

1.1

Introduction Langage C

Jean-Charles Régin

Licence Informatique 2ème année

Remerciements

1.2

- Carine Fédèle

Plan du cours

1.3

- 6 cours
- Le cours et les TP
 - ▣ <http://deptinfo.unice.fr/~regin/cours/cours/LangageC/introC.htm>

Bibliographie

1.4

- *The C Programming Language*
Kernighan B.W., Ritchie D.M.
Prentice Hall, 1978
- *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
- *Langage C - Manuel de référence*
Harbison S.P., Steele Jr. G.L.
Masson, 1990
Traduit en français par J.C. Franchitti

Langage C

1.5

- Inventé en 1972 par Dennis Ritchie
- Langage de bas niveau
- Créé pour porter des systèmes d'exploitation (Unix)
- But : un compilateur peut-être écrit en 2 mois
- A Permis de réécrire des programmes assembleurs
- Programmation indépendante de la machine
- Portable : présent sur presque toutes les architectures ayant été inventées

Langage C

1.6

- Utilisé du micro-contrôleur au super-ordinateur
- Présent dans les systèmes embarqués
- Sont écrits en C
 - ▣ Unix
 - ▣ Linux
 - ▣ Windows
 - ▣ Tous les Compilateurs GNU
 - ▣ Python, R
 - ▣ GNOME
 - ▣ Les machines virtuelles JAVA 😊

Langage C

1.7

- Un code C optimisé permet d'obtenir le code le plus rapide
- Fortran, C, C++ sont équivalents
- Java est un petit peu plus lent (mais pas beaucoup)

Tiobe : popularité des langages

1.8

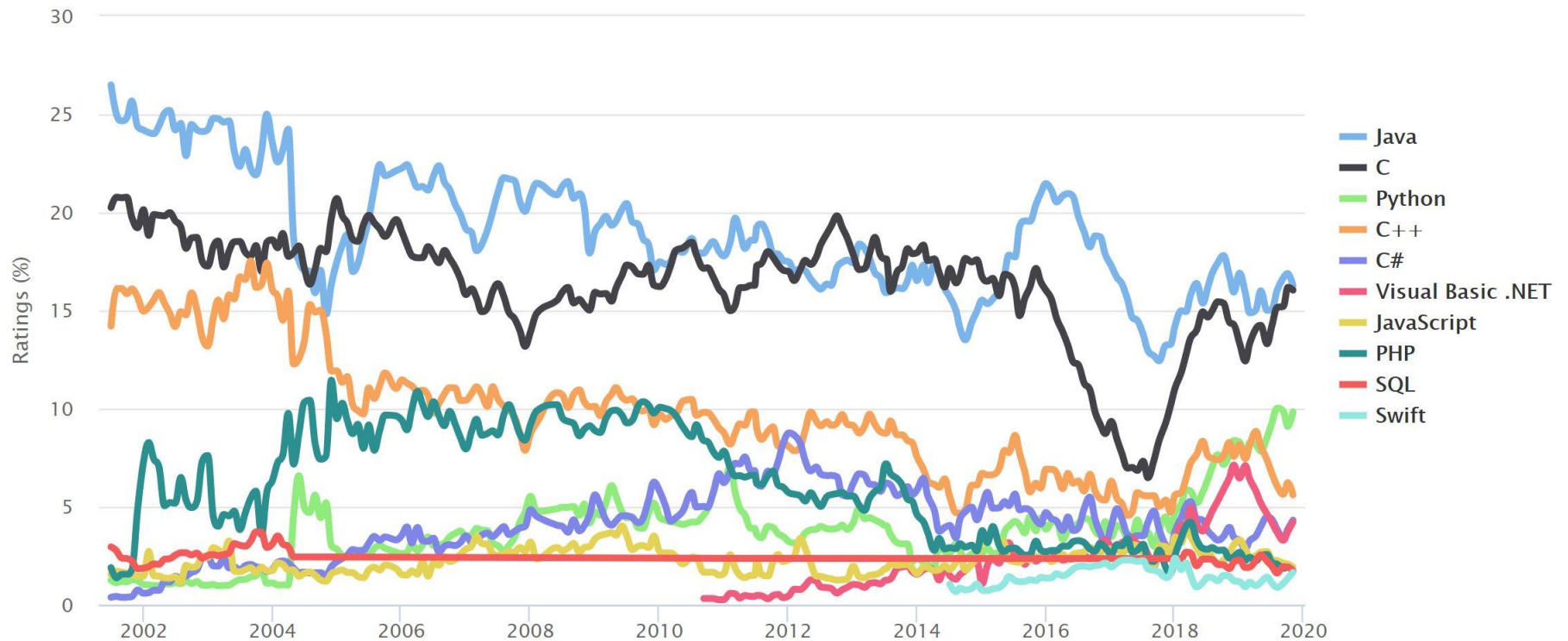
Nov 2019	Nov 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.246%	-0.50%
2	2		C	16.037%	+1.64%
3	4	⬆	Python	9.842%	+2.16%
4	3	⬇	C++	5.605%	-2.68%
5	6	⬆	C#	4.316%	+0.36%
6	5	⬇	Visual Basic .NET	4.229%	-2.26%
7	7		JavaScript	1.929%	-0.73%
8	8		PHP	1.720%	-0.66%
9	9		SQL	1.690%	-0.15%
10	12	⬆	Swift	1.653%	+0.20%
11	16	⬆	Ruby	1.261%	+0.17%
12	11	⬇	Objective-C	1.195%	-0.28%
13	13		Delphi/Object Pascal	1.142%	-0.28%
14	25	⬆	Groovy	1.099%	+0.50%
15	15		Assembly language	1.022%	-0.09%
16	14	⬇	R	0.980%	-0.43%

Tiobe : popularité des langages

1.9

TIOBE Programming Community Index

Source: www.tiobe.com



Langage C

1.10

- Très utilisé car
 - ▣ Bibliothèque logicielle très fournie
 - ▣ Nombre de concepts restreint
 - ▣ Permet de gérer des exécutables qui n'ont besoin de rien pour s'exécuter et pour lesquels on peut contrôler parfaitement la mémoire utilisée (noyau de Syst. Exploit. Logiciel Embarqué)
 - ▣ Contrôle de bas niveau : très puissant

Langage C

1.11

□ Avantages

- ▣ 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

Langage C

1.12

- Inconvénients
 - ▣ Pas d'objets
 - ▣ Pas de gestion des exceptions
 - ▣ Peu de contrôles (on peut tout faire : débordement de tableaux ...)
 - ▣ Faiblement typé (on peut toujours convertir)
 - ▣ Peu devenir franchement complexe

Langage C

1.13

- De très nombreux langages sont inspirés du C
 - ▣ C++ (C avec objets (encapsulation, héritage, généricité))
 - ▣ Java
 - ▣ PhP

Langage C

1.14

- Environnement de développement
 - ▣ Allez jeter un œil sur la page wikipedia du langage C (en français et en anglais)
 - ▣ Visual Studio community C++ est gratuit!
 - **Définir des .c au lieu de .cpp pour avoir la compilation c**
 - ▣ Visual Code est gratuit

Langage C

1.15

- Langage normalisé (C99)

Langage C

1.16

- L'apprentissage du C permet de mieux comprendre le fonctionnement
 - ▣ D'un ordinateur (CPU, mémoire, périphériques)
 - ▣ D'un OS
- Un informaticien se doit d'avoir fait du C dans sa vie (sera toujours utile)

Test

1.17

- `void f(int i){i=8;}`
- `int i=5;`
 `f(i);`
 `k=i;`
- Valeur de k ? On vote

But du cours

1.18

- Vous faire comprendre plein de choses sur la programmation et sur comment écrire un programme efficace
- Exemple 10^{16} d'Excel

Le C et le « snobisme » en programmation

1.19

```
□ void strcpy(char* dest, char* src){  
    while (*dest++=*src++);  
}
```

Langage C : pour débiter

1.20

- 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).
- 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.

Un premier exemple

1.21

```
□ int main (void) {  
    int i;  
    int x = 3;  
    int y = 4; /* y doit être positif */  
    double z = 1.0;  
    i = 0;  
    while (i < y) {  
        z = z * x;  
        i = i + 1;  
    }  
    return 0;  
}
```

Quelques règles

1.22

- Ce n'est pas obligatoire dans le langage mais suivez ces règles :
 - ▣ On met toujours des `{}`: `if (x > 3) {y = 4;}`
 - ▣ On évite plusieurs instructions sur la même ligne
 - `i=i+1; j=j+2; //` on sépare sur 2 lignes
 - ▣ On évite plusieurs déclarations sur la même ligne
 - `int i, j=2, k=5; //` à éviter

Quelques règles

1.23

- Les variables sont écrites uniquement en minuscule
- Les macros ou constantes définies à l'aide de `#define` sont toutes en majuscules
- Les noms de fonctions commencent par une minuscule
- Les noms de fonctions utilisent l'une des 2 formes
 - ▣ Tout en minuscule avec des `_` pour séparer
 - `fahrenheit_to_celcius`
 - ▣ En « collant » tout et mettant des majuscules pour séparer les mots
 - `fahrenheitToCelcius`

On compile et on exécute

1.24

- On compile et on exécute.
- Compilation :
 - ▣ Avec un compilateur (gcc ou celui de l'IDE)
 - ▣ On prend le fichier source en entrée (monfichier.c)
 - ▣ En sortie cela crée un fichier a.out (sous Linux, monfichier.exe sous windows)
- `gcc -Wall -pedantic -ansi foo.c`
- Ce programme C calcule x^y , x et y étant donnés
- (x vaut 3, et y vaut 4). Il n'affiche cependant rien du tout...

Affichage (écriture)

1.25

- `int x;`
- `fprintf(stdout, "%d ", x);`
 - ▣ Écrit un entier dans le fichier `stdout` (sortie standard)
- `printf("%d ", x);`
 - ▣ Écrit directement sur la sortie standard

On affiche quelque chose

1.26

```
□ #include <stdio.h>
  int main (void) {
      int x = 3;
      int y = 4;
      double z = 1.0;
      fprintf(stdout, " x = %d, y = %d", x, y);
      while (y > 0) {
          z *= x;
          y -= 1;
      }
      fprintf(stdout, " , z = %.2f \n", z);
      return 0;
  }
```

Compilation et exécution

1.27

- On compile et on exécute.
 - ▣ `gcc -Wall -pedantic -ansi foo.c`
 - ▣ `a.out`
- `x = 3, y = 4, z = 81.00`
- 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` (standard input-output. `.h` pour « headers »).
- Les fonctions de bibliothèque d'E/S font partie de la bibliothèque C: `libc`

Compilation et exécution

1.28

- Le compilateur C utilisé est celui du projet : gcc.
- Du fichier source au fichier exécutable, différentes étapes sont effectuées :
 - ▣ 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 (ensemble de fichiers objets prédéfinis).

Compilation et exécution

1.29

- Les fichiers objets sont suffixés par `.o` sous Unix et `.obj` sous Windows
- Les bibliothèques sont suffixées par `.a` `.sl` `.sa` sous Unix et par `.lib` sous Windows
- Les bibliothèques chargées dynamiquement (lors de l'exécution du programme et non pas lors de l'édition de liens) sont suffixées par `.so` sous Unix et `.dll` sous Windows.

Options du compilateur

1.30

- 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 (avec les noms des fonctions).
 - ▣ -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.

Options du compilateur

1.31

- Démonstration avec Visual Studio

Calcul d'une puissance

1.32

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


Définition de fonctions

1.33

- 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`
 - $3^4 = 81.00$
 - $3^0 = 1.00$

Déclarations : prototype

1.34

- ```
double puissance (int a, int b) {
 /* corps de la fonction puissance
 déclarations
 instructions */
}
```
- Si on utilise une fonction avant sa définition alors il faut la déclarer en utilisant ce que l'on appelle un prototype :
  - ▣ `double puissance (int, int);`
- Le compilateur en a besoin pour s'y retrouver
- 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.

# Lecture au clavier

1.35

- `int x;`
- `fscanf(stdin, "%d ", &x);`
  - ▣ Lit un entier dans le fichier `stdin` (entrée standard)
- `scanf("%d ", &x);`
  - ▣ Lit directement sur l'entrée standard

# Puissance : lecture au clavier

1.36

```
□ #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;
 int y;
 fprintf(stdout, " x = ");
 fscanf(stdin, "%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;
}
```

# Lecture au clavier

1.37

- Dans les précédentes versions, pour modifier les valeurs de  $x$  et de  $y$ , il fallait modifier le texte source du programme, le recompiler et l'exécuter.
- En C, on peut demander à l'utilisateur des valeurs sur l'entrée standard.
- `gcc -Wall -pedantic -ansi foo.c`
- `a.out`
- `x = 3`
- `y = 4`
- `x = 3, y = 4, x^y = 81.00`

# Un autre exemple

1.38

- 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 :

variable c : caractère

début

    lire(c)

    tantque non findefichier(entrée) faire

        écrire(c)

        lire(c)

    fintantque

fin

# En C :

1.39

```
□ #include <stdio.h>
 int main (void) {
 char c;
 c = fgetc(stdin);
 while (c != EOF) {
 fputc(c, stdout);
 c = fgetc(stdin);
 }
 return 0;
}
```

# En C : (en plus court)

1.40

```
□ #include <stdio.h>
 int main (void) {
 char c;
 while ((c = fgetc(stdin)) != EOF) {
 fputc(c, stdout);
 }
 return 0;
}
```



# Un autre exemple

1.41

- Compter le nombre de caractères lus sur l'entrée standard et écrire le résultat sur la sortie standard.
- En pseudo-langage :  
variable nb : entier  
début  
    nb := 0  
    tantque non fdf(entrée) faire  
        nb := nb + 1  
    fintantque  
    afficher(nb)  
fin

# Une première version

1.42

- Une première version (les fonctions non déclarées avant leur utilisation sont considérées comme renvoyant un entier) :
- ```
#include <stdio.h>
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 (void) {
    fprintf(stdout, "nb 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;
}
```

# En C :

1.43

```
□ #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 (void) {
 fprintf(stdout, "nb de caractères = %ld \ n", compte());
 return 0;
}

□ long compte (void) {
 long nb = 0;
 for (; fgetc(stdin) != EOF; nb++) {
 /* Rien */;
 }
 return nb;
}
```

# Compilation séparée

1.44

- On veut **réutiliser** le plus possible les codes existants
  - ▣ On organise le code : on le sépare par thème, par module :
    - Les fonctions de maths
    - Les fonctions d'entrée/sortie (affichage/saisie)
    - ...
  - ▣ On met un ensemble de code source de fonctions (**définition des fonctions**) dans le même fichier .c
  - ▣ Pour donner accès aux autres à ces fonctions, on doit donner leur signature (type de retour, nombre de paramètres, type des paramètres) = **déclaration**. On met ces déclarations dans un fichier publique le .h

# Compilation séparée

1.45

- fichier `math.h` : fichier de « **déclarations** »
- **`double puissance (int , int);`**  
/\* Rôle : retourne  $a^b$  (ou 1.0 si  $b < 0$ ) \*/

# Compilation séparée

1.46

□ 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;
}
```

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

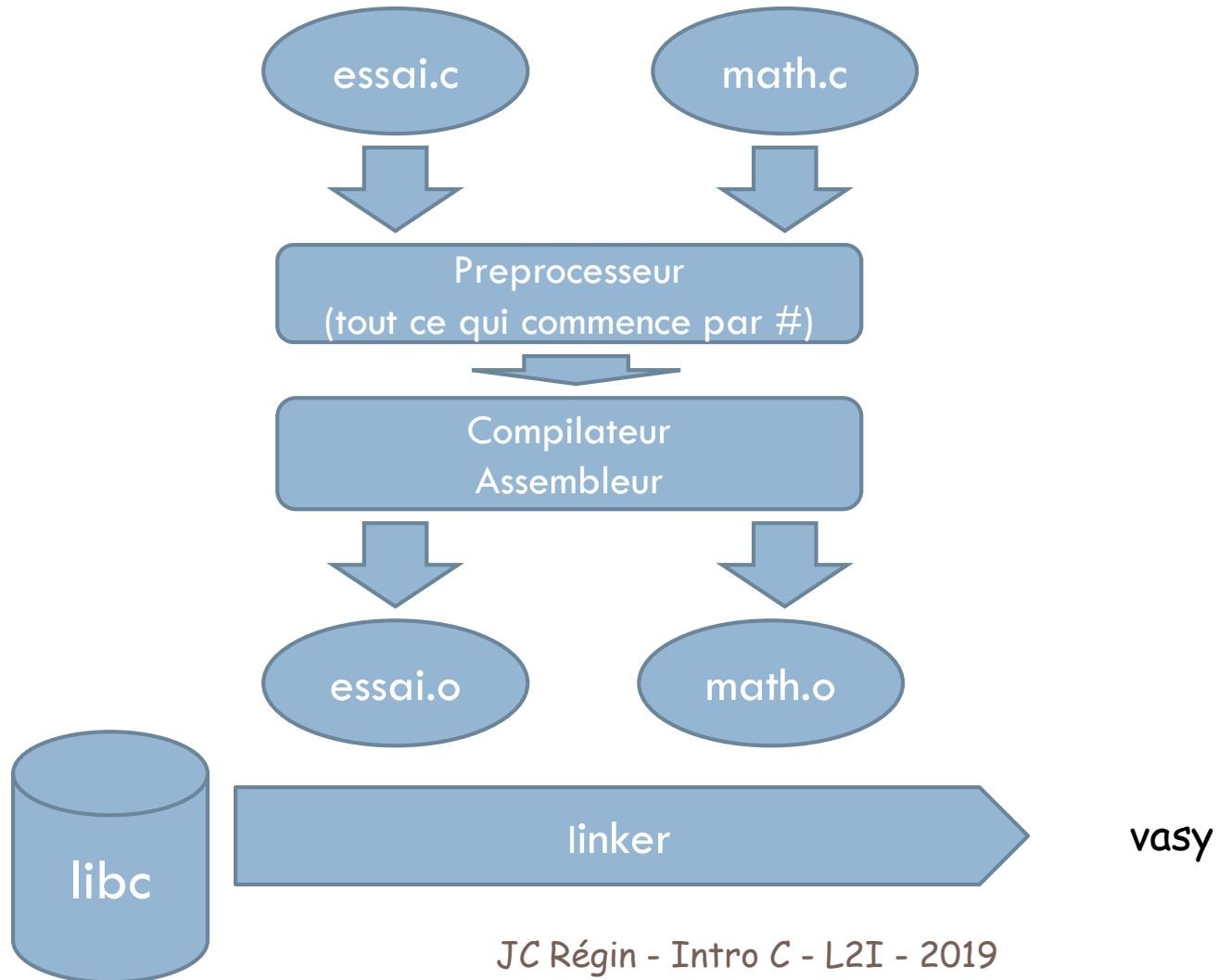
Déclarations et code

1.48

- On va s'organiser un peu
 - ▣ Il faut une déclaration avant utilisation
 - Prototypes mis dans un fichier : les .h
 - Code source des définitions : les .c
 - Certains fichiers sont déjà compilés (les objets) : les .o
 - On a des bibliothèques (comme libc ou les maths) : les .so
 - ▣ Il faut arriver à gérer tout cela :
 - On compile les .c avec les .h,
 - On ne veut pas tout recompiler quand un .c est modifié, mais une modification d'un .h peut avoir de l'importance
 - On utilise les autres .o et les lib
 - ▣ Le gestionnaire : l'utilitaire make avec les makefile; ou bien votre interface de développement

Compilation

1.49



Éléments lexicaux

1.50

- Commentaires : `/* */`
- Identificateurs : suite de lettres non accentuées, 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`

Séquences d'échappement

1.51

<code>\a</code>	Sonnerie
<code>\b</code>	Retour arrière
<code>\f</code>	Saut de page
<code>\n</code>	Fin de ligne
<code>\r</code>	Retour chariot
<code>\t</code>	Tabulation horizontale
<code>\v</code>	Tabulation verticale
<code>\\</code>	Barre à l'envers
<code>\?</code>	Point d'interrogation
<code>\'</code>	apostrophe
<code>\"</code>	guillemet
<code>\o \oo \ooo</code>	Nombre octal
<code>\xh \xhh</code>	Nombre hexadécimal

Constantes

1.52

- Constantes de type entier en notation décimale, octale ou hexadécimale
 - int unsigned int long long long ou __int64
 - 123 12u 100L 1234LL
 - 0173 0123u 125L 128LL
 - 0x7b 0xAb3 0x12UL 0xFFFFFFFFFFFFFFFFLL

- Constantes de type réel
 - float double long double
 - 123e0 123.456e+78
 - 12.3f 12.3L 12.3 12F

Constantes

1.53

- Constantes de type caractère (char) : un caractère entre apostrophes
 - 'a' '\141' '\x61'
 - '\n' '\0' '\12'

- en C, un caractère est considéré comme un entier (conversion unaire).
 - `char ca = 'a'; /* tout est équivalent */`
 - `char ca = 97;`
 - `char ca = '\141';`
 - `char ca = '\x61';`

Constantes

1.54

- Constantes de type chaîne (char *) : chaîne placée entre guillemets
 - ""
 - "here we go"
 - "une chaîne sur \
deux lignes"
 - "et"
 - "une autre"

Variables

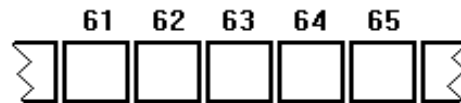
1.55

- 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).

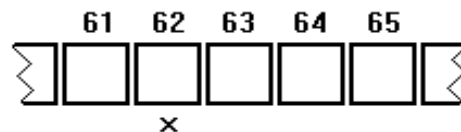
Variable

56

- `int x; // Réserve un emplacement pour un entier en mémoire.`
- `x = 10; // Ecrit la valeur 10 dans l'emplacement réservé.`
- Une variable est destinée à contenir une valeur du type avec lequel elle est déclarée.
- Physiquement cette valeur se situe en mémoire.



- `int x;`

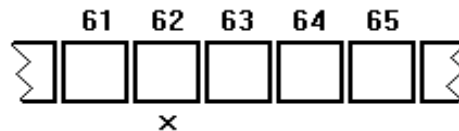


Variable

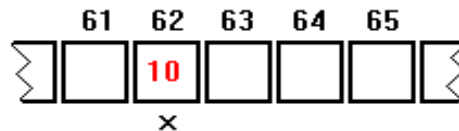
57

- `int x;` // Réserve un emplacement pour un entier en mémoire.
- `x = 10;` // Ecrit la valeur 10 dans l'emplacement réservé.

□ `int x;`



□ `x=10;`



□ `&x` : adresse de x en C : ici 62

▣ Adresse = numéro de la case mémoire

Types élémentaires

1.58

- Signé ou pas :
 - ▣ unsigned : non signé pas de négatif si n bits : $0 \dots (2^n - 1)$
 - ▣ signed : signé, négatifs, si n bits $-2^{n-1} \dots (2^{n-1} - 1)$
- Type spécial : void
 - ▣ ne peut pas être considéré comme un type « normal »
- Type caractère :
 - ▣ char , signé par défaut : -128..+128
 - ▣ unsigned char : 0..255 parfois appelé byte
- Type entier :
 - ▣ short, signé par défaut en général sur 16 bits
 - ▣ int, signé par défaut, sinon unsigned int peut s'écrire unsigned : 32 bits en général
 - ▣ long, signé par défaut . Attention 32 ou 64 bits
 - ▣ long long sur 64 bits

Types élémentaires

1.59

- Type réel :
 - ▣ float, signé, en général 32 bits
 - ▣ double, 32 ou 64 bits
 - ▣ long double, souvent 64 bits
- **PAS de type booléen** : 0 représente le faux, et une valeur différente de 0 le vrai.
- **ATTENTION** nombre de bits du type n'est pas dans le langage.

Principe du C

1.60

- **On écrit des valeurs dans des cases mémoires**
- **On lit des cases mémoire et on interprète le contenu de ce qu'on a lu**

- x est un int. J'écris 65 dans x: `int x=65;`
- Je lis la valeur de x : c'est 65
- Je place 65 dans un char: `char c=65`
- J'affiche le résultat: j'obtiens la lettre A
 - ▣ J'ai interprété le résultat; la valeur n'a pas changé
 - ▣ Pour afficher des caractères on utilise une table de conversion, dite table ASCII. Dans cette table la valeur 65 correspond à A

Table ASCII

1.61

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

Principe du C

1.62

- **On écrit des valeurs dans des cases mémoires**
- **On lit des cases mémoire et on interprète le contenu de ce qu'on a lu**
- **Ce qui est écrit est toujours écrit en binaire**
 - ▣ Codage des entiers (int, long, ...)
 - ▣ Codages des flottants (float, double, ...)
 - ▣ Ce n'est pas la même chose !
 - ▣ Attention à l'interprétation

Représentation sur 32 bits (ou 64)

1.63

- Sur 32 bits, on peut représenter au plus 2^{32} informations différentes
- Si on représente des entiers on représente donc des nombres de 0 à 4 Milliards
- Comment représenter des nombres plus grands ?
- Comment représenter des nombres à virgule ?

On va donner un sens différents aux bits

Représentation en base 2

1.64

- Système de numération (base 2, 10, 16)
- Représentation des entiers
- Représentation des nombres réels en virgule flottante

Bases 2, 10, 16

65/31

- \forall la base B , un nombre N s'écrit
$$N = \sum_{i=0}^{\infty} a_i b^i$$
- Base 10 :
 - ▣ valeurs possibles de a_i 0, 1, 2, ..., 7, 8, 9
 - ▣ $N = 1011_{10} = 1*10^3 + 0*10^2 + 1*10^1 + 1*10^0 = 1011_{10}$
- Base 2 :
 - ▣ valeurs possibles de a_i 0, 1
 - ▣ $N = 1011_2 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11_{10}$
- Base 16 :
 - ▣ valeurs possibles de a_i 0, 1, 2, ..., 9, A, B, C, D, E, F
 - ▣ $N = 1011_{16} = 1*16^3 + 0*16^2 + 1*16^1 + 1*16^0 = 4113_{10}$

Bases 2, 10, 16

66/31

- Taille bornée des entiers stockés
 - ▣ Soit un entier M représenté sur n symboles dans une base b , on a $M \in [0, b^n - 1]$
 - ▣ Exemples :
 - sur 3 digits en décimal, on peut représenter les entiers $\in [0, 999]$
 - sur 3 bits en binaire, on peut représenter les entiers $\in [0, 7]$
 - sur 3 symboles en hexadécimal, on peut représenter les entiers $\in [0, 4095]$

Conversion 2->10, 16->10

67/31

- \forall la base b , un nombre N s'écrit
$$N = \sum_{i=0}^{\infty} a_i b^i$$
- Exemples :

- $010100_2 = 0*2^0 + 0*2^1 + 1*2^2 + 0*2^3 + 1*2^4 + 0*2^5 = 4 + 16 = 20_{10}$

- $1111_2 = 1*2^0 + 1*2^1 + 1*2^2 + 1*2^3 = 1 + 2 + 4 + 8 = 15_{10}$

- $012_{16} = 2*16^0 + 1*16^1 + 0*16^2 = 2 + 16 = 18_{10}$

- $1AE_{16} = 14*16^0 + 10*16^1 + 1*16^2 = 14 + 160 + 256 = 430_{10}$

Conversion 10->2

68/31

- Comment à partir de N retrouver les a_i ?
- Divisions successives
 - ▣ Jusqu'à l'obtention d'un quotient nul
 - ▣ Attention : lecture du bas vers le haut
- Tableau de puissance de 2
 - ▣ Parcourir le tableau des 2^i de gauche à droite
 - Si $N \geq 2^i$, alors mettre 1 dans case 2^i et $N = N - 2^i$
 - Sinon mettre 0 dans case 2^i
 - ▣ Continuer jusqu'à 2^0
- Exemple = $6_{10} = 110_2$

$$N = \sum_{i=0}^{\infty} a_i 2^i$$

6 | 2
0 | 3 | 2
1 | 1 | 2
1 | 0

2^3	2^2	2^1	2^0
0	1	1	0

Conversion 10->16

69/31

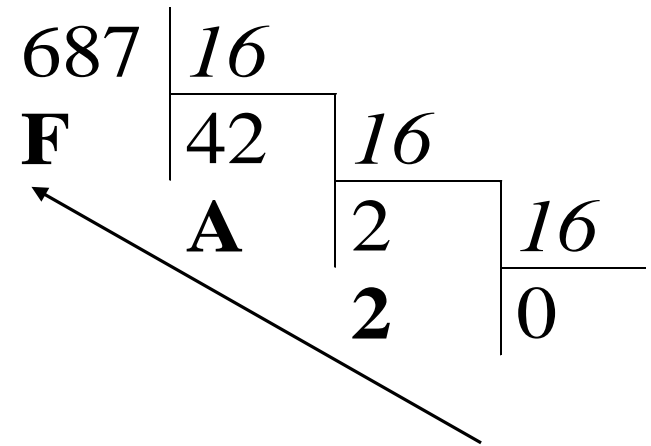
□ Idem base 2

□ Divisions successives

$$N = \sum_{i=0}^{\infty} a_i 16^i$$

▣ Jusqu'à l'obtention d'un quotient nul

▣ Attention : lecture du bas vers le haut



□ Tableau de puissance de 16

▣ Parcourir le tableau des 16^i de gauche à droite

■ Si $N \geq 16^i$, alors mettre 1 dans case 16^i et $N=N-16^i$

■ Sinon mettre 0 dans case 16^i

▣ Continuer jusqu'à 16^0

□ Exemple = $687_{10} = 2AF_{16}$

16^3	16^2	16^1	16^0
0	2	A	F

Conversion 2->16, 16->2

70/31

- 2->16 :
 - ▣ Séparer le nombre binaire en **quartet** (de droite à gauche)
 - ▣ Convertir chaque quartet en hexadécimal
- Exemple : 11011110001010000_2
 - ▣ Séparation en quartet : 1 1011 1100 0101 0000
 - ▣ Conversion : $1\ 1011\ 1100\ 0101\ 0000_2 = 1\ B\ C\ 5\ 0_{16}$
- 16->2 :
 - ▣ Conversion de chaque symbole par un quartet
- Exemple : $AF23_{16}$
 - ▣ En quartet : $A_{16}=1010_2$, $F_{16}=1111_2$, $2_{16}=0010_2$, $3_{16}=0011_2$
 - ▣ Conversion : $AF23_{16} = 1010\ 1111\ 0010\ 0011_2$

Exemples

71/31

- $1010_2 =$
- $1101000_2 =$
- $12_{16} =$
- $3A_{16} =$
- $27_{10} =$
- $35_{10} =$

Réels en virgule flottantes

1.72

- On représente un nombre avec une *mantisse* et un *exposant* : vous connaissez la notation scientifique :
 - ▣ $1,234 \times 10^3 (=1234)$
- Imaginez que la mantisse et l'exposant doivent être représentés sur un **nombre fixe de chiffres** (respectivement 5 et 2 par exemple), et qu'il faille ôter 50 de l'exposant (le *biais*, pour avoir des exposants négatifs) pour 1234^{53} , on aurait alors
 - ▣ Mantisse = 12340
 - ▣ Exposant = $53 - 50 = 3$
 - ▣ soit $(1 + 2/10 + 3/10^2 + 4/10^3) \times 10^{53-50}$

Réels en virgule flottante

1.73

- Donc, $0,0005$ (5×10^{-4}) serait représenté par $m=50000$
 $e=46$ ($50000 | 46$)
- Le plus grand nombre représentable est $99999 | 99$
- Le plus petit (strictement positif) est $00001 | 00 = 1.10^{-4}$
 $\times 10^{-50} = 10^{-54}$
- Il est clair qu'on ne peut pas représenter **exactement**
1,23456 sur cet exemple, puisque la mantisse ne peut
avoir que 5 chiffres

Réels binaires en virgule flottante

1.74

- La représentation binaire en virgule flottante est analogue : avec par exemple une mantisse sur 5 bits, un exposant sur 3 bits et un biais D valant 3
 - ▣ $10110 \mid 110 : (1+1/4+1/8) \times 2^{6-3} = 11$, ou, autre façon de calculer, $(11 \times 2^{-3}) \times 2^3$ (puisque $1011_2 = 11_{10}$, $1,0110_2 = 11_{10} \times 2^{-3}$)
- La mantisse $b_0b_1b_2\dots b_m$ (en forme *normalisée*, on a toujours $b_0 = 1$) représente le nombre **rationnel** $m = b_0 + b_1/2 + b_2/2^2 + \dots b_m/2^m$
- l'exposant $x_nx_{n-1}\dots x_1x_0$, représente l'**entier** $x = x_n2^n + \dots x_12 + x_0$,
- et la valeur représentée par le couple (m,x) est $m \cdot 2^{x-D}$

Réels binaires en virgule flottante

1.75

- La conversion de $0,xxxxx$ en binaire se fait en procédant par **multiplications successives**
- $0,375 = \frac{1}{4} + \frac{1}{8}$
 - ▣ $0,375 \times 2 = 0,75$ on obtient 0
 - ▣ $0,75 \times 2 = 1,5$ on obtient 1
 - ▣ $0,5 \times 2 = 1$ on obtient 1
 - ▣ 0 c'est fini
 - ▣ Le résultat est $0,011_2$

Réels binaires en virgule flottante

1.76

- La norme IEEE 754 définit 2 formats (un peu plus compliqués que le modèle précédant)
 - ▣ Simple précision : mantisse sur 23 bits, exposant sur 8, $D=127$
 - ▣ Double précision : mantisse sur 52 bits, exposant sur 11, $D=1023$
- Les calculs se font en précision étendue : *mantisse sur au moins 64 bits, exposant sur au moins 15*

Réels binaires en virgule flottante

1.77

- La représentation entière ne permet de compter que jusqu'à l'ordre de grandeur 4×10^9 (sur 32 bits), alors que la représentation flottante permet de représenter les nombres entre les ordres de grandeur 10^{-37} et 10^{37} (sur 32 bits) ou 10^{-308} et 10^{308} (sur 64 bits) : on serait tenté de n'utiliser qu'elle !
- Mais cette faculté d'exprimer de très petits ou de très grands nombres se paye par une approximation puisque seuls peuvent être représentés **exactement les nombres de la forme $(b_0 + b_1/2 + b_2/2^2 + \dots + b_m/2^m) \times 2^{x-D}$**
- On a donc des erreurs d'arrondi : par exemple sur 32 bits, $1000 \times (1/3000 + 1/3000 + 1/3000) \neq 1$
- $0,3_{10}$ n'est pas exactement représentable en binaire !
- **sur n bits, on représente moins de nombres réels flottants *différents* que de nombres entiers !**

Réels binaires en virgule flottante

1.78

- La conversion de $0,xxxxx$ en binaire se fait en procédant par **multiplications successives**
- $0,3$
 - ▣ $0,3 \times 2 = 0,6$ on obtient 0
 - ▣ $0,6 \times 2 = 1,2$ on obtient 1
 - ▣ $0,2 \times 2 = 0,4$ on obtient 0
 - ▣ $0,4 \times 2 = 0,8$ on obtient 0
 - ▣ $0,8 \times 2 = 1,6$ on obtient 1
 - ▣ $0,6 \times 2$: on boucle
 - ▣ Le résultat est $0,0100110011001\dots_2$: ça ne s'arrête jamais !
- Problème $0,3$ a une représentation fini en base 10 mais pas en base 2 : je ne peux pas représenter tous les nombres à virgule

Norme IEEE 754

1.79

- Le standard IEEE en format simple précision utilise 32 bits pour représenter un nombre réel x :
- $x = (-1)^S \times 2^{E-127} \times 1,M$
 - ▣ S est le bit de signe (1 bit),
 - ▣ E l'exposant (8 bits),
 - ▣ M la mantisse (23 bits)
- Pour la double et quadruple précision, seul le nombre de bits de l'exposant et de la mantisse diffèrent.

Type énuméré

1.80

```
#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 (void) {
    enum Lights feux = red;
    enum Cards jeu = spade;
    enum Operator op = Min;

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


Type énuméré

1.81

- 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.

Types structurés

1.82

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

```
int main (void) {  
    int t[4];  
    int i;  
    int j;  
    int 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; printf("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;  
}
```

Tableaux

1.83

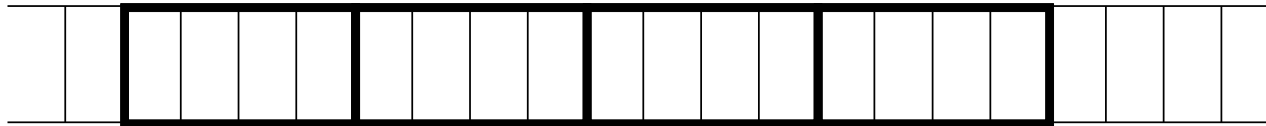
`int t[4];`

`t[0]`

`t[1]`

`t[2]`

`t[3]`



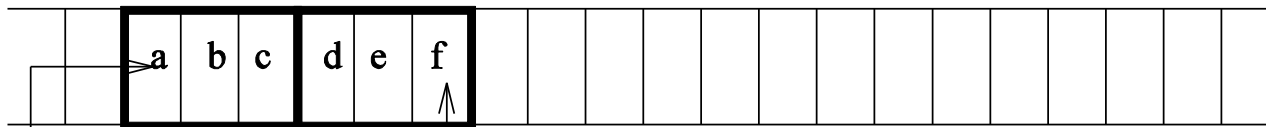
`&t[0]`

`t = &t`

`char w[][3];`

`w[0]`

`w[1]`



`&w[0]`

`w = &w`

`w[0][0]`

`w[1][2]`

Tableaux

1.84

- 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.

Tableaux

1.85

- La dimension doit être connue **statiquement** (lors de la compilation)

```
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 */
```

- On verra plus tard comment définir des tableaux de façon **dynamique** (taille connue à l'exécution)

Tableaux : initialisation

1.86

```
#include <stdio.h>

void init (int t[], int n, int v) {
    int i;
    for (i = 0; i < n; i++)
        t[i] = v;
}

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

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, 10);
    return 0;
}
```

Chaines de caractères

1.87

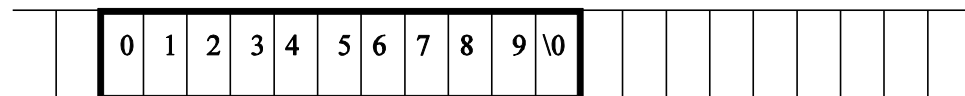
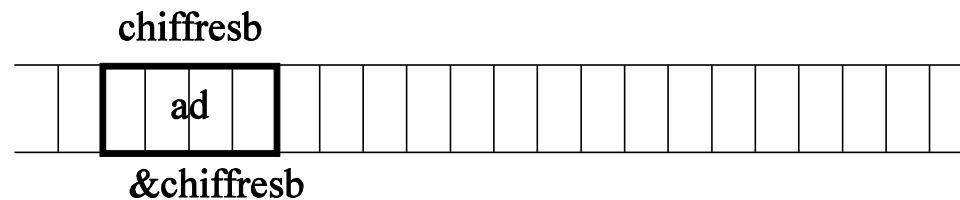
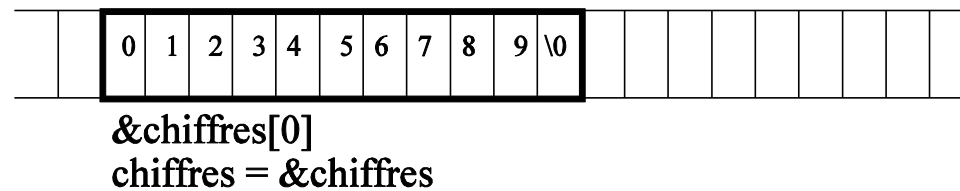
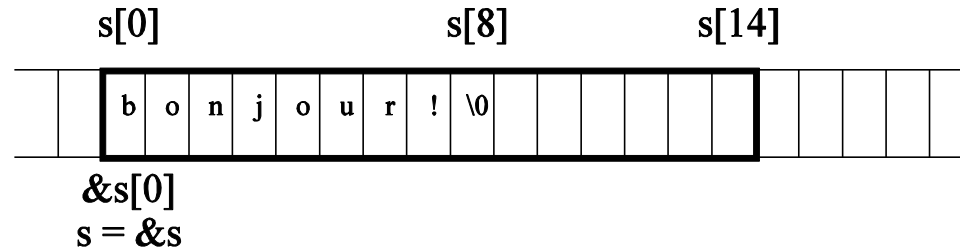
- Ce sont des tableaux de caractères : ce n'est donc pas un vrai type.
- Par convention, elles se terminent par le caractère nul `\0` (valeur 0).
- Il n'y a pas d'opérations pré-définies (puisque ce n'est pas un type), mais il existe des fonctions de bibliothèque, dont le fichier de déclarations s'appelle `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";`

- Ecriture sur la sortie standard :
 - ▣ `printf("%s",chiffres);`
 - ▣ `fputs(chiffres,stdout);`

Chaines de caractères

1.88



`&"0123456789" = ad` (adresse dans mémoire constante)

Chaines de caractères : initialisation

1.89

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

int main (void) {
    char chaine[] = "bonjour ";
    char chRes[256];

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

Structures

1.90

```
#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 (void) {
    struct personne p;
    struct personne 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;
```

Structures

1.91

```
int main (void) {
    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];
        }
        printf("moy de %d, ne en %d = %.2f\n",
            etud[e].num, etud[e].neLe.annee,
            somme/(double)3);
    }
    return 0;
}
```

Structures

1.92

```
struct personne {  
    int num; /* numéro de carte */  
    struct date neLe;  
    short tn[3]; /* tableau de notes */  
};
```

- 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.

Union

1.93

```
union aword {  
    unsigned int i;  
    unsigned char b[4];  
    unsigned short s[2];  
};
```

```
□ const aword a = { buf[k]}; /* initialisation */  
if (a.i) cnt += (lut[a.b[0]] + lut[a.b[1]] + lut[a.b[2]] +  
lut[a.b[3]]);
```

Union

1.94

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

Champs de bits

1.95

```
#include <stdio.h>

typedef struct {
    unsigned rouge : 10;
    unsigned vert : 10;
    unsigned bleu : 10;
    unsigned : 2;
} couleur;

int main (void) {
    couleur rouge = {0x2FF, 0x000, 0x000}, vert = {0x133, 0x3FF, 0x133},
    bleu = {0x3FF, 0x3FF, 0x2FF};
    printf("rouge = %x\tvert = %x\tbleu = %x\n", rouge, vert, bleu);
    printf("rouge = %u\tvert = %u\tbleu = %u\n",rouge, vert, bleu);
    return 0;
}
```

Champs de bits

1.96

- 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).

- Utiliser pour coder plusieurs choses sur un mot :
 - ▣ Je veux coder les couleurs sur un mot machine (ici 32 bits)
 - ▣ 3 couleurs donc 10 bits par couleur au lieu d'un octet

Champs de bits

1.97

- Dépend fortement de la machine, donc difficilement transportable.
- Architectures droite à gauche : **little endian** (petit bout)
 - ▣ Octet de poids faible en premier (adresse basse)
 - ▣ x86
- Architectures gauche à droite : **big endian** (gros bout)
 - ▣ Octet de poids fort en premier (adresse basse)
 - ▣ Motorola 68000
- Souvent remplacé par des masques binaires et des décalages

Définition de types : typedef

1.98

- Mot-clé typedef (très utilisé). Permet de définir de nouveau type (aide le compilateur)

```
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;  
    Tree right;  
} Node;  
typedef int (*PFI) (char *, char *);
```

Instructions : instruction vide

1.99

- 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...
  - ▣ Mettre toujours des { }, avec les if et les for

# Expression, affectations et instructions

1.100

- En général
  - ▣ instruction = action
  - ▣ expression = valeur
- 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 '='.

# Affectation composée

1.101

- `partie_gauche ?= expression`  
? 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 -= 6; /* a = a - 6; */  
    a *= 3 - 5; /* a = a * (3 - 5) */  
    a *= (3 - 5); /* a = a * (3 - 5) */  
    return 0;  
}
```

Bloc

1.102

- 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;  
        printf(" i = %d, j = %d \n ", i, j);  
        i++; j++;  
    }  
    printf(" i = %d, j = %d \n", i, j); /* valeur de i et j ? */  
    return 0;  
}
```

Instruction conditionnelle

1.103

- 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.**

Instruction Conditionnelle

1.104

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
    int x, y = -3, z = -5;
    printf(" valeur de x? ");
    scanf("%d ", &x);
    if (x == 0)
        x = 3;
    else if (x == 2) {
        x = 4; y = 5;
    }

    else
        z = x = 10;
    printf(" x = %d, y = %d, z = %d \n", x, y, z);
    if (0) x = 3; else x = 4; /* à éviter */
    return 0;
}
```


Instruction Conditionnelle

1.105

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
    int x, y = -3, z = -5;
    printf(" valeur de x? ");
    scanf("%d ", &x);
    if (x == 0) {
        x = 3;
    } else {
        if (x == 2) {
            x = 4; y = 5;
        } else {
            z = x = 10;
        }
    }
    printf(" x = %d, y = %d, z = %d \n", x, y, z);
    if (0) x = 3;
    else x = 4;
    return 0;
}
```

Aiguillage (switch)

1.106

- 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**)

Switch

1.107

```
#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++;
    }
    printf(" chiffres = %d, blancs = %d , autres = %d \n", nbc, nbb, nba);
    return 0;}
```

Les boucles

1.108

- **while** (*expression_entière*)
instruction

```
#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 ';
    printf(" \n %s\ n ", tab);
    return 0;
}
```

Les boucles

1.109

□ **do**
 instruction
while (expression_entière);

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define NB 3
int main (int argc, char *argv[]) {
    char rep[NB];
    do {
        printf(" Voulez vous sortir ? " );
        fgets(rep, NB, stdin);
        rep[0] = toupper(rep[0]);
    } while (strcmp(rep, "O\ n"));
    return 0;
}
```

Les boucles

1.110

- **for (init; cond_continuation; rebouclage)**
instruction
- équivalent à
init;
while (cond_conti) {
instruction;
rebouclage;
}

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

```
#define MAX 30
```

```
int main (int argc, char *argv[]) {  
    char tab[MAX] = "toto";  
    int i;  
    printf("'%s'\n",tab);  
    for (i = 0; i < MAX; i++)  
        tab[i] = '\0';  
    printf("'%s'\n",tab);  
    return 0;  
}
```

Les boucles

1.111

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i;
    i = 0;
    while (i < 10) {
        i++;
    }
    printf(" i = %d \n ", i);
    i = 0;
    do {
        i++;
    } while (i < 10);
    printf(" i = %d \n ", i);
    for (i = 0; i < 10; i++);
    printf(" i = %d \n ", i);
    return 0;
}
```

Instructions de débranchement

1.112

- **break**
 - ▣ Utilisée dans un switch ou dans une boucle
 - ▣ Se débranche sur la première instruction qui suit le switch ou la boucle
- **continue**
 - ▣ Utilisée dans les boucles
 - ▣ Poursuit l'exécution de la boucle au test (ou au rebouclage)
- **goto**
 - ▣ « saute » à l'étiquette donnée
 - ▣ l'étiquette, etq par exemple, est placée comme suit
 - etq :
- **return**
 - ▣ Provoque la sortie d'une fonction
 - ▣ La valeur de retour est placée derrière le mot-clé

Fonction exit

1.113

- Fonction exit
 - ▣ Met fin à l'exécution d'un programme
 - ▣ 0 en paramètre indique l'arrêt normal

Débranchements : exemple

1.114

```
#include <stdio.h>

void f(int j){
    int i = 0;
    while (i < j) {
        /* instruction? */
        i++;
    }
    printf(" f i n   de   f ,   i = %d \n ", i);
}

int main (int argc, char *argv[]) {
    int i = 100;
    f(i);
    printf(" f i n   de   main ,   i = %d \n ", i);
    return 0;
}
```

Débranchements : exemple

1.115

Instruction?	Résultat
break;	fin de f, i = 0
	fin de main, i = 100
continue;	Ca boucle
return;	fin de main, i = 100
exit(2);	
/* aucune instruction */	fin de f, i = 100
	fin de main, i = 100

Opérateurs

1.116

Affectation

- 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 */
i = i++; n'est pas défini

Opérateurs de calcul

1.117

- Arithmétiques $+ * - / \%$
 - Relationnels $< <= > >= == !=$
 - Logiques $! || \&\&$
 - Bit à bit $\sim \& | ^ << >>$
-
- $3 + 4$ $3 * 4 + 5$
 - $3 / 4$ $3.0 / 4$
 - $3 \% 4$ $!(n \% 2)$
 - $(x = 1) || b$
 - $(x = 0) \&\& b$

Opérateurs de calcul

1.118

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    short i;
    for (i = 0; i < 10; i++){
        printf("i = %hx ,~i = %hx",i,~i);
        printf("!i = %hx ,i << 1 = %hx", !i, i << 1);
        printf("i >> 1 = %hx \n ",i >> 1);
    }
    return 0;
}
```

Opérateurs de calcul

1.119

□ Résultat :

- $i = 0, \sim i = \text{ffff}, !i = 1, i \ll 1 = 0, i \gg 1 = 0$
- $i = 1, \sim i = \text{fffe}, !i = 0, i \ll 1 = 2, i \gg 1 = 0$
- $i = 2, \sim i = \text{fffd}, !i = 0, i \ll 1 = 4, i \gg 1 = 1$
- $i = 3, \sim i = \text{fffc}, !i = 0, i \ll 1 = 6, i \gg 1 = 1$
- $i = 4, \sim i = \text{fffb}, !i = 0, i \ll 1 = 8, i \gg 1 = 2$
- $i = 5, \sim i = \text{fffa}, !i = 0, i \ll 1 = \text{a}, i \gg 1 = 2$
- $i = 6, \sim i = \text{fff9}, !i = 0, i \ll 1 = \text{c}, i \gg 1 = 3$
- $i = 7, \sim i = \text{fff8}, !i = 0, i \ll 1 = \text{e}, i \gg 1 = 3$
- $i = 8, \sim i = \text{fff7}, !i = 0, i \ll 1 = 10, i \gg 1 = 4$
- $i = 9, \sim i = \text{fff6}, !i = 0, i \ll 1 = 12, i \gg 1 = 4$

Opérateurs sur les types

1.120

- 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`.
 - ▣ `size_t` : taille d'un mot machine

Opérateurs sur les types

1.121

```
#include <stdio.h>
#define imp(s, t) printf(" 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);
    printf(" sizeof t1 = %d, sizeof t2 = %d \n", sizeof t1, sizeof t2);
    printf(" sizeof t1 [0] = %d, sizeof t2 [1] = %d \n", sizeof t1[0], sizeof t2[1]);
    return 0;
}
```

Opérateurs sur les types

1.122

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

\$

Opérateurs sur les types

1.123

```
#include <stdio.h>
#include <string.h>
void f (int t[]) {
    printf(" f : sizeof t = %d, sizeof t [0] = %d \n",sizeof t, sizeof t[0]);
}
void g (char s[]) {
    printf("g: sizeof s = %d, sizeof s[0] = %d \n",sizeof s, sizeof s[0]);
    printf("g: l g r de s = %d \n ", strlen(s));
}
int main (int argc, char *argv[]) {
    int t1[10];
    char s1[] = " 12345";
    printf(" main : sizeof t1 = %d \n", sizeof t1);
    f(t1);
    printf(" main : sizeof s1 = %d, s t r l e n ( s1) = %d \n",sizeof s1, strlen(s1));
    g(s1);
    return 0;
}
```

Opérateurs sur les types

1.124

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

\$

Casting ou transtypage

1.125

Conversion explicite (« casting » ou transtypage)
(type) expression

```
#include <stdio.h>  
int main (int argc, char *argv[]) {  
    printf("%.2 f ", 3/(double)4);  
    printf("%d ", (int)4.5);  
    printf("%d ", (int)4.6);  
    printf("%.2 f ", (double)5);  
    fputc( ' \n' , stdout);  
    return 0;  
}
```

Opérateur de condition

1.126

condition ? expression_si_vrai : expression_si_faux

Seul opérateur ternaire du langage.

```
#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;
    printf(" x = %d , y  %d \n ", x, y);
    return 0;
}
```

Opérateur virgule

1.127

- Le résultat de `expr1, expr2, ..., exprn` est le résultat de `exprn`
- `expr1, expr2, ..., expr(n-1)` sont évaluées, mais leurs résultats oubliés (sauf si effet de bord)

Opérateur virgule

1.128

```
#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];
    printf("%d \n ", (a = 1, b = 2));
    printf("%d \n ", (a = 1, 2));
    return 0;
}
```


Opérateurs

1.129

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

Pointeurs

1.130

- Les pointeurs sont très utilisés en C.
- En général, ils permettent de rendre les programmes
 - ▣ plus compacts,
 - ▣ plus efficaces,
- L'accès à un objet se fait de façon indirecte.

Pointeurs

1.131

- `int x; //` alloue de la mémoire pour `x`
- `x=10; //` met 10 dans la(les) case(s) mémoire correspondant à `x`
- On ne peut pas décider de l'emplacement mémoire (adresse) de `x`
- On ne peut pas changer l'adresse de `x`
- Les pointeurs vont permettre de faire cela
- `int *p; //` **pointeur représenté avec une ***
- Toute variable contient une valeur : la valeur d'un pointeur est une adresse
`p= &x;`

Pointeurs

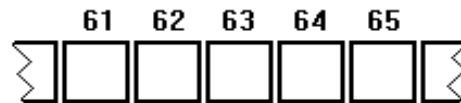
1.132

- `int *p;` pointeur
- Toute variable contient une valeur : la valeur d'un pointeur est une adresse `p = &x;`
- Déréférencement : opérateur `*`
- Permet d'interpréter le contenu de la case située à l'adresse du pointeur :
 - ▣ `p = &x;` // `&x` vaut 1234
 - ▣ `*p` contenu la mémoire à la case 1234
 - On peut la lire (`y = *p;`)
 - On peut la modifier (`*p = 22;`)

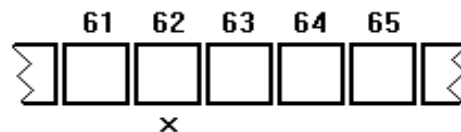
Pointeurs

133

- `int x; // Réserve un emplacement pour un entier en mémoire.`
- `x = 10; // Ecrit la valeur 10 dans l'emplacement réservé.`
- Une variable est destinée à contenir une valeur du type avec lequel elle est déclarée.
- Physiquement cette valeur se situe en mémoire.



- `int x;`

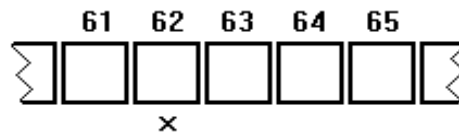


Pointeurs

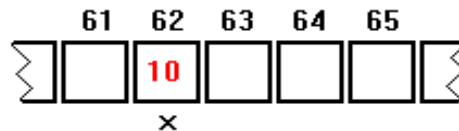
134

- `int x;` // Réserve un emplacement pour un entier en mémoire.
- `x = 10;` // Ecrit la valeur 10 dans l'emplacement réservé.

□ `int x;`



□ `x=10;`



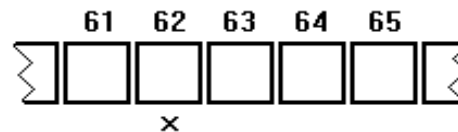
□ `&x` : adresse de x : ici 62

Pointeurs

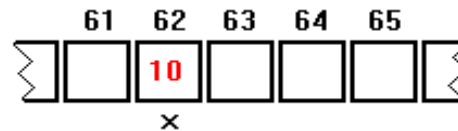
135

- `int x; // Réserve un emplacement pour un entier en mémoire.`
- `x = 10; // Ecrit la valeur 10 dans l'emplacement réservé.`

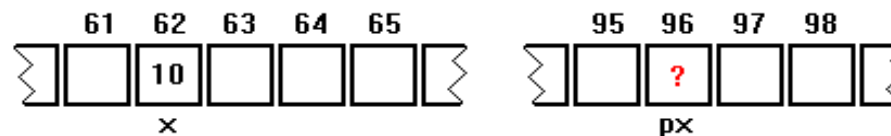
□ `int x;`



□ `x=10;`



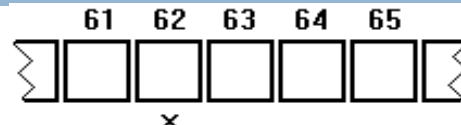
□ `int* px; // pointeur sur un entier`



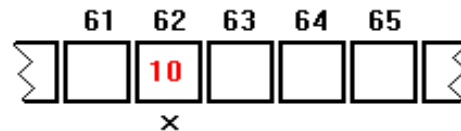
Pointeur Implémentation

136

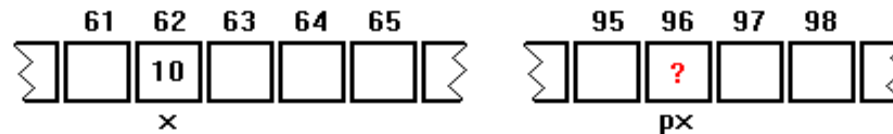
□ `int x;`



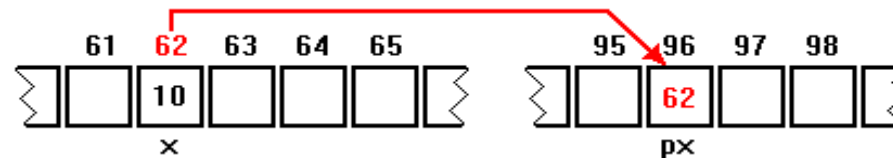
□ `x=10;`



□ `int* px; // pointeur sur un entier`



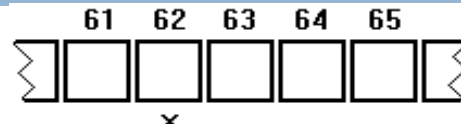
□ `px=&x (adresse de x)`



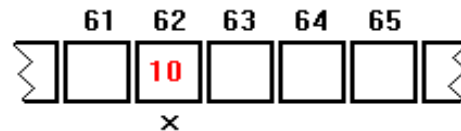
Pointeur Implémentation

137

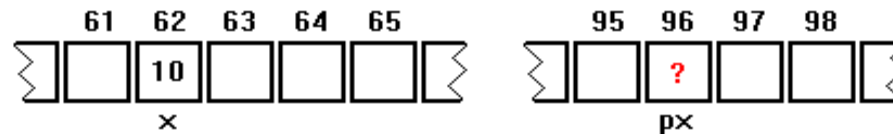
□ `int x;`



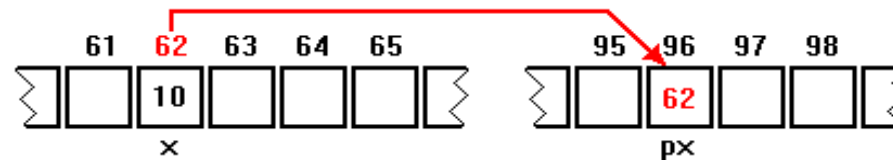
□ `x=10;`



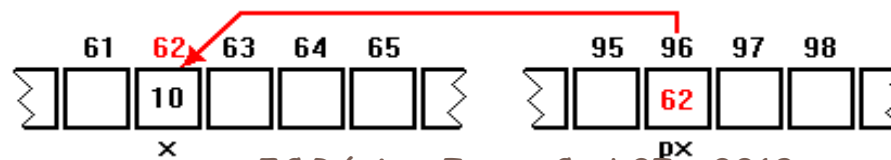
□ `int* px; // pointeur sur un entier`



□ `px=&x (adresse de x)`



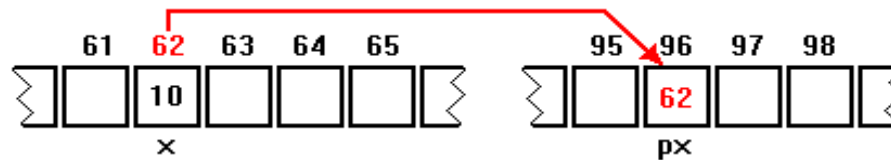
□ `int y=*px`



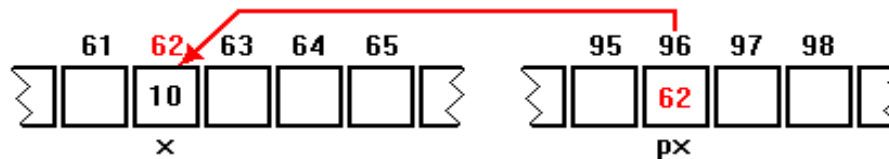
Pointeur Implémentation

138

- Si `px` contient l'adresse de `x`



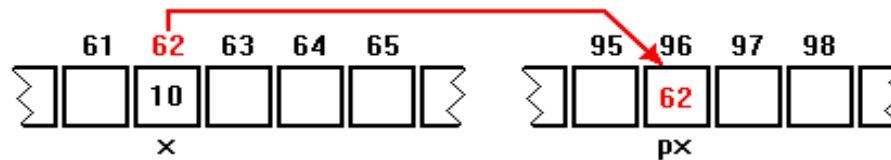
- Alors `*px` contient la valeur de l'adresse de `x`, donc la valeur de `x`



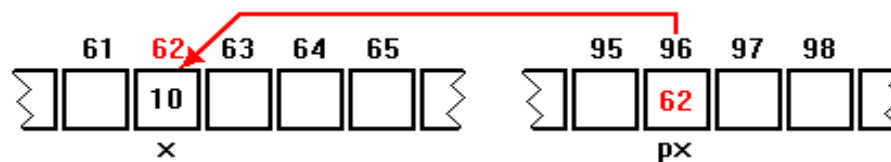
Pointeur Implémentation

139

- Si px contient l'adresse de x



- Alors $*px$ contient la valeur de l'adresse de x , donc la valeur de x

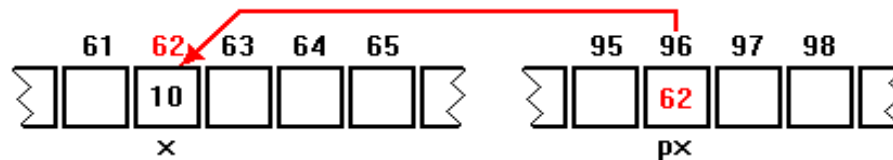


- Si je change la valeur à l'adresse de x , alors je change la valeur de x ,

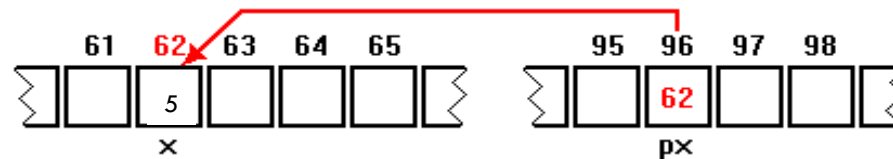
Pointeur Implémentation

140

- Si `px` contient l'adresse de `x`, donc `*px` la valeur de `x`



- Si je change la valeur à l'adresse de `x`, alors je change la valeur de `x` : `*px=5`



Pointeur Implémentation

141

- `px=&x`
- Si je modifie `*px` alors je modifie `x` (même case mémoire concernée)
- `px=&y`
- Si je modifie `*px` alors je modifie `y` (même case mémoire concernée)
- `px=&bidule`
- Si je modifie `*px` alors je modifie `bidule` (même case mémoire concernée)
- `px` désigne **l'objet pointé**
- `*px` modifie **l'objet pointé**

Pointeurs : type

1.142

□ Types des pointeurs :

- ▣ on a vu qu'on avait besoin d'interpréter le contenu des cases mémoires (2 octets pour un short, 4 pour un int, big-endian, small-endian...). On va donc typer les pointeurs pour obtenir ce résultat
- ▣ `int *p` veut dire que `*p` sera un int, donc `int y = *p` est parfaitement ok.

Pointeurs

1.143

- A quoi cela sert-il ?
 - ▣ accès direct à la mémoire
 - ▣ passage de paramètres
 - ▣ partage d'objets
 - ▣ indirection
 - ▣ passage par référence
 - ▣ allocation dynamique

Pointeurs

1.144

- A quoi cela sert-il ?
 - ▣ accès direct à la mémoire
 - ▣ **passage de paramètres**
 - ▣ partage d'objets
 - ▣ indirection
 - ▣ passage par référence
 - ▣ allocation dynamique

Pointeurs : passage de paramètre

1.145

- Passage de paramètres par référence

```
struct etudiant {  
    char nom[50];  
    char prenom[50];  
    char adresse[255];  
    int notes[10];  
};  
  
double calculMoyenne(struct etudiant etud){  
    int i, sum=0;  
    for(i=0;i<10;i++)  
        sum += etud.notes[i];  
    return (double)sum/10;  
}
```

- Que se passe t'il quand on passe un étudiant en paramètre ?

Pointeurs : passage de paramètre

1.146

- Passage de paramètres par référence

```
struct etudiant {  
    char nom[50];  
    char prenom[50];  
    char adresse[255];  
    int notes[10];  
};  
  
double calculMoyenne(struct etudiant etud){  
    int i, sum=0;  
    for(i=0;i<10;i++)  
        sum += etud.notes[i];  
    return (double)sum/10;  
}
```

- **Il y a création d'une variable locale**
 - ▣ on réalloue de la mémoire pour cette structure locale
 - ▣ la structure est entièrement copiée ! **Donc au moins 365 opérations !**

Pointeurs : passage de paramètre

1.147

- Passage de paramètres par référence

```
struct etudiant {  
    char nom[50];  
    char prenom[50];  
    char adresse[255];  
    int notes[10];  
};  
  
double calculMoyenne(struct etudiant etud){  
    int i, sum=0;  
    for(i=0;i<10;i++)  
        sum += etud.notes[i];  
    return (double)sum/10;  
}
```

- Comment éviter cela ?

- On définit un tableau global de structure et on passe l'indice de l'élément dans le tableau (je ne vois pas d'autres solutions sans les pointeurs)
- **GROS défauts**
 - Cela impose des données globales (je ne peux pas passer le tableau ! Sinon copie !)
 - Cela impose une structure de tableau, comment gère t'on les suppressions/ajouts ?
 - Je dois connaître la taille du tableau au début du programme
- C'est pratiquement injouable

Pointeurs : passage de paramètre

1.148

- Passage de paramètres par référence

```
struct etudiant {  
    char nom[50];  
    char prenom[50];  
    char adresse[255];  
    int notes[10];  
};  
  
double calculMoyenne(struct etudiant etud){  
    int i, sum=0;  
    for(i=0;i<10;i++)  
        sum += etud.notes[i];  
    return (double)sum/10;  
}
```

- **Solution : on utilise un pointeur sur la structure d'un étudiant**

```
double calculMoyenne(struct etudiant *etud){  
    int i, sum=0;  
    for(i=0;i<10;i++)  
        sum += (*etud).notes[i];  
    return (double)sum/10;  
}
```

- On ne passe que l'adresse mémoire et on travaille avec cette adresse mémoire : il n'y a pas de copie locale

Pointeurs

1.149

- A quoi cela sert-il ?
 - ▣ accès direct à la mémoire
 - ▣ passage de paramètres
 - ▣ **partage d'objets**
 - ▣ indirection
 - ▣ passage par référence
 - ▣ allocation dynamique

Pointeurs : partages d'objets

1.150

- Pour chaque note on veut connaitre celui qui a la meilleure note
 - ▣ Premier (meilleure note) en C
 - ▣ Premier en Système Exploitation
 - ▣ Premier en Algorithmique
 - ▣ Premier en Anglais ...

- Comment faire ?

Pointeurs : partages d'objets

1.151

- Pour chaque note on veut connaître celui qui a la meilleure note
 - ▣ Premier (meilleure note) en C
 - ▣ Premier en Système Exploitation
 - ▣ Premier en Algorithmique
 - ▣ Premier en Anglais ...
- On fait une copie à chaque fois : mauvaise solution, problème de synchronisation entre les copies
- On utilise des indices pour désigner chaque étudiant : problème demande une gestion sous la forme de tableau des étudiants (avec les inconvénients vu)
- On utilise des pointeurs !

Pointeurs : partages d'objets

1.152

- Pour chaque note on veut connaître celui qui a la meilleure note
 - ▣ Premier (meilleure note) en C
 - ▣ Premier en Système Exploitation
 - ▣ Premier en Algorithmique
 - ▣ Premier en Anglais ...

- On utilise des pointeurs !

```
struct etudiant *pC;  
struct etudiant *pSE;  
struct etudiant *pAlgo;  
struct etudiant *pAnglais;
```

Un meme élément peut être partagé!

Pointeurs

1.153

- A quoi cela sert-il ?
 - ▣ accès direct à la mémoire
 - ▣ passage de paramètres
 - ▣ partage d'objets
 - ▣ **indirection**
 - ▣ passage par référence
 - ▣ allocation dynamique

Pointeurs : indirection

1.154

- L'indice d'un tableau est différent du contenu : on fait une indirection
- C'est pareil avec les pointeurs
- Plus petit élément d'un ensemble (ppelt)
 - ▣ Avec un tableau d'élément de type `t` : ppelt est un indice
 - ▣ Avec n'importe quelle structure de données d'élément de type `t` : ppelt est un pointeur de type `t` (`t *ppelt`)

Pointeurs

1.155

- A quoi cela sert-il ?
 - ▣ accès direct à la mémoire
 - ▣ passage de paramètres
 - ▣ partage d'objets
 - ▣ indirection
 - ▣ **passage par référence**
 - ▣ allocation dynamique

Pointeurs : passage par référence

1.156

- Comment modifier un paramètre avec une fonction ?

```
double moyenne(int* t, int nbelt){  
    /* on calcule la moyenne et on la retourne */  
    return moy;  
}
```

- Je veux faire :

```
void modifierMoyenne(int* t, int nbelt, double moy){  
    /* je veux modifier la moyenne moy*/  
    moy = 12;  
}
```

- Cette solution ne marche pas car moy est copié dans une variable locale

Pointeurs : passage par référence

1.157

- Un pointeur contient une adresse mémoire. Si on déréférence on touche directement à l'adresse mémoire. C'est ce qu'il nous faut !

```
void modifierMoyenne(int* t, int nbelt, int* moy){  
    /* je veux modifier la moyenne moy*/  
    *moy = 12;  
}
```

- Cette solution marche car on modifie ce qui est à l'adresse mémoire passée en paramètre. C'est l'adresse qui est passée et non plus la valeur

Pointeurs

1.158

- A quoi cela sert-il ?
 - ▣ accès direct à la mémoire
 - ▣ passage de paramètres
 - ▣ partage d'objets
 - ▣ indirection
 - ▣ passage par référence
 - ▣ **allocation dynamique**

Pointeurs : allocation dynamique

1.159

- On ne peut pas toujours connaître la taille d'une structure de données au moment où on écrit le code source
 - ▣ Nombre d'étudiants à l'université ? Nombre d'arbres dans un jardin ?
- On peut surestimer mais cela peut créer des problèmes
- Allocation dynamique
 - ▣ Comment allouer n éléments, avec n qui sera défini à l'exécution ?
 - ▣ Comment définir cette variable dans le programme ?

Pointeurs : allocation dynamique

1.160

- Allocation dynamique
 - ▣ Comment allouer n éléments, avec n qui sera défini à l'exécution ?
 - ▣ Comment définir cette variable dans le programme ?
- Solution : on travaille en fait avec des cases mémoires avec les pointeurs
 - ▣ on n'a pas besoin que la mémoire soit définie,
 - ▣ on a juste besoin de savoir que c'est la mémoire à un certain endroit qui va être utilisée.
- On allouera cette mémoire (cela veut dire on réservera cette place mémoire) après quand on connaîtra la taille, en utilisant des fonction spéciales : malloc, calloc, realloc

Pointeurs : allocation dynamique

1.161

```
int* tab; /* tab est un tableau */
```

```
int n;
```

```
tab = calloc(n,sizeof(int)); /* réserve de la place pour  
                             n entiers */
```

```
/* calloc retourne une adresse mémoire et dit à l'OS  
   que la place est réservée à cet endroit */
```

Pointeurs : allocation mémoire

1.162

- En utilisant les fonctions standard suivantes, l'utilisateur a la garantie que la mémoire allouée est contigüe et respecte les contraintes d'alignement.
- Ces fonctions se trouvent dans le fichier de déclarations `stdlib.h`.
 - ▣ **`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 x 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)

Pointeur

163

- **Un pointeur est un type de données dont la valeur fait référence (référencie) directement (pointe vers) à une autre valeur.**
- Un pointeur référence une valeur située quelque part en mémoire en utilisant son adresse
- Un pointeur est une variable qui contient une adresse mémoire

Pointeur

164

- Un pointeur est un type de données dont la valeur **pointe vers** une autre valeur.
- Obtenir la valeur vers laquelle un pointeur pointe est appelé **déréférencer** le pointeur.
- Un pointeur qui ne pointe vers aucune valeur aura la valeur **NULL ou 0**
- **En Java TOUT est pointeur**

Pointeurs : déclarations

1.165

Comment déclarer un pointeur ?

```
type *nom_pointeur;
```

```
char *p;
```

```
int *p1, *p2;
```

```
struct {int x, y;} *q;
```

```
void *r;
```

```
int *s1[3];
```

```
int (*fct)(void);
```

```
int (*T[5])(void);
```

```
double *f(void);
```

Pointeurs : syntaxe

1.166

- Adresse d'une variable `x` est désignée en utilisant &
 - ▣ exemple `&x;`
- Pointeur « universel » : `void *`
- 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` est équivalent à `(*q).x`

Références fantômes !

1.167

```
#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;
}
```

Que se passe t'il ?

1.168

```
#include <stdio.h>

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

    printf("sizeof(char) = %d, sizeof(int) = %d,\n, sizeof(void *) %d\n\n",
           sizeof(char), sizeof(int), sizeof(void *));
    printf(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';
    printf("i = %d, c = %c,\npc = %p, *pc = %c,\n pi = %p, *pi = %d\n",
           i, c, pc, *pc, pi, *pi);
    return 0;
}
```


- 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
- \$

Que se passe t'il ?

1.170

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

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

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

- 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
- \$

Passage par référence

1.172

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

Passage par référence

1.173

```
#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;

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

Passage par référence

1.174

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

void imp (int t[], int lg) {
    int i;
    for (i = 0; i < lg; i++)
        printf("%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;
}
```

Pointeurs et tableaux

1.175

- 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]`
- 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]` est équivalent à `*(a + i)`

Pointeurs : opérations arithmétique

1.176

- Un pointeur contenant une adresse, on peut donc lui appliquer des opérations arithmétiques
- $\text{pointeur} + \text{entier}$ donne un pointeur
- $\text{pointeur} - \text{entier}$ donne un pointeur
- $\text{pointeur} - \text{pointeur}$ donne un entier
- Le dernier point est **FORTEMENT déconseillé** car très peu portable.

Pointeurs : opérations arithmétiques

1.177

```
#include <stdio.h>

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

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

Exécution :

\$ a.out

t1 = 0xbffff780, t2 = 0xbffff730

t1 + 3 = 0xbffff78c, &t1[3] = 0xbffff78c

&t1[3] - 3 = 0xbffff780

t1 - t2 = 20

t2 - t1 = -20

\$

```
#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++)
        printf("%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;
}
```

```

#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;
    printf("les %d nombres? ", n);
    for (i = 0; i < n; i++)
        scanf("%d", &t[i] /* t + i */);
}
void copieTab (int t[], int *p, short n) {
    /* Antécédent : t, p et n initialisés */
    /* Conséquent : $forall$ i $in$ [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++)
        printf("%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;
}

```

Pointeurs et chaînes de caractères

1.180

- 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'};

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

Parcours de chaînes de caractères

1.181

```
#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;
}
```

Pointeurs et fonctions

1.182

- 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 *

Pointeurs et fonctions

1.183

- 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[]) {
    printf("%p\n", (void *)printf);
    printf("%p\n", (void *)&printf);
    printf("%p\n", (void *)max);
    printf("%p\n", (void *)max(1, 16));
    return 0;
}
```

Fonctions en paramètre

1.184

- 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++)
        printf("%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;
}
```


Pointeur polymorphe (void*)

1.185

```
#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;

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

Pointeur polymorphe (void*)

1.186

```
#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
}
```

Pointeur polymorphe (void*)

1.187

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

#define imp(t, s1, s2, x, s3, y) \
    printf("%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;
}
```

Pointeur polymorphe (void*)

1.188

```
#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));

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

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

    return 0;
}
```

Conversion de type

1.189

- Une conversion est en fait un changement de représentation.
 - ▣ Un entier en un double
 - ▣ Un double en un entier
 - ▣ Etc...

- **Attention** : le résultat d'une conversion de type peut être indéterminé.

Conversions implicites

1.190

- Elles sont provoquées par des opérateurs arithmétiques, logiques et d'affectation, lorsque les types des opérandes sont différents mais comparables.
- Pour les expressions arithmétiques et logiques :
 - ▣ les conversions sont effectuées du type le plus faible vers le plus fort. (char vers int, short vers int ...)
- Pour les affectations :
 - ▣ le résultat de la partie droite est converti dans celui de la partie gauche. (int x = 3/4;)

Conversions explicites

1.191

- Elles sont faites par le transtypage.
- La valeur de l'expression ainsi construite est le résultat de la conversion de l'expression dans le type.

```
int i;  
double d;  
enum E {rouge=1, bleu=2, vert=3} couleur;  
...  
d = 3;  
couleur = (enum E)((int) d);  
i = (int) couleur;  
(float) 3; /* \donc 3.0 */  
(int) 5.1267e2; /* \donc 512 */
```

Ligne de commande

1.192

- 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`.

Ligne de commande : exemple

1.193

- \$ commande -option fic1 199
- Dans le programme C
 - ▣ argc = 4
 - ▣ argv a 5 éléments significatifs et on peut les représenter comme suit :
 - argv[0] : “command”
 - argv[1] : “-option”
 - argv[2] : “fic1”
 - argv[3] : “199”
 - argv[4] : NULL

Ligne de commande

1.194

```
#include <stdio.h>

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

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

    while (*++argv){
        printf("%s*\n", *argv);
    }

    return 0;
}
```

Nombre variable de paramètres

1.195

- La liste variable de paramètres est dénotée par ... derrière le dernier paramètre fixe. Il y a **au moins un** paramètre fixe.
 - ▣ `int printf(const char *format, ...);`
- 4 macros sont définies dans le fichier `stdarg.h`
 - ▣ Le « type » **`va_list`** sert à déclarer le pointeur se promenant sur la pile d'exécution.
`va_list ap;`
 - ▣ La macro **`va_start`** initialise le pointeur de façon à ce qu'il pointe après le dernier paramètre nommé.
`void va_start (va_list ap, last);`

Nombre variable de paramètres

1.196

- 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);

Nombre variable de paramètres

1.197

```
#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++)
        printf("%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;
}
```

Nombre variable de paramètres

1.198

```
#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[]) {
    printf("%d\n", max(12, 18, 17, 20, 1, 34, 5, -1));
    printf("%d\n", max(12, 18, -1, 17, 20, 1, 34, 5, -1));
    return 0;
}
```

Entrées/Sorties

1.199

- Fichier de déclarations `stdio.h`
- E/S simples
 - ▣ Écriture d'un caractère sur la sortie standard

```
char c = '1';  
putchar(c);  
putchar('\n');
```

- ▣ Lecture d'un unique caractère sur l'entrée standard

```
int c;  
c = getchar();
```

Entrées/Sorties

1.200

- En fin de fichier standard, valeur EOF
- Écriture d'une chaîne de caractères (avec retour à la ligne) sur la sortie standard

```
char *s = "coucou";  
puts(s);  
puts("bonjour");
```

- Lecture d'une chaîne de caractères sur l'entrée standard (jusqu'à EOF ou `\n`) et `\0` est mis à la fin

```
char s[256];  
printf("Nom? ");  
gets(s);
```


Entrées/Sorties formatées

1.201

- Écriture sur la sortie standard

```
printf("décimal = %d, hexa = %x\n", 100, 100);  
printf("nb réel = %f, %g\n", 300.25, 300.25);
```

- Lecture sur l'entrée standard

```
int i;  
double x;  
printf("i = ? "); scanf("%d", &i);  
printf("x = ? "); scanf("%lf", &x);
```

Entrées/Sorties formatées

1.202

- Écriture dans une chaîne de caractères

```
char s[256];  
sprintf(s, "%s", "il fait beau");  
printf("*%s*\n", s);
```

- Lecture dans une chaîne de caractères

```
int i;  
sscanf("123", "%d", &i);  
printf("%d\n", i);
```

Conversion de base : printf

1.203

caractère	type du paramètre
d,i	Nombre décimal
o	Nombre octal non signé
x,X	Nombre hexadécimal non signé
u	Non décimal non signé
c	Caractère isolé
s	Chaîne de caractères
f	[-]m.dddddd
e, E	[-]m.dddddde/E+/-xx
p	pointeur
%	%

Conversion de base : scanf

1.204

caractère	Type de l'argument
d, i, n	Entier décimal
o, u, x	Entier décimal non signé
c	caractère
s	Chaîne de caractères
e, f, g	Nombre en virgule flottante
%	%

Entrées/Sorties : fichier de caractères

1.205

- **FILE *** est un descripteur de fichier

- Trois fichiers standard déclarés dans `stdio.h`
 - ▣ `FILE *stdin, *stdout, *stderr;`

- Déclaration d'un descripteur de fichier
 - ▣ `FILE *fd;`

Entrées/Sorties : fichier de caractères

1.206

- Ouverture d'un fichier (liaison entre le nom logique et le nom physique)
 - ▣ FILE *fopen (const char *filename, const char *type);
- "r" pour lecture, "w" pour écriture et "a" pour ajouter en fin de fichier, "b" à la fin du mode pour les fichiers binaires

- Fermeture d'un fichier
 - ▣ int fclose (FILE *stream);

Entrées/Sorties : fichier de caractères

1.207

```
FILE *fl, *fe;
```

```
fl = fopen("../ficLec", "r");
```

```
/* si ../ficLec n'existe pas, fl == NULL
```

```
sinon fl contient le descripteur de fichier correspondant au  
fichier physique de nom ../ficLec */
```

```
fe = fopen("ficEcr", "w");
```

```
/* si problème, fe == NULL
```

```
sinon fe contient le descripteur de fichier correspondant au  
fichier physique de nom ficEcr;  
si ficEcr existe, effacement du contenu,  
sinon création du fichier vide */
```

Écriture dans un fichier

1.208

- un caractère
 - ▣ `int fputc (int c, FILE *stream);`
 - ▣ `int putc (int c);`

- une chaîne
 - ▣ `int fputs (const char *s, FILE *stream);`

- un peu plus compliqué
 - ▣ `int fprintf (FILE *stream, const char *format, ...);`

Lecture dans un fichier

1.209

- un caractère
 - ▣ `int fgetc (FILE *stream);`
 - ▣ `int getc (FILE *stream);`

- une chaîne
 - ▣ `char *fgets (char *s, int n, FILE *stream);`

- un peu plus compliqué
 - ▣ `int fscanf (FILE *stream, const char *format, ...);`

Commande cat (cat.c)

1.210

```
#include <stdio.h>

void copy (FILE *f) {
    int c;

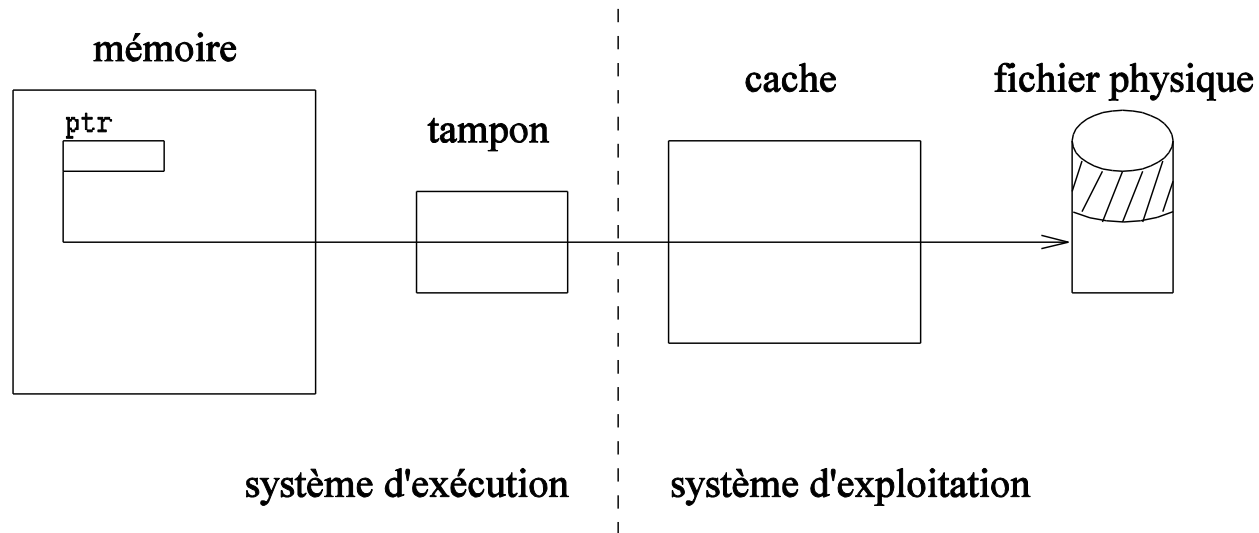
    while ((c = fgetc(f)) != EOF)
        fputc(c, stdout);
}

int main (int argc, char *argv[]) {
    FILE *f;
    if (argc == 1)
        copy(stdin);
    else {
        while (--argc) {
            if ((f = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr, "Ouverture impossible %s\n",
                    *argv);
                return 1;
            }
            else {
                copy(f);
                fclose(f);
            }
        }
    }
    return 0;
}
```

E/S sur fichiers quelconques

1.211

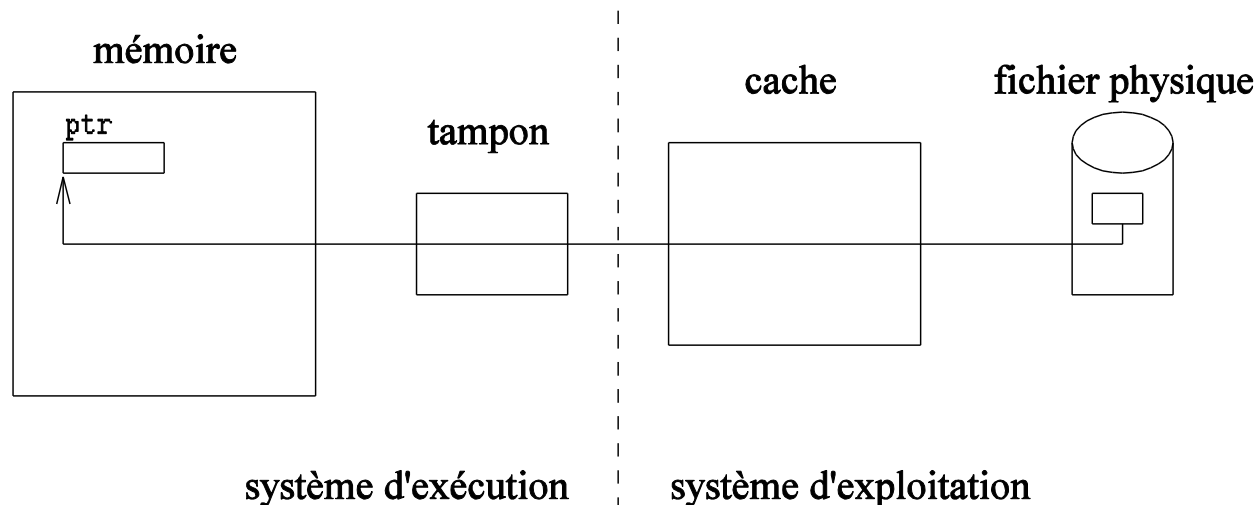
- Ouverture fopen et fermeture fclose
- Écriture
- `size_t fwrite (const void *ptr, size_t size, size_t nitems, FILE *stream);`



E/S sur fichiers quelconques

1.212

- Ouverture fopen et fermeture fclose
- Lecture
- `size_t fread (void *ptr, size_t size, size_t nitems, FILE *stream);`



E/S sur fichiers quelconques

1.213

```
/* fl ouvert en lecture, fe en écriture */  
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
char c;  
fread(&c, sizeof(char), 1, fl);  
fwrite(&i, sizeof(int), 5, fe);  
fwrite(&i, sizeof(int), 1, fe);  
fwrite(&c, sizeof(int), 1, fe);  
fwrite(&i, sizeof(char), 10, fe);
```

E/S Positionnement

1.214

□ Déplacement en octets

▣ `int fseek (FILE *stream, long offset, int ptrname);`

`ptrname`

<code>SEEK_SET==0</code>	Début de fichier
<code>SEEK_CUR==1</code>	Position courante
<code>SEEK_END==2</code>	Fin de fichier

- Indication de position
 - ▣ long ftell (FILE *stream);

- Fin de fichier
 - ▣ int feof (FILE *stream);

Fichiers

1.216

```
#include <stdio.h>

FILE *ecrire (char *nom, int n) {
    int i;
    FILE *f = fopen(nom, "w");

    for (i = 0; i < n; i++)
        fwrite(&i, sizeof(int), 1, f);
    fclose(f);
    return f;
}

FILE *lireEtAfficher (char *nom) {
    int i;
    FILE *f = fopen(nom, "r");

    while (fread(&i, sizeof(int), 1, f))
        printf("%d ", i);
    fputc('\n', stdout);
    fclose(f);
    return f;
}

int main (int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "%s filename\n", argv[0]);
        exit(2);
    }
    {
        ecrire(argv[1], 20);
        lireEtAfficher(argv[1]);
    }
    return 0;
}
```


Structure des programmes

1.217

- Deux types de durée de vie
 - ▣ **statique** ou permanente (c'est-à-dire la durée de vie est celle du programme)
 - ▣ **automatique** ou dynamique (c'est-à-dire la durée de vie est celle du bloc qui la déclare)
- Trois types de portée
 - ▣ bloc (ou fonction)
 - ▣ fichier (au sens .c = fichier source)
 - ▣ programme

Classes de variables

1.218

- La classe de mémorisation est spécifiée par
 - ▣ auto
 - ▣ extern
 - ▣ static
 - ▣ register.
- auto : pas ou plus utilisé
- extern : variable globale définie dans une autre fichier
- register : variable mise dans un registre
- static : variable dans un bloc conservant sa valeur d'un appel à l'autre

Variables et durée de vie

1.219

- Définition vs. déclaration
- **Définition d'une variable :**
Variable réellement créée, mémoire allouée
- ```
{ /* bloc */
 int v; /* variable automatique */
 /* déclare v et alloue la mémoire nécessaire
 au rangement d'un entier */
}
```
- La définition est UNIQUE

# Variables et durée de vie

1.220

- **Déclaration d'une variable :**  
**Pas de mémoire allouée, juste la nature de la variable est donnée.**
- `extern int v;`  
`/* définit v comme étant une variable de type entier */`
- **déclaration de référence : MULTIPLE**

# Variables internes

1.221

- Paramètres
- Variables automatiques (ou locales) : elles sont locales à un bloc
  - ▣ naissent à l'appel de la fonction (ou à l'entrée d'un bloc
  - ▣ meurent lorsque la fonction se termine (ou quand on sort du bloc).
- Classe auto

# Variables externes

1.222

- Elles servent à la communication entre fonctions (comme les paramètres) :
  - ▣ visibles du point de déclaration jusqu'à la fin du fichier physique ;
  - ▣ définies hors de toute fonction (niveau 0) ;
  - ▣ ont des valeurs permanentes, durant l'exécution du programme ;
  - ▣ initialisées lors de la définition.
- Chaque fonction doit la déclarer si elle veut l'utiliser :
  - ▣ de façon explicite, grâce à `extern`.
  - ▣ de façon implicite, par contexte (si la déclaration apparaît avant son utilisation).

# Variables static

1.223

- ❑ Les variables internes statiques sont locales à une fonction particulière mais « restent en vie » d'un appel sur l'autre.
- ❑ Les variables externes statiques sont locales au fichier source (fichier physique). Cela permet de ne pas les exporter.

```
void f(void) {
 static int S = 0;
 int L = 0; /* automatique */
 L++; S++;
 printf(" L=%d , S=%d \n ", L, S);
}
```

# Variables registres

1.224

- Cela permet d'indiquer au compilateur que la variable va être beaucoup utilisée et que le compilateur va pouvoir la ranger dans un registre (car l'accès à un registre est plus rapide qu'un accès à la mémoire).
- Seuls les variables automatiques et les paramètres formels d'une fonction peuvent avoir cette caractéristique. L'ensemble des types autorisés varie.
- Il est impossible de connaître l'adresse d'une variable **register**.



# Variables volatiles

1.225

- Indiquer à l'optimiseur qu'une variable peut changer de valeur même si cela n'apparaît pas explicitement dans le source du programme.

```
int main (int argc, char *argv[]) {
 volatile int var = -1;
 int i = 1;
 while (i)
 i = var; /* sans volatile, cette instruction est supprimée
 par l'optimiseur */
 return 0;
}
```

- Variables susceptibles d'être modifiées indépendamment du déroulement normal du programme : variable modifiée sur réception d'un signal ou d'une interruption, variable implantée à une adresse directement utilisée par la machine

# Initialisation des variables

1.226

- Si pas d'initialisation explicite, les variables sont initialisées à n'importe quoi.
- Si initialisation explicite, les variables static et extern doivent l'être avec une expression constante car initialisations élaborées à la compilation. Les initialisations des variables auto et register sont élaborées à chaque entrée dans le bloc (exécution) a , donc n'importe quelle expression est acceptée.

# Fonctions

1.227

- Par défaut, toute fonction est `extern` et elle est supposée rendre un `int` ou `char` s'il n'y a pas eu de déclaration explicite avant.
- Une fonction peut être `static` explicitement (dans ce cas-là, on limite la portée de la fonction au fichier physique).

# Domaine d'application

1.228

- Le domaine d'application d'une déclaration = région de texte du programme C dans laquelle cette déclaration est active.
  - ▣ **Variables globale** : Le domaine d'application d'un identificateur déclaré dans une déclaration de plus haut niveau s'étend entre son emplacement de déclaration et la fin du fichier physique.
  - ▣ **Paramètre formel** : Le domaine d'application d'un identificateur déclaré dans une déclaration de paramètre formel s'étend entre son emplacement de déclaration et la fin du corps de la fonction.

# Domaine d'application

1.229

## □ Domaine d'application

- ▣ **Variable automatique** : Le domaine d'application d'un identificateur déclaré au début d'un bloc s'étend entre son emplacement de déclaration et la fin du bloc.
- ▣ **Étiquette d'instruction** : Le domaine d'application d'une étiquette d'instruction englobe l'ensemble du corps de la fonction dans laquelle elle apparaît.
- ▣ **Macro** : Le domaine d'application d'un nom de macro de préprocesseur s'étend entre `#define` et la fin du fichier physique ou jusqu'à un `#undef` correspondant.

# Conseils

1.230

- Avoir un seul emplacement de définition (fichier source) pour chaque variable externe (omettre `extern` et avoir un initialiseur).
  - ▣ `int cpt = 0;`
- Dans chaque fichier source référençant une variable externe définie dans un autre module, utiliser `extern` et ne pas fournir d'initialiseur.
  - ▣ `extern int cpt;`

# Modularité

1.231

- Forme très simple de «modularité» reposant sur la notion de fichiers (et inclusion de fichiers).
- Rappels sur les déclarations de variables :
  - ▣ au début d'un bloc : locale, temporaire;
  - ▣ au niveau 0 : globale au programme, permanente;
  - ▣ déclaration static : locale (fichier ou fonction), permanente;
  - ▣ déclaration extern : référence à une définition.
- Rappel sur le mot-clé static : il permet de rendre une variable externe ou une fonction «privée» à un fichier; il permet de déclarer des variables internes permanentes.

# Pourquoi la modularité ?

1.232

- Un module est une unité de programme, c'est-à-dire un ensemble de fonctions réalisant un même traitement et un ensemble de variables utilisées par ces fonctions.
- Le découpage d'un programme en modules est indispensable à
  - ▣ la lisibilité;
  - ▣ la maintenance;
  - ▣ la ré-utilisation
- Dans un programme en langage C, on définira un module nom au moyen du couple de fichiers :
  - ▣ nom.c : le fichier d'implémentation contenant les définitions de toutes les fonctions et variables du module;
  - ▣ nom.h : le fichier de définitions contenant les déclarations de types, de constantes, de variables et de fonctions (prototypes).



# Règles de modularité

1.233

- Les fonctions et variables internes du module sont déclarées locales à ce module au moyen du mot-clé `static`.
- Un fichier de déclarations ne contient aucune définition de variable, mais seulement :
  - ▣ des définitions de types
  - ▣ des déclarations de variables
  - ▣ des déclarations de fonctions.
- Le fichier de déclarations d'un module est inclus dans le source de ce module, afin de permettre un contrôle des déclarations par le compilateur.

# Exemple

1.234

```
#ifndef _GENERATOR_H
#define _GENERATOR_H

extern void generator_reset (int);
extern int generator_current (void);
extern int generator_next (void);

#endif
```

# Example

1.235

```
#include "generator.h"

static int value = 0;

void generator_reset (int beg) {
 value = beg;
}

int generator_current (void) {
 return value;
}

int generator_next (void) {
 return value++;
}
```

# Modularité compilation

1.236

- compilation :
  - ▣ gcc -c -Wall -pedantic -ansi generator.c
- Ne crée que le fichier objet et pas d'exécutable.
- un fichier utilisateur main.c :

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

int main (int argc, char *argv[]) {
 printf("%d\n", generator_current());
 generator_reset(4);
 printf("%d\n", generator_next());
 printf("%d\n", generator_current());
 return 0;
}
```

- Compilation :
  - ▣ gcc -Wall -pedantic -ansi generator.c main.c
- Crée l'exécutable a.out.

# Options de compilations

1.237

- Générales (principes généraux mais syntaxe spécifique gcc)
  - ▣ -c : compile et assemble les fichiers sources et stoppe avant l'édition de liens (fichier .o ou .obj).
  - ▣ -S : stoppe après la compilation propre; n'assemble donc pas (fichier .s).
  - ▣ -E : stoppe après le passage du préprocesseur (sur la sortie standard).
  - ▣ -o file : (output) redirige la sortie sur le fichier file.
  - ▣ -v : (verbose) option intéressante, car liste toutes les commandes appelées par gcc.
  
- Démo Visual C++ (définir IDE, front end)

# Options de compilations

1.238

## □ Répertoires

- ▣ -I dir : ajoute dir à la liste des répertoires où chercher les fichiers à inclure. (i majuscule)
- ▣ -L dir : ajoute dir à la liste des répertoires où chercher les bibliothèques -l.

## □ Pour déboguer

- ▣ -g : met les informations nécessaires dans l'exécutable pour le débogueur.
- ▣ -g format : pour un format précis (gdb, coff, xcoff, dwarf).

# Options de compilations

1.239

## □ Optimisations

- ▣ -O1, -O2, -O3 : afin d'optimiser.
- ▣ -O0 : pour ne pas optimiser.

## □ Cible

- ▣ -b machine : pour la cross-compilation.
- ▣ -V version : quelle version utiliser (option utilisée bien sûr si plusieurs versions sont installées).

# Options de compilations

1.240

## □ Avertissements

- ▣ -w : pour supprimer les avertissements du compilateur.
- ▣ -Wall : à utiliser si on veut avoir des programmes «vraiment propres».
- ▣ -pedantic -ansi : pour être sûr de «coller» la norme ansi.



# Outil make

1.241

- Vocation de **make** = gérer la construction de logiciels modulaires.
- Réaliser des compilations (ou toute autre action) dans un certain ordre (compatible avec des règles de dépendance) : fichier **makefile**.
- Dans le cas où make réalise des actions autres que la compilation, cet outil est équivalent à un script SHELL, si ce n'est qu'il y a la gestion des règles de dépendance en plus.
- make ne produit les cibles que si les cibles dont elles dépendent sont plus récentes qu'elles.

# Utilitaire make et makefile

1.242

- ❑ Existe partout (make sur Linux, nmake sur windows)
- ❑ Exécute une suite d'instructions contenues dans un fichier dit « makefile »
- ❑ Souvent le fichier « makefile » s'appelle Makefile
- ❑ Structure du fichier
  - entree: dépendances
  - action à réaliser
- ❑ Attention tabulations importantes !!!! Avant actions

# makefile

1.243

- ❑ **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**

- **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
- `$`

# Structure générale du makefile

1.245

- Ce fichier de texte contient une suite d'entrées qui spécifient les dépendances entre les fichiers.
- Il est constitué de lignes logiques (une ligne logique pouvant être une ligne physique ou plusieurs lignes physiques séparées par le signe barre à l'envers).
- Les commentaires débutent par le signe `#` et se terminent à la fin de ligne.
- Contenu (ordre recommandé) :
  - ▣ définitions de macros
  - ▣ règles implicites
  - ▣ règles explicites ou entrées

# Règles explicites

1.246

cible<sub>1</sub> ... cible<sub>m</sub> : dépendance<sub>1</sub> ... dépendance<sub>n</sub>

action<sub>1</sub>

action<sub>2</sub>

...

action<sub>p</sub>

- Traduction : mettre à jour les cibles : cible<sub>1</sub> ... cible<sub>m</sub> quand les dépendances dépendance<sub>1</sub> ... dépendance<sub>n</sub> sont modifiées en effectuant les opérations action<sub>1</sub>, action<sub>2</sub> ... action<sub>p</sub>
- La première entrée est la cible principale

# makefile : exemple

1.247

```
onyva : entiers.o matrices.o pal.o
 @gcc -o onyva entiers.o matrices.o pal.o
 @echo "exécutable $@ créé"
entiers.o : entiers.h entiers.c
 @echo "compilation de $*.c..."
 @gcc -c -Wall -pedantic -ansi entiers.c
matrices.o : matrices.h entiers.h elements.h matrices.c
 @echo "compilation de $*.c..."
 @gcc -c -Wall -pedantic -ansi matrices.c
pal.o : matrices.h pal.c
 @echo "compilation de $*.c..."
 @gcc -c -Wall -pedantic -ansi pal.c

clean :
 -rm -f *.o; rm -f onyva; rm -f *~

print :
 a2ps entiers.h entiers.c | lpr
 a2ps matrices.h matrices.c | lpr
 a2ps elements.h pal.c | lpr
```

- Si une action s'exécute sans erreur (code de retour nul), **make** passe à l'action suivante de l'entrée en cours, ou à une autre entrée si l'entrée en cours est à jour. Si erreur (et - absent), **make** arrête toute exécution.
- Les actions peuvent être précédées des signes suivants :
  - ▣ - si l'action s'exécute avec un code de retour anormal (donc erreur), **make** continue
  - ▣ @ l'impression de la commande elle-même est supprimée
  - ▣ @-, -@ pour combiner les précédents



# Commandes usuelles

1.249

- Il est bien pratique d'avoir des entrées d'impression, de nettoyage ou d'installation.

impression :

```
-lpr *.c *.h
```

menage :

```
@-rm *.o *.out core *~
```

install :

```
mv a.out /usr/bin/copy
```

```
chmod a+x /usr/bin/copy
```

# Appel de make

1.250

- `make [-f nom_du_makefile] [options] [nom_des_cibles]`
- Options :
  - ▣ `-f` : si option manquante, make prendra comme fichier de commandes un des fichiers `makefile`, `Makefile`, `s.makefile` ou `ous.Makefile` (s'il le trouve dans le répertoire courant)
  - ▣ `-d` : permet le mode «Debug», c'est-à-dire écrit les informations détaillées sur les fichiers examinés ainsi que leur date
  - ▣ `-n` : imprime les commandes qui auraient dû être exécutées pour mettre à jour la cible principale (mais ne les exécute pas).

# Commandes usuelles

1.251

- Options (suite) :
  - ▣ -p : affiche l'ensemble complet des macros connues par make, ainsi que la liste des suffixes et de leurs règles correspondantes
  - ▣ -s : n'imprime pas les commandes qui s'exécutent; make fait son travail en silence
  - ▣ -S : abandonne le travail sur l'entrée courante en cas d'échec d'une des commandes relatives à cette entrée (L'option opposée est -k).
  - ▣ -t : permet de mettre à jour les fichiers cible
  - ▣ nom\_des\_cibles : si aucun nom n'est donné, la cible principale sera la première entrée explicite du makefile.

# make : exemple d'appel

1.252

- ❑ make
- ❑ make pal.o
- ❑ make clean
- ❑ make onyva
- ❑ make (@print@)

# Makefile : Macros

1.253

- Définition de macros :
- Syntaxe
  - ▣ `chaîne1 = chaîne2`
- `chaîne2` est une suite de caractères se terminant au caractère `#` de début de commentaire ou au caractère de fin de ligne (s'il n'est pas précédé du caractère d'échappement `\`).
- Dans la suite du makefile, chaque apparition de `$(chaîne1)` sera remplacée par `chaîne2`.
- Exemples :
  - ▣ `OBJETS=f1.o f2.o f3.o`
  - ▣ `SOURCES=f1.h f1.c f2.h f2.c f3.h f3.c`
  - ▣ `REPINST=/usr/bin`

- Remplacement d'une sous-chaîne par une autre dans une chaîne :
- Syntaxe : `$(chaîne:subst1=subst2)`
  - ▣ `subst1` est remplacé par `subst2` dans chaîne.
- Exemples :
  - ▣ `$(OBJETS:f2.o=)`
  - ▣ `$(OBJETS:f2.o=fn.o)`
  - ▣ `$(REPINST:bin=local/bin)`

# Macros internes

1.255

- `$*` le nom de la cible courante sans suffixe
- `$@` le nom complet de la cible courante
- `$<` la première dépendance
- `$$` la liste complète des dépendances
- `$$?` la liste des dépendances plus récentes que la cible

- Les variables d'environnement sont supposées être des définitions de macros. Par défaut :
  - ▣ Les variables d'environnement l'emportent sur les macros internes définies par défaut.
  - ▣ Les macros définies dans le makefile l'emportent sur les variables d'environnement.
  - ▣ Les macros définies dans une ligne de commande l'emportent sur les macros définies dans le makefile.
- L'option -e change tout ça de telle façon que les variables d'environnement l'emportent sur les macros définies dans le makefile.



# Exemple

1.257

```
CC=gcc
CFLAGS=-Wall -pedantic -ansi
OBJETS=entiers.o matrices.o pal.o
SOURCES=*.h *.c Makefile ALIRE
CARDIR=/usr/profs/Licence

onyva : $(OBJETS)
 @$ (CC) -o onyva $(OBJETS)
 @echo "$(USER), l'exécutable $@ est créé"
entiers.o : entiers.h entiers.c
 @echo "compilation de $*.c..."
 @$ (CC) -c $(CFLAGS) entiers.c
matrices.o : matrices.h entiers.h elements.h \
 matrices.c
 @echo "compilation de $*.c..."
 @$ (CC) -c $(CFLAGS) matrices.c
pal.o : matrices.h pal.c
 @echo "compilation de $*.c..."
 @$ (CC) -c $(CFLAGS) pal.c

clean :
 -rm -f *.o; rm -f onyva; rm -f *~

print :
 a2ps entiers.h entiers.c | lpr
 a2ps matrices.h matrices.c | lpr
 a2ps elements.h pal.c | lpr

copy :
 tar czf $(CARDIR)/binomes.tgz $(SOURCES)
 chmod o+r $(CARDIR)/binomes.tgz
```

# Règles implicites

1.258

- Elles servent à donner les actions communes aux fichiers se terminant par le même suffixe.

`.SUFFIXES:` liste de suffixes

`.source.cible :`

actions

- Dans `.SUFFIXES:`, on définit les suffixes standard utilisés par les outils pour identifier des types de fichiers particuliers.
- Traduction : À partir de `XX.source`, on produit `XX.cible` grâce à actions.
- Pour supprimer les règles implicites par défaut, appeler `make` avec l'option `-r`, ou écrire `.SUFFIXES: seulement`.

# Règles implicites

1.259

- Exemple : Pour tous les fichiers sources C (ayant comme suffixe .c), on appelle le compilateur C avec l'option -c.

.SUFFIXES: .out .o .h .c

.c.o:

gcc -c -Wall -pedantic -ansi \$\*.c

# Exemple : règles implicites par défaut

1.260

```
CC=gcc
CFLAGS=-Wall -pedantic -ansi
OBJETS=entiers.o matrices.o pal.o
SOURCES=*.h *.c Makefile ALIRE
CARDIR=/usr/profs/Licence

onyva : $(OBJETS)
 @$(CC) -o onyva $(OBJETS)
 @echo "$(USER), l'exécutable $@ est créé"
entiers.o : entiers.h entiers.c
matrices.o : matrices.h entiers.h elements.h \
 matrices.c
pal.o : matrices.h pal.c

clean :
 -rm -f *.o; rm -f onyva; rm -f *~

print :
 a2ps entiers.h entiers.c | lpr
 a2ps matrices.h matrices.c | lpr
 a2ps elements.h pal.c | lpr

copy :
 tar czf $(CARDIR)/binomes.tgz $(SOURCES)
 chmod o+r $(CARDIR)/binomes.tgz
```

# Exemple : changement des règles implicites par défaut

1.261

```
CC=gcc
CFLAGS=-Wall -pedantic -ansi
OBJETS=entiers.o matrices.o pal.o
SOURCES=*.h *.c Makefile ALIRE
CARDIR=/usr/profs/Licence
.c.o:
 @echo "compilation de $*.c..."
 @$ (CC) -c $(CFLAGS) $*.c

onyva : $(OBJETS)
 @$ (CC) -o $@ $(OBJETS)
 @echo "$(USER), l'exécutable $@ est créé"
entiers.o : entiers.h entiers.c
matrices.o : matrices.h entiers.h elements.h \
 matrices.c
pal.o : matrices.h pal.c

clean :
 -rm -f *.o; rm -f onyva; rm -f *~

print :
 a2ps entiers.h entiers.c | lpr
 a2ps matrices.h matrices.c | lpr
 a2ps elements.h pal.c | lpr

copy :
 tar czf $(CARDIR)/binomes.tgz $(SOURCES)
 chmod o+r $(CARDIR)/binomes.tgz
```

# Exemple : avec demande à l'utilisateur

1.262

```
SHELL=zsh
```

```
CC=gcc
```

```
CFLAGS=-Wall -pedantic -ansi
```

```
.c:
```

```
 @echo "avec Debug? "
```

```
 @-read -q REP; \
```

```
 case ${REP} in \
```

```
 y) $(CC) $(CFLAGS) -g -o $@ $*.c;; \
```

```
 n) $(CC) $(CFLAGS) -o $@ $*.c;; \
```

```
 esac
```

# Autre exemple

1.263

```
SHELL=zsh
CC=gcc
CFLAGS=-Wall -pedantic -ansi
.c.o:
 @print "compilation de $*.c..."
 @$ (CC) $ (CFLAGS) -c $*.c

all :
exécution d'un script shell pour demander
à l'utilisateur quelle implémentation il
désire
 @demande
pileL : pileL.o pal.o
 @print "edl pour impl. liste..."
 @$ (CC) -o pileL pileL.o pal.o
pileT : pileT.o pal.o
 @print "edl pour imp. tableau..."
 @$ (CC) -o pileT pileT.o pal.o
pileT.o : pileT.c pile.h
pileL.o : pileL.c pile.h
pal.o : pal.c pile.h

clean :
 -rm -f *~ core pileL pileT *.o
print :
 a2ps pal.c pile.h pileT.c pileL.c | lpr
```

# Autre exemple

1.264

```
echo -n "Liste ou tableau (L/T)? "
read rep
case $rep in
t|T) make pileT;;
l|L) make pileL;;
*) echo "pas possible...";;
esac
```



# make sous windows

1.265

- Il existe un ou plusieurs utilisateurs make sous chaque système
- Attention : souvent pas compatible entre eux, mais fonctionnement commun
- make sous Windows = nmake
- Sous windows il faut exécuter vcvars32 ou un équivalent
  - ▣ Allez dans démarrer / Tous les programme / Microsoft Visual Studio 2010 / Visual Studio Tools puis cliquer sur un « command prompt »
  - ▣ Vous pouvez exécuter nmake /help
  - ▣ Nécessaire pour avoir plusieurs versions de compilateurs installées sur la même machine

# make

1.266

- Les IDE créent des makefile.
- Visual Studio : positionnez la souris sur un projet, puis faites un clique droit, puis Propriétés. Dans la partie gauche sous C/C++ et Linker il y a une entrée Command Line : elle permet de voir la commande qui est effectivement appelée pour compiler et pour linker

# Le préprocesseur

1.267

- Fonctions du préprocesseur
- Il est appelé avant chaque compilation par le compilateur. Toutes les directives commencent par un # en début de ligne.
  - ▣ Inclusion textuelle de fichiers (`#include`)
  - ▣ Remplacements textuels (`#define`)
    - Définition de constantes
    - Définition de macros
  - ▣ Compilation conditionnelle (`#if` `#ifdef` `#ifndef` `#else` `#endif`)
- Remarque : Récursivité des définitions.

# Définition des constantes

1.268

- `#define nom expression`
- `#undef nom`
- Dans le fichier concerné, `nom` sera remplacé textuellement par `expression` (sauf dans les chaînes de caractères et les commentaires).
- Exemples :
  - ▣ `#define FALSE 0`
  - ▣ `#define TRUE 1`
  - ▣ `#define NULL ((char*) 0)`
  - ▣ `#define T_BUF 512`
  - ▣ `#define T_BUFDBLE (2 * T_BUF)`

# Remarques

1.269

- ❑ Certains préprocesseurs produisent un message d'avertissement s'il y a re-définition d'une macro, mais remplacent la valeur par la nouvelle.
- ❑ D'autres ont une pile de définitions.
- ❑ La norme ansi ne permet pas l'empilement.

# Inclusion de fichiers sources

1.270

- `#include "nom_du_fichier"`
- `#include <nom_du_fichier>`
- Avec les `<>`, le préprocesseur ne va chercher que dans le ou les répertoires standards.
  - ▣ `/usr/include`
  - ▣ `/include`
- Avec les guillemets, le préprocesseur va chercher à l'endroit spécifié, puis dans le ou les répertoires standards.
- On peut passer une option au compilateur pour lui expliquer où chercher (`-I`)

# Définition de macros

1.271

- `#define nom(par1, ..., parn) expression`
- Dans `expression`, il est recommandé de parenthéser les `pari` afin d'éviter des problèmes de priorité lors du remplacement textuel des paramètres (rien à voir avec le passage des paramètres lors d'un appel de sous-programme).
- Exemples :
  - ▣ `#define getchar() getc(stdin)`
  - ▣ `#define putchar(c) putc(c, stdout)`
  - ▣ `#define max(a, b) (((a) > (b)) ? (a) : (b))`
  - ▣ `#define affEnt(a) fprintf(stdout, "%d", a)`
  - ▣ `#define p2(a) ((a) * (a))`

# Définition de macros

1.272

- Une macro est définie à partir du `#define` jusqu'à la fin de ligne
- Pour passer à la ligne, sans utiliser une fin de ligne, on doit utiliser le caractère `\`
- Exemple

```
#define PRINT_TAB(tab,n)\
 int i;\n for(i=0;i<n;i++){ printf("%d ", tab[i]);}\n printf("\n");
```



# Définition de macros

1.273

- `##` va permettre de concaténer
- Macro `concat(a,b)` je veux concaténer `a` et `b` ?
  - ▣ `#define concat(a,b) a##b`
- ATTENTION a la priorité des opérateurs : on parenthèse
  - `#define max(a,b) (a < b) ? b : a;`
  - `#define max(a,b) ((a) < (b)) ? (b) : (a);`
- ATTENTION les macros font du remplacement de texte
  - ▣ `#define max(a,b) ((a) < (b)) ? (b) : (a);`
  - ▣ `max(i++,j++)` : mauvais résultat !

# Macros prédéfinies

1.274

- `__LINE__` ligne courante dans le fichier source
- `__FILE__` nom du fichier source
- `__DATE__` date de compilation du programme
- `__HEURE__` heure de compilation du programme
- `__STDC__` à 1 si implémentation conforme à ansi

# Compilation conditionnelle

1.275

```
#if expression_constante
#ifdef expression_constante
#ifndef expression_constante
 liste_instructions_ou_déclarations
#else
 liste_instructions_ou_déclarations
#endif
```

# Compilation conditionnelle

1.276

- Il est possible d'emboîter des commandes de compilation conditionnelle.
- La compilation conditionnelle permet :
  - ▣ la paramétrisation à la compilation des structures de données;
  - ▣ de gagner de la place en ôtant le code inutile à l'exécution;
  - ▣ de prendre des décisions à la compilation plutôt qu'à l'exécution.

# Exemple

1.277

```
#include <stdio.h>

#if 0
/* partie de programme en commentaires */
...
#endif

 int main (int argc, char *argv[]) {
 printf("%d\n", __STDC__);
#if __STDC__
 printf("ansi\n");
#else
 printf("non ansi\n");
#endif
 return 0;
 }
```

# Autres directives

1.278

- `#line` fournit un numéro de ligne
- `#elif` sinon si
- `defined(nom)` détermine si `nom` est défini comme une macro de préprocesseur
- `#error "mess"` arrête la compilation avec le message d'erreur
- `#warning "mess"` produit l'avertissement `mess` à la compilation

# Options du compilateur (très utilisées)

1.279

- -Dmacro=defn définit la macro avec la chaîne defn comme valeur
- -Umacro pour ôter la définition de la macro

# Exemple

1.280

```
#ifndef _GENERATOR_H
#define _GENERATOR_H

extern void generator_reset (int);
extern int generator_current (void);
extern int generator_next (void);

#endif
```



# Example

1.281

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

static int value = 0;

void generator_reset (int beg) {
 value = beg;
}

int generator_current (void) {
 return value;
}

int generator_next (void) {
 return value++;
}

#ifdef SELFEXEC
int main (int argc, char *argv[]) {
 generator_reset(argv[1] != NULL ?
 atoi(argv[1]) : 0);
 do {
 printf("%d\n", generator_current());
 } while (generator_next() < 10);
 return 0;
}
#endif
```

# Assert et NDEBUG

1.282

```
#include <assert.h>
```

```
void assert(int exp);
```

- La macro `assert` est utilisé pour tester des erreurs. Si `exp` est évalué à 0, alors `assert` écrit des informations sur la sortie erreur standard (`stderr`).
- Si la macro `NDEBUG` est définie alors `assert` est ignorée

# Assert et NDEBUG

1.283

- Code

```
x = t[i] + 2;
```

- Dangereux si  $i$  n'est pas dans les bornes ( $i \geq 0$  et  $i < n$ )

- Problème tester coute cher

- Solution assert

```
assert(i >= 0 && i < n);
```

```
x = t[i] + 2;
```

- Avec NDEBUG définie (Not Debug) ne fait rien du tout = cout nul

- Avec NDEBUG non définie, alors on teste les valeurs et si erreur alors message du type

- ▣ « Assertion failed in line XXX »

# Edition de liens

1.284

- L'édition de lien permet de constituer un module binaire exécutable à partir de bibliothèques et de fichiers objets compilés séparément, en résolvant les références (externes) qui n'ont pas été résolues lors des passes précédentes du processus de compilation.
- Elle extrait des bibliothèques les modules utiles aux fonctions effectivement utilisées.
- Chaque objet externe doit avoir une et une seule définition dans l'ensemble des modules à assembler.

# Bibliothèque (library)

1.285

- C'est un fichier « un peu spécial » contenant la version objet d'une fonction ou de plusieurs traitant d'un sujet particulier (ou la version objet d'un « module »).
- Sous UNIX, les répertoires standard des bibliothèques sont `/lib` ou `/usr/lib`.
- La bibliothèque d'archives standard C est en fait le fichier `/usr/lib/libc.a`, et elle contient entre autres `fprintf.o`, `atoi.o`, `strncat.o`, etc.
- La bibliothèque d'archives mathématique est en fait le fichier `/usr/lib/libm.a`, contenant entre autres `e_pow.o`, `s_sin.o`, etc

# Utilisation explicite d'une bibliothèque

1.286

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

#define dem(n, v) printf(n " = ? ");\
 fscanf(stdin, "%d", &v)

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

 dem("x", x);
 dem("y", y);
 printf("%d^%d = %.2f\n", x, y, pow(x, y));
 return 0;
}
```

# Compilation et édition de liens

1.287

- ❑ `gcc -Wall -pedantic -ansi -c power.c`
- ❑ `gcc power.o : ERROR`
- ❑ `...: In function 'main':`  
`...: undefined reference to 'pow'`  
`...: ld returned 1`
- ❑ OK :
  - ❑ `gcc power.o -lm`
  - ❑ `gcc power.o /usr/lib/libm.a`
  - ❑ `gcc power.o /usr/lib/libm.so`

# Edition de liens statique

1.288

- Elle extrait le code de la fonction et le recopie dans le fichier binaire.
- Tout est donc contenu dans le fichier binaire, ce qui permet une exécution directe du programme.
- Inconvénients :
  - ▣ Problème de mémoire : le code de la fonction est chargé en mémoire autant de fois qu'il y a de processus l'utilisant.
  - ▣ Problème de version : si la bibliothèque change, les applications déjà construites continueront d'utiliser l'ancienne version...



# Edition de liens statique

1.289

- `gcc power.o -static /usr/lib/libm.a`
  - ▣ `taille de a.out = 1656301 octets`
- `gcc power.o -static -lm`
  - ▣ `taille de a.out = 1656301 octets`

# Edition de liens dynamique

1.290

- Dans ce cas, l'éditeur de liens ne résout plus totalement les références, mais construit une table de symboles non résolus contenant des informations permettant de localiser ultérieurement les définitions manquantes.
- Les résolutions sont alors seulement faites lors de l'exécution.
  - ▣ liaison dynamique **immédiate** (lors du chargement du programme);
  - ▣ liaison dynamique **différée** (à la première référence d'un objet).
- Inconvénient : ralentissement du chargement du programme.

# Edition de liens dynamique

1.291

- Les bibliothèques de fonctions «reliables,» dynamiquement sont appelées
  - ▣ objets partagés (fichiers .so)
  - ▣ bibliothèques partagées (fichiers .sl)
  - ▣ Dynamic link library (dll) sous Windows
  
- `gcc power.o /usr/lib/libm.so`
  - ▣ taille de `a.out` = 14298 octets
  
- `gcc power.o -lm`
  - ▣ taille de `a.out` = 14298 octets

# Options de compilation

1.292

- Tous les fichiers dont les noms n'ont pas de suffixe connus sont considérés par gcc comme des fichiers objets (et sont donc reliés par l'éditeur de liens).
- `-lnom_de_biblio` : l'éditeur de liens va chercher, dans la liste des répertoires standard des bibliothèques, la bibliothèque `nom_de_biblio`, qui est en fait un fichier nommé `lib` (`nom_de_biblio.a` ou `.so` et `.sl` si dynamique).
- `-Lnom_de_chemin` : ajoute `nom_de_chemin` à la liste des répertoires standard des bibliothèques.
- `-static` : sur les systèmes acceptant l'édition de liens dynamique, elle permet d'éviter l'édition de liens avec des bibliothèques partagées.

# Options de compilation

1.293

- Pour Windows :
- "*nom\_de\_biblio.lib*" : l'éditeur de liens va chercher, dans la liste des répertoires standard des bibliothèques, la bibliothèque *nom\_de\_biblio*, qui est en fait un fichier nommé lib
- /LIBPATH:"*nom\_de\_chemin*" : ajoute *nom\_de\_chemin* à la liste des répertoires standard des bibliothèques.
- /MT : édition de lien statique
- /MD : édition de liens dynamique

# Construction de bibliothèques

1.294

- Il est nécessaire de fournir un (ou des) `.h`, qui servira d'interface avec la bibliothèque :
  - ▣ définitions de constantes symboliques et de macros;
  - ▣ définitions de types;
  - ▣ déclarations de variables globales externes;
  - ▣ déclarations de fonctions (en fait, leur prototype).
- Et bien sûr, il y a le (ou les) `.c`, qui définit les variables et les fonctions déclarées dans le (ou les) `.h`, et autres objets privés.

# Création de bibliothèques

1.295

- Après la mise au point de la fonction ou du module, il suffit de créer la bibliothèque :
- Bibliothèque d'archives : réunir un ensemble d'«objets» en une seule bibliothèque d'archives, en vue de leur reliure statique.
  - ▣ `ar [options] archive fichiers`
- Options :
  - ▣ `-t` : affiche le contenu de l'archive
  - ▣ `-r` : remplace ou ajoute le(s) fichier(s) dans l'archive
  - ▣ `-q` : ajoute le(s) fichier(s) à la fin de l'archive (sans contrôler leur existence)
  - ▣ `-d` : supprime le(s) fichier(s) spécifié(s) de l'archive
  - ▣ `-x` : extrait le(s) fichier(s) de l'archive sans que le contenu de l'archive soit modifié

# Examples

1.296

- ❑ `ar -r lib/libdiv.a generator.o`
- ❑ `ar -t lib/libdiv.a`
- ❑ `generator.o`
- ❑ `ar -r lib/libdiv.a pow.o`
- ❑ `ar -t lib/libdiv.a`
- ❑ `generator.o`
- ❑ `pow.o`



# Bibliothèque partagée

1.297

- Bibliothèque partagée (produite par gcc ou ld)
- Exemple :
- `gcc -c -shared -o pow.so pow.c`
- `gcc power.o pow.so`



# TODO

1.299

- Faire plus sur les opérations de décalage
- Après slide 181 (parcours des chaînes de caractères) faire le détail des boucles et donner le détail de strcpy
  - ▣ Progressivement on retire tout: on commence par strlen et compter le nombre de caractères avec une boucle i, puis des [] etc..
  - ▣ Expliquer le problème avec les strlen dans les autres appels: c'est important pour ceux qui viennent du Java
  - ▣ Ensuite faire le imp
  - ▣ Puis montrer le strcpy()

Expliquer que `ch=« toto »` ne peut pas marcher

Expliquer que `ch==« toto »` ne fait que comparer des pointeurs

# TODO

1.300

- Pour les pointeurs reprendre l'exemple donné en TP avec tous les types de passage de paramètres
- Voir le petit enonce
- 
- Leur donner la signature de `qsort` (voir wikipedia) et leur demander de trier un tableau en appelant `qsort`
- Leur demander de faire une fonction `for_each` qui applique une operation à tous les éléments d'un tableau