Le langage C

Jean-Marc CIEUTAT
Enseignant-Chercheur
ESTIA/LIPSI

Planning & objectifs

- Planning :
  - 10 h de Cours
  - 6 h de Travaux Dirigés
  - 14 h de Travaux Pratiques

- Objectif :
  - Apprendre le langage de programmation le plus utilisé en informatique industrielle

Avertissements

- La maîtrise d'un langage informatique s'acquiert devant un ordinateur en programmant.

- Il faut au moins 6 mois de pratique pour devenir un bon programmeur en langage C, puis en encore 6 mois pour devenir un bon programmeur en langage C++.

- Conseils :
  - Lecture préalable du livre « Introduction au langage C » de Bernard Cassagne.
  - Programmer les exemples ou exercices du cours.

Références

- B.W. Kernighan & D.M. Ritchie. Le langage C. 2ème édition. Prentice Hall, 1990.
- Steve Bourne. Le système UNIX. InterEditions. 1985
- ISO/IEC. Programming languages – C. ISO/IEC 1999.
- ACCU. Association des utilisateurs de C et de C++. http://www.accu.org/
- B. Stroustrup. The C++ programming Language. Prentice Hall INC., Englewood Cliffs, New Jersey 1986

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les chaînes de caractères
- Les pointeurs
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

Historique

- En 1976 :

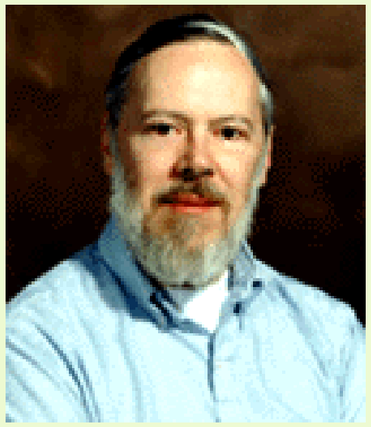
  - Bell Labs abandonne le projet MULTICS.

  - Ken Thompson, programmeur système chez Bell Labs, se trouve désoeuvré. MULTICS devait être un nouveau système d'exploitation multi-tâches et multi-utilisateurs utilisable pour la commande de systèmes de télécommunications, mais le projet est abandonné parce que trop coûteux, et que les perspectives sur la base matérielle utilisée sont trop restreintes : à cette époque, le mini-ordinateur le plus performant était le PDP-8 de Digital Equipment Corporation.

  Ken Thompson décide alors de développer un système d'exploitation équivalent écrit en assembleur : c'est la naissance d'UNIX.

Historique (suite)

- Un autre programmeur de Bell Labs, Dennis Ritchie utilise le Langage BCPL, mais le trouve inadapté à ses besoins. Il va concevoir un langage dérivé de BCPL, et l'appellera B (vraissemblablement la première lettre de BCPL). B ne sortira jamais officiellement des tiroirs de Dennis Ritchie.



Aidé par Brian Kernighan, B va connaître un nouveau développement. Le Langage ainsi créé se nommera C.

Historique (suite)

- Dennis Ritchie parvient à persuader Ken Thompson de réécrire UNIX sur une machine plus performante, un PDP11. Au lieu d'utiliser l'assembleur, on va utiliser le langage de Ritchie, le C. Cette décision est à considérer comme un des plus importants tournants de l'histoire de l'informatique : Pour la première fois, on réalise un système d'exploitation écrit dans un langage indépendant de la machine cible, un système d'exploitation portable au niveau source.

Cette grande première va faire le succès d'UNIX et du langage C. 90 % du noyau d'UNIX est écrit en langage C ; le compilateur C est lui-même écrit en langage C.

Propriétés du langage C

- Des générateurs de code pour la plupart des processeurs du marché (large couverture)
- Présent sur presque toutes les plate-formes de développement (large diffusion)
- Le langage le plus utilisé pour les systèmes temps réel et les systèmes embarqués (code généré très efficace)
- Souvent considéré comme un “assembleur de haut niveau”
- Facile à appréhender, permet d’acquérir une bonne connaissance des ordinateurs
- Programmation très concise
- Et le C ANSI vit le jour
- Migration de C vers C++

Comment créer un programme C ?

☺ Choix de l'algorithme (phase de **réflexion** basée sur l'Algorithmique et Structure de Données).

☺ Traduction de l'algorithme en instructions C et saisie des instructions dans un fichier texte, le **fichier source** avec l'extension *.c ou *.cpp.

💻 Compilation en 3 étapes :
Phase de pré-processing des fichiers *.c ➔ inclusion des fichiers d'en-tête .h,
Compilation des fichiers *.c ➔ génération des **fichiers objets** *.o ou *.obj,
Édition de liens avec les différentes librairies du langage C et les fichiers objets ➔ obtention du **fichier exécutable** *.exe.

☺ Test du programme.

Quelques exemples en C : « Hello World »

fichier

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

Le pré-processeur est utilisé pour partager (ajouter) les informations entre les fichiers sources

fonction

```
void main()
```

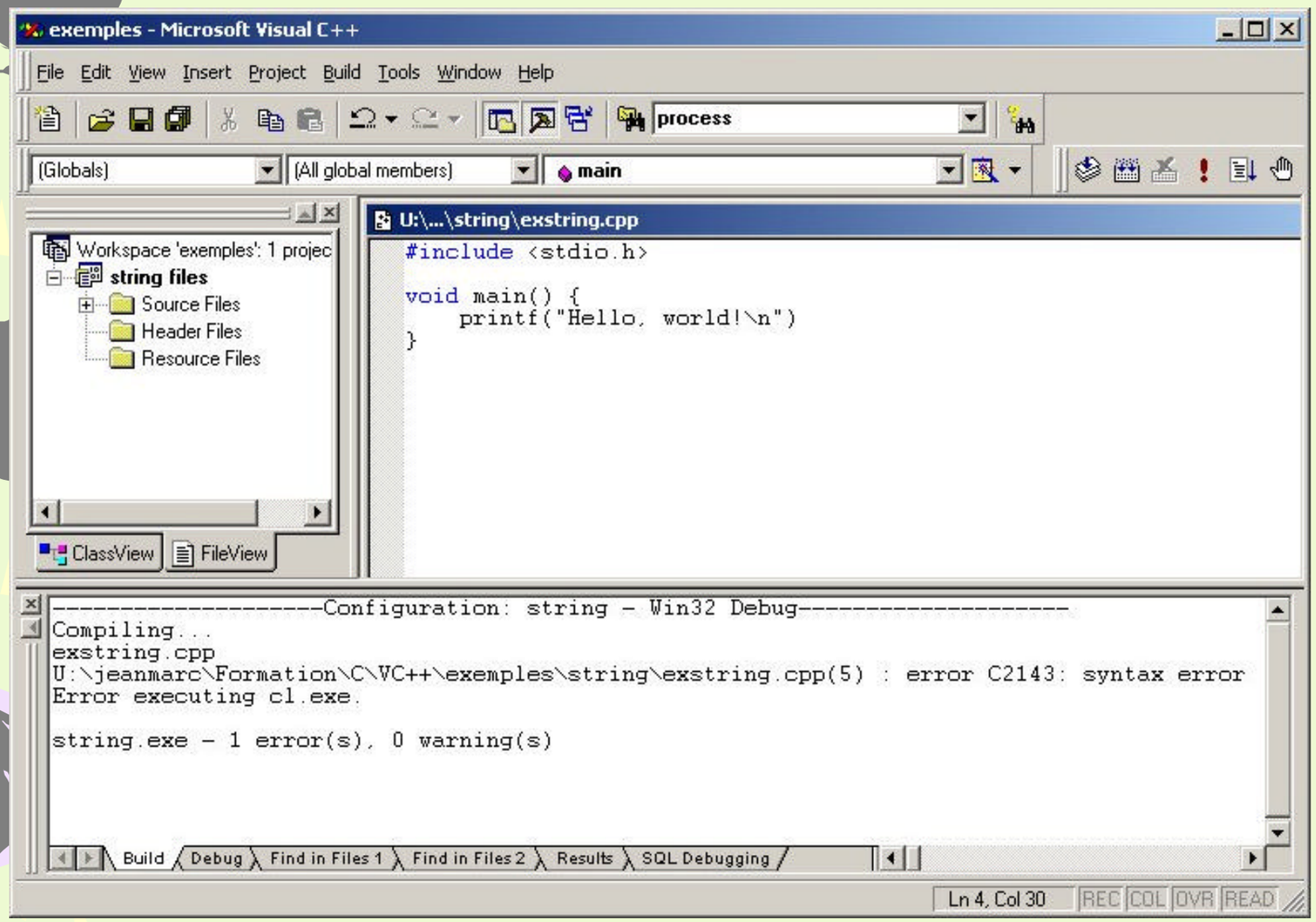
bloc

```
{
  printf(“Hello, world! \n”);
}
```

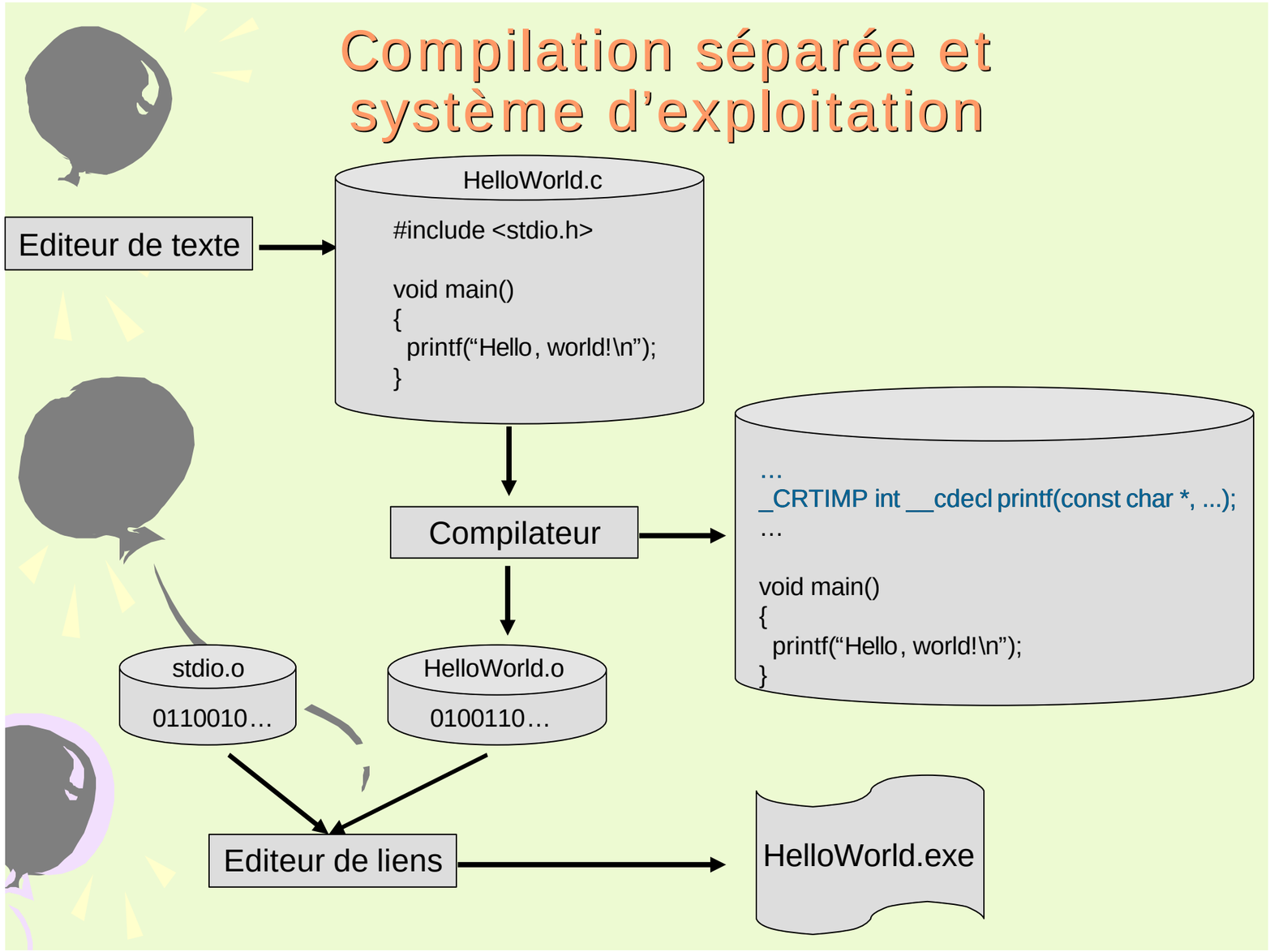
Un programme C est une collection de fonctions avec, comme point d’entrée, une fonction spécifique, la fonction main

Les E/S ne font pas partie du langage mais sont traitées au sein d’une librairie

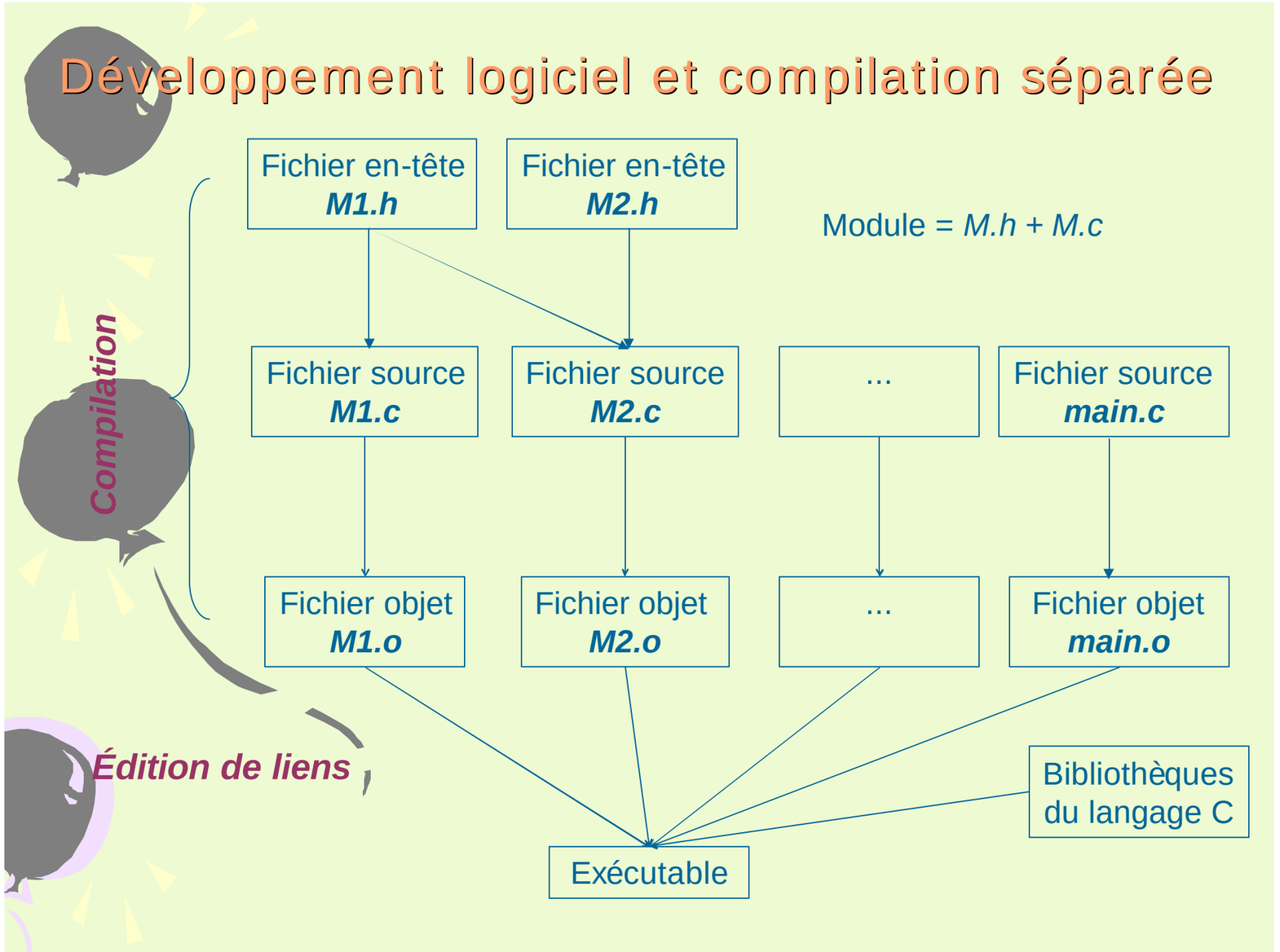
☹ syntax error



Compilation séparée et système d'exploitation

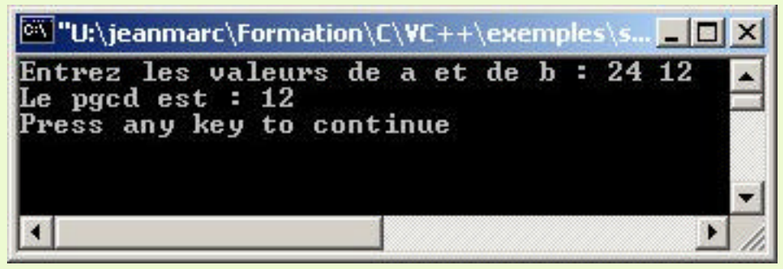


Développement logiciel et compilation séparée



Exemples en C : L'algorithme d'Euclide

```
#include <stdio.h>

int pgcd(int m, int n)
{
  int r;
  while ((r = m % n) != 0) {
    m = n;  n = r;
  }
  return (n);
}

void main()
{
    int a, b;
    printf("Entrez les valeurs de a et de b : ");
    scanf("%d %d", &a, &b);
    printf("Le pgcd est : %d\n", pgcd(a, b));
}
```



Idées fondamentales du langage C

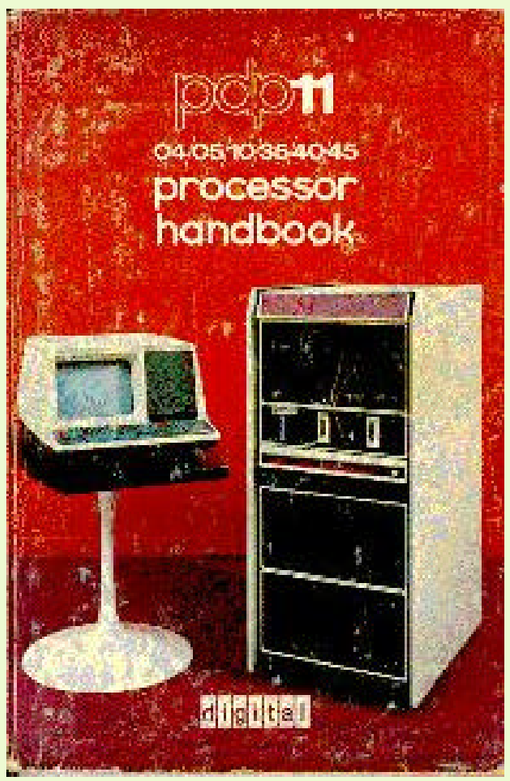
- A introduit un nouveau style de programmation basé sur la concision ((r = m % n) != 0)

- C'est un langage de programmation haut niveau : possibilité de définir des types abstraits de données (instruction typedef), compilation séparée, collections de fonctions dans un fichier source, ...

- Mais c'est également un langage de programmation de bas niveau de type assembleur : le langage de programmation se limite aux fonctionnalités efficacement traduites en langage machine et la mémoire est banalisée comme une suite d'octets (un caractère est codé par son code ASCII qui occupe un octet).

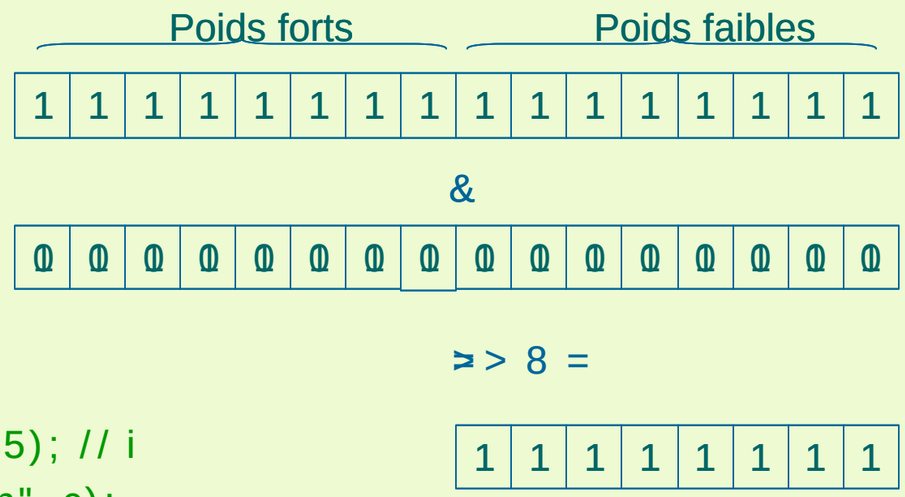
Le programme d'Euclide compilé sur le PDP-11

```
.globl _gcd
.text
_gcd:
   jsr  r5,rsave
L2:mov  4(r5),r1
   sxt  r0
   div  6(r5),r0
   mov  r1,-10(r5)
   jeq  L3
   mov  6(r5),4(r5)
   mov  -10(r5),6(r5)
   jbr  L2
L3:mov  6(r5),r0
   jbr  L1
L1:jmp  rretrn
```



La programmation système

```
#include <stdio.h>

void main() {
    unsigned short i = 65535;
    unsigned char c;
    c = (unsigned char) (i & 255); // i
    printf("poids faibles = %u\n", c);
    c = (unsigned char) ((i & 65280)>> 8);
    printf("poids forts = %u\n", c);
}

// RESULTAT : 255, 255
```



Les erreurs de programmation

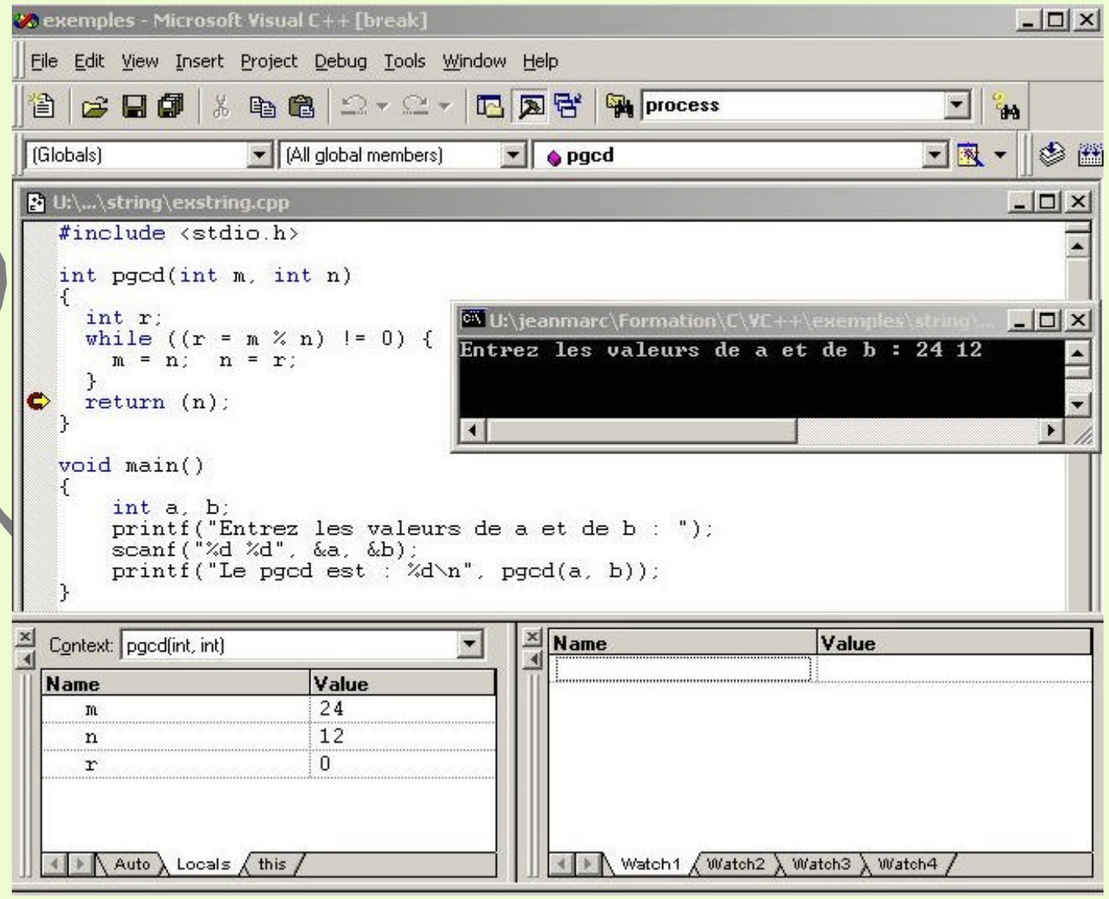
Il existe deux types d'erreurs :

- ☹ Les erreurs syntaxiques et sémantiques (faute de frappe, nom de variable utilisé et non déclaré, ponctuation manquante, …) détectées pendant la compilation :
  - Conséquence → le binaire n'est pas généré
  - Que faire ? → Corriger le fichier source et recompiler.

- ☹ Les erreurs logiques et dynamiques (le programme opérationnel mais il ne procure pas les résultats désirés) détectées au moment de l'exécution :
  - Conséquence → Résultats erronés, bogues, boucles infinies, …
  - Que faire ? → Revoir la conception, rechercher les erreurs, corriger le programme et recompiler.

Le debogueur

- Créer dans le programme des points d'arrêts, exécuter pas à pas le programme, afficher le contenu des variables ...



Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

Éléments du langage

```
#include <stdio.h>
#define PI 3.14159

float Surface;

// declaration exacte du main
void main(int argc, char *argv[])
{
    float Rayon;
    float Calcul(float r);

    printf("Rayon = ? ");
    scanf("%f", &Rayon);
    Surface = Calcul(Rayon);
    printf("Surface = %f\n", Surface);
}

/* calcul de la surface d'un cercle */
float Calcul(float r)
{
    return (PI*r*r);
}
```

Un programme : suite de commentaires, de déclarations et d'instructions espacées par des séparateurs, respectant une syntaxe précise.

Une instruction pré-processeur débute par un #

Cartouche de commentaires ou commentaire simple.

Pour être définie, une fonction doit être suivie d'un bloc d'instructions { …}

La visibilité d'une variable dépend de l'emplacement de sa déclaration.

Structure d'un programme C

- Un programme C :
  - Une ou plusieurs fonctions réparties dans un ou plusieurs fichiers. Une des fonction doit être le main.

- Une fonction C :
  - Une entête appelée prototype de fonction (type de retour, nom ou identificateur de la fonction, type des paramètres)
  - Instruction composée (bloc d'instructions) constituant le corps de la fonction

Les identificateurs

- Un identificateur est un nom de type, constante, variable ou de fonction.
- Il se compose de lettres ('a' - 'z' , 'A' - 'Z'), de chiffres ('1' - '9') ou '_' et ne peut pas être un mot-clé du langage. Il débute par une lettre ou '_'.
- On distingue les minuscules des majuscules (NB est différent de nb).
- Les 31 premières lettres sont significatives afin que deux identificateurs puissent être distincts.
- On différencie la *déclaration* (existence d'un nom, de sa qualité et de son type) de la *définition* (affectation d'une valeur pour une variable, bloc d'instructions associé à une fonction)

Exemples d'identificateurs

- Identificateurs valides :

  x somme _temperature
  y2 TABLE mon_fichier

- Identificateurs invalides :

  4ème // ne doit pas commencer par un chiffre
  taux change // surtout pas d'espace
  x#y // caractère # non autorisé
  sans-commande // pas plus que le caractère -

Les mots réservés

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

- On peut les classer par famille :

auto, const, enum, extern, register, static, struct, typedef, union, void
→ déclarations

char, double, float, int, long, short, signed, unsigned, sizeof,
→ types de base

break, case, continue, default, do, else, for, goto, if, return, switch, while
→ instructions

Instructions

- Une instruction est soit une instruction simple terminée par un ; soit une instruction composée délimitée par un bloc d'instructions { … }

- En combinant des identificateurs constantes, variables et fonctions avec des opérateurs arithmétiques, on construit des expressions arithmétiques qui quand elles se terminent par un ; forment une instruction simple.

- En combinant ces identificateurs avec des opérateurs de comparaison et logiques, on forment des expressions logiques qui, ajoutées à un mot-clé du langage, forment une instruction composée.

Exemples d’instructions

```
X = R * cos(angle*PI/180.0); // instruction d’affectation
Y = R * sin(angle*PI/180.0); // instruction d’affectation

/* instruction conditionnelle
    ou instruction d’alternative simple */
if ((lettre == 'a') || (lettre == 'e') || (lettre == 'i') ||
   (lettre == 'o') || (lettre == 'u') || (lettre == 'y'))
        printf(“voyelle \n“);
else
     printf("consonne \n");
```

Les caractères spéciaux

!	*	+	\	"	<
#	(	=	|	{	>
%	)	~	;	]	/
^	-	[	:	,	?
&	_	}	'	.	(espace)

- On peut les classer par famille :

  +, -, *, / et l'opérateur modulo % → opérateurs arithmétiques

  >, >=, <, <=, ==, !=, &&, || → opérateurs de comparaison et opérateurs logiques

  ++, –– → opérateurs d'incrémentation et de décrémentation

  &, |, ^, <<, >>, ~ → opérateurs de traitements de bits

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

Les types de base

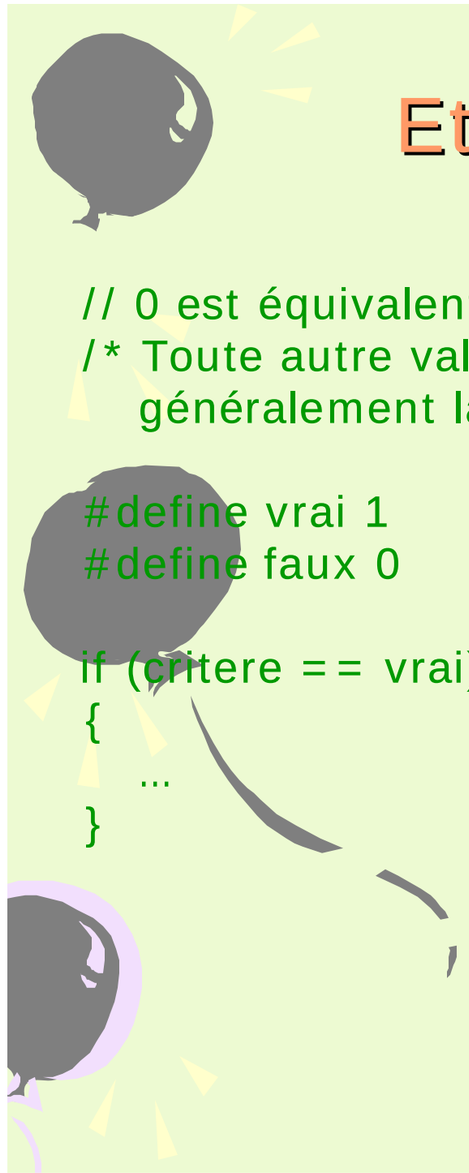
- C est un langage typé (pas fortement typé)

Type de base (machine de 32 bits)	Nb bits	Min	Max
char	8	-128	+127
unsigned char	8	0	+255
short int	16	-32768	+32767
unsigned short int	16	0	+65535
long int	32	-2147483648	+2147483647
unsigned long int	32	0	+4294967295
float	32	$\pm 3.40^{e-38}$	$\pm 3.40^{e+38}$
double	64	$\pm 1.78^{e-308}$	$\pm 1.78^{e+308}$
void	-	-	-

Et int sans short ou long ?

- sizeof(T) : nombre d'octets occupés par un objet de type T

Et le type booléen ?



```
// 0 est équivalent à faux
/* Toute autre valeur est équivalente à vrai,
   généralement la valeur 1 */

#define vrai 1
#define faux 0

if (critere == vrai) // Le langage C préfère la concision : if (critere)
{
   ...
}
```

Le type char

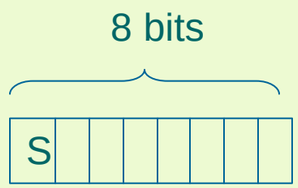
- Représenté par un code ASCII

- Dualité caractère/entier :

```
char c = 'a' ; // entre apostrophes
char c = 12 + 3;     // entier ou caractère ?
```

- Deux constantes utiles :

```
'\t'     // tabulation horizontale
'\n'     // passage à la ligne
```



mais aussi '\a' (bip sonore), '\f' (saut de page), ...

- Programmation système :

```
unsigned char Buffer[256];
```

→ exercice : Quel est le contenu de char c = 'a' ?

La table ASCII

code	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB
10	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US		!	"	#	$	%	&	‘
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	8	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	DEL		

Fonctions portant sur les caractères : <ctype.h>

isalnum	Vérifie si c est un caractère alphanumérique
isalpha	Vérifie si c est un caractère alphabétique
isascii	Vérifie si c est un caractère ascii
iscntrl	Vérifie si c est un caractère de contrôle \n, \t
isdigit	Vérifie si c est un caractère numérique
islower	Vérifie si c est un caractère minuscule

Autres fonctions portant sur les caractères : <ctype.h>

isprint	Vérifie si c est un caractère imprimable
ispunct	Vérifie si c est un caractère de ponctuation
isspace	Vérifie si c est un caractère d'espacement
isupper	Vérifie si c est un caractère majuscule
tolower	Convertit c en minuscule s'il est en majuscule
toupper	Convertit c en majuscule s'il est en minuscule

Les types entiers : short, int, long

- Signés par défaut, sinon unsigned est à préciser
- Entiers courts et longs : On peut écrire short int ou short, long int ou long
- L'implémentation de int dépend du compilateur (sizeof(int) ou <limits.h>)
- Les constantes littérales entières (dépend de la taille) :

```
100  // int
100u // unsigned int
100l // long
```

- Un nombre octal commence par 0 (zéro), un nombre héxadécimal par 0x, sinon décimal :

```
037 // nombre 31 en octal : 3*8 + 7
0x1F // nombre 31 en hexadécimal : 1*16 + 15
```

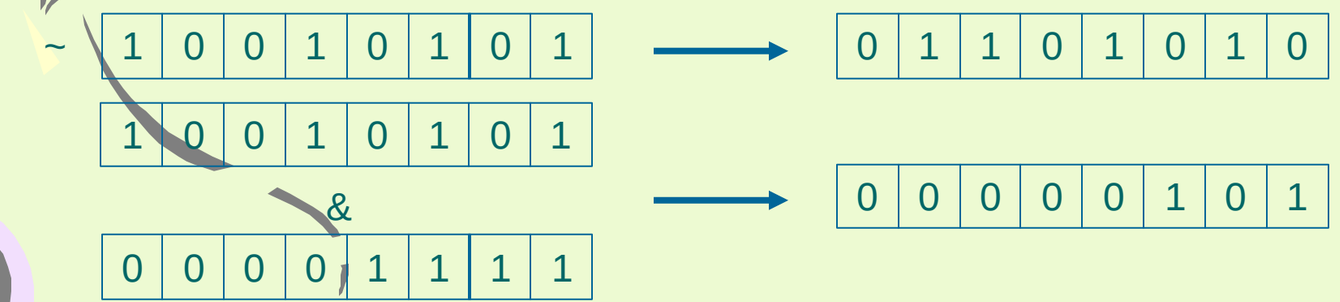
→ exercice : Ecrire l'entier 32567 en formats hexadécimal et octal

Le fichier <limits.h>

```
/* Number of bits in a char.  */
#define CHAR_BIT    8
/* No multibyte characters supported yet.  */
#define MB_LEN_MAX    1

/* Min and max values a signed char can hold.  */
#define SCHAR_MIN    (-128)
#define SCHAR_MAX    127

/* Max value an unsigned char can hold.  (Min is 0).  */
#define UCHAR_MAX    255U

/* Min and max values a char can hold.  */
#define CHAR_MIN SCHAR_MIN
#define CHAR_MAX SCHAR_MAX

/* Min and max values a signed short int can hold.  */
#define SHRT_MIN (-32768)
#define SHRT_MAX 32767

/* Max value an unsigned short int can hold. (Min is 0). */
#define USHRT_MAX 65535U

/* Min and max values a signed int can hold.  */
#define INT_MIN (-INT_MAX-1)
#define INT_MAX 2147483647

/* Max value an unsigned int can hold.  (Min is 0).  */
#define UINT_MAX 4294967295U

/* Min and max values a signed long int can hold.  */
#define LONG_MIN (-LONG_MAX-1)
#define LONG_MAX 2147483647

/* Max value an unsigned long int can hold. (Min is 0). */
#define ULONG_MAX 4294967295U
```

Les opérateurs de traitements des bits

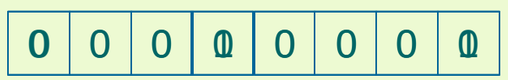
Opérateur	Signification
&	Et de bit à bit
|	OU inclusif de bit à bit
^	OU exclusif de bit à bit
<<	décalage à gauche
>>	décalage à droite
~	complément à un (masquage)



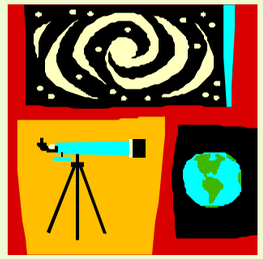
→ exercice : soit deux entiers i et j de valeur 0X0FF et 0XF0F, quel est le résultat héxadécimal de i & j, i | j, i ^ j, i << 2, j >> 2 ?

Exemple de traitement binaire

```
#include <stdio.h>

void main() {
   unsigned char c = 1;
   printf("premiere valeur : %d\n", c);
   c = c << 4;
   printf("deuxieme valeur : %d\n", c);
}

// RESULTAT : 1 , 16
```



Les nombres réels



- float et double (☹ double par défaut)

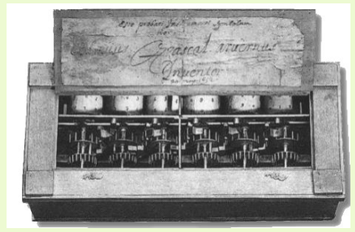
- Les constantes littérales réelles :

```
123.4  // double par défaut
1e-2  // double par défaut
123.4f  // float
123.4l  // double
```

Le fichier <float.h>

```
/*      Float definitions */

#define FLT_MANT_DIG        24
#define FLT_EPSILON         1.19209290e-07f
#define FLT_DIG             6
#define FLT_MIN_EXP         -125
#define FLT_MIN             1.17549435e-38f
#define FLT_MIN_10_EXP      -37
#define FLT_MAX_EXP         128
#define FLT_MAX             3.40282347e+38f
#define FLT_MAX_10_EXP      38

/*      Double definitions */

#define DBL_MANT_DIG        53
#define DBL_EPSILON         2.2204460492503131e-16
#define DBL_DIG             15
#define DBL_MIN_EXP         -1021
#define DBL_MIN             2.2250738585072014e-308
#define DBL_MIN_10_EXP      -307
#define DBL_MAX_EXP         1024
#define DBL_MAX             1.79769313486231570e+308
#define DBL_MAX_10_EXP      308
```

Les opérateurs arithmétiques



- Il en existe 5 : +, -, *, /, %

  a/b // le résultat dépend du type de a et du type de b
  10/4 // le résultat est 2
  10.0/4.0 // le résultat est 2.5
  18%4 // le reste de la division est 2

- Priorité entre les opérateurs : * et / sont plus prioritaires que + et –. Les opérateurs arithmétiques sont plus prioritaires que les opérateurs de comparaison, qui sont eux-mêmes plus prioritaires que les opérateurs logiques.

→exercice : Soit deux entiers de valeurs 10 et 3, que valent respectivement les valeurs de i+j, i-j, i*j, i/j et i%j ?

→exercice : Que vaut 2+3*5 ?

Les opérateurs arithmétiques

- ++ et –– sont des opérateurs d'incrémentation et de décrémentation

  i++ // est équivalent à i + 1

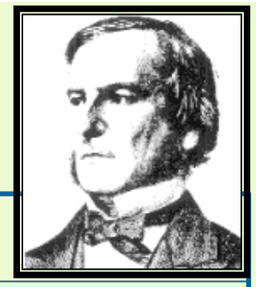
  j–– // est équivalent à j – 1

```
void main() {
    int x = 0;
    int y;
    y = ++x;
}
```

```
void main() {
    int x = 0;
    int y;
    y = x++;
}
```

→exercice : Que vaut les valeurs de y dans les deux programmes ?

Les opérateurs de comparaison et les opérateurs logiques



Expression	Valeur
(x == y)	1 si x est égal à y, 0 sinon
(x != y)	1 si x est différent de y, 0 sinon
(x < y)	1 si x est plus petit que y, 0 sinon
(x > y)	1 si x est plus grand que y, 0 sinon
(x <= y)	1 si x est plus petit ou égal à y, 0 sinon
(x >= y)	1 si x est plus grand ou égal à y, 0 sinon
x && y	1 si les deux conditions sont vraies, 0 sinon
x || y	1 si au moins une des deux conditions est vraie, 0 sinon

→exercice : Soit deux entiers i et j de valeurs 1 et 2, que valent respectivement les valeurs de i && j, i || j, !i, !j ?

Combinaisons entre opérateurs

expression	résultat	équivalence	lecture
opérateurs arithmétiques			
`i += 10`	110	`i = i + 10`	ajoute 10 à i
`i += j`	115	`i = i + j`	ajoute j à i
`i -= 5`	110	`i = i - 5`	retranche 5 à i
`i -= j`	105	`i = i - j`	retranche j à i
`i *= 10`	1050	`i = i * 10`	multiplie i par 10
`i *= j`	5250	`i = i * j`	multiplie i par j
`i /= 10`	525	`i = i / 10`	divise i par 10
`i /= j`	105	`i = i / j`	divise i par j
`i %= 10`	5	`i = i % 10`	i reçoit le reste de la division entière de i par 10
opérateurs de masquage			
`i &= 8`	0	`i = i & 8`	ET de i avec 8
`i |= 8`	8	`i = i | 8`	OU de i avec 8
`i ^= 4`	0x0C	`i = i ^ 4`	OU exclusif de i avec 4
opérateurs de décalage			
`i <<= 4`	0xC0	`i = i << 4`	décale i à gauche de 4 positions
`i >>= 4`	0x0C	`i = i >> 4`	décale i à droite de 4 positions

→ exercice : soit int i = 10, que valent respectivement les valeurs de i =/ 2, i += 3, i *= 2, i %= 3, i |= 10, i << 2, i &= 19, i ^ = 7 ?

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

Les déclarations de type

- Exemples :

Typedef Type AliasType;

typedef unsigned short Age;

typedef float Poids;

Age AgeMaxime = 19;

Poids PoidsMaxime = 65.0;

Les déclarations de constantes et de variables

- Tout identificateur doit être déclaré avant d'être utilisé :
  Type NomVariable , … ; // forme générale d'une déclaration
  double rayon;

- Une variable peut être initialisée lors de sa déclaration :
  double XCentre = 0.0, YCentre = 0.0;

- Une constante est précédée du mot-clé const :
  const float PI = 3.14159;

- Différence entre la phrase d'affectation et l'initialisation:
  NomVariable = expression;
  XCentre = YCentre = 1.0;

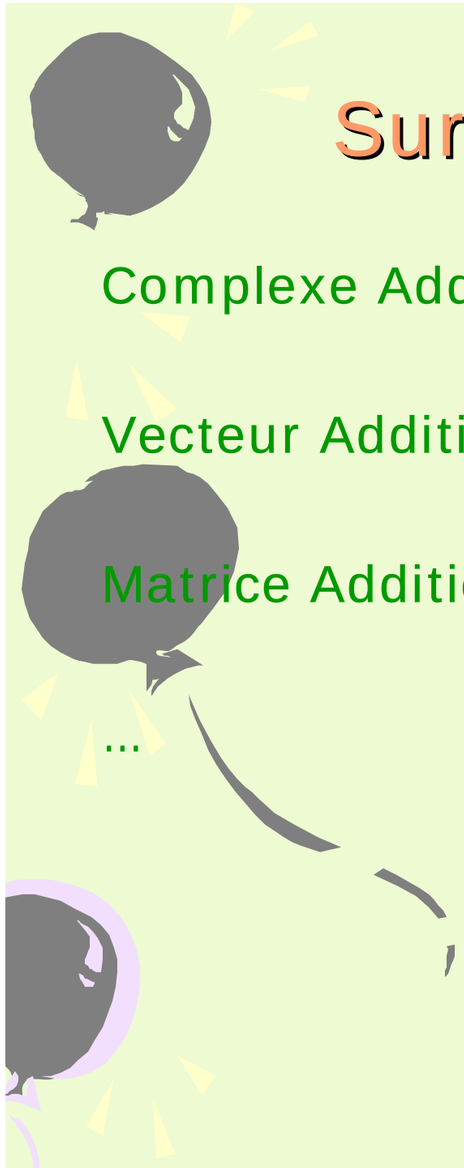
La phrase d'affectation

```
// ☹ Le langage C est un langage faiblement typé

NomVariable = expression;

int i = 6;
float f;
double d;

f = i; // autorisé mais avertissement
// expliciter la conversion de type (casting)
d = (double) f;
```

Les conversions entre types : le « casting »

```
char c = '1';
int i;
float f = 3.14159;

i = (int) c; // Résultat : 99
i = (int) f; // perte de la partie fractionnaire
```

Surcharge de fonctions



Complexe Addition(Complexe C1, Complexe C2);

Vecteur Addition(Vecteur V1, Vecteur V2);

Matrice Addition(Matrice M1, Matrice M2);

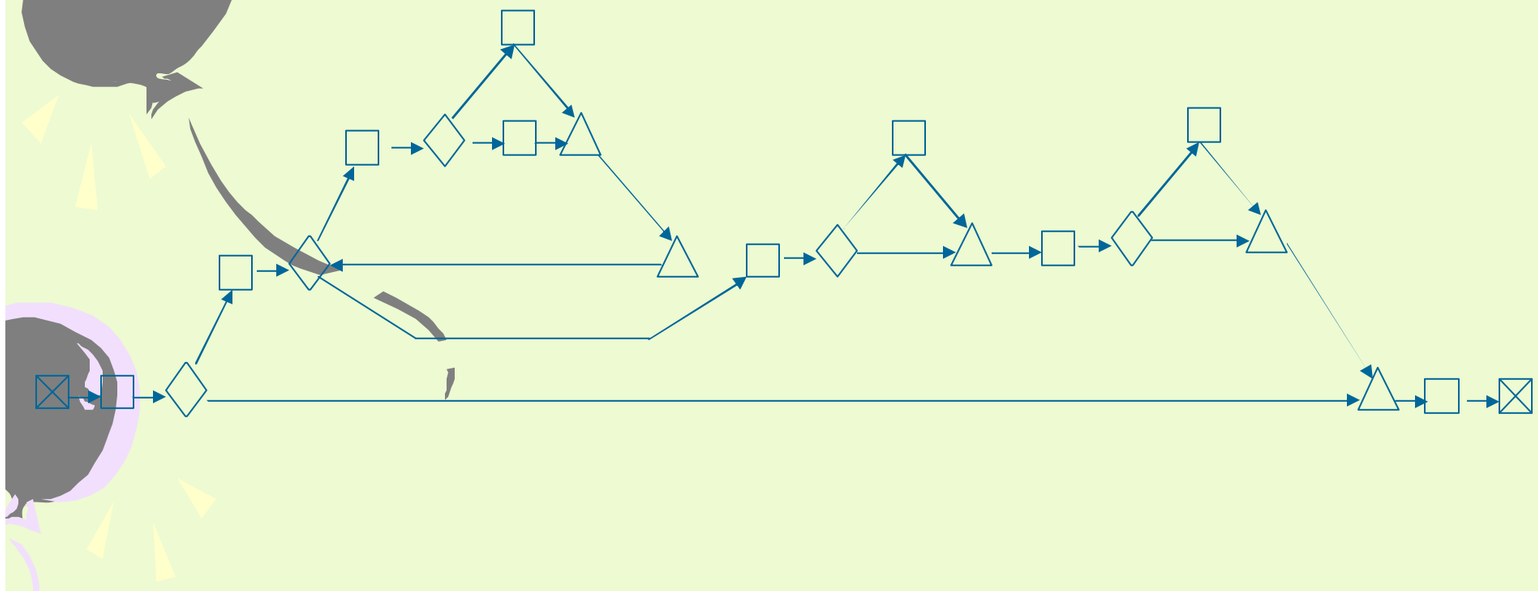
...

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

Programmation structurée

- ☺ Pas de goto
- 😐 Polémique 1 point d'entrée / 1 point de sortie avec :
  - break, continue, return, exit



Le « si / alors / sinon »

if (expression)
    instruction$_1$

if (expression)
    instruction$_1$
else
    instruction$_2$

- Exemples :

```
if (x == 0)
    y = 0;
```

```
if ((x >= 1) && (x <= 9))
        y = x;
else
        y = 0;
```

- Expression conditionnelle :

    (expr1) ? expr2 : expr3;

    x = (a > b) ? a : b; // x = max(a, b)

→ exercice : écrire l'expression conditionnelle de l'exemple

Autre exemple

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

#define MAXI 100

void main(void) {
    int x, prop, cpt;
    srand( (unsigned)time( NULL ) );
    x = (rand()%MAXI)+1;
    cpt = 0;
    do {
        cpt++;
        printf("Entrer un nombre entre 1 et %d :",MAXI);
        scanf("%d",&prop);
        if (prop<x)
                printf("\nTrop petit : ");
        else
                if (prop>x)
                        printf("\nTrop grand : ");
    }
    while (prop != x);
    printf("\nLe nombre etait %d, vous avez gagne en %d coups",x,cpt);
}
```

Le « selon / faire » (1/2)

switch (expression) {
case val_1 :
$instruction_1$
...
case val_n :
$instruction_n$
default :
$instruction_{default}$
}

- L'expression doit avoir une valeur entière
- default est optionnel

```
switch (c) {
    case 'a' : case 'e' : case 'i' :
    case 'o' : case ‘u' : case ‘y' :
            printf("voyelle\n");
            break;
    default :
            printf("consonne\n");
}
```

Le « selon / faire » (2/2)

- Utiliser break pour n'exécuter qu'une seule branche :

```
int x, y, z;

switch (operateur) {
    case '+' : z = x + y; break;
    case '-' : z = x - y; break;
    case '*' : z = x * y; break;
    case '/' : z = x / y; break;
    default :
        printf("Erreur\n");
}
```

La boucle « tant que / faire »

- Exécute l'instruction tant que l'expression est différente de 0 :

while (expression)
instruction

- Exemples :

```
int i = 0;
while (palindrome[i] == palindrome[n-1-i]) {
        i++;
        if (i >= n/2)
                break;
}
if (i == n/2)
   printf("palindrome");
```

```
while (1) {
   // boucle infinie
}
```

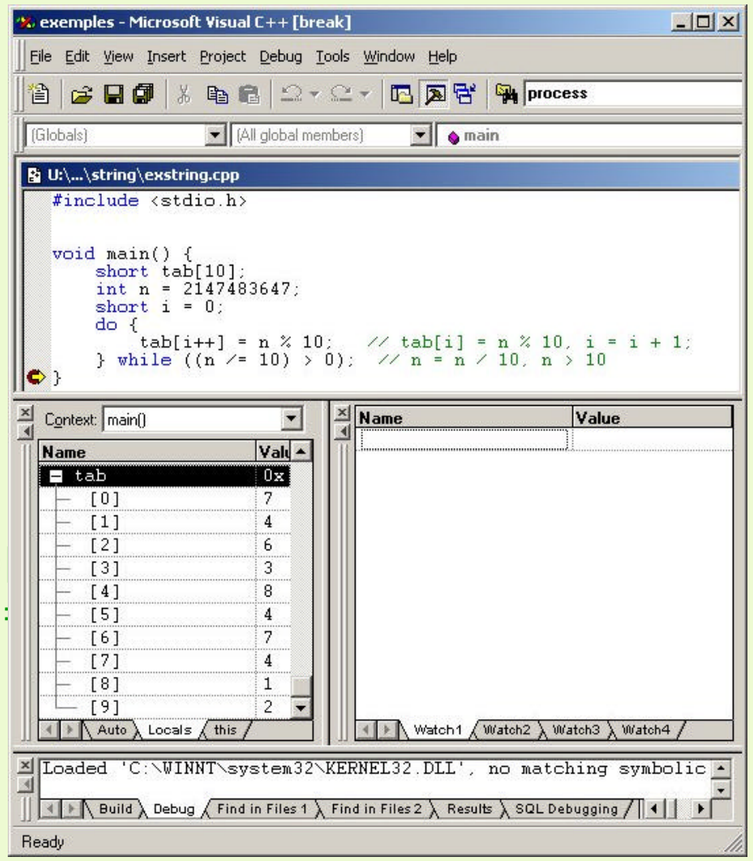
La boucle « faire / tant que »

- Comme la boucle « tant que / faire » mais la condition est évaluée à la fin :

do
instruction
while (expression)

- Exemple :

```
short tab[10];
int n = 2147483647;
short i = 0;
do {
   tab[i++] = n % 10;   // tab
} while ((n /= 10) > 0);  // n =
```



La boucle « pour »

- $expression_1$ est exécutée avant la première itération,
  $expression_2$ conditionne le passage dans la boucle,
  $expression_3$ est exécutée après chaque itération :

  for ($expression_1$; $expression_2$; $expression_3$)
  instruction

- Exemple :

```
char tab [256];
for (int i = 0; (c = getchar()) != '  '; i++) {
    if (c == ' \t ')
        continue;
    tab[i] = c;
}
```

NB : La variable i n'est plus visible après la boucle « pour »

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

Les énumérations

- Déclaration :

  enum NomEnum { ident$_1$ [= valeur$_1$], …, ident$_n$ [= valeur$_n$] };

- Exemple :

```
// Les plus grandes fêtes
enum villes {Rio =  1, Pampelune = 2, Munich = 3, Bayonne = 4};
villes Fetes;

if (Fetes == Bayonne) {
        printf (“Remise des cles : le premier mercredi du mois d’aout\n“);
        printf (“La ville est en fete du mercredi au dimanche“);
}

// Jours de la semaine
enum Jour {lundi, mardi, mercredi, jeudi, vendredi, samedi ,
dimanche}
```

Les tableaux (1/4)

- Un tableau est une séquence d'objets identiques en mémoire

- `int tab[10];` se traduit par la réservation de 10 entiers contigus en mémoire

- Mais le premier indice du tableau est 0. C'est également l'adresse du tableau : *tab, tab[0] signifient la même chose.

- L'initialisation d'un tableau se fait au moyen d'accolades :

  int tab[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

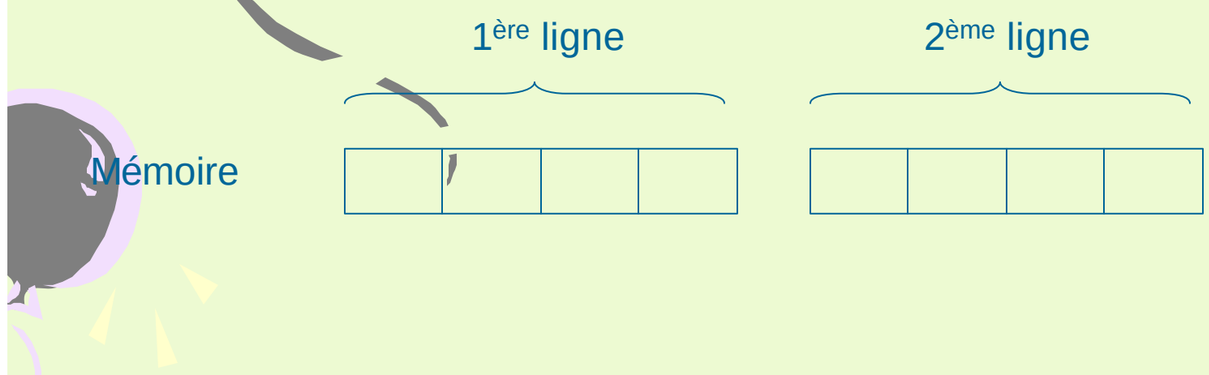
Les tableaux multi-dimensions (2/4)

- Les dimensions se lisent de droite à gauche :

```
// un tableau de 10 tableaux de trois tableaux de 2 entiers
 int  tab[10] [3] [2];
```

- Une matrice est un tableau à deux dimensions :

```
int mat[2] [4]
```



Exemple de tableaux (3/4)

```
#define size 4

int matrice [size] [size] =
 {{ 5, 20, 1, 16} , { 4, 7, 31, 23} , { 0, 8, 12, 3} , { 25, 2, 10, 8}};

// Transposé d'une matrice
for (int i = 0; i < size; i++)
   for (int j = 0; j < i; j++) {
      int permutation;

      permutation = matrice[i] [j];
      matrice[i] [j] = matrice[j] [i];
      matrice[j] [i] = permutation;
}
```

Exemple de tableaux (4/4)

```
#define size 4

int mat1 [size] [size] =
 {{ 5, 20, 1, 16} , { 4, 7, 31, 23} , { 0, 8, 12, 3} , { 25, 2, 10, 8}};
int mat2 [size] [size] =
 {{ 2, 17, 21, 15} , { 23, 35, 3, 2} , { 6, 26, 16, 4} , { 8, 17, 5, 39}};
int mat3 [size] [size];

// Multiplication de deux matrices
int somme;
for (int i = 0; i < size; i++)
   for (int j = 0; j < size; j++) {
      for (int k=0; k < size; k++)
            somme = somme + mat1[i] [k] * mat2 [k] [j];
      mat3 [i] [j] = somme;
      somme = 0;
   }
```

Les structures

- Association de plusieurs champs caractérisant une même entité

- Déclaration :
  struct Ident { Champ$_1$; … ; Champ$_n$};

```
struct Date  {
        int jour, mois, année;
} Naissance = { 18, 4, 1987 };
Date Rentree = { 18, 9, 2006 };
```

- On accède à un champ à l'aide du ' . ' :
  `RentreeProchaine.annee = 2007;`

- Quelle est la place mémoire occupée par une structure ?

Types abstraits de données

```
typedef struct pile {
    int SommetDePile;
    int Buffer [1000];
} Pile;
```

```
typedef struct point2D {
    double x;
    double y;
} Point2D;
```

```
typedef struct nombrecomplexe {
    double PartieReelle;
    double PartieImaginaire;
} NombreComplexe;
```

```
typedef struct cercle {
    Point2D Centre;
    double Rayon;
} Cercle;

Cercle CercleTrigonométrique =
    {{ 0, 0}, 1};
```

Types abstraits de données (suite)

```
typedef struct point2D {
    double x;
    double y;
} Point2D;

typedef struct triangle {
    Point2D p1, p2, p3;
} Triangle;

Point2D Centre(Triangle T) {
    Point2D P;
    P.x =
    (T.p1.x+T.p2.x+T.p3.x)/3;
    P.y =
    (T.p1.y+T.p2.y+T.p3.y)/3;
    return P;
}
```

```
void main() {
    Point2D p;
    Triangle t;

    printf("Premier point : \n");
    scanf("%lf %lf \n", &t.p1.x, &t.p1.y);
    printf("Deuxieme point : \n");
    scanf("%lf %lf \n", &t.p2.x, &t.p2.y);
    printf("Troisieme point : \n");
    scanf("%lf %lf \n", &t.p3.x, &t.p3.y);

    p = Centre(t);
    printf("Le centre est : \n");
    scanf("%lf %lf \n", p.x, p.y);
}
```

Les unions

- ☹ Allocation de plusieurs champs dans le même espace mémoire :

```
union PlusieursTypes {
        int ival;
        float fval;
        char *sval;
};

PlusieursTypes MaVariable.ival = 135;
```

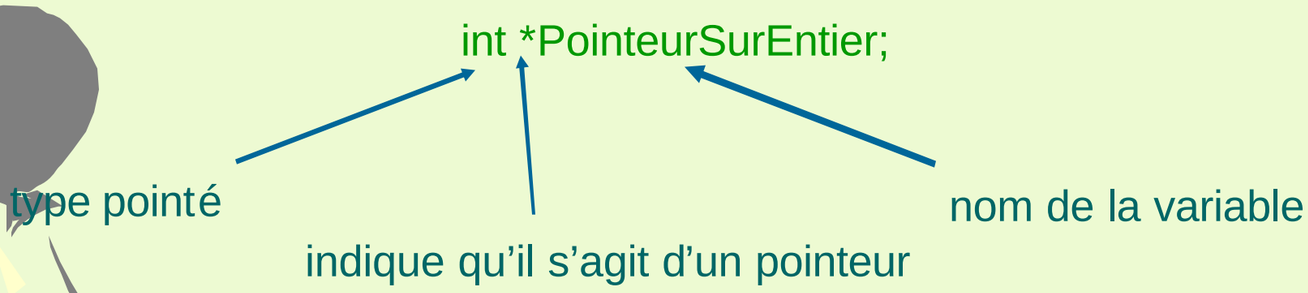
NB : Programmation potentiellement dangereuse

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

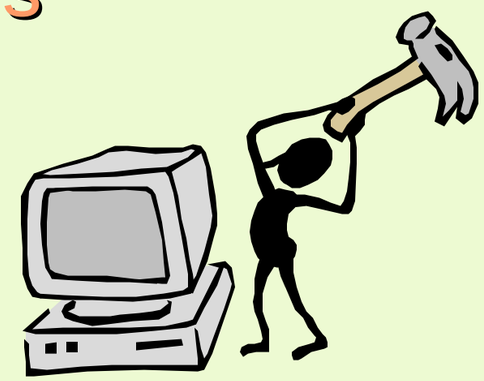
Définition de pointeurs

- Un pointeur est une variable qui contient **l'adresse** d'une variable ou d'une fonction. Comme toute variable, un pointeur doit être déclaré préalablement à son utilisation :



- Les programmeurs trouvent dans la notion de pointeur les techniques **d'adressage indirect**, ce qui donne une grande puissance au langage C.

Définition de pointeurs (suite)



- Avant d'utiliser un pointeur, il faut lui affecter une valeur, sinon 💣

```
int *PointeurSurEntier;
int i;

PointeurSurEntier = &i;
```

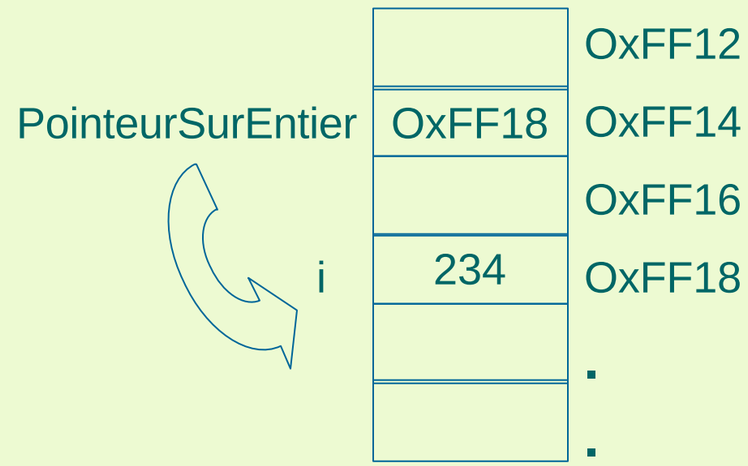
```
struct Liste {
        int objet;
        Liste *Suivant;
};

struct Liste *Premier = NULL;
```

- & fournit l'adresse d'une variable.
- NULL signifie pas d'adresse valide (adresse nulle).

Opérateurs unaires & et *

- Un pointeur est une adresse qui occupe 2 ou 4 octets selon les machines

```
int *PointeurSurEntier;
int i;

PointeurSurEntier = &i;

*PointeurSurEntier = 234;
```

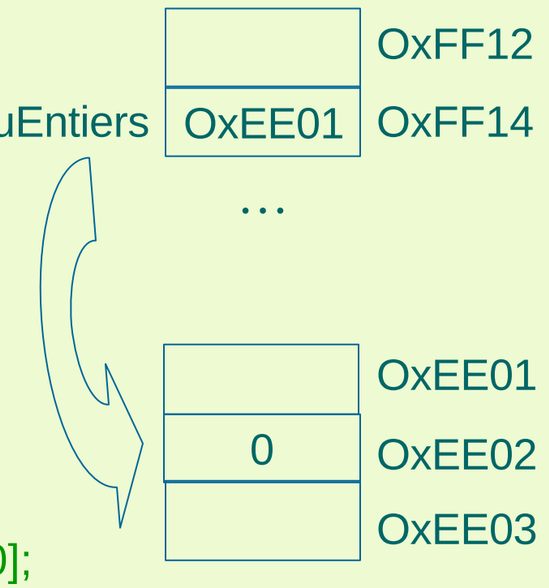


- Il ne faut pas confondre ces opérateurs avec les opérateurs multiplication et ET logique (ET de bit à bit).

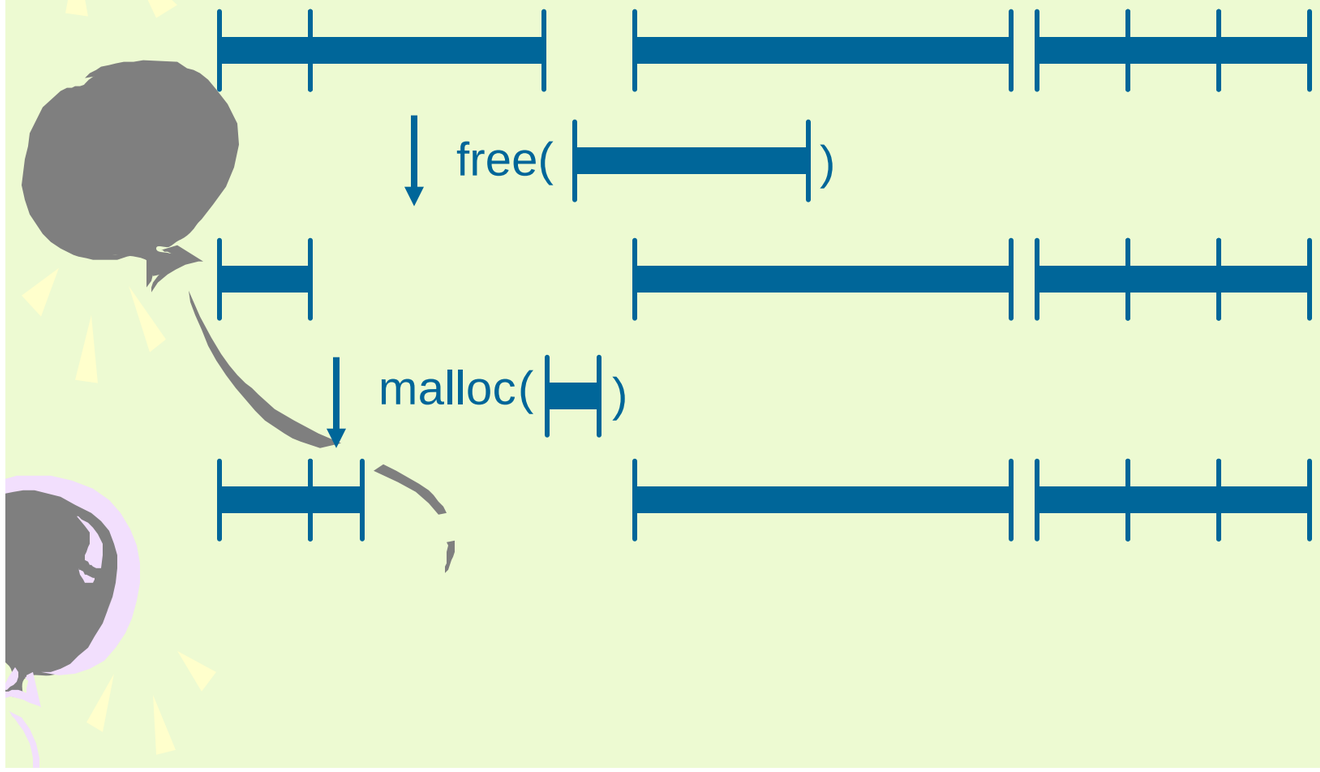
malloc() et free()

- L'utilisation la plus courante des pointeurs est l'allocation dynamique de la mémoire (librairie « stdlib.h »).

```
int *TableauEntiers;
int *NouveauTableauEntiers;

TableauEntiers = (int *) malloc(sizeof(int)*3);

TableauEntiers[1] = 0; // est autorisé
*(TableauEntiers++) = 0; // resultat identique

NouveauTableauEntiers = TableauEntiers;
NouveauTableauEntiers = &TableauEntiers[0];
```



Gestion de la mémoire dynamique



realloc()

- 💣 Attention aux conversions de type : ici (void *) pour les allocations de mémoire

```
int NombreDeSommets;
float *Vecteur;

// Premiere allocation
NombreDeSommets = 1;
Vecteur = (float *) malloc(sizeof(float));

float valeur;
while ( fscanf(fichier, "%f ", valeur) != 'EOF') {
   // Allocations suivantes
   NombreDeSommets++
    Vecteur = (float *) realloc(Vecteur , NombreDeSommets*sizeof(float));
}

// Utilisations
for (int i = 0; i <= NombreDeSommets; i++)
    Vecteur[i] = … ;
```

Opérateurs arithmétiques appliqués sur les pointeurs

- L'unité d'incrémentation ou de décrémentation d'un pointeur est toujours convertie en octets :

```
char *pChar;
int *pEntier;

pChar++; // progresse d'un octet
pEntier++; // progresse de deux octets
```

- TabEntiers + i est équivalent à TabEntiers[i]

- 💣 ++ est prioritaire sur * : *p++ est différent de (*p)++

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

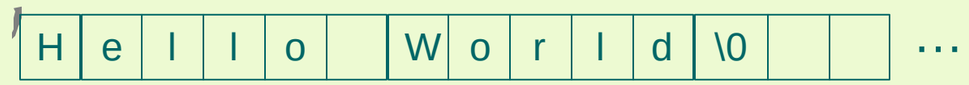
Les chaînes de caractère

- Une chaîne de caractères est un tableau de caractères alphanumériques (texte) se terminant par '\0' (longueur effective) :

```
char UneChaine[] = "Hello World";
char *LaMemeChaine = "Hello World";
char ToujoursLaMemeChaine[256] = "Hello World";
```

- La taille effective d'une chaîne de caractères est de n (strlen() + 1) octets :

UneChaine



Lecture d'une chaîne de caractères

```
char ch[256];
int i = 0;
while ((char c = getchar()) != '\n' ) {
    ch[i] = c;
    i++;
}
ch[i] = '\0';

// autre manière
scanf("%s", ch);
```

- Contrairement à printf(), le paramètre de la fonction scanf() est une adresse (scanf("%d", &i)). La fonction s'arrête lorsqu'un espace, une tabulation ou un retour à la ligne est rencontré.

- Autre fonction gets(ch) (idem getline()) qui lit jusqu'au retour à la ligne et le remplace directement par '\0' :

Manipulation de chaînes de caractères

- unsigned int strlen(const char *s) : retourne le nombre de caractères de la chaîne sans compter '\0'.

- int strcmp(const char *s1, cpnst char *s2) : compare les deux chaînes s1 et s2, retourne 0 si elles sont égales et un nombre différent sinon.

- char *strcpy(char *dest, char *src) : recopie src dans dest.

- char *strcat(char *dest, char *src) : concatène src à dest.

- ... (voir la librairie « string.h »

Exemple de chaîne de caractères

```
// programme qui enlève les 'e' dans un texte

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

void main() {
    char texte [81];
    char *adr = texte;
    printf("donner un texte de 80 lettres : \n");
    gets(texte);
    while (adr = strchr(adr,'e'))
        strcpy(adr, adr+1);
    printf("voici votre texte : %s", texte);
}

// commnt cla march ?
```

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

Les fonctions

- Une fonction est une unité de traitement (un ensemble fonctionnellement homogène)

- Les fonctions permettent de :
  - Partitionner les traitements en éléments plus petits (décomposition fonctionnelle),
  - Réutiliser des briques déjà existantes,
  - Gérer la communication entre les modules,
  - Simplifier à la fois la réalisation et la mise au point du programme (compilation séparée)

- Tout programme comporte une fonction particulière, le main.

Déclaration et définition d'une fonction

- La déclaration d'une fonction détermine comment effectuer son appel en décrivant son nom, le type des paramètres et le type de la valeur qu'elle renvoie.

- La définition d'une fonction définit la suite d'instructions constituant le corps de la fonction :

TypeRetour NomFonction (Type Param$_1$, … , Type Param$_n$) {
// Corps de la fonction
return resultat; // pas toujours présente
}

- Une procédure est une fonction qui ne retourne pas de résultat (mot-clé void)

Exemples de fonctions et de procédures

```
#include « stdio.h »

typedef struct point2D {
    double x;
    double y;
} Point2D;

double Distance (Point2D p1, Point2D p2);

void AfficheDistance (Point2D p1, Point2D p2) {
    printf(“Nombre elements : %f \n“, Distance(p1, p2));
}

double Distance (Point2D p1, Point2D p2) {
    return (sqrt( pow(p1.x-p2.x,2) + pow(p1.y-p2.y,2));
}
```

Localisation des fonctions

- Pour qu'une fonction puisse être appelée, il faut que, dans le même fichier, elle soit préalablement déclarée ou définie. Si elle n'est que préalablement déclarée, elle peut être définie en suivant dans le même fichier ou définie dans un autre fichier (précédée du mot-clé extern).

- Une fonction s'appelant elle-même est une fonction récursive :

```
int Factorielle (int n) {
   if (n==0)
      return(1);
   return(n*Factorielle (n-1));
}
```

Passage par valeur et par adresse des paramètres

- Par défaut, à l'exception des tableaux, les paramètres sont passés par valeur.

```
void Echange (int x, int y) {
   int z = x;
   x = y;
   y = z;
}

void Exemple(void) {
   int a = 1;
   int b = 2;
   Echange (a, b);
}
```

```
void Echange (int *x, int *y) {
   int z = *x;
   *x = *y;
   *y = z;
}

void Exemple(void) {
   int a = 1;
   int b = 2;
   Echange (&a, &b);
}
```

La fonction main()

- Elle contient le programme principal
- Elle est appelée en premier lors de l'exécution
- Sa signature est :

```
int main (int argc, char *argv[]) {
  ...
  return resultat;
}
```

- Elle peut retourner un résultat, notamment pour préciser le bon déroulement de l'exécution.

Récupération des paramètres d'exécution

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

void main(int argc, char *argv[]) {
    int unEntier;
    char *uneChaine;
    if (argc > 1) {
        unEntier = atoi(argv[1]);
        uneChaine = (char *) malloc(10*sizeof(char));
        if (uneChaine != NULL)
            sprintf(uneChaine, "%d", unEntier);
        printf("atoi : %d itoa : %s\n", unEntier, uneChaine);
    }
}
```

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

Affichages/Lectures simples

- Affichage d'un caractère sur la sortie standard :

```
int putchar(int c);
putchar('A');
```

- Affichage d'une chaîne de caractères sur la sortie standard :

```
int puts(char *s);
puts("Hello World");
```

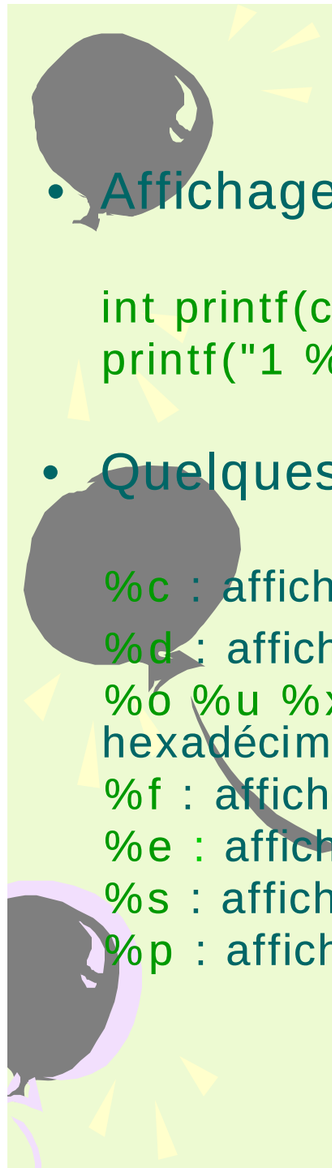
- Lecture d'un caractère sur l'entrée standard :

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

- Lecture d'une chaîne de caractères sur l'entrée standard :

```
char * gets(char *s);
char *chaine; (void) gets(chaine);
```

Affichage formaté



- Affichage formaté sur la sortie standard :

```
int printf(const char* format, ...);
printf("1 %s vaut %f %s\n", "euro", 6.55957, "francs");
```

- Quelques formats :

%c : affichage d'un caractère
%d : affichage d'un entier signé
%o %u %x : affichage d'un entier non signé en octal, décimal, hexadécimal
%f : affiche un double sous la forme [-]m.dddddd
%e : affiche un double sous la forme [-]m.ddddddde [-]xx
%s : affiche une chaîne de caractères
%p : affiche un pointeur

Lecture formatée

- Lecture formatée sur l'entrée standard :

```
int scanf(const char* format, ...);
scanf("Le quotient de  %d / %d vaut %f\n", &a, &b, &x);
```

- Quelques formats :

%c : Lecture d'un pointeur sur un caractère

%d : Lecture d'un pointeur sur un entier signé

%o %u %x : Lecture d'un pointeur sur entier en octal, décimal, hexadécimal

%f %e : Lecture d'un pointeur sur un nombre réel

%s : Lecture d'une chaîne de caractères

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

Ouverture de fichiers

- Les E/S standards :

  FILE *stdin : entrée standard
  FILE *stdout : sortie standard
  FILE *stderr : erreur standard

- Ouverture d'un fichier :

  FILE *fopen(const charpath, const char *mode)
  avec :
  - path = chemin et nom du fichier à ouvrir
  - mode = type d'accès :
    - r : ouverture en lecture seule
    - w : ouverture en écriture seule (destruction du fichier si il existe)
    - a : ouverture en écriture seule (copie à la fin du fichier)
    - r+ : ouverture en lecture/écriture
    - ...

E/S dans un fichier

- Entrées/sorties simples :

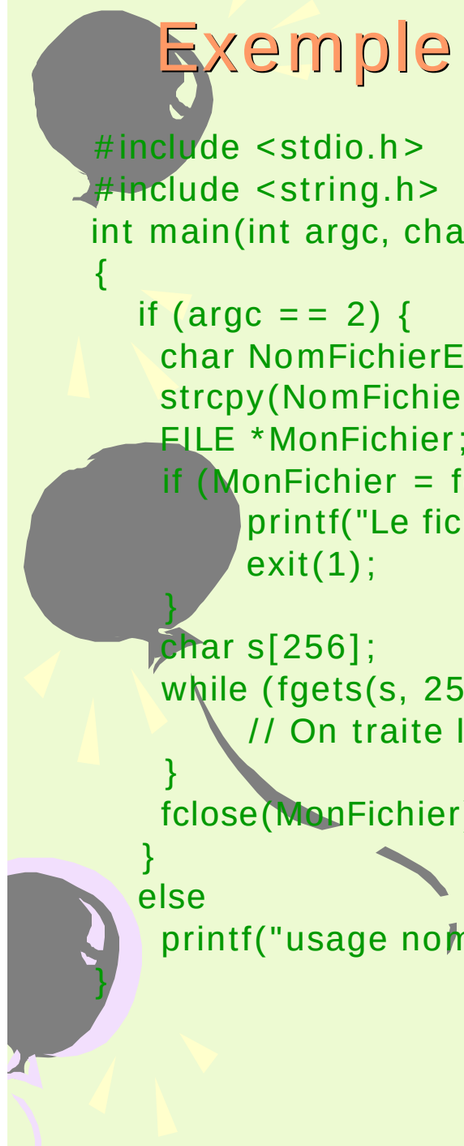
```
int fputc(int c, FILE *stream);
int fputs(char *s, FILE *stream);
int fgetc (FILE *stream);
char * fgets(char *s, int size, FILE *stream);
```

- Entrées/sorties formatées :

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

- Le caractère EOF indique la fin du fichier.

Exemple de manipulation de fichiers



```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    if (argc == 2) {
      char NomFichierEnLecture[256];
      strcpy(NomFichierEnLecture, argv[1]); // On recupere le nom du fichier
      FILE *MonFichier; // On ouvre le fichier en lecture seulement
      if (MonFichier = fopen(NomFichierEnLecture, "r") == NULL) {
            printf("Le fichier %s n'existe pas\n", NomFichierEnLecture);
            exit(1);
      }
      char s[256];
      while (fgets(s, 256, MonFichier) != NULL) {
            // On traite la chaine de caracteres s
      }
      fclose(MonFichier); // Penser a refermer le fichier
    }
    else
      printf("usage nomprogramme <nomfichier>\n");
}
```

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

Module = fichier en-tête + fichier corps

- Inclusion d'un fichier en-tête :

```
#include "NomFichier.h"
#include "CheminEtNomFichier.h"
```

- Se prémunir des inclusions multiples :

```
#ifndef NOMFICHIER
#define NOMFICHIER
#include "NomFichier.h"
#endif
```

- Que doit contenir un fichier en-tête ?
  - Les constantes et macros
  - Les types
  - Les fonctions
  - Éventuellement des variables (précédées du mot-clé extern)

Instructions pré-processeur

- cc -DTAILLE=256 ...

```
#ifndef TAILLE
#define TAILLE 256
#endif
```

- Utilisation de #define

```
#define NbElements(t) sizeof(t)/sizeof(t[0])
```

Structure d’un fichier en-tête

Structure d’un fichier en-tête .h en langage C :

- cartouche de commentaires au début (date, auteur, version, contenu du fichier),
- directives d'inclusion adressées au pré-processeur,
- déclarations de constantes,
- déclarations de types,
- « déclarations de variables »,
- déclarations de sous-programmes.

Structure d’un fichier corps

Structure d’un fichier corps .c en langage C :

- cartouche de commentaires au début (date, auteur, version, contenu du fichier),
- directives d'inclusion adressées au pré-processeur,
- déclarations des variables externes et des sous-programmes externes,
- déclarations des variables globales au fichier,
- définition des sous-programmes en C,
- main (paramètres de lancement : argc et argv).

Modes de déclaration des variables

- Une variable globale est statique par défaut (static). On peut également déclarer une variable locale comme étant static.

- Toute variable définie dans un autre fichier doit être déclarée comme étant extern

- Une variable locale est déclarée comme étant auto par défaut. On peut la déclarer register

→ Voir les règles de visibilité

Modes de déclaration des variables (suite)

```
#include <stdio.h>

int pgcd(int m, int n) {
    static NombreAppels = 0;
    NombreAppels ++;
    int r;
    while ((r = m % n) != 0) {
          m = n;  n = r;
    }
    return (n);
}

void main()
{
    int a, b;
    printf(“Entrez les valeurs de a et de b : “);
    scanf(“%d %d”, &a, &b);
    printf(“Le pgcd est : %d\n”, pgcd(a, b));
}
```

Options de compilation

- -DConstanteSymbolique=valeur
- -E : arrêt de la compilation après exécution du pré-processeur
- -ANSI : compatibilité avec le C ANSI
- -c nomfichier : permet de récupérer un fichier binaire *nomfic.o*
- -o nomexecutable : permet de renommer l'exécutable au lieu de *a.out*
- *...*

Plan

- Historique et principes fondamentaux
- Éléments du langage
- Les types de base et leurs opérateurs
- Les déclarations
- Les instructions de contrôle
- Les types composés
- Les pointeurs
- Les chaînes de caractères
- Les fonctions
- Les entrées-sorties
- Les fichiers
- Structure d’un fichier C
- Les problèmes fréquemment rencontrés

Problèmes fréquemment rencontrés

- Inclusions multiples de fichiers en-tête
- Division entière et division de nombre réel
- Confusion entre l'opérateur == et l'opérateur =
- Combinaisons entre opérateurs
- Instruction vide ; et boucle infinie
- Réaffectation de chaînes de caractères
- Débordement d'un tableau
- Passage de paramètre par valeur et par adresse
- ...

Mais aussi : non déterminisme

Questions