slide 101

8.5 Que se passe-t-il ? (2)

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

int main (int argc, char *argv[]) {
  int i = 0;
  char c, *pc;
  int *pi;

  fprintf(stdout, "sizeof(char) = %d, "
          "sizeof(int) = %d,\n"
          "sizeof(void *) = %d\n\n", sizeof(char),
          sizeof(int), sizeof(void *));
  fprintf(stdout, "&i = %p, &c = %p,\n"
          "&pc = %p, &pi = %p\n",
          (void *)&i, (void *)&c, (void *)&pc,
          (void *)&pi);
  c = 'a';
  pc = calloc(1, sizeof(char));
  pi = calloc(1, sizeof(int));
  *pi = 50; *pc = 'B';
  fprintf(stdout, "i = %d, c = %c,\n"
          "pc = %p, *pc = %c,\n"
          "pi = %p, *pi = %d\n", i, c, pc, *pc,
          pi, *pi);
  return 0;
}
```

Slide 102

Exécution:

```
$  a.out
sizeof(char) = 1, sizeof(int) = 4,
sizeof(void *) = 4

&i = 0xbffff7b4, &c = 0xbffff7b3,
&pc = 0xbffff7ac, &pi = 0xbffff7a8
i = 0, c = a,
pc = 0x80497c8, *pc = B,
pi = 0x80497d8, *pi = 50
$
```

Slide 103

8.6 Paramètres par référence

En C, le passage des paramètres se fait toujours *par valeur*.

Les pointeurs permettent de simuler le passage par référence.

```
#include <stdio.h>
void Swap (int a, int b) {
  int aux;
  aux = a;
  a = b;
  b = aux;
}
int main (int argc, char *argv[]) {
  int x = 3, y = 5;
  fprintf(stdout, "av: x = %d, y = %d\n",
          x, y);
  Swap(x, y);
  fprintf(stdout, "ap: x = %d, y = %d\n",
          x, y);
  return 0;
}
```

Slide 104

```
#include <stdio.h>

void Swap (int *a, int *b) {
  int aux;

  aux = *a;
  *a = *b;
  *b = aux;
}

int main (int argc, char *argv[]) {
  int x = 3, y = 5;

  fprintf(stdout, "av: x = %d, y = %d\n", x, y);
  Swap(&x, &y);
  fprintf(stdout, "ap: x = %d, y = %d\n", x, y);
  return 0;
}
```

Slide 105

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

void imp (int t[], int lg) {
  int i;
  for (i = 0; i < lg; i++)
    fprintf(stdout, "%d ", t[i]);
  fputc('\n', stdout);
}

void MAZ (int t[], int lg) {
  int i;
  for (i = 0; i < lg; i++)
    t[i] = 0;
}

void Mess (int t[], int lg) {
  int i;
  t = malloc(sizeof(int) * lg);
  for (i = 0; i < lg; i++)
    t[i] = 33;
  imp(t, lg);
}

#define MAX 10
int main (int argc, char *argv[]) {
  int t[MAX];
  MAZ(t, MAX); imp(t, MAX);
  Mess(t, MAX); imp(t, MAX);
  return 0;
}
```

8.7 Pointeurs et tableaux

Slide 106

Notions très liées en C.

Le nom du tableau correspond à l'adresse de départ du tableau (en fait l'adresse du premier élément). Si `int t[10];`, alors `t` = `&t[0]` et `t` = `&t`.

Si on passe un tableau en paramètre, seule l'adresse du tableau est passée (il n'existe aucun moyen de connaître le nombre d'éléments).

En fait, l'utilisation des crochets est une simplification d'écriture. La formule suivante est appliquée pour accéder à l'élément `i`.

`a[i]` $\Leftrightarrow$ `*(a + i)`

Slide 107

Il existe des opérations arithmétiques sur les pointeurs :

pointeur + entier $\Rightarrow$ pointeur

pointeur - entier $\Rightarrow$ pointeur

pointeur - pointeur $\Rightarrow$ entier

Slide 108

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
  int t1[10], t2[20];

  fprintf(stdout, "t1 = %p, t2 = %p\n", t1, t2);
  fprintf(stdout, "t1 + 3 = %p, &t1[3] = %p\n", t1 + 3, &t1[3]);
  fprintf(stdout, "&t1[3] - 3 = %p\n", &t1[3] - 3);
  fprintf(stdout, "t1 - t2 = %d\n", t1 - t2);
  fprintf(stdout, "t2 - t1 = %d\n", t2 - t1);
  return 0;
}
```

```
$  a.out
t1 = 0xbffff780, t2 = 0xbffff730
t1 + 3 = 0xbffff78c, &t1[3] = 0xbffff78c
&t1[3] - 3 = 0xbffff780
t1 - t2 = 20
t2 - t1 = -20
$
```

Slide 109

```c
#include <stdio.h>

void aff (int t[], int n) {
  /* Antécédent : n initialisé */
  /* Rôle : affiche les n premiers éléments de t
     séparés par des blancs et terminés par une fdl */
  int i;
  for (i = 0; i < n; i++)
    fprintf(stdout, "%d ", t[i]);
  fputc('\n', stdout);
}

int main (int argc, char *argv[]) {
  int tab[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  aff(tab, sizeof tab / sizeof tab[0]);
  aff(tab + 5, 5);
  return 0;
}
```

Slide 110

```
#include <stdio.h>
#include <stdlib.h>
#define M 5
void saisieTab (int t[], short n) {
  /* Antécédent : t et n initialisés */
  /* Rôle : saisie de n entiers dans t */
  short i;
  fprintf(stdout, "les %d nombres? ", n);
  for (i = 0; i < n; i++)
    fscanf(stdin, "%d", &t[i] /* t + i */);
}
void copieTab (int t[], int *p, short n) {
  /* Antécédent : t, p et n initialisés */
  /* Conséquent : ∀ i ∈ [0..n-1] p[i] = t[i] */
  short i;
  for (i = 0; i < n; i++)
    *p++ = t[i];
}
void affTab (int t[], short n) {
  /* Antécédent : n initialisé */
  /* Rôle : affiche les n premiers éléments de t
     séparés par des blancs et terminés par fdl */
  short i;
  for (i = 0; i < n; i++)
    fprintf(stdout, "%d ", t[i]);
  fputc('\n', stdout);
}
int main (int argc, char *argv[]) {
  int *p = calloc(M, sizeof(int)), t[M];
  saisieTab(t, M); affTab(t, M); affTab(p, M);
  copieTab(t, p, M); affTab(t, M); affTab(p, M);
  return 0;
}
```

Slide 111

8.8 Pointeurs et chaînes de caractères

Une *constante* chaîne de caractères a comme valeur l'adresse mémoire de son premier caractère (on ne peut la modifier).

Son type est donc pointeur sur caractères.

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  char y[] = "1234";
  char *x = "1234";
  char z[10] = "1234";
  char tc[] = {'1', '2', '3', '4'};

  fprintf(stdout, "x=%p &x=%p\n", x, (void *)&x);
  fprintf(stdout, "y=%p &y=%p\n", y, (void *)&y);
  return 0;
}
```

Slide 112

Pour parcourir une chaîne

```
#include <stdio.h>

void aff (char *s) {
  while (*s) {
    fputc(*s, stdout);
    s++;
  }
  fputc('\n', stdout);
}

int main (int argc, char *argv[]) {
  char s1[] = "bonjour␣vous!",
    *s2 = "et␣vous␣aussi";
  aff(s1);
  aff(s2);
  return 0;
}
```

Slide 113

8.9 Pointeurs et fonctions

En C, le type pointeur sur fonction existe :

```
typedef int (*tpf) (int);
/* déclaration de tpf comme étant un type pointeur sur fonction prenant
   un int en paramètre et renvoyant un int */

float (*vpf) (double, char *);
/* déclaration de vpf comme étant une variable de type pointeur sur
   fonction prenant en paramètres un double et un char * et renvoyant
   un float */

char *f (int);
/* déclaration de f comme étant une constante de type pointeur sur
   fonction prenant en paramètre un int et renvoyant un char * */
```

Slide 114

Lorsqu'on définit une fonction, en fait on déclare une constante de type pointeur sur fonction et sa valeur est l'adresse de la première instruction de la fonction.

```
#include <stdio.h>

int max (int a, int b) {
  return a > b ? a : b;
}

int main (int argc, char *argv[]) {
  fprintf(stdout, "%p\n", (void *)fprintf);
  fprintf(stdout, "%p\n", (void *)&fprintf);
  fprintf(stdout, "%p\n", (void *)max);
  fprintf(stdout, "%p\n", (void *)max(1, 16));
  return 0;
}
```

Slide 115

Fonction en paramètre : il suffit de passer le type du pointeur sur fonction (concerné) en paramètre.

```
#include <stdio.h>
void imp (int t[], int lg) {
  int i;
  for (i = 0; i < lg; i++)
    fprintf(stdout, "%d ", t[i]);
  fputc('\n', stdout);
}
int maz (int a) {
  return 0;
}
int plus1 (int a) {
  return a + 1;
}
void modifierTableau (int t[], int lg,
                      int (*f) (int)) {
  int i;
  for (i = 0; i < lg; i++)
    t[i] = f(t[i]);
}
#define MAX 10
int main (int argc, char *argv[]) {
  int t[MAX];
  modifierTableau(t, MAX, maz); imp(t, MAX);
  modifierTableau(t, MAX, plus1); imp(t, MAX);
  modifierTableau(t, MAX, plus1); imp(t, MAX);
  return 0;
}
```

Slide 116

8.10 Pointeur polymorphe `void*`

Reprenons l'exemple du sous-programme `Swap`

```
#include <stdio.h>

void Swap (int *a, int *b) {
  int aux;

  aux = *a;
  *a = *b;
  *b = aux;
}

int main (int argc, char *argv[]) {
  int x = 3, y = 5;

  fprintf(stdout, "av: x = %d, y = %d\n", x, y);
  Swap(&x, &y);
  fprintf(stdout, "ap: x = %d, y = %d\n", x, y);
  return 0;
}
```

Slide 117

Version avec **`void *`**, afin d'éviter la duplication de code pour le sous-programme si on veut échanger des entiers, des réels, etc.

```
#include <stdlib.h>
#include <string.h>
#include "swapPoly1.h"

void Swap (void *a, void *b, short taille) {
  void *aux = malloc(taille);

  memcpy(aux, b, taille);
  memcpy(b, a, taille);
  memcpy(a, aux, taille);
#if 0
  /* c'est complètement faux:
     on déréférence un void *!!! */
  *aux = *b;
  *b = *a;
  *a = *aux;
#endif
}
```

Slide 118

Et le programme principal :

```
#include <stdio.h>
#include "swapPoly1.h"

#define imp(t, s1, s2, x, s3, y) \
 fprintf(stdout, "%s: %s = " t ", %s = " t "\n", \
         s1, s2, x, s3, y)

int main (int argc, char *argv[]) {
  int a = 3, b = 9;
  float x = 13, y = 19;
  double m = 23.45, n = 25.0;
  char c1 = 'a', c2 = 'B';
  imp("%d", "avant", "a", a, "b", b);
  Swap(&a, &b, sizeof(int));
  imp("%d", "après", "a", a, "b", b);
  imp("%.2f", "avant", "x", x, "y", y);
  Swap(&x, &y, sizeof(float));
  imp("%.2f", "après", "x", x, "y", y);
  imp("%.2f", "avant", "m", m, "n", n);
  Swap(&m, &n, sizeof(double));
  imp("%.2f", "après", "m", m, "n", n);
  imp("%c", "avant", "c1", c1, "c2", c2);
  Swap(&c1, &c2, sizeof(char));
  imp("%c", "après", "c1", c1, "c2", c2);
  return 0;
}
```

Slide 119

Version différente (on ne fait pas la même chose)

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

void Swap (void **a, void **b) {
  void *aux = malloc(sizeof(void *));

  aux = *b;
  *b = *a;
  *a = aux;
}

int main (int argc, char *argv[]) {
  int a = 3, b = 4;
  int *x = &a, *y = &b;
  double *m = malloc(sizeof(double));
  double *n = malloc(sizeof(double));

  fprintf(stdout, "av: a=%d, b=%d, x=%d, y=%d\n",
          a, b, *x, *y);
  Swap(&x, &y);
  fprintf(stdout, "ap: a=%d, b=%d, x=%d, y=%d\n",
          a, b, *x, *y);

  *m = 3.456;
  *n = 1.2345;
  fprintf(stdout, "av: *m=%f, *n=%f\n", *m, *n);
  Swap(&m, &n);
  fprintf(stdout, "ap: *m=%f, *n=%f\n", *m, *n);

  return 0;
}
```

9 Divers

9.1 Ligne de commande

Slide 120

En fait, la fonction `main` a plusieurs paramètres permettant de faire le lien avec UNIX. Son prototype est :

```
int main (int argc, char *argv[]);
```

On utilise par convention `argc` et `argv` (ce ne sont pas des identificateurs réservés).

- Le paramètre `argc` (entier) indique le nombre de paramètres de la commande (incluant le nom de celle-ci).
- Le paramètre `argv` (tableau de chaînes de caractères) contient la ligne de commande elle-même : `argv[0]` est le nom de la commande ; `argv[1]` est le premier paramètre ; etc. ; `argv[argc] == NULL`.

Exemple :

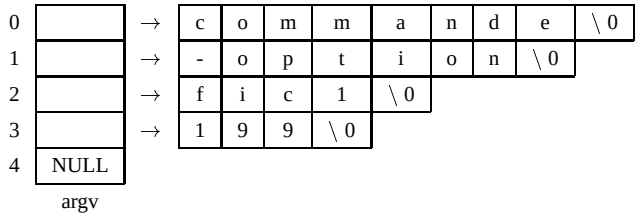
Appel UNIX

```
$  commande -option fic1 199
$
```

Slide 121

Dans le programme C :

- `argc = 4`
- `argv` a cinq éléments significatifs, et on peut le représenter comme suit



Slide 122

```c
#include <stdio.h>

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

  for (i = 1; i < argc; i++)
    fprintf(stdout, "*%s*\n", argv[i]);

  while (*++argv)
    fprintf(stdout, "*%s*\n", *argv);
  /*  version avec for
      for (argv++; *argv;
        fprintf(stdout, "*%s*\n", *argv++)); */
  return 0;
}
```

Slide 123

9.2 Nombre variable de paramètres

La liste variable de paramètres est dénotée par `...` derrière le dernier paramètre fixe. Il y a *au moins* un paramètre fixe.

```c
int fprintf(FILE * stream, const char *format, ... );
```

Quatre macros sont définies dans le fichier de déclarations `stdarg.h`:

- Le « type » `va_list` sert à déclarer le pointeur « se promenant » sur la pile d'exécution.

  ```c
  va_list ap;
  ```

- La macro `va_start` initialise le pointeur de façon à ce qu'il pointe sur le dernier paramètre nommé.

  ```c
  void va_start (va_list ap, last);
  ```

Slide 124

- La macro `va_arg` retourne la valeur du paramètre en cours, et positionne le pointeur sur le prochain paramètre. Elle a besoin du nom du type pour déterminer le type de la valeur de retour et la taille du pas pour passer au paramètre suivant.

```
type va_arg (va_list ap, type);
```

- La macro `va_end` permet de terminer proprement.

```
void va_end (va_list ap);
```

Slide 125

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

void imp (int nb, ...) {
  int i;
  va_list p;

  va_start(p, nb);
  for (i = 0; i < nb; i++)
    fprintf(stdout, "%d ", va_arg(p, int));
  fputc('\n', stdout);
  va_end(p);
}

int main (int argc, char *argv[]) {
  imp(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  imp(5, 'a', 'b', 'c', 'd', 'e');
  imp(2, 12.3, 4.5);
  return 0;
}
```

Slide 126

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

int max (int premier, ...) {
  /* liste d'entiers >= 0 terminée par -1 */
  va_list p;
  int M = 0, param = premier;

  va_start(p, premier);
  while (param > 0) {
    if (param > M)
      M = param;
    param = va_arg(p, int);
  }
  va_end(p);
  return M;
}

int main (int argc, char *argv[]) {
  fprintf(stdout, "%d\n", max(12, 18, 17, 20, 1, 34, 5, -1));
  fprintf(stdout, "%d\n", max(12, 18, -1, 17, 20, 1, 34, 5, -1));
  return 0;
}
```