Le langage C

Xavier Jeannin
UREC / CNRS
Ce cours de 5 jours a été dispensé dans le cadre de la formation permanente du CNRS pour des ingénieurs chercheurs.
Xavier.Jeannin@urec.cnrs.fr

©Xavier Jeannin 2000-2005

Bibliographie

- **Le langage C, 2 éme édition**
  - B.W. Kenighan D. M. Ritchie, éd. Masson
- **Programmer en langage C**
  - C. Delannoy, éd. Eyrolles
- **La référence C norme ANSI/ISO**
  - C. Delannoy, éd. Eyrolles
- **Traps and Pits Fall**
  - Koenig Addison-Wesley
- **Source en C** : Noyau Linux
- **WWW** etc

Plan

- Généralité (Découverte du langage C à travers un exemple)
- Type de base
- Les opérateurs
- Entrées et sorties conversationnelles
- Les instructions de contrôle
- Programmation et modularité : les Fonctions
- Tableaux et pointeurs (références)
- Chaîne de caractères
- Les structures et les unions
- les fichiers
- Gestion dynamique de la mémoire
- Préprocesseur
- Possibilités proches de la machine
- Supplément sur les fonctions
- Make
- Deboggeur (gdb)

©Xavier Jeannin 2000-2005

Plan 1

- Généralité
  - Avant propos sur le langage C
    - Historique
    - Intérêt du langage C
  - Découverte du langage C à travers un exemple
    - Exemple de programme
    - Structure d'un programme C
    - Règle d'écriture
    - Création d'un programme C
- Type de base
  - Notion de type
  - Les types entiers
  - Les constantes entières
  - Les types flottants
  - Le type caractère
  - Les constantes
  - Les enums

©Xavier Jeannin 2000-2005

Plan 2

- Les opérateurs
  - Notion d'instructions et d'expressions
  - Les opérateurs arithmétiques
  - Les priorités et associations
  - Les conversions
  - Les opérateurs relationnels
  - Les opérateurs logiques
  - Les opérateurs d’affectation (Lvalue)
  - Les opérateurs d’incrémentation
  - Les opérateurs d ’affectation d ’élargie
  - L ’opérateur CAST
  - L ’opérateur conditionnel
  - L ’opérateur séquentiel
  - L ’opérateur SIZEOF
  - Tableau récapitulatif des priorités des opérateurs

©Xavier Jeannin 2000-2005

Plan 3

- Entrées et sorties conversationnelles
  - La fonction PRINTF et PUTCHAR
  - La fonction SCANF et GETCHAR
- Les instructions
  - Les contrôles : IF, SWITCH
  - Les boucles : FOR, DO … WHILE, WHILE
  - Les branchements conditionnels : BREAK, CONTINUE,GOTO
- Programmation et modularité : les Fonctions
  - Tout n’est que fonction
  - Exemple d ’une fonction
  - Arguments et valeur de retour
  - Déclaration et appel d’une fonction, fichier en-tête
  - Transmission des arguments par valeur
  - Variables globales
  - Variables locales automatiques et statiques
  - Fonctions récursives
  - Compilation séparée, portée des variables
  - tableau récapitulatif des classes de variables.

©Xavier Jeannin 2000-2005

Plan 4

- Tableaux et pointeurs (références)
  - Tableaux à un indice
  - Tableaux à plusieurs indices
  - Initialisation
  - les pointeurs
  - arithmétique des pointeurs
  - passage par adresse
  - Pointeur constant et nom de tableau
  - conversion de pointeur et pointeur vide
  - Les tableaux transmis en argument
  - Pointeur sur fonction
- Chaîne de caractères
  - initialisations
  - lire et écrire une chaîne
  - fonctions manipulant des chaînes

Plan 5

- Les structures et les unions
  - tableaux de structures
  - les structures récursives
  - unions
- les fichiers
  - accès séquentiel
  - accès direct
  - les fichiers textes
  - les entrées-sorties formatées et les fichiers textes
  - Les ouvertures de fichiers
  - les fichiers prédéfinis stdin etc...
- Gestion dynamique de la mémoire
  - MALLOC et FREE
  - CALLOC et REALLOC

Plan 6

- Préprocesseur
  - #INCLUDE
  - #DEFINE
  - macro
  - La compilation conditionnelle
  - autres directives
- Possibilités proches de la machine
  - Complément sur le type entier, unsigned int
  - Complément sur le type caractère , unsigned char
  - Opérateurs de bits
  - Les champs de bit
- Supplément sur les fonctions
  - les fonctions à arguments variables
  - Les arguments de la fonction main
- Make
- GDB

©Xavier Jeannin 2000-2005

Généralités

Historique

Langage C a été conçu dans les années 1970 par Dennie Ritchie aux Laboratoires Bell/ATT. Le but de ce langage était de développer une version portable du système d'exploitation UNIX d'où un langage de programmation structuré, mais très " près" de la machine.
Il provient de deux langages : BPCL développé en 1967 par Martin Richards et B développé en 1970 chez AT&T par Ken Thompson.

Il fut limité à l'usage interne de Bell jusqu'en 1978, date à laquelle Brian Kernighan et Dennie Ritchie publièrent les spécifications définitives du langage : **The C programming Language.**

Au milieu des années 1980, la popularité du langage était établie. De nombreux compilateurs apparaissent et comportent des incompatibilités. L'American National Standart Institute' (ANSI) formaient un groupe de travail pour normaliser le langage C qui aboutit en 1988 avec la parution du manuel :

The C programming Language -2`eme´edition

Bjarne Stroustrup crée le C++ dans les années 1990.

Intérêt du Langage C

- **Polyvalent** : Il permet le développement de systèmes d'exploitation, de programmes scientifiques et de programmes de gestion.
- **Langage structuré et évolué** qui permet néanmoins d'effectuer des opérations de bas niveau (assembleur d'Unix).
- **Prés de la machine**
- **Indépendant de la machine**
- **Portabilité** (en respectant la norme !) :
  - Recompilation des sources sur la nouvelle plate-forme.
- **Compact**
- **Rapide**
  - code très proche du langage machine.
- **Extensible**
  - nombreuses bibliothèques
- **Père syntaxique du C++, Java, PHP etc**

©Xavier Jeannin 2000-2005

Désavantages du Langage C

- **Trop permissif**
  - Programmation spaghetti et autres …
  - Warning interdit
- **Programme difficile à maintenir si**
  - La programmation est trop compacte
  - Les commentaires sont insuffisants
- **Portabilité**
  - Dès que l'on utilise des bibliothèques externes, la portabilité devient difficile.

©Xavier Jeannin 2000-2005

Découverte du langage C à travers un exemple

- 1 exemple de programme
- Structure d'un programme C
- Règles d'écriture
- Création d'un programme C

©Xavier Jeannin 2000-2005

```
#include <stdio.h>
#define NB_INT 4
/*  Ce programme calcule le produit de NB_INT
    nombres entiers introduits au clavier   */
int main(void)
   {int i, nb;
    float prod;

    nb = 0;                                /* Initialisation des variables */
    for (i = 0; i < NB_INT; i++)           /* Boucle principale */
        {printf("Entrez un nombre  :"); /* Lire la valeur du nombre suivant */
         scanf("%d", &nb);
         /* calculer le produit */
         if (i == 0)                        /* pas de multiplication au 1er passage  */
                prod = nb;
                 else
                prod = prod * nb;
        }
    printf("le produit est %f \n",prod);   /* Impression du résultat */
    printf("Fin du travail\n");
    return 0;
   }
```

Structure d'un programme C

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

déclaration de fonctions et de variables

main()
{
    déclaration de fonctions et de variables
    instructions
}
fct1()
{
    déclaration de fonctions et de variables
    instructions
}
fct2()
{……
}
```

Zone des déclarations

Zone des fonctions

©Xavier Jeannin 2000-2005

Règles d'écriture

Les identificateurs

- un identificateur est composé de lettres et de chiffres, le caractère souligné _ est considéré comme une lettre
- le premier caractère est forcément une lettre
- différenciation entre minuscule et majuscule : Count ≠ count
- Les 32 premiers caractères sont significatifs

Exemple :

taux taux_de_change Euro _diff _77

©Xavier Jeannin 2000-2005

Règles d'écriture

Les mots clefs :

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

Les séparateurs :

l'espace, un signe de ponctuation, fin de ligne

```
intx , y;       /* incorrecte */
int x,y,z;      /* correcte */
int x, y, z;  /* correcte et lisible */
```

©Xavier Jeannin 2000-2005

Règles d'écriture

Une instruction peut s'étendre sur un nombre de lignes quelconque mais attention à conserver une programmation claire !

```
#include <stdio.h>
#define NB_INT 4
main()    {int i, nb; float
prod
;    nb = 0;
for (i = 0; i < NB_INT; i++)
 { printf("Entrez le premier nombre  :"); scanf("%d", &nb1);
           if (i == 0)prod = nb;         else
                    prod = prod
* nb;}
printf("le produit est %f \n",prod);     printf("Fin du travail\n");return;}
```

©Xavier Jeannin 2000-2005

Règles d'écritures

Les commentaires :

Texte en les caractères /* et */. Pas d'imbrication de commentaire

```
/* commentaire */
/**********************
   commentaires
   sur plusieurs lignes
**********************/

float tva_rate;                 /*  tva depend de la classe du produit : luxe etc... */
for (i = debut; i < fin; i++)   /*  Boucle principale */
```

©Xavier Jeannin 2000-2005

Style d'écriture

- Usage commun : indentation →, accolade : {}

```
int main(int argc, char argv[][])
{   int i = 1;
    float taux;

    i ++;
    if(i > 2){
          taux = 19.8;
          i ++;
    }
    else{
          taux = 5.8;
          i --;
    }
    for (i = debut; i < fin; i++) {
          taux = taux - 5;
    }

}
```

©Xavier Jeannin 2000-2005

Style d'écriture

- Variante : indentation →, accolade : {}

```
int main(int argc, char argv[][])
{   int i = 1;
    float taux;

    i ++;
    if(i > 2)
          {taux = 19.8;
          i ++;
          }
    else
          {taux = 5.8;
          i --;
          }
    for (i = debut; i < fin; i++)
          {taux = taux - 5;
          }

}
```

Création d'un programme C

Édition d'un fichier ".c" : pgrm.c

Compilation :

- gcc pgrm.c
  - étape 1 le préprocesseur cpp :
    - traite les directives du source qui le concernent (#include #define) ==> pgrm.i
  - étape 2 : assembler pour créer : pgrm.obj
  - étape 3 : édition de liens Linker
    - création d'un exécutable nommé a.out
    - permet de lier à l'objet les fonctions des librairies externes (printf, scanf…)

Rendre exécutable a.out pour Linux : chmod u+x a.out

©Xavier Jeannin 2000-2005

Type de base

Notion de type

Type :

Le type sert à donner une signification à une information rangée en mémoire.

Quel que soit le type d'une information, nombre 65 ou la lettre A, tout est rangé en mémoire sous forme de bits (binary digit 0,1) regroupés en paquets de 8 nommés octets (byte) :

0100 0001 = A

Les types de base :

entier **int**

réel, nombre flottant **float**

caractère **char**

©Xavier Jeannin 2000-2005

Les types entiers

Les entiers : les entiers sont définis grâce au type INT

- **short int**
- **int**
- **long int**
- modificateur **unsigned**

Par exemple : les short et int peuvent correspondre à 16 bits (de -32 768 à 32 767) et les long correspondent à 32 bits (de -2 147 483 648 à 2 147 483 647).

Attention : Tous les entiers n'ont pas la même taille selon les machines !

©Xavier Jeannin 2000-2005

Représentation des entiers en mémoire

- **Représentation des entiers**

1	0000 0001	01
2	0000 0010	02
127	0111 1111	7F
0	0000 0000	00
-1	1111 1111	FF
-2	1111 1110	FE
-127	1000 0001	81
-128	1000 0000	80

- les négatifs sont représentés en "complément à deux", ce qui signifie que pour un nombre négatif, on prend la valeur absolue calculée en binaire puis on inverse tous les bits (0 => 1, 1 => 0), puis on ajoute 1.

©Xavier Jeannin 2000-2005

Constantes entières

Une constante entière peut s'écrire dans différents systèmes : **décimal, octal ou hexadécimal**.

Les constantes décimales :

+92 -46

notation octale :

précédé du chiffre 0 92 = 0134

notation hexadécimale :

précédé du chiffre 0x ou 0X 92 = 0x5C

Une constante entière est par défaut de type int.

Elle est de type long si elle est suffixée par les lettres l ou L, et non signée lorsqu'elle est suffixée par les lettres u ou U .

Base 10 : 12lu Base octale : 014u Base hexadécimale 0xCl

©Xavier Jeannin 2000-2005

Les types flottants

Les flottants :

Les nombres réels sont représentés de manière approchée grâce au type float.

- **float**
- **double**
- **long double**

Notation scientifique, flottant, virgule flottante : $\mathbf{M * B^e}$

exemple : 124.765 ----> $1.24756 * 10^2$

un réel peut être représenté par une mantisse M et un exposant e; dans notre exemple, la mantisse est 1.24756 et l'exposant est 2.

La base dépend de la machine et est en général 2 ou 16.

Attention La représentation des réels par les flottants n'est qu'une approximation.

©Xavier Jeannin 2000-2005

Type flottant

Précision de calcul

Précision :

La limitation du nombre de décimales impose une erreur dite de troncature.

Erreur de 10^{-6} pour le type float et 10^{-10} pour le type double.

Domaine couvert :

on est assuré de couvrir de 10-37 à 10+37 pour le type float.

Constante :

Notation décimale et exponentielle **:**

125.3 -0.089 -.3 .34

2.34E4 2.34+E4 23.4E+3

5.678E-29 56.78E-30 5678E-32

Par défaut, les constantes sont créées en type double; on peut forcer le type des constantes en faisant suivre la constante par f ou F pour float et l ou L pour long double

©Xavier Jeannin 2000-2005

Erreur numérique

Dépassement de capacité

Sur dépassement de capacité

- Int (entier) :
  - comportement de modulo
- Float (réel)
  - Message d'erreur + arrêt
  - convention IEEE : + Inf -Inf Nan (Not a number)

Sous dépassement de capacité

- Float (réel)
  - 0 zéro
  - Message d'erreur + arrêt

Division par zéro

- Message d'erreur + arrêt
- convention IEEE : +Inf -Inf Nan (Not a number)

©Xavier Jeannin 2000-2005

Type caractère, char

Char :

char mon_char = ‘A’ ;

Les caractères disponibles dépendent de l’environnement mais on dispose des caractères avec lesquels on écrit les programmes : les minuscules, les majuscules, les chiffres, les séparateurs, des signes de ponctuations.

Les caractères sont stockés sur un octet : ‘ A’ ‘ + ’ ‘ ? ’

On peut noter les caractère en code ASCII octal \ et hexadécimal \x

‘ A ’ ‘ \x41 ’ ‘ \101 ’

note : '\0' jusqu’à '\7' renvoie le code ASCII 0 à 7, '\7' est la cloche. Si \ est de tout autre caractère que ceux cités plus haut et dans le tableau qui suit, le \ est ignoré. Par exemple : \9 renvoie le code ASCII de ‘9’ 57

Attention : 'A' est différent de "A"

©Xavier Jeannin 2000-2005

Type caractère, char

Séquences d'échappement	**Notation en C**	**code ASCII**	**Abréviation**
sonnerie	\a	7	BEL
retour arrière	\b	8	DS
tabulation hor.	\t	9	HT
tabulation ver.	\v	11	VT
retour `a la ligne	\n	10	LF
nouvelle page	\f	12	FF
retour chariot	\r	13	CR
guillemets	\"	34	"
apostrophe	\’	39	’
point d ’interr.	\?	63	?
anti-slash	\\	92	\
caractère nul	\0	0	NUL

©Xavier Jeannin 2000-2005

Les constantes

- **Grâce au préprocesseur**

  #define **NB** 15

  int i = **NB**;

- **les constantes symboliques :** const int nb = 15;
  - const int constante_symbolique = 15;
  - int i = constante_symbolique;
  - ne peuvent faire partie d'une expression constante (les indices de tableaux)

- **les valeurs numériques**

  15 => int

  15L => long int

  15.0 => double (les vbles réelles sont par défaut doubles)

  15.0F => float

©Xavier Jeannin 2000-2005

Les constantes énumérées

•**enum** permet de déclarer des constantes énumérées

enum couleur {jaune, rouge, vert, bleu}

enum logique {vrai, faux}

couleur est un nouveau type énuméré.

enum couleur c1, c2; enum logique f;

les composants d'un enum sont des entiers ordinaires numérotés de 0 à (n - 1) sauf initialisation.

```
int i;
c1 = jaune;
i = vert;
c2 = c1 + 2    /* vert */
c1 = f;        /* bizarre mais accepté */
jaune = 2;     /* impossible jaune n'est pas une Lvalue */
```

©Xavier Jeannin 2000-2005

Les constantes énumérées

- **Initialisation :**

  enum couleur {jaune = 1, rouge = 5, vert , bleu =9}

vert vaut 6.

Les Enum peuvent être remplacés par des define :

```
enum bool {false, true};
#define false 0
#define true  1
```

Les opérateurs

Notion d'instructions et d'expressions

Dans les langages en général :

Notion d'**expression** : Les expressions sont formés entre autres à partir d'opérateurs et possèdent une valeur mais n'effectuent rien

Notion d'**instruction** : les instructions peuvent contenir des expressions et effectuent une action et ne possèdent pas de valeur.

Par exemple, l'instruction d'affectation :

```
a = b * c + d + 8;
```

En général, dans les langages, les notions d'expression et d'instruction sont distinctes.

©Xavier Jeannin 2000-2005

Les opérateurs en C

Dans le langage C :

k = i = 5; interprétée comme **k = (i = 5);**

i = 5 est une expression et renvoie 5

Il existe une **instruction** qui est une **expression terminée par un point virgule**

a = 5; est une instruction

Exemple : l'opérateur d'incrémentation :

```
a = i++;
```

Autre exemple :

```
int resultat_du_test;
resultat_du_test = ( i < 0 );
```

resultat_du_test vaut vrai (tout sauf zéro 0) ou faux (0 zéro) selon la valeur de i.

©Xavier Jeannin 2000-2005

Les opérateurs arithmétiques

- **Les opérateurs arithmétiques :**
  - L'addition : a + b
  - La multiplication : a * b
  - La division : a / b
  - Modulo : a % b, a et b doivent être entiers, modulo renvoie le reste de la division de a par b;

- **par exemple :**

a = c * b + a

©Xavier Jeannin 2000-2005

Les priorités

- **Les priorités**
  - comme dans l'algèbre traditionnelle * et / ont une priorité supérieure à + -
  - les parenthèses permettent d'outrepasser les règles de priorité.

3 * 4 + 5 <==> (3 * 4) + 5

3 * (4 + 5) <==> 3 * (4 + 5)

- Les priorités sont définies et sont utilisées pour fournir **un sens à une expression.**

©Xavier Jeannin 2000-2005

Association

- En cas de priorités identiques, les calculs s'effectuent de gauche à droite, "**associativité gauche droite**" sauf pour l'affectation et quelques autres opérateurs.

```
a = 1;
b = a << 2 << 3;
printf(" a = %d b = %d \n", a, b);
b = a << (2 << 3);
printf(" a = %d b = %d \n", a, b);
```

```
a = 1 b = 32

a = 1 b = 65536
```

- **Association droite de l'affectation :**

  a = b = c = d = 5 <==> (a = (b = (c = (d = 5))))

Ordre d'évaluation des opérandes

- Par ailleurs, **l'ordre d'évaluation des opérandes d'un opérateur n'est pas imposé par la norme.**

a = f() + g()

Attention ! aux effets de bord, dans notre exemple l'ordre d'évaluation de f() et g() n'est pas déterminé.

- En revanche **l'évaluation de gauche à droite** est **garantie** par la norme pour les opérateurs logiques "&&" "||" et pour l'opérateur virgule ",".

©Xavier Jeannin 2000-2005

Ordre d'évaluation des opérateurs commutatifs

- Attention ! L'ordre d'évaluation de 2 opérateurs "**commutatifs**" n'est pas défini et ne peut être forcé par les parenthèses.

a + b + c ===> a + b puis + c

ou a + b + c ===> b + c puis +a

Les conversions

Les conversions d'ajustement de type :

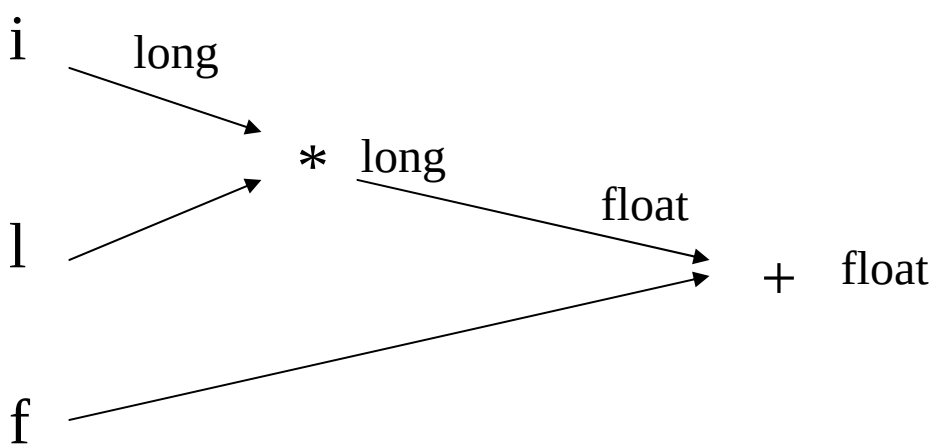
Les conversions se font selon une hiérarchie qui permet de ne pas dénaturer la valeur.

int > long > float > double > long double

Exemple :

i * l + f

i est un int, l est un long, f est un float



Les conversions

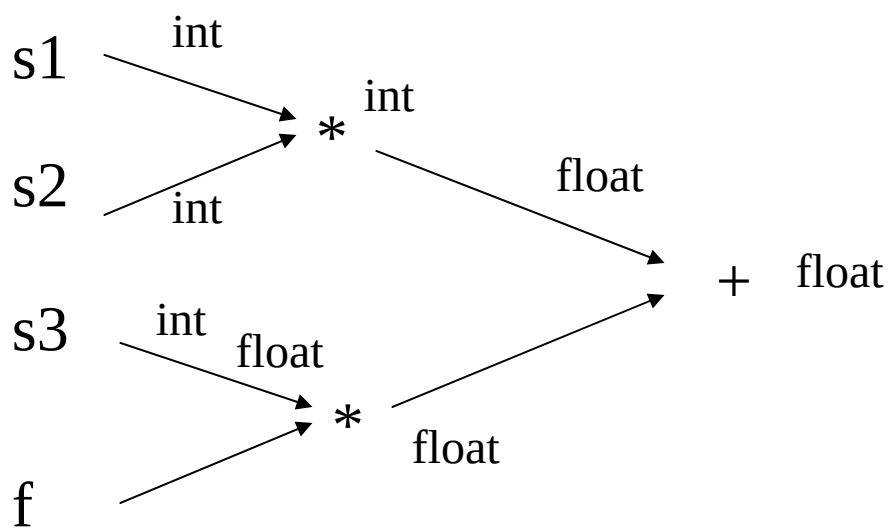
Les promotions numériques :

Les opérateurs arithmétiques ne sont pas définis pour les types short et char. Le langage C convertit automatiquement ces types en int.

Exemple :

s1 * s2 + s3 * f

s1, s2, s3 sont des short, f est un float.



Les conversions

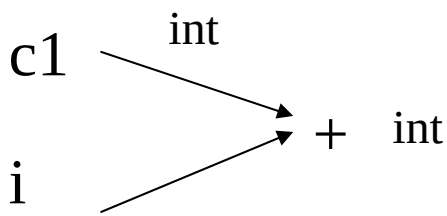
Les promotions numériques des caractères :

L'entier associé à un caractère donné n'est pas toujours le même.

Exemple :

c1 + i

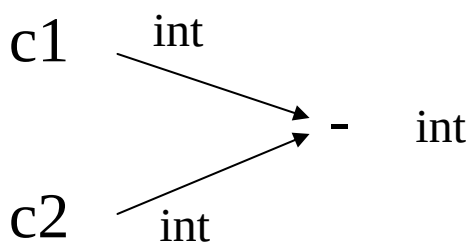
c1 est un char, i est un entier.



Exemple :

c1 - c2

c1 et c2 sont des char.



©Xavier Jeannin 2000-2005

Les promotions numériques cas particuliers

Les unsigned char renvoient un chiffre entre 0 à 255 alors que les char renvoient un nombre entre -128 et 127

Les vieux compilateurs pratiquent la promotion numérique du float vers le double compilateur

Les arguments d'appel aux fonctions :

- si la fonction a été déclarée avec ses arguments, la conversion se fait en fonction du type dans le prototype de la fonction.
- Si le type de l'argument n'a pas été déclaré :
  - suivre les mêmes règles que précédemment
  - promotion float en double

```
printf("%d,%f ", char, double);
```

©Xavier Jeannin 2000-2005

Les opérateurs relationnels

Le résultat d'une comparaison est un entier et non un booléen

- 0 si le résultat de la comparaison est faux.
- 1 si le résultat de la comparaison est vrai, tout autre nombre que 0 est considéré comme vrai

Opérateur	**signification**
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à
!=	différent de

©Xavier Jeannin 2000-2005

Les opérateurs logiques

- **(condition1) && (condition2**) condition1 et condition2.

Et	Faux	Vrai
Faux	Faux	Faux
Vrai	Faux	Vrai

©Xavier Jeannin 2000-2005

Les opérateurs logiques

- **(condition1) || (condition2)** condition1 ou condition2.

Ou	Faux	Vrai
Faux	Faux	Vrai
Vrai	Vrai	Vrai

- **!(condition)** not condition.

Not	
Faux	Vrai
Vrai	Faux

©Xavier Jeannin 2000-2005

Les opérateurs logiques

Les opérateurs logique acceptent n'importe quel opérande numérique.

- int a, b, n; a && b a || b !n
- if(n == 0) peut s'écrire if(!n)

Vrai et faux

- **0 correspond à faux**
- **toute valeur non nulle correspond à vrai**

©Xavier Jeannin 2000-2005

Les opérateurs logiques

L'évaluation des conditions se fait de gauche à droite et seulement si cela est nécessaire.

(a > b) && (c > d)

L'expression (c > d) n'est pas évaluée si l'expression (a > b) est fausse

Attention aux effets de cette règle, par exemple :

```
if ((i < NBFOIS) && (( n = get_int()) > 0))
    res = sqrt(n);
else
    { …
    }
```

©Xavier Jeannin 2000-2005

L'opérateur d'affectation

X = 1

A gauche de l'affectation, on peut mettre une expression; mais à droite de l'affectation, il faut placer un objet qui est une référence en mémoire, **une Lvalue**.

3 + X = 12 incorrect

Lvalue :

C'est la partie gauche d'une affectation qui possède une adresse mémoire :

un identificateur, membre d'un tableau etc ...

L'associativité de l'affectation est de droite à gauche :

X = Y = 5

Conversion

Si le résultat de l'évaluation de la partie droite de l'affectation est de type différent de celui de la lvalue alors il y a conversion imposée dans le type de la lvalue, d'où le risque de perte d'information.

Attention lors d'une conversion float/int, on perd la partie décimale !

©Xavier Jeannin 2000-2005

L'opérateur d'incrémentation

Les opérateurs d'incrémentation « ++ » et de décrémentation « -- » permettent de remplacer les deux instructions suivantes :

i = i + 1; <==> i++;

j = j - 1; <==> j--;

++ est :

- –un opérateur de **pré-incrémentation** si la lvalue est placée à droite.
- –un opérateur de **post-incrémentation** si la lvalue est placée à gauche.

-- est

- –un opérateur de **pré-décrémentation** si la lvalue est placée à droite.
- –un opérateur de **post-décrémentation** si la lvalue est placée à gauche.

Exemple : que vaut i et j ?

```
i = 6;
j = --i + 4;
j = i++ + 4;
```

©Xavier Jeannin 2000-2005

L'affectation élargie

X = X + 7 <==> X **+=** 7

encore plus fort

X = X * 7 <==> X ***=** 7

de manière générale

Lvalue **=** Lvalue **opérateur** expression <==> Lvalue **opérateur =** expression

Les opérateurs concernés sont :

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

©Xavier Jeannin 2000-2005

L'opérateur cast

On peut forcer la conversion d'une expression grâce à l'opérateur **cast**

(type) expression_a_ caster

Permissivité excessive du C.

Exemple :

```
double d;
int i,j;

i = 20; j = 7;
d = i / j;                  /*  d vaut 2          */
d = (double)(i/j);          /*  d vaut 2.0        */
d = (double)(i)/j;          /*  d vaut 2.857143   */
d =  20 / 7;                /*  d vaut 2 */
d =  20.0 / 7.0;            /*  d vaut 2.857143   */
```

©Xavier Jeannin 2000-2005

L'opérateur conditionnel

si a > b alors max = a;
sinon max = b;

peut s'écrire :

max = a > b ? a : b

Format de l'opérateur conditionnel

cond **?** valeur de retour si cond est vraie **:** valeur de retour si cond est fausse

Attention aux priorités !

max = a > b ? a : b;

fonctionne mais pour éviter les confusions :

max = (a > b ? a : b);

est plus claire.

©Xavier Jeannin 2000-2005

L'opérateur séquentiel

L'opérateur séquentiel est la virgule.

```
a * b  , c + d
```

a * b est évalué d'abord puis c + d. Cette expression renvoie la valeur de c + d.

if (i++, a > b) … <==> i++; if (a > b)...

for (i = 0, j = 10; . .. ; ...) <==> i = 0; f or (j = 10; . .. ; ...)

©Xavier Jeannin 2000-2005

L'opérateur sizeof

L'opérateur sizeof() renvoie la taille en octet de l'opérande ou du type.

int i;

float j;

sizeof(i); ==> 2

sizeof(j); ==> 4

On peut utiliser sizeof avec le nom des types :

sizeof(int); ==> 2

sizeof(float); ==> 4

gain de portabilité

pratique avec les structures

©Xavier Jeannin 2000-2005

Tableau récapitulatif des priorités

(ordre décroissant)

Catégorie	**Opérateurs**	**associativité**
Référence	( ) [ ] → .	→→→→
unaire	+ - ++ -- ! ~ * & (cast) sizeof	←←←←
Arithmétique	* / %	→→→→
Arithmétique	+ -	→→→→
Décalage	<< >>	→→→→
Relationnel	< <= > >=	→→→→
Relationnel	== !=	→→→→
Manipulation de bits	&	→→→→
Manipulation de bits	^	→→→→
Manipulation de bits	|	→→→→
Logique	&&	→→→→
Logique	||	→→→→
Conditionnel	? :	←←←←
Affectation	= += -= *= /= %= &= ^= |= <<= >>=	←←←←
séquentiel	,	→→→→

©Xavier Jeannin 2000-2005

Les entrées-sorties conversationnelles

Les entrées-sorties conversationnelles

La fonction printf

printf("premier argument %format_arg2 % format_ arg3", arg2, arg3);

Le premier argument de printf contient une chaîne de caractères qui est affichée à l'écran à l'exception des mots commençant par le caractère % .

Example :

```
int i = 6;
float f = 2.3456;
printf("Rang : %d , valeur %10.5f", i, f);
```

affiche à l'écran

Rang : 6, valeur 2.34560

©Xavier Jeannin 2000-2005

Les entrées-sorties conversationnelles

Le nombre d'arguments de printf est variable et déterminé par le premier argument (une chaîne de caractères)

Le premier argument composent ce que l'on nomme le "**format**" ; dans notre exemple "Rang : %d , valeur %10.5f"

%x est appelé le **code format**, il représente un code de conversion ; dans notre exemple %d et %10,5F

Exemple de code format :

%c %f %s %Lf %3d

Printf renvoie le nombre de caractères écrits, négatif si erreur.

©Xavier Jeannin 2000-2005

Les formats de printf

c : char caractère (convient aussi pour int ou short compte tenu des conversions systématiques).

d : int les entiers (convient aussi pour char ou short compte tenu des conversions systématiques).

u : unsigned int (convient aussi pour unsigned char ou unsigned short compte tenu des conversions systématiques).

ld :long int.

lu : unsigned long int.

f :double ou float écrit notation en décimale ex : 999.123456 ou 999.123400 (compte tenu des conversions systématiques float --> double).

e : double ou float écrit en notation exponentielle, mantisse 1à 10, 6 chiffres après la décimale ex : -x.xxxxxxe+yyy -x.xxxxxxe-yyy

s : chaîne de caractères, fournir l'adresse (pointeur) de la chaîne.

©Xavier Jeannin 2000-2005

Exemple de formats de printf

```
printf("%3d",i);              /* entier avec 3 caratères minimum */
i =2;                         ^^2
i = 4567;                     4567
i = -4567;                    -4567
printf("%f",f);               /* décimale avec 3 caractères minimum */
f =99.12345;                  99.12345
printf( "10%f",f);            /* décimale avec 10 caractères minimum */
f =99.;                       ^99.000000
f =-99.;                      -99.000000
printf( "20.9%f",f);          /* décimale avec 20 caractères minimum */
f =99.;                       ^^^^^^^99.000000000
f =99.;                       ^^^^^^-99.000000000
printf( "%e",f);              /* notation exponentielle */
f =99.1234;                   9.912340e+001
f =-99.1234;                  -9.912340e+001
```

©Xavier Jeannin 2000-2005

Précision de printf

Le nombre placé après le % dans le code format est appelé **gabarit**

```
printf(  "%10.3f",f);          /* décimale */
f =99.1236;                     ^^^^99.124
printf( "%12.3e",f);           /* exponentielle */
f =99.1263E8;                   ^^^^9.913e+08
```

calage à gauche signe moins -

```
printf(  "% -10.3f",f);
```

le caractère * dans le gabarit ou dans la précision signifie que la valeur effective est fournie dans la liste des arguments de printf.

```
printf(  "%8.*f",n,f);
n = 1    f = 99.12345   ^^^^99.1
```

©Xavier Jeannin 2000-2005

printf

printf(format, liste d’expression);

format

- constante chaîne de caractères entre "".
- une chaîne de caractères (pointeur sur une une chaîne de caractères).

liste d’expression

- suite d’expressions séparées par des virgules correspondant au format.

Printf renvoie le nb de caractères réellement affichés ou -1 en cas d’erreur.

©Xavier Jeannin 2000-2005

Erreur printf

Cas d'erreur :

- si le type de l'expression ne correspond pas au format
  - même taille mémoire ===> erreur d'interprétation de l'expression
  - taille mémoire différente ===> décalage et affichage de n'importe quoi

- si le nb de code format est différent de nb d'expressions
  - printf("%d", n, p); ===> affiche n
  - printf("%d %d ", n); ===> affiche n et n'importe quoi

©Xavier Jeannin 2000-2005

putchar

Putchar(c) est équivalent à printf("%c",c)

- pas d'analyse de format ==> plus rapide.
- putchar(i) n'est pas une vraie fonction mais une macro équivalente à putc(int i, stdout)
- putc(int i,stdout) est une macro équivalente à fputc(c, stdout)
- le fait que putchar soit une macro entraîne l'obligation de la directive #include<stdio.h>

©Xavier Jeannin 2000-2005

scanf

scanf(format, liste d'adresses);

scanf("%arg2%arg3", &arg2, &arg3);

Scanf permet de saisir des données entrées au clavier. Le format détermine le type des données saisies. Il est nécessaire de fournir l'adresse des variables qui contiendront les données.

Code format :

c	char	f ou e	décimale ou exponentielle
d	int	lf ou le	double
u	unsigned int	s	chaîne
hd	short int		
ld	long int		
lu	unsigned long		

©Xavier Jeannin 2000-2005

Le tampon de scanf

- Les informations entrées au clavier sont placées dans un **tampon**. Scanf explore ce tampon caractère par caractère au fur et à mesure des besoins. L'espace et la fin de ligne '\n' sont considérés comme des séparateurs (il peut y en avoir d'autre).
- Fonctionnement de scanf
  - pour les nombres : scanf avance jusqu'au premier caractère différent d'un séparateur, puis scanf prend en compte tous les caractères jusqu'à la rencontre d'un séparateur ou en fonction d'un gabarit ou d'un caractère invalide.
  - pour les nombres : pas de conversion implicite.
  - pour les caractères %c : scanf renvoie le prochain caractère dans le buffer séparateur ou non séparateur.
- Exemple (^ = espace, @ = fin de ligne) : scanf("%d%d",&n,&p);
  - 45^23@ ===> n = 45 p = 23
  - 45@@23@ ===> n = 45 p = 23
  - scanf("%d%c",&n,&c);
  - 45^a ===> n = 45 c = ' '

©Xavier Jeannin 2000-2005

Gabarit de scanf

- Saisie avec gabarit :
  - scanf avance jusqu'au premier caractère différent d'un séparateur, puis scanf prend en compte tous les caractères jusqu'à la rencontre d'un séparateur ou lorsque le nombre de caractères indiqués dans le gabarit est atteint.
  - Lecture d'un nombre d'au plus X chiffres.

- Exemple (^ = espace, @ = fin de ligne) : scanf("%3d%3d",&n,&p);
  - 45^23@ ===> n = 45 p = 23
  - 12345@ ===> n = 123 p =45

©Xavier Jeannin 2000-2005

L'espace et caractères invalides dans le code format

- **L'espace** dans le code format indique à scanf d'avancer jusqu'au prochain caractère qui n'est pas un séparateur (c'était déjà le cas pour les nombres).
- Exemple (^ = espace, @ = fin de ligne) :
  - scanf("%d^%c",&n,&c);
  - 45^^a ===> n = 45 c = 'a'

- **Caractère invalide**
  - scanf("%d^%c",&n,&c);
  - 45a@ ===> n = 45 c = 'a'

- La rencontre d'un caractère invalide (le point dans un entier) provoque l'arrêt du traitement du code format.

©Xavier Jeannin 2000-2005

Scanf (suite)

- Arrêt prématuré de scanf
  - Scanf renvoie le nombre d'arguments lus correctement.
  - Exemple (^ = espace, @ = fin de ligne) :
    - scanf("%d^%d^%c",&n, &p, &c);
    - 45a@ ===> n = 45 p = inchangé c = inchangé code retour 1
- Synchronisation
  - printf("entrez la valeur 1 \n"); scanf("%d",&val1);
  - printf("entrez la valeur 2 \n"); scanf("%d",&val2);
  - vous tapez : 1^2@
  - val1 = 1 val2 = 2 mais il n'a pas attendu pour l'introduction de val2
  - La fin de page '\n' déclenche le traitement par scanf des caractères lus
  - scanf lit des lignes jusqu'à pouvoir réaliser son traitement.

©Xavier Jeannin 2000-2005

Erreur scanf

Cas d'erreur :

- si le type de l'expression ne correspond pas au format
  - même taille mémoire ===> introduction d'une mauvaise valeur
  - taille lvalue < taille code format ===> écrasement en mémoire de la zone consécutive

- si le nb de codes formats est différent du nb d'expressions
  - scanf cherche à satisfaire le format
  - scanf("%d", &n, &p); ===> n est le seul entré
  - scanf("%d %d ", &n); ===> n est entré et un entier quelque part dans la mémoire est entré aussi.

Erreur scanf (suite)

Mauvaise réponse de l'utilisateur :

- trop peu d'information ou trop d'information : cf. plus haut.
- un caractère invalide n'est pas retiré du buffer en cas arrêt prématuré.
  - n = 21;
  - p = 22;
  - scanf("%d", &n); ===> on entre le caractère &, n vaut 21
  - scanf("%d", &p); ===> **on ne peut pas entrer l'entier p**, p vaut 22;

©Xavier Jeannin 2000-2005

getchar

getchar() est équivalent scanf("%c",c)

- pas d’analyse de format ==> plus rapide.
- getchar(i) n’est pas une vraie fonction mais une macro.
- getchar utilise le même tampon que scanf.

©Xavier Jeannin 2000-2005

Les instructions

Les instructions de contrôle

Type d'instructions :

- Simple : i = 5; return 5; goto Label;
- Structurée : choix, boucles.
- Bloc : { }

l'instruction **vide** est : ;

Notion de bloc :

- Un bloc est une suite d'instructions placées entre { et }
- Un bloc peut contenir un bloc
- Une instruction structurée peut contenir un bloc.

©Xavier Jeannin 2000-2005

IF

If (expression)
instruction1
else
instruction2

Exemple :

```
if  (i > 1)
    printf("succes\n");
else
    printf("erreur\n");

if  (++i > 1) printf("succes\n");
    correspond à
i = i + 1;
if  (i > 1) printf("succes\n");
```

If (expression)
instruction1

```
if  (i++ > 1) printf("succes\n");
    correspond à
i = i + 1;
if  (i  -  1> 1) printf("succes\n");
```

©Xavier Jeannin 2000-2005

IF

Exemple :

```
if  (i > 1)
    {printf("succes\n");
    …
     printf("fin\n");
    }
else
    {printf("echec\n");
    …
     printf("fin\n");
    }
```

```
if  ( (i++ > max)  && ( (c = getchar()) != '\n' ) )
    ne correspond pas à
i = i + 1;
c = getchar();
if  ((i > max)  && (  c != '\n' ) )
```

©Xavier Jeannin 2000-2005

IF imbriqués

Le else se rapporte toujours au dernier if rencontré auquel un else n'a pas encore été attribué.

Exemple :

```
if  ( i > 0)
    if( j > 0)  printf("j positif");
    else prinf("j negatif");

if  ( val < 1000) taux_remise = 5;
    else if  ( val < 10000) taux_remise = 10;
       else if  ( val < 50000) taux_remise = 15;
          else taux_remise = 20;
```

©Xavier Jeannin 2000-2005

switch

Exemple :

```
switch (n)
    {case 0 : printf("zero\n");
     case 1  : printf("un\n");
               break;
      case 2 : printf("deux\n");
               break;
     default : printf("defaut\n");
               break;
    }
    printf("fin");
```

si n == 0 le pgrm affiche
zero
un
fin

si n == 1 le pgrm affiche
un
fin

si n == 2 le pgrm affiche
deux
fin

si n == 23 le pgrm affiche
defaut
fin

©Xavier Jeannin 2000-2005

switch

```
switch (expression )
    {case constante_entière1 : [ liste instruction1 ]
     case constante_entière1 : [ liste instruction2 ]
     case constante_entière2 : [ liste instruction3 ]
            [ default :  liste instruction ]
    }
```

constante_entière1 : entier ou char 'a'

[ ] : facultatif

©Xavier Jeannin 2000-2005

Les boucles : do … while

Exemple :

```
do
    {printf ("entrez un entier > 0  (0 pour arreter)",\n)
     scanf("%d",&i);
     printf(" l'entier est : %d \n", i );
    }
while ( i != 0);
```

Syntaxe :

```
do
    instruction
while ( expression);
```

©Xavier Jeannin 2000-2005

Les boucles : while

Exemple :

```
i = 1;
while ( i != 0)
    {printf ("entrez un entier > 0  (0 pour arreter)",\n);
     scanf("%d",&i);
     printf(" l'entier est : %d \n", i );
    }
```

Syntaxe :

```
while ( expression)
    instruction
```

©Xavier Jeannin 2000-2005

Les boucles : for

Exemple :

```
for( i = 0; i < 5; i++)
    {printf ("boucle for : ");
     printf("  %d  fois \n", i + 1 );
    }
```

Syntaxe : **for ([exp1]; [exp2]; [exp3])**
instruction

est équivalent à

```
exp1 ;
while(exp2)
   {instruction
   exp3 ;
   }
```

Les boucles : for

remarque : si exp2 est absent du for, il est considéré comme vrai.

```
For ( ; ; )
        { /* boucle infini */
        }
```

on peut arrêter une boucle par l'instruction **break**.

©Xavier Jeannin 2000-2005

Break

```
for( i = 0; i < 5; i++)
    {printf ("boucle for : ")
    if ( i == 2 ) break;
     printf("  %d  fois \n", i +1 );
    }
```

donne comme résultat :

```
boucle for 1 fois
boucle for 2 fois
boucle for
```

break permet de sortir de la boucle la plus interne.

©Xavier Jeannin 2000-2005

continue

```
for( i = 0; i < 5; i++)
    {printf ("boucle for : ")
    if ( i == 2 ) continue;
     printf("  %d  fois \n", i +1 );
    }
```

donne comme résultat :

```
boucle for 1 fois
boucle for 2 fois
boucle forboucle for 4 fois
boucle for 5 fois
```

"continue" permet de passer prématurément au tour de boucle suivant.

"continue" s'applique à la boucle la plus interne.

©Xavier Jeannin 2000-2005

goto

```
for( i = 0; i < 5; i++)
            {printf ("boucle for : ")
            if ( i == 2 ) goto sortie;
             printf("  %d  fois \n", i +1 );
            }
sortie : printf("\nfin\n");
```

donne comme résultat :

```
boucle for 1 fois
boucle for 2 fois
boucle for
fin
```

goto permet de se brancher à un emplacement quelconque du programme dans la même fonction.

Goto est à éviter.

©Xavier Jeannin 2000-2005

Programmation modulaire et les fonctions

Programmation modulaire

On découpe les programmes en plusieurs parties :

- programme difficile à lire
- pas de séquence répétitives
- écriture modulaire, programmation structurée
- les modules sont décomposés eux-mêmes en sous-modules
- partage d'outils communs
- compilation séparée.

En général, on utilise deux types de module :

- Les fonctions prennent des arguments et ramènent un résultat scalaire simple comme en mathématique.
- Les procédures (sous-programmes) prennent des arguments, font une action et modifient des valeurs de variables.

©Xavier Jeannin 2000-2005

Fonctions du C

En C, il n'y a que des fonctions

mais elles peuvent :

– faire des actions

– ne pas renvoyer de valeur

– fournir des résultats complexes (structures, etc)

– modifier la valeur des arguments et la valeur des variables dites "globales".

Le langage C autorise la compilation séparée (programme source en plusieurs parties), ce qui facilite la création de grand programme.

©Xavier Jeannin 2000-2005

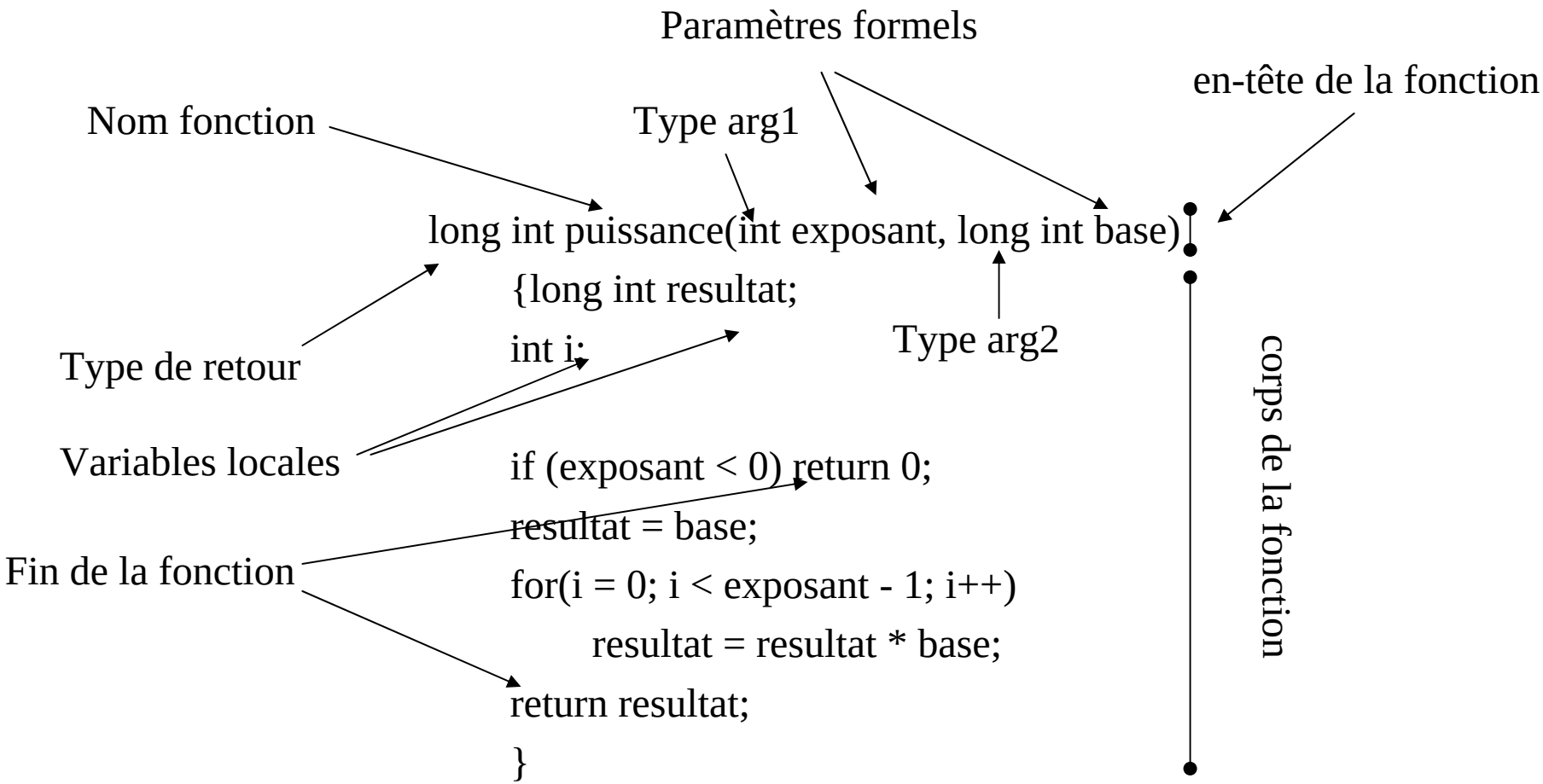
```
include <stdio.h>                       /* Exemple d'appel de fonction */
main()  /* pgrm principale */
        {long int puissance(int, long int);          /* declaration de la fonction */
        int  n, exp;
        long int r;

        printf("Entrez un nombre entier :");    /* Lire la valeur du nombre suivant */
        scanf("%d", &n);
        printf("Entrez l'exposant (positif) :");        /* lire l'exposant */
        scanf("%d", &exp);
        /* ici theoriquement des controles son necessaires */

        r = puissance(exp + 1, (long int)n);              /* calcul de la puissance */
        printf("le resultat est %ld \n",r);               /* Impression du resultat */
        }
/* puissance calcule "base" a la puissance "exposant" */
long int puissance(int exposant, long int base)
        {long int resultat;
        int i;

        if (exposant < 0) return 0;
        resultat = base;
        for(i = 0; i < exposant - 1; i++)
                resultat = resultat * base;
        return resultat;
        }
```

Exemple de fonction



©Xavier Jeannin 2000-2005

Déclaration et appel de fonction

```
main()  /* pgrm principale */
      {long int puissance(int, long int);          /* declaration de la fonction */
      int  n, exp;
      long int r;

      printf("Entrez un nombre entier :");    /* Lire la valeur du nombre suivant */
      scanf("%d", &n);
      printf("Entrez l'exposant (positif) :");       /* lire l'exposant */
      scanf("%d", &exp);
      /* ici theoriquement des controles sont necessaires */

      r = puissance(exp + 1, (long int)n);              /* calcul de la puissance */
      printf("le resultat est %ld \n",r);               /* Impression du resultat */
      }
```

Déclaration de la fonction

Paramètres effectifs

©Xavier Jeannin 2000-2005

return

- renvoie la valeur résultat
- si l'argument de return n'est pas du même type que celui déclaré dans l'entête le compilateur convertit.
- fonction sans valeur de retour :
  - void sans_de_retour(int n)
  - on peut utiliser return sans argument.
- fonction sans paramètre:
  - int sans_de_param(void)

©Xavier Jeannin 2000-2005

Vieux en-têtes

```
long int puissance (exposant, base)
int exposant;
long int base;
    {
    ……..
    }
```

©Xavier Jeannin 2000-2005

Les déclarations de fonctions

- La déclaration n'est nécessaire que lorsque la fonction appelante se trouve plus haut que l'appelée. La déclaration reste fortement conseillée.
- Déclaration partielle possible (interdit en C++):
  - long int puissance();
  - **à éviter.**
- Sans déclaration, une fonction renvoie int et ces arguments sont quelconques.
- Les noms d'arguments sont admis dans le prototype (la déclaration).
  - long int puissance(int exposant, long int base);
- **on place la déclaration soit**
  - **dans la fonction appelante**
  - **avant la définition de la première fonction.**

©Xavier Jeannin 2000-2005

Les déclarations de fonctions

- La déclaration permet au compilateur :
  - **si la définition se situe dans le même fichier source, le compilateur vérifie si les types d'argument formel est le même que celui du prototype.**
  - **Le contrôle sur le nombre et le type d'argument mentionnée dans les appels.**
  - **La mise en place de conversion. Si le compilateur ne connaît pas le type des arguments, il applique la conversion automatique.**
    - **Char et short --> int et float --> double**
  - **Seule une fonction déclarée peut recevoir un float, short ou un char**

©Xavier Jeannin 2000-2005

Les fichiers d'en-têtes

- Les fichiers .H sont inclus au source par la directive #include
- ils contiennent, entre autre, des déclarations.
  - **Contrôle sur le nombre et le type d'argument mentionné dans les appels.**
  - **Mise en place des conversions adéquates.**
- #include<math.h> pour utiliser la fonction sqrt()

©Xavier Jeannin 2000-2005

Transmission des arguments par valeur

- **Les arguments sont transmis par valeur**
- Ceci implique qu'une modification de la valeur d'un argument dans la fonction appelé n'influe pas sur la fonction appelante.
- Les arguments sont copiés dans la pile.
- Exemple :

```
void appelante(void)
    {int i = 10;                          I vaut 10
    change(i);
    printf("valeur de i =%d=", i);        I vaut 10
    }
void change(int i)
      { i = 20;                           I vaut 20
      }
```

Les arguments

- **Pour remédier au passage par valeur :**
  - passer l'adresse de l'argument : scanf("%d",&i);
  - utiliser des variables globales.

- **Remarque :**
  - l'ordre d'évaluation d'un argument n'est pas garanti

    f(i++, i);
  - l'argument effectif peut être une expression grâce au passage par valeur. Si le passage était par adresse ou référence, on ne pourrait transmettre qu'une "lvalue"

©Xavier Jeannin 2000-2005

Les variables globales

- Les variables globales sont déclarées à l'extérieur des fonctions
- exemple :

```
#include <stdio.h>
void change(void);
int i = 10;
void main(void)
      {printf("valeur de i =%d=\n", i);    /* i vaut 10*/
      change();
      printf("valeur de i =%d=\n", i);        /* i vaut 20*/
      }
void change(void)
    { i = 20;                    /* i vaut 20*/
    }
```

©Xavier Jeannin 2000-2005

Les variables globales

- **La portée :**
  - Les variables globales ne sont connues que dans la partie du programme source suivant leur déclaration.
  - On nomme cet espace la **portée** de la variable

- **la classe d'allocation :**
  - les variables globales font parties de la classe d'allocation statique.
  - La place mémoire est attribuée au début de l'exécution du programme dans le "tas".
  - Elles sont initialisées à zéro sauf indication contraire.

©Xavier Jeannin 2000-2005

Les variables locales

- **Les variables locales sont déclarées à l'intérieur d'une fonction ou d'un bloc.**
- **La portée :**
  - Les variables locales ne sont connues que dans la fonction ou le bloc où elles sont déclarées.
- **Les variables locales automatiques :**
  - Un espace mémoire est alloué à chaque entrée dans la fonction et libéré à chaque sortie dans la **pile**. Elles ne sont pas "rémanentes".
  - Pas d'initialisation automatique.
  - Initialisation par constante, constante symbolique, variable.
- **Les variables locales statiques : static** int i;
  - Un espace mémoire permanent est alloué. Sa valeur se conserve d'un appel à l'autre. Elles sont "rémanentes".
  - Initialisation à 0.

©Xavier Jeannin 2000-2005

Les variables locales

- exemple :

```
#include <stdio.h>
void f(void);
void main(void)
      {int i;
       for(i = 0; i < 5; i++)
                  f();
     }
void f(void)
     {static int n = 100;
     int i = 10;
     i++;
     n++;
     printf("passage numero %d, i = %d\n", n,i);
     }
```

- exécution :

```
passage numero 101, i = 11
passage numero 102 , i = 11
passage numero 103 , i = 11
passage numero 104 , i = 11
passage numero 105 , i = 11
```

©Xavier Jeannin 2000-2005

Les fonctions récursives

- exemple :

```
long fac(int n)
    {
    if ( n  > 1) return (fac(n - 1) * n)) ;
      else return(1);
    }
```

- À chaque appel il y a allocation de mémoire pour les variables locales et les arguments

- On ne commence à dépiler que lorsqu'on exécute un return, et on libère la mémoire

©Xavier Jeannin 2000-2005

Compilation séparée

- Source 1

```
int x;
void main(void)
    {…..

    }
void f11(void)
    {…….
    }
```

- Source 2

```
extern int x;
void f21(void)
    {x = 12;
    …..
    }
void f22(void)
    {…….
    }
```

- **Le mot clef extern ne réserve pas de mémoire mais précise le type.**

©Xavier Jeannin 2000-2005

Compilation séparée

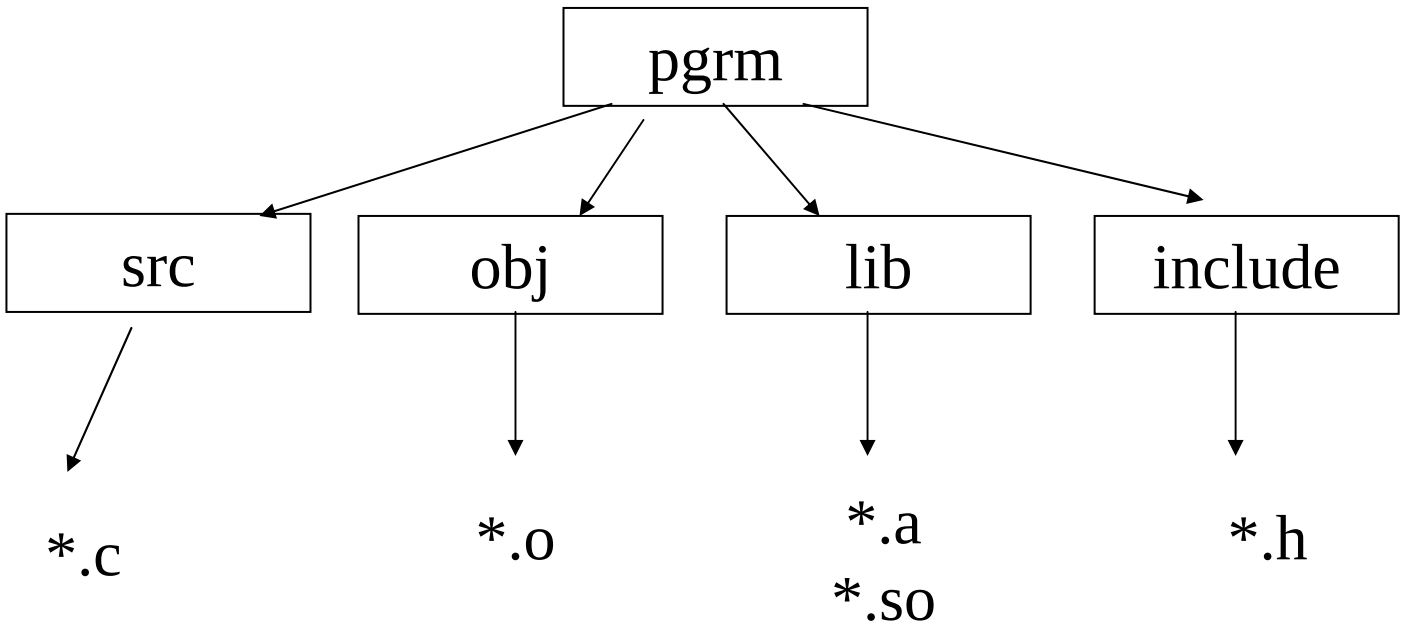
- Nous compilons séparément source1.c et source2.c et obtenons 2 objets source1.o et source2.o

- Comment faire correspondre au symbole x de source2 l'adresse de x du source1?
  - Dans l'objet de source1 on conserve le nom x et son adresse
  - Dans l'objet de source2 on conserve le nom x, on indique qu'il faut chercher à l'extérieur et qu'il faut fournir l'adresse
  - le linker retrouvera l'adresse de x dans source1pour le reporter en source2

©Xavier Jeannin 2000-2005

Compilation séparée

Organisation d'un programme :



©Xavier Jeannin 2000-2005

Compilation séparée

- **Vbles globales cachées**
  - exemple

```
static int a;
main()
        {…
        }
```

  - impossible de référencer "a" dans un autre fichier.

©Xavier Jeannin 2000-2005

Compilation séparée

- **Classes d'allocation**
  - classe statique, allouée une seule fois au moment de l'édition de lien.
    - les vbles globales et les vbles locales avec le mot "**static**".
  - Classe automatique allouée à chaque entrée dans la fonction et libérée a chaque sortie
    - les vbles locales sans le mot "**static**".
    - **Register** demande de mettre une vble de classe automatique dans un registre.

©Xavier Jeannin 2000-2005

Compilation séparée

- **Remarque :**
  il n'est pas nécessaire de mettre mot clef "extern" dans la déclaration de la fonction pour utiliser une fonction définie dans un fichier source différent (mais le mettre n'est pas une erreur).

- **Fonctions cachées :** **static** int foo(int a);
  - limite l'usage de cette fonction au seul fichier source

©Xavier Jeannin 2000-2005

Tableau récapitulatif

Type de variable	**Déclaration**	**Portée**	**Classe d'allocation**
globale	En dehors de toute fonction	Le fichier source suivant sa déclaration N'importe quel fichier source avec extern	**statique**
Globale cachée	En dehors de toute fonction avec l'attribut **static**	Uniquement le fichier source suivant sa déclaration	
Locale rémanente	Au début fonction avec l'attribut **static**	La fonction	
Locale à une fonction	Au début fonction	La fonction	**automatique**

©Xavier Jeannin 2000-2005

Les tableaux
et
les pointeurs

Définitions

Les tableaux :

• c'est une collection d'objets identiques (de même type) désignés par un identificateur unique (le pointeur sur le tableau).

int **tab**[2][5]

Les pointeurs :

• Il s'agit d'une variable contenant l'adresse d'un autre objet, il est caractérisé par le type de l'objet sur lequel il pointe (variable, fonction, premier élément d'un tableau)

int *point;

int entier; ==> &entier est l'adresse mémoire (le pointeur) de la variable entier.

©Xavier Jeannin 2000-2005

Les tableaux à un indice

- **Exemple d'utilisation du tableau :** stocker les notes d'une classe de 20 élèves.

  float notes[20]

- les indices commencent toujours à 0 et finissent donc au nombre d'éléments moins 1 dans notre exemple de 0 à 19.

  Attention il n'existe aucun contrôle sur les bornes.

  notes[100] = 0; est valide mais où avons-nous écrit ?

- **Utilisation d'un tableau**
  - notes[2] est une lvalue
  - Affectation : notes[2] = 18;
  - Affectation globale de tableau impossible. t1 = t2 non valide.
  - incrémentation : notes[2]++; --notes[16];

©Xavier Jeannin 2000-2005

Les indices

- Les indices peuvent être le résultat d'une expression arithmétique renvoyant un entier positif ou négatif.

  notes[(2* n) + 1]

- De même on peut utiliser des chars comme indice

  C1 C2 sont de type char : notes[c1 + 3], notes[c1]

- On peut utiliser des constantes comme indices mais pas les constantes symboliques (const)

```
#define NB 5
int tab[NB]; valide
int tab2[NB+2]; valide
const int nb = 50;
int tab[nb];  non valide
```

©Xavier Jeannin 2000-2005

Les tableaux à plusieurs indices

- Déclaration :

int notes[5][3]

notes est un tableau de 5 lignes et 3 colonnes

- Principe : les tableaux à plusieurs indices en C sont plutôt des tableaux de tableau de …

  notes[5][1] est un int
  notes[5] est un tableau de 3 int
  notes est un tableau dont chaque élément est un tableau de 3 int

- pas de limitation du nombre d'indices.

- Rangement en mémoire int t[5][3] ==>

  t[0][0], t[0][1], t[0][2], t[1][0], t[1][1], t[1][2], …. t[4][0], t[4][1], t[4][2]

donc

t[0][5] est en faite t[1][2]

©Xavier Jeannin 2000-2005

Initialisation des tableaux

- **Généralement :**
  - Comme les scalaires, les tableaux de classe statique (globale + locale statique) sont initialisés à zéro. Les tableaux de classe automatique (locale non statique) ne sont pas initialisés.
  - On ne peut initialiser les tableaux qu'avec des constantes non-symboliques
- **les tableaux à une dimension :**

  int t[5] = { 1, 2, 3, 4, 5}

  on peut ne mentionner que les premiers éléments : int t[5] = { 1, 2, 3}

  on peut omettre la dimension : int t[] = { 1, 2, 3, 4, 5}
- **les tableaux à plusieurs dimensions :**

```
int t[3][4] = { { 1, 2, 3, 4},
                { 11,12, 13, 14},
                { 21,22, 23, 24} }
int t[3][4] = {  1, 2, 3, 4, 11,12, 13, 14, 21,22, 23, 24}
```

  on peut ne mentionner que les premiers éléments.

©Xavier Jeannin 2000-2005

Les pointeurs

- **Les pointeurs :**
  - Il s'agit d'une variable contenant l'adresse d'un autre objet, le pointeur est aussi déterminé par le type sur lequel il pointe (variable, fonction, premier élément d'un tableau)

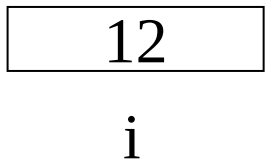
int *point;

- ***point** représente le contenu de l'entier pointé par point.
- Réciproquement, int entier; **&entier** est l'adresse mémoire (le pointeur) de la variable entier.

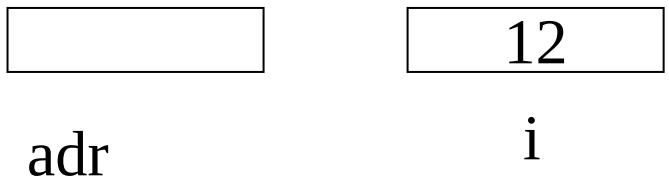
©Xavier Jeannin 2000-2005

Utilisation

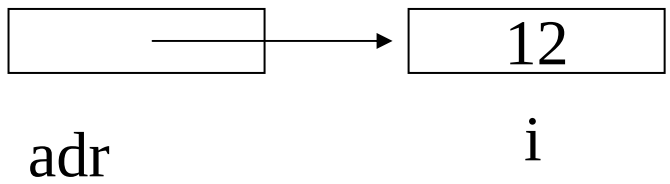
int i = 12;



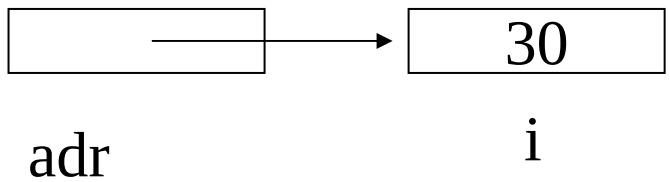
int *adr;



adr = &i;



*adr = 30;



©Xavier Jeannin 2000-2005

Exemples

```
int   n = 10, p = 10, *adr1, *adr2 ;

adr1 = &n;
adr2 = &p;
*adr1 = *adr2 + 2; ==>  ????

*adr1 += 3   ===> ???

(*adr2)++    ===> ????
```

Remarque

- int n, *adr1;

  adr1 et *adr1 sont des *lvalue*, c'est à dire que *adr1++;* ou **adr1= 3;* sont valides.

  Au contraire &adr1 et &n ne sont pas des *lvalue*, car ces adresses sont nécessairement fixes puisque calculées par le compilateur; elles ne peuvent pas être modifiées.

  (&adr1)++*;* ou &n= *3;* seront rejetées par le compilateur.

- **Réservation mémoire :**

  int *adr1; réserve de la mémoire pour le pointeur mais pas pour un entier.

©Xavier Jeannin 2000-2005

Arithmétique des pointeurs

• **int *adr, adr + 1 représente l'adresse de l'entier suivant l'adresse de l'entier contenue dans adr.**

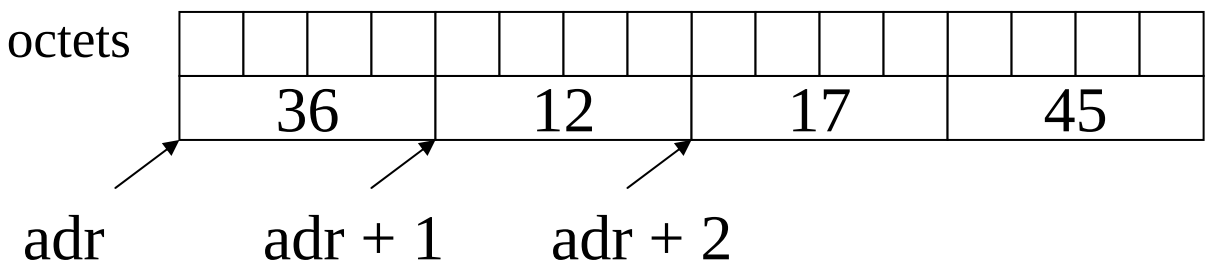
Le décalage se fait par rapport à la taille de l'objet pointé (size of)

• On peut aussi :

– incrémenter : adr++;

– ajouter : adr += 12;

• **Exception** : les pointeurs de fonction.



©Xavier Jeannin 2000-2005

Arithmétique des pointeurs

- **Comparaison de pointeur**

  le pointeur est caractérisé par une adresse et un type. On ne peut donc comparer que des pointeurs de même type.

```
int t[10] , *p;
for( p = t;  p < t +10;  p++)
    *p = 1;
```

```
int t[10] , *p;
for(i = 0;  i < 10;  i++)
        t[i] = 1;
```

- **Soustraction de pointeur**

  entre pointeur de même type; renvoie le nombre d'éléments du type entre les deux pointeurs

©Xavier Jeannin 2000-2005

Affectation de pointeur

• **On ne peut pas affecter 2 pointeurs de type différent.**
On peut affecter 2 pointeurs du même type uniquement.

– float *pf1, *pf2;

– pf1 = pf2;

• **On peut forcer l'affectation de 2 pointeurs de type différent avec l'opérateur cast :**

– char *c; int *i;

– i = (int *) c;

– Mais attention aux contraintes d'alignement

©Xavier Jeannin 2000-2005

Le pointeur NULL

- Le pointeur nul est un pointeur qui ne pointe sur rien.
  - – On peut donc affecter l'entier 0 à n'importe quel pointeur.
  - – Il est conseillé d'utiliser la constante NULL (definie dans stdio.h).

    P = 0; (conversion int → pointeur) p = NULL
  - – On peut comparer un pointeur à NULL.

    if (p == 0) p == NULL;
  - – Comme n'importe quelle variable, un pointeur de la classe d'allocation statique (par exemple les variables globales) est initialisé à 0 / NULL mais cela est dépendant de l'implémentation. Par prudence, il vaut mieux initialiser le pointeur.

```
int *Pa = NULL;
```

©Xavier Jeannin 2000-2005

Passage par adresse

```
include <stdio.h>                    /* Exemple d'appel de fonction */
void echange(int *a, int *b);        /* declaration de la fonction */

main()  /* pgrm principale */
    {int i = 10, j = 20;
    printf("valeur de i =%d= j =%d=", i,j);          i =10= j =20=
    echange(&i, &j);
    printf("valeur de i =%d= =%d=", i,j);            i =20= j =10=
    }
void echange(int *a, int *b)
    { int x;
    x = *a;
    *a = *b;
    *b = x;
    }
```

©Xavier Jeannin 2000-2005

Pointeurs constants

- int * const pc est un pointeur constant sur un entier, pc est non modifiable.

- const int *pc est un pointeur sur un entier constant, *pc est non modifiable.

Les tableaux et les pointeurs

Les tableaux sont des pointeurs constants

```
int t[10];
t          <==>   &t[0]
t + 1      <==>   &t[1]
t + i      <==>   &t[i]
t[i]       <==>   *(t + i)
```

Attention ! t n'est pas une lvalue : int t[10]; t = 2; impossible

```
int i, t[10];
for( i = 0;  i < 10;  i++)
    *(t + i) = 1;
```

```
int  *p, t[10];
for( p = t;  p < t+10;  p++)
        *p = 1;
```

©Xavier Jeannin 2000-2005

Cas des tableaux à 2 indices

int t[3][4];

t est un tableau des 3 éléments, chacun contenant un tableau de 4 éléments.

t + 1 est pointeur sur tableau de 4 entiers (une ligne du tableau)

t pointe au même endroit que &t[0][0] pointe au même endroit que t + 0

t + 1 pointe au même endroit que &t[1][0] pointe au même endroit que t + 1

Note : t représente l'adresse du début du tableau, l'adresse de la première ligne du tableau et l'adresse du premier élément du tableau

Le cas de sizeof

sizeof(t) est la taille de tout le tableau

sizeof(t + 0) est la taille d'une ligne du tableau

©Xavier Jeannin 2000-2005

Conversion des pointeurs

- **Conversion de pointeur**

  On peut convertir des pointeurs notamment en utilisant l'opérateur cast mais les contraintes d'alignement peuvent générer des erreurs, adresse obligatoirement paire etc

  Par exemples :

  ```
  char *pc; int *pi;
  pi = (int *)pc;
  ```

©Xavier Jeannin 2000-2005

Le pointeur générique : void *

- **Le pointeurs générique : void ***
  - – Sert quand une fonction manipule des adresses de type inconnu. Autrefois, on utilisait char *.
  - – Sert dans l'allocation dynamique de mémoire (malloc).
  - – Est compatible avec tous les type de pointeurs.
- Conversion de pointeur .

  type * ==> void * valide et bon

  void * ==> type * valide mais il faut contrôler car cela peut-être dangereux à cquse des contraintes d'alignement
  - – float *pf; void *pv;
  - – pf = pv;
- Interdiction du type void * :
  - – Pas de calcul arithmétique void *p,r; p + i et p - r n'ont pas de sens.
  - – Pas de déférencement (*pv, pv[])

Usage du pointeur vide

```
Void raz (void *adr, int n)
    {int i;
    char * ad  = (char *)adr;
    for(i = 0; i < n ; i++) *(ad + i) =0;
    }

/* appel */
int t[10];
longint d[4];
raz(t, 10*sizeof(int));
raz(d, 4 * sizeof(longint));
```

©Xavier Jeannin 2000-2005

Les tableaux transmis en argument

Lors du transfert d'un tableau en argument, on transfert seulement l'adresse du tableau

Les tableaux ne sont pas transmissible par la pile.

Dans la fonction appelée on reçoit un pointeur.

Cas à une dimension

```
int t[10], r[20];
f(t, 10);              /* syntaxe d'appel */
f(&r[0], 20);          /* syntaxe d'appel */
/* syntaxe possible de l'entête de la fonction */
void f(int tt[10], int n) ou void f(int *tt, int n)ou void f(int tt[ ] , int n)
        {int i;
        for(i = 0; i < n ; i++,tt++)
            *tt = 1; /* tt est une copie de l'adresse de r ou  de t, donc une lvalue */
        }
```

©Xavier Jeannin 2000-2005

Les tableaux transmis en argument

La dimension n'est pas indispensable pour la première dimension.

Si la dimension n'est pas fixe, il faut transmettre sous forme d'un argument supplémentaire

void f(int tt[10], int n)

Le dimension indiquée (10) n'a aucune signification pour le compilateur.

Les tableaux transmis en argument

Cas d'un tableau à plusieurs dimensions

- **Il n'est pas indispensable pour la première dimension mais elle est indispensable pour les autres dimensions et doit être fixée dans l'entête de la fonction.**

```
void f(int t[][15], int dim1)
    {int i, j;
    for(i = 0; i < dim1 ; i++)
        for(j = 0; j < 15 ; j++)
            t[i][j] = 1;
    }
```

- **f ne peut pas servir pour un tableau dont la seconde dimension n'est pas 15 :**

  int tab[10][15]; f(tab,10); f(&tab[0],10);

  si int tab[10][5]; f(tab,10); ne convient pas.

©Xavier Jeannin 2000-2005

Les tableaux transmis en argument

Cas d'un tableau à plusieurs dimensions

- **Les arguments d'appels peuvent être :**

  f(tab) ou f(&tab[0]) mais pas f(&tab[0][0]) qui un pointeur sur un entier.

©Xavier Jeannin 2000-2005

Les tableaux transmis en argument

Cas d'un tableau à plusieurs dimensions

Traiter le tableau comme un tableau à un indice et se servir des pointeurs :

```
int t[12][15];
f((int *) t, 12, 15);       /* remarquez le cast */

void f (int *adr; int n ; int p)
        {..
}
```

Accès au éléments du tableau dans la fonction

adr pointe sur le premier élément de la première ligne du tableau

adr + p pointe sur le premier élément de la seconde ligne du tableau

adr + i*p pointe sur le premier élément de la ligne de rang i du tableau

adr + i*p + j pointe sur l'élément de rang j de la ligne de rang i du tableau

t[i][j] <==> *(adr + i*p + j) dans la fonction

©Xavier Jeannin 2000-2005

Les tableaux transmis en argument

Cas d'un tableau à plusieurs dimensions

Retrouver le formalisme des tableaux :

```
int t[12][15];
f((int *) t, 12, 15);       /* remarquez le cast */

void f (int *adr; int n ; int p)
        {int **ad;

        ad = malloc(n * sizeof (int *));
        for (i = 0; i <  n ; i++)
                {ad[i] = adr + i * p;

                }

        for (i = 0; i <  n ; i++)
                for (j = 0; j <  p ; j++)
                        ad[i][j] = 0;
        }
```

©Xavier Jeannin 2000-2005

Les pointeurs de fonctions

En C, on peut déclarer un pointeur de fonction.

int (* fct) (double, int);

(* fct) est une fonction renvoyant un entier et prenant un double et un entier en argument alors que fct est un pointeur sur une fonction renvoyant un entier et prenant un double et un entier en argument.

Si on déclare : int (* fct)(double, int); et int f1(double, int);

on peut réaliser les affectations suivantes :

fct = f1;

fct = f2;

on utilise fct comme suit :

(* fct)(1.2, 3); ou par simplification fct(1.2, 3);

©Xavier Jeannin 2000-2005

Les pointeurs de fonctions en argument

Vous pourriez écrire une fonction de tri qui accepte en argument la fonction de comparaison. Vous pourriez ainsi faire des tris ascendants descendants ou autres.

```
void sort( char(* compare)(char, char), char *t, int size)
    {…
    }

char descendant(char a, char b)
    {return((a > b)? a : b);
    }

sort(descendant, tab_a_trier, 10);
```

©Xavier Jeannin 2000-2005

Les chaînes caractères
String

Définition et convention

Il n'y a pas de type chaîne ou string en C, on recourt à **un tableau de char**.
Il existe une convention de représentation des chaînes.

- Pour le compilateur les chaînes constantes sont notées entre double quotes "bonjour"
- Pour les fonctions de traitement des chaînes qui indiquent la fin de la chaîne par le caractère NUL '\0' : char chaine[10]; "bonjour" ==> bonjour\0

chaine[0] = 'b'	chaine[5] = 'u'
chaine[1] = 'o'	chaine[6] = 'r'
chaine[2] = 'n'	chaine[7] = '\0'
chaine[3] = 'j'	chaine[8] = indéfini
chaine[4] = 'o'	chaine[9] = indéfini

- **Rappel : 'a' != "a"**
  car "a" vaut 'a' suivi de '\0' ou 65 suivi de 0 en ASCII.

©Xavier Jeannin 2000-2005

Ecriture des constantes chaînes caractères

- **Backslash \ permet**
  - de désigner des caractères spéciaux :
    \n (fin de ligne) **\t** (tab) **\r** (carriage return) etc...
  - de désigner le slash \\ les guillemets **\"** et **\'**
  - des caractères sous forme octal **\123** et hexadécimal **\x123456**, attention seulement les 3 caractères suivants sont concernés pour les octales
- **\n est la fin de ligne (Line Feed)**
  - sous PC sous Windows **\n = \r\n,** substitution automatique
  - sous Unix pas de substitution de \n
- **guillemets simples et doubles**
  - " la chaine est non "valide"" alors que "la chaine est \"valide\""
  - " la chaine \'A\' est aussi valide"

©Xavier Jeannin 2000-2005

Ecriture des constantes chaînes caractères

- **Concaténation de constante chaînes adjacentes**
  - "bon" "jour" <==> "bonjour"
  - "voici un exemple de chaîne sans changement de lignes "
    " et qui s'écrit cependant sur 2 lignes "
  - " cela marche aussi \
    est ce dans la norme ??? "
- **Pourcentage %**
  - **%%** permet de représenter le caractère pourcentage.

©Xavier Jeannin 2000-2005

Initialisation

- **Vous n'avez pas le droit d'écrire :**

```
char ch[20];
ch = "bonjour";
```

- Vous pouvez utiliser :

```
char ch[20] = "bonjour";
char ch[20] = {'b','o','n','j','o','u','r','\0'};
```

  les 12 derniers caractères sont initialisés à 0 ou non en fonction de la classe d'allocation.

```
char ch2[] = "bonjour"; /* ch2 fait 8 caractères exactement */
```

- Vous pouvez aussi utiliser :

```
char *jour[7] =  {"lundi", "mardi", "mercredi", "jeudi", "vendredi",
"samedi", "dimanche"}
```

  jour est **pointeur sur un tableau de chaînes** .

©Xavier Jeannin 2000-2005

Initialisation

- **Attention au constante chaîne :**
  char *ch = "bonjour"; est valide ; le compilateur remplace "bonjour" par l'adresse de l'emplacement où il a rangé la chaîne.

  Mais **vous ne pouvez pas modifier "bonjour" par** :

  ```
  char *ch;
  ch = "bonjour";
  *(ch + 2) = 'X';     /* accepté mais comportement indéterminé */
  ```

  On peut se prémunir contre cette faute par la déclaration

  ```
  char const *ch =  "bonjour";
  ```

- Par contre, si la déclaration est char *ch[20] = "bonjour"; La modification fonctionne.

Lire et écrire des chaînes

Plusieurs possibilités :

- le format **%s** avec les fonctions **scanf** et **printf**.
- les fonctions spécifiques : lecture **gets** et écriture **puts** d'une seule chaîne à la fois.

```
#include <stdio.h>
main()
    {char nom[20], prenom[20], ville[20];
    printf("quelle est votre ville :");
    gets(ville);
    printf("donnez votre nom et prenom :\n");
    scanf("%s %s",nom, prenom);
    printf(" %s %s habite à %s \n", prenom, nom, ville);
    }
```

©Xavier Jeannin 2000-2005

Lire et écrire des chaînes

- **Scanf avec %s :**
  - Le délimiteur espace empêche de saisir des chaînes comportant des espaces.
  - Le délimiteur n'est pas consommé.
- **Gets :**
  - Lit une chaîne et une seule.
  - Le seul délimiteur est la fin de ligne (\n).
  - Le délimiteur (\n ou EOF) est consommé mais remplacé par \0 et est rajouté à la fin.
  - Renvoie NULL si erreur ou le pointeur sur la chaîne saisie.
- **Puts :**
  - Ajoute un saut de ligne à la fin de chaque chaîne.

©Xavier Jeannin 2000-2005

Fiabiliser la lecture de chaîne

- Pour parer aux pbs posés par scanf :
  - On peut utiliser gets mais il y a un pb avec l'arrêt de la lecture et le débordement de buffer donc plutôt **fgets(stdin, …).**
  - Puis decoder la chaîne avec sscanf

sscanf(adresse, format, liste vbles)

- Maîtrise du buffer, donc nous pouvons appeler sscanf sans nous soucier de la position du pointeur dans le buffer.

©Xavier Jeannin 2000-2005

Fiabiliser la lecture de chaîne

```
#define  LG_LIGNE 41

char ligne[LG_LIGNE], mot [31],c;

int n;
float x;

while (1)
        {printf("Donnez un entier, un flottant et une chaine de caractere :\n");
        fgets(ligne, LG_LIGNE, stdin);
        n_val_ok = sscanf(ligne, "%d %e %30s", &n, &x, mot);
        if(n_val_ok == 3) break;
        }
printf("'Merci pour l\'entier %d le flottant %e la chaine %s \n", n, x, mot);

/* traitement du cas ou l'utilisateur a fourni une chaine trop longue */
if((strlen(ligne) == LG_LIGNE - 1) && (ligne[LG_LIGNE - 2] != '\n'))
        do c = getchar(); while(c !='\n');
```

©Xavier Jeannin 2000-2005

Les fonctions de manipulation de chaînes

- Les prototypes des fonctions manipulant les chaînes se trouvent principalement dans string.h.
- Les fonctions travaillent sur l'adresse des chaînes.
- Les fonctions possèdent 2 variantes : l'une travaillant sans contrôle, l'autre permettant une limitation de l'effet de la fonction à un certain nombre de caractères.
- Les fonctions renvoient en général NULL en cas d'échec.
- **strlen :** longueur de la chaîne;

  strlen(chaîne)

  strlen("bonjour"); ==> 7

  char *ch = "coucou"; strlen(adr); ==> 6

©Xavier Jeannin 2000-2005

Concaténation de chaînes

- **strcat :** concatène 2 chaînes

  strcat(but, source)

  ```
  char ch1[50] = "Comment";
  char *ch2 = "ca va";
  strcat(ch1,ch2);
  ch1 ==> "Commentca va"
  ```

- **strncat :** concatène 2 chaînes avec contrôle de la longueur

  strcat(but, source, lgmax)

  ```
  char ch1[50] = "Comment";
  char *ch2 = "ca va";
  strcat(ch1, ch2, 2);
  ch1 ==> "Commentca"
  ```

©Xavier Jeannin 2000-2005

Comparaison de chaînes

- **strcmp :** compare 2 chaînes, ordre des codes de caractère (ASCII).

  strcmp(ch1, ch1)

  - positif si ch1 > ch2
  - nul si ch1 = ch2
  - négatif si ch1 < ch2

  strcmp("bonjour", "madame"); ==> négatif

  strcmp("bonjour", "andre"); ==> positif

©Xavier Jeannin 2000-2005

Comparaison de chaînes

- **strncmp :** compare 2 chaînes, ordre des codes de caractère (ASCII).

  strnmp(ch1, ch2, lgmax)

  strncmp("bonjour", "bon",4); ==> négatif

  strncmp("bonjour", "bon",2); ==> zéro

©Xavier Jeannin 2000-2005

Copie de chaînes

- **strcpy :** copy source dans dest.

  strcpy(dest, source)

- **strncpy :** de même que strcpy avec longeur max.

  strncpy(dest, source, lgmax)

  char ch1[20] = "123456789"

  char ch2[20] = "coucou"

  strncpy(ch1, ch2, 3); ==> ch1 = "cou456789"

  strncpy(ch1, ch2, strlen(ch2) + 1); ==> ch1 = "coucou"

©Xavier Jeannin 2000-2005

Recherche dans une chaîne

- On peut rechercher l'occurrence d'un caractère d'une sous chaîne, les fonctions renvoient un pointeur sur l'endroit recherché ou NULL dans le cas contraire.

strchr(chaîne, caractère) en partant du début

strrchr(chaîne, caractère) en partant de la fin

strstr(chaîne, sous-chaîne) en partant du début

©Xavier Jeannin 2000-2005

Conversion

- Les fonctions de conversion ignorent les espaces de début de chaîne et utilisent les caractères suivants pour fabriquer une valeur numérique.
- Un caractère illégal arrête le traitement.
- Si aucun caractère n'est exploitable, renvoie nul.

atoi(chaîne) int

atol(chaîne) long

atof(chaîne) double

©Xavier Jeannin 2000-2005

Les structures

Définition

La structure permet de regrouper différentes variables en une seule.

Exemple :

```
struct produit
   {int id;
   int qte;
   float prix;
   };

struct produit oeuf;
```

- "produit" est **un modèle de structure**, il n'y a pas de réservation de place mémoire. "id", "qte" et "prix" sont nommés **champs**.
- "oeuf"est une **variable** de type "produit", **la place mémoire est réservée** lors de la déclaration de "œuf".

©Xavier Jeannin 2000-2005

Accès au champs

On peut concentrer l'écriture (pas recommandé):

```
struct produit
    {int id;
    int qte;
    float prix;
    }oeuf, tomate, mozarella;
```

- **L'accès au champs se fait par l'opérateur point :**

```
œuf.prix = 1.15;
printf("%f", œuf.prix);
scanf("%f", &œuf.prix);
œuf.prix ++;
```

©Xavier Jeannin 2000-2005

L'affectation

L'affectation entre 2 structures de même modèle :

struct produit œuf, tomate.

tomate = œuf;

ce qui équivaut à :

```
tomate.prix = oeuf.prix;
tomate.qte = oeuf.qte;
tomate.id = oeuf.id;
```

Remarque : l'affectation globale entre 2 tableaux est impossible, mais on peut la réaliser en créant une structures contenant un tableau.

©Xavier Jeannin 2000-2005

Initialisation des structures

- Les structures suivent la règle des classes d'allocation.
- Il est possible d'initialiser explicitement avec des expressions constantes et non des expressions quelconques, comme les tableaux.
  - **Struct produit tomate = { 2, 2000, 2.15 };**
- La taille d'une structure dépend des **contraintes d'alignement** :

```
struct taille
  {int x;
  char c;
  } t ;
  printf(" taille %d \n", sizeof(t));
```

La taille de t **devrait être 5**, mais en fait **sizeof renvoie la taille réelle 8.**

©Xavier Jeannin 2000-2005

Typedef

- **Typedef permet de définir des types synonymes**

```
typedef  int entier;
entier i, j, k;

typedef  int *ptint;
ptint  pti, ptj, ptk;

struct produit
   {int id;
   int qte;
   float prix;
   };
typedef struct produit t_produit;
t_produit œuf, tomate;
```

©Xavier Jeannin 2000-2005

Les structures comportant des tableaux

```
struct eleve
   {int id;
   char nom[20];
   int note[10];
   } gaston;
```

gaston.note[4] = note_geo;

gaston.nom est l'adresse du tableau "nom".

```
strcpy(gaston.nom, "gaston");
```

gaston.nom[2] est la lettre 's'

```
struct eleve mirza = {10, "mirza" , {0, 1, 2, 3, 4, 5}};
```

©Xavier Jeannin 2000-2005

Les tableaux de structures

```
struct point
  {char nom[10];
  int x;
  int y};
struct point courbe[50];
```

"point" est le modèle de structure, courbe est tableau de 50 éléments de type "point"

courbe[3].x est valide, courbe.x[3] n'a pas de sens.

courbe[3] est une structure "point" le 4 éme point de la courbe.

courbe[3].nom[2] pour accéder au 3 éme caractére de nom du 4 éme point de la courbe.

courbe est l'adresse du début du tableau.

Initialisation :

```
struct point courbe2[25] = {{'a', 1, 2}, {'b', 1, 2}, , {'d', 1, 2}};
```

©Xavier Jeannin 2000-2005

Les structures imbriquées

```
struct date {int jour; int mois; int annee};
struct person
  {char nom[10]; char prenom[10];
  struct date date_embauche;
  struct date date_poste;
  } gaston;

gaston.date_embauche.annee = 1985;

gaston.date_poste.annee = gaston.date_embauche.annee;
```

©Xavier Jeannin 2000-2005

Portée des modèles de structure

- Si le modèle est déclaré :
  - à l'intérieur d'une fonction, la portée est limitée à la fonction.
  - À l'extérieur d'une fonction, la portée est limitée à tout le fichier source mais pas dans les autres fichiers sources car la directive "extern" ne peut s'appliquer (seulement aux variables).

- Par contre on peut utiliser un fichier ".h" inclus par la directive include

©Xavier Jeannin 2000-2005

Les structures en arguments

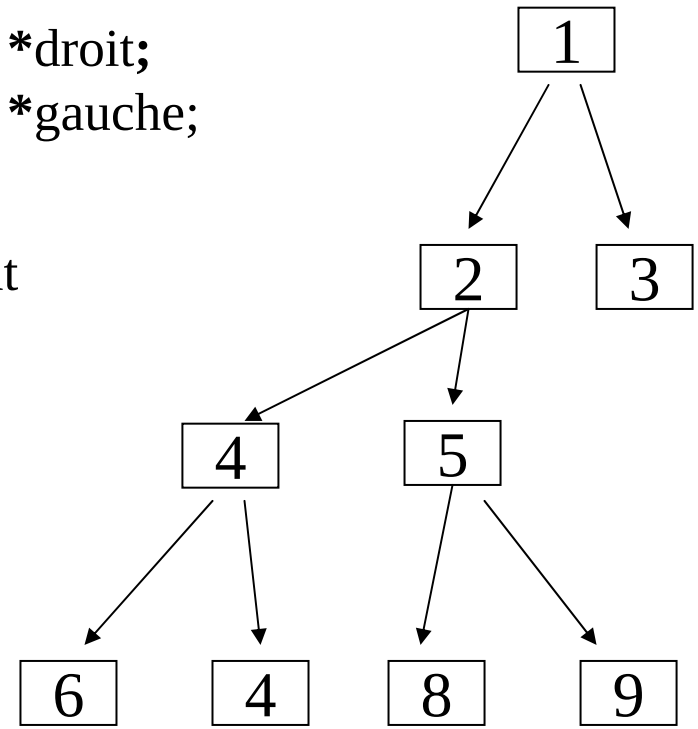
- On peut transmettre une structure par valeur (contrairement au tableau) et on travaille dans l'appelée sur une copie.
- Pour changer les valeurs de la structure, on envoie un pointeur :
  f(&date_embauche)
  définition de l'appelée :
  void f(struct date *d);

  Dans l'appelée on accède au champs par : **(*d).annee = 1986**
  ou on peut utiliser : **d->annee = 1986**
- Structure en valeur de retour
  struct date f(…….)
  {….
  }

©Xavier Jeannin 2000-2005

Les structures récursives

- struct noeud
  {int val
  struct nœud ***droit;**
  **struct nœud ***gauche;
  };
- droit et gauche sont obligatoirement des pointeurs



©Xavier Jeannin 2000-2005

Les unions

- Les unions permettent de stocker dans un espace mémoire des variables de plusieurs types différents.

```
union prime
        {int fixe;
        double taux;
        } sac, *pt_sac;
```

- La syntaxe est la même que pour une structure : sac.val pt_sac->taux
- struct salaire
        {char nom[10];
        union prime prime_mensuelle;
        } employe[50];
- employe[5]. prime_mensuelle.fixe, employe[6]. prime_mensuelle.taux
- L'initialisation explicite se fait par le premier type ici int

©Xavier Jeannin 2000-2005

Les fichiers

Les fichiers

- Les entrées / sorties que nous avons déjà vues ne sont qu'un cas particulier des fichiers. On nomme en jargon les fichiers des flux (stream).

- **Type d'accès aux fichiers**
  - l'accès séquentiel : traiter les informations séquentiellement comme sur une bande magnétique.
  - l'accès direct : se placer immédiatement sur l'information à traiter sans parcourir celles qui la précèdent.

©Xavier Jeannin 2000-2005

Exemples

```
#include <stdio.h>                /* Exemple d'ecriture de fichier */
main()  /* pgrm principale */
    {char nom_fich[21];  int n;
    FILE *fichier;

    printf(" Entrez un  nom de fichier :");    /* Lire le nom du fichier */
    scanf("%20s", nom_fich);
    fichier = fopen( nom_fich, "w");
    do
        {printf("donnez  un entier  (0 pour finir) :  \n");
        scanf("%d", &n);
        if(n != 0)
                n =fwrite(&n , sizeof(int), 1, fichier) ;
        }
    while(n != 0);
    fclose(fichier);
    }
```

©Xavier Jeannin 2000-2005

Les fichiers

Un fichier est déclaré par :

FILE *fichier;

fopen ouvre le fichier en mode écriture "w". Le fichier est crée s'il n'existe pas sinon il est recréé (remis à zéro).

fichier = fopen(nom_fich, "w");

fwrite ecrit le tampon "n" et cela 1 fois dans fichier. Fwrite a besoin de la taille entier.

fwrite(&n , sizeof(int), 1, fichier) ;

ferme le fichier.

fclose(fichier);

©Xavier Jeannin 2000-2005

Lecture séquentielle

```
 #include <stdio.h>                /* Exemple de lecture de fichier */
main()  /* pgrm principale */
     {char nom_fich[21];  int n;
     FILE *fichier;

     printf(" Entrez un  nom de fichier :");    /* Lire le nom de fichier  */
     scanf("%20s", nom_fich);
     fichier = fopen( nom_fich, "r");
     do
          {fread(&n , sizeof(int), 1, fichier) ;
          if(! feof(fichier) )printf("\n  %d",n);
          }
     while(! feof(fichier));
     fclose(fichier);
     }
```

©Xavier Jeannin 2000-2005

Lecture séquentielle

fichier = fopen(nom_fich, "r");

fopen ouvre le fichier en mode ouvrir pour la lecture.

fread(&n , sizeof(int), 1, fichier) ;

fread lit dans le fichier "fichier" 1 * sizeof(int) octets dans &n.

fread renvoie 0, s'il y a une erreur ou si la fin de fichier est atteinte.

- Pour distinguer entre une erreur et une fin normale de fichier, on utilise feof(fichier) qui renvoie vrai à la fin du fichier. Il faut lire après le fichier pour que feof renvoie vrai.

- L'indicateur de fin de fichier est remis à zéro grâce à :
  - l'appel à la fonction clearerr
  - lorsque l'on lit avec succès de nouvelles informations

©Xavier Jeannin 2000-2005

Lecture directe

```
 #include <stdio.h>              /* Exemple de lecture directe de fichier */
main()  /* pgrm principale */
    {char nom_fich[21];  int n, res  = 1; long  pos; FILE *fichier;
    printf(" Entrez un  nom de fichier :");    /* Lire la valeur du nombre suivant */
    scanf("%20s", nom_fich);
    fichier = fopen( nom_fich, "r");
    while  (pos)
          {printf(" Entrez la position de l'entier recherché (q pour arrêter) : ");
          res = scanf("%ld", &pos);
          if (res)
                {fseek(fichier, sizeof(int) * (pos -1), SEEK_SET) ;
                fread(&n , sizeof(int), 1, fichier) ;
                printf(" valeur = %d \n",n);
                }
          }
    fclose(fichier);
    }
```

©Xavier Jeannin 2000-2005

Lecture directe

fseek(fichier, sizeof(int) * (pos -1), SEEK_SET) ;

- le type de "sizeof(int) * (pos -1)" est long car la position dans un fichier dépassent Maxint.
- SEEK_SET == 0 : déplacement depuis la début du fichier en octets.
- SEEK_CUR == 1 : déplacement depuis la position courante en octets.
- SEEK_END == 2 : déplacement depuis la fin du fichier en octets.

- Fseek renvoie si 0 si le positionnement s'est bien déroulé
  un valeur > 0 sinon (cas de dépassement du fichier)
  cette norme n'est pas toujours appliquée
- On peut utiliser ftell qui renvoie la position

```
pos = ftell(fichier);
```

©Xavier Jeannin 2000-2005

Création de fichier par accès direct

- En écriture, on voit que la possibilité de faire des trous existe.

  fseek(fichier, sizeof(int) * (pos -1), SEEK_SET) ;

  fwrite(&n , sizeof(int), 1, fichier) ;

- Ceci provoque en général la création de zones non initialisées
- Pour résoudre ces problèmes de trou on peut :
  - soit initialiser les zones non remplies explicitement.
  - soit gérer une table des zones écrites.

©Xavier Jeannin 2000-2005

Les fichiers de texte

- Les fichiers de type texte ou "fichiers texte" sont des fichiers que vous pouvez manipuler avec un éditeur de texte ou lister (more cnrs.txt).
- Chaque octet représente un caractère. '2' '7' représente l'entier 27.
- En général, on y trouve des caractères fin de ligne (\n) qui créent des suites de lignes.
- fread et fwrite réalisent des transferts brut entre mémoire et fichier.

©Xavier Jeannin 2000-2005

Fichiers de Texte

- Portabilité.
- transcodage.
  - **\r\n est transformé en \n en lecture.**
  - **\n est transformé en \r\n en écriture.**
- espaces de fin de ligne peuvent être tronqués en lecture.
- Il peut y avoir un transcodage des caractères mais pas sur les imprimables, \t ,\n
- Le compilateur doit accepter les lignes d'au moins 254 car. et il peut y en avoir une borne max
- Accès directe difficile à gérer

©Xavier Jeannin 2000-2005

Fichiers binaires

- Les informations dans le fichier sont telles que représentées en mémoire.
- Pb de portabilité :
  - **même représentation Complément A2.**
  - **Même implémentation cf sizeof(type).**
- Les fichiers seront utilisables par des programme C de la même implémentation que le créateur du fichier.

©Xavier Jeannin 2000-2005

Les fonctions

- Formellement les fonctions ne sont pas dédiés à un type d'accès mais il est conseillé :

- Fread, fwrite --> binaires

- fscanf, fprintf --> texte
- fgets, fputs --> texte

- Fgetc, fputc --> mixte
- getc, putc --> mixte

©Xavier Jeannin 2000-2005

Les fichier textes

- On peut utiliser sur ces fichiers des fonctions de formatage

  fscanf(fichier, format, liste d'adresses);

  fprintf(fichier, format, liste d'adresses);

  fgetc(fichier);

  fputc(‘c’, fichier);

  fgets(chaîne, lgmax, fichier);

  fputs(chaîne, fichier);
- les valeurs de retour et la signification des arguments sont les mêmes que pour les fonctions conversationnelles à l'exception des détails suivants :
  - fgets prend en argument une longueur max qui contrôle le nb maximal de caractères entrés (y compris \0).
  - fgetc renvoie un int et non pas un char car lorsque getc détecte la fin de fichier il renvoie la constante EOF en générale -1. La détection de fin de fichier se fait quand il n'y a plus de caractères disponibles.

©Xavier Jeannin 2000-2005

Les fichiers de texte

- **Le cas du Windows :**

- Sous DOS la fin de ligne est représentée par la succession de 2 caractères \r\n retour chariot et fin de ligne.
- Le C ne voit lui qu'un seul caractère, donc l'implémentation des fonctions de lecture et d'écriture :
  - en lecture : remplacer le couple \r\n par \n
  - en écriture : remplacer \n par le couple \r\n
- Deuxième conséquence pour déterminer le type de fichier, on ajoute la lettre t pour "texte" ou la lettre b pour "binaire".
- Dans ce type d'implémentation, les fonctions formatées ne sont utilisables que sur les fichiers textes.

©Xavier Jeannin 2000-2005

Gestion des erreurs

- Tester systématiquement fopen
- Ferror() est peu exploitable
- Errno est peu exploitable
- feof est exploitable uniquement en lecture
- les valeurs de retour des fonctions de positionnement du pointeur sont peu fiables
- les valeurs de retour des fonctions de lecture et écriture sont à examiner systématiquement.

©Xavier Jeannin 2000-2005

Gestion des erreurs

- Ferror(flux)
  - renvoie vrai ou faux.
  - inaccessible si fopen a échoué
  - toutes les erreurs ne sont pas répercutées par cette fonction

- Errno est un indicateur peu fiable
  - indifférencié selon les fonctions
  - la norme n'oblige pas sa prise en compte

©Xavier Jeannin 2000-2005

Gestion des erreurs

- Fwrite(..,..,nblocs, fich)
  - retour = nblocs <==> écriture OK
  - retour < nblocs <==> écriture pas OK, manque de place, erreur matériel.
- Fread(..,..,nblocs, fich)
  - feof = Faux et retour = nblocs <==> lecture OK
  - feof = Faux et retour < nblocs <==> erreur de lecture
  - feof = Vrai et retour = 0 <==> fin normal de fichier
  - feof = Vrai et retour != 0 <==> fin anormal de fichier

©Xavier Jeannin 2000-2005

Gestion des erreurs

- Retour = fprintf(fich, ...)
  - retour < 0 <==> erreur d'écriture OK, manque de place, erreur matériel.
  - Sinon <==> écriture OK.
- Retour = fscanf(fich, …..)
  - feof = Faux et retour = nb de valeurs attendus <==> lecture OK
  - feof = Faux et retour < nb de valeurs attendus <==> erreur de lecture
  - feof = Vrai et retour = EOF <==> fin normal de fichier
  - feof = Vrai et retour != EOF <==> fin anormal de fichier

©Xavier Jeannin 2000-2005

Gestion des erreurs

- Retour = fputs(….,fich)
  - retour != EOF <==> écriture OK
  - retour = EOF <==> erreur d'écriture

- Retour = fgets(….,fich)
  - feof = Faux et retour != NULL <==> lecture OK
  - feof = Faux et retour = NULL <==> erreur de lecture
  - feof = Vrai et retour = NULL <==> fin normal de fichier
  - feof = Vrai et retour != NULL <==> fin anormal de fichier

©Xavier Jeannin 2000-2005

Les modes d'ouverture

- **r :** lecture seule, le fichier doit exister.
- **w :** écriture seule, le fichier est créé s'il n'existe pas ou s'il existe son contenu est perdu.
- **a :** écriture en fin de fichier, le fichier est créé s'il n'existe pas ou s'il existe son contenu sera étendu.
- **r+ :** mise à jour (lecture écriture), le fichier doit exister. On ne peut enchaîner les deux opérations lecture et écriture successivement sans effectuer un fseek.
- **w+ :** création pour mise à jour, le fichier est crée s'il n'existe pas ou s'il existe son contenu est perdu (de même que r+).
- **a+ :** extension pour mise à jour, le fichier est créé s'il n'existe pas ou s'il existe son contenu sera étendu.
- **t ou b :** lorsque l'implémentation distingue les fichiers textes des binaires. Cette option s'ajoute aux autres.

©Xavier Jeannin 2000-2005

Les fichiers prédéfinis

- **stdin :** unité d'entrée, par défaut le clavier.
- **stdout :** unité de sortie, par défaut l'écran.
- **stderr :** unité d'affichage des messages d'erreur, par défaut l'écran.
- if (DEBUG)

```
fprinf(stderr,"analyseur : taux %f\n", taux);
```

- **redirection de programme :**

  monpgrm < donnees > resultat

©Xavier Jeannin 2000-2005

Gestion dynamique

Classe d'allocation

- **statique :** les données occupent un emplacement défini lors de la compilation.
- **automatique :** les données sont créées dynamiquement au fur et à mesure de l'exécution du pgrm, dans la pile (stack) .
- **dynamique** : les données sont créées ou détruites à l'initiative du programmeur. Elle sont créées dans le tas (heap).

©Xavier Jeannin 2000-2005

Malloc et free

```
long *p;
p = (long *)malloc (100 * sizeof(long));
for( i = 0; i < 100; i++) *(p +i ) = 1;
```

- nous avons ainsi 100 long, compte tenu de l'arithmétique des pointeurs, il est important que"p" soit de type pointeur sur long.

void *malloc(size_t taille); /* stdlib.h */

- malloc renvoie un pointeur générique qui est converti dans le type du pointeur. Size_t est un est prédéfinie par TYPEDEF ce qui implique que le type de size_t dépend de l'implémentation.
- Malloc renvoie NULL si l'allocation échoue.
- Malloc aligne en fonction de la plus grande contrainte d'alignement car malloc renvoie le pointeur que l'on doit affectater or "*type <== *void" n'est pas sûr.

©Xavier Jeannin 2000-2005

Malloc et free

```
char *p1, *p2, *p3;
p1 = (char *)malloc (100 * sizeof(char));
printf("allocation de 100 octets en %p", p1);
p2 = (char *) malloc (50 * sizeof(char));
printf("allocation de 50 octets en %p", p2);
free(p1);
printf("libération de 100 octets en %p", p1);
p3 = (char *) malloc (100 * sizeof(char));
printf("allocation de 100 octets en %p", p3);
```

- allocation de 100 octets en 0x80496a8
- allocation de 50 octets en 0x8049710
- libération de 100 octets en 0x80496a8
- allocation de 50 octets en 0x80496a8

Écart de 104 et non de 100 : 4 octets de service. La taille réelle allouée dépend de l'implémentation

©Xavier Jeannin 2000-2005

Calloc et realloc

void *calloc(size_t nb_blocs, size_t taille); /* stdlib.h */

- calloc alloue "nb_blocs" consécutifs de taille "taille".
- calloc met à zéro chacun des octets de la zone allouée

void *realloc(void *pointeur, size_t taille); /* stdlib.h */

- realloc modifie la taille d'une zone déjà allouée.
- "pointeur" est l'adresse de la zone à modifier
- realloc renvoie la nouvelle adresse de la zone, nul si échec.
- Si la "taille" est supérieure à l'ancienne, il y a conservation des données, quitte à recopier ces dernières.
- Si la "taille" est inférieure à l'ancienne, il y a conservation des données, jusqu'à la nouvelle taille.

©Xavier Jeannin 2000-2005

Le préprocesseur

Le préprocesseur

- Le préprocesseur est un programme qui agit automatiquement avant la compilation. Ce programme transforme le source à partir d'un certain nombre de "**directives**".

- Ces directives commencent par #, pas forcément en première colonne; mais # doit être le premier caractère de la ligne.

- le préprocesseur permet
  - l'incorporation de fichiers sources (#include)
  - la définition de symboles et de macros (#define)
  - la compilation conditionnelle

©Xavier Jeannin 2000-2005

#include

- Permet d'introduire avant la compilation. On utilise cette directive pour incorporer les en-têtes de fonctions d'une librairie que l'on veut utiliser (par exemple #include <stdlib.h>). Pratique pour centraliser et éviter les répétitions des déclarations extern prototype et variable

- **#include<nom_fichier>**

  recherche le fichier mentionné dans un emplacement défini par l'implémentation du compilateur (en Unix /usr/include).
- **#include "nom_fichier"**

  recherche le fichier mentionné dans le même emplacement que le source.

- On peut incorporer des "includes" dans des fichiers include
- Attention à ne pas redéclarer plusieurs fois la même fonction et variable.

©Xavier Jeannin 2000-2005

#define

- Permet de
  - **définir des symboles**
  - **définir des macros**

- **#define NB_MAX 5**
  remplace le texte "NB_MAX" par le texte "5"
  #define NB_MIN NB_MAX - 5
  NB_MIN n'est pas remplacé par "0" mais par "5 - 5"

- **#define ENTIER int**
  ENTIER i, *p;

©Xavier Jeannin 2000-2005

#define

- **#define ligne printf("\n")**
  Attention à ne pas mettre de ";" superflu.

- #define N = 5
  int tab[N];
  Attention à la substitution **int tab[=5];**

- Réciproquement **#undef N** annule la définition

  Avec certains compilateurs, on peut vérifier les substitutions grâce à certaine options du compilateur.

  gcc -E ...

©Xavier Jeannin 2000-2005

Macros

#define carre(a) a * a

#define deux(a) a + a

carre(z);

deux(z) * y <==> z + (z * y) et non (z + z)* y ce que l'on souhaitait

- on résout le problème grâce aux parenthèses

  #define deux(a) (a + a)

  deux(z) * y <==> (z + z)* y
- en fait la macro ne peut avoir d'espace dans le nom de la macro.

  #define deux (a) (a + a)

  deux(z) * y <==> (a) (a + a) (z)* y
- On peut prolonger une macro sur plusieurs lignes grâce à \

  #define   trois(a) (a + a \

  + a)

©Xavier Jeannin 2000-2005

Compilation conditionnelle

Le préprocesseur permet d'inclure ou d'exclure des portions des fichiers sources

- **test sur l'existence d'un symbole**

#ifdef DEBUG	**réciproquement**	**#ifndef** DEBUG
instruction1		instruction1
#else		**#else**
instruction2		instruction2
#endif		**#endif**

(pas de #elif avec #ifdef)

L'instruction1 sera incorporée au code si DEBUG a été defini, sinon ce sera l'instruction2. On définit par **#define** DEBUG et on supprime par **#undef** DEBUG

La norme ANSI a introduit :

#ifdef DEBUG <==> #**if defined**(DEBUG)

#ifndef DEBUG <==> #**if ! defined**(DEBUG)

©Xavier Jeannin 2000-2005

Compilation conditionnelle

- **test sur la valeur d'un symbole**

```
#define DEBUG 1
#if DEBUG == 1
instruction1
#elif  DEBUG <= 4
instruction2
#else
instruction3
#endif
```

L'instruction1 sera incorporée au code si DEBUG vaut 1, l'instruction2 si DEBUG vaut moins de 4 sinon ce sera l'instruction3.

(#elif ne s'applique qu'à #if et non #ifdef)

- Attention tout identificateur non remplacé par une constante est remplacé par 0.

```
#DEFINE NPT 12
 If NPT < NPT_MAX        /*  NPT_MAX vaut 0 */
```

Compilation conditionnelle

#ifdef se fonde sur l'existence du symbole

#if se fonde sur une expression arithmétique

À #if on peut rajouter l'opérateur « defined » qui renvoie

- 0 si non existence
- 1 si existence

Par exemple :

```
#if (CODE ==1) || defined(DEBUG)
```

©Xavier Jeannin 2000-2005

Autres directives

#pragma

La directive #pragma, qui n'existe que depuis la norme ANSI, est entièrement dépendante de l'implantation. Les concepteurs d'environnement ont une liberté complète pour décider de son utilisation.

#line constant ["nom de fichier"]

change le numéro de ligne et le nom du fichier pour les messages de compilations. Si "nom de fichier" est absent, le nom du source est conservé.

#error message : permet de stopper la compilation en affichant le message d'erreur donné en paramètre ;

: ne fait rien (directive nulle) ;

Manipulation de chaînes de caractères dans les macros

- Le préprocesseur permet d'effectuer des opérations sur les chaînes de caractères. Tout argument de macro peut être transformé en chaîne de caractères dans la définition de la macro s'il est précédé du signe #. Par exemple, la macro suivante :

  #define CHAINE(s) #s

  transforme son argument en chaîne de caractères.

  CHAINE(2+3) devient "2+3"

- Lors de la transformation de l'argument, toute occurrence des caractères " et \ est transformée respectivement en \" et \\ pour conserver ces caractères dans la chaîne de caractères de remplacement.

©Xavier Jeannin 2000-2005

Manipulation de chaînes de caractères dans les macros

- Le préprocesseur permet également la concaténation de texte grâce à l'opérateur ##. Les arguments de la macro qui sont séparés par cet opérateur sont concaténés (sans être transformés en chaînes de caractères cependant). Par exemple, la macro suivante :

  ```
  #define NOMBRE(chiffre1,chiffre2) chiffre1##chiffre2
  ```

  permet de construire un nombre à deux chiffres :

  NOMBRE(2,3) est remplacé par le nombre décimal 23.

- Le résultat de la concaténation est ensuite analysé pour d'éventuels remplacements additionnels par le préprocesseur.

©Xavier Jeannin 2000-2005

Autres directives

- Le préprocesseur définit un certain nombre de constantes de compilation automatiquement. Ce sont les suivantes :
    - __LINE__ : donne le numéro de la ligne courante ;
    - __FILE__ : donne le nom du fichier courant ;
    - __DATE__ : renvoie la date du traitement du fichier par le préprocesseur ;
    - __TIME__ : renvoie l'heure du traitement du fichier par le préprocesseur ;

©Xavier Jeannin 2000-2005

Possibilités proche de la machine

Possibilités proche de la machine

- Ces fonctionnalités du C proches de l'assembleur sont par définition peu portables.
- **Représentation des entiers**

1	0000 0000 0000 0001	00 01
2	0000 0000 0000 0010	00 02
255	0000 0000 1111 1111	00 FF
0	0000 0000 0000 0000	00 00
-1	1111 1111 1111 1111	FF FF
-2	1111 1111 1111 1110	FF FE
-255	1111 1111 0000 0001	FF 01

- les négatifs sont représentés en "complément à deux", ce qui signifie que pour un nombre négatif,on prend la valeur absolue calculée en binaire puis on inverse tous les bits (0 => 1, 1 => 0), puis on ajoute 1.

©Xavier Jeannin 2000-2005

Possibilités proches de la machine

- Les short, int et long sont codés en complément à deux.
- On peut qualifier ces types de signed ou unsigned. En non signé, les entiers sur 16 bit passent de [-32768; 32767] à [0; 65535].
- Un entier pourra avoir 2 valeurs suivant la manière dont il est considéré. Par exemple, sur 16 bits 1111111111111111 vaut -1 en non-signé et 65535 en signé.
- **Conversion de signe :**
  - Si on ne mélange pas les signés et les non signés conversion classique short => int => long
  - Le mélange de signés et non signés est autorisé; cela provoque en général une conversion signé vers non signé. Les conversions prévues par la norme se contentent de conserver le motif binaire.
  - Il en va de même, pour le cast ou la conversion forcée par affectation.

©Xavier Jeannin 2000-2005

Les unsigned char

- La norme ne stipule pas si un char est signé ou non.
- Il n'y a qu'un seul code de 8 bits de 256 combinaisons.
- Mais si on considère la valeur numérique, alors le signe compte comme dans
  c + 1; c++; printf("%d", c);
- La conversion du char en int se fait :
  - c non signé : 1000 0111 ==> 0000 0000 1000 0111
  - c signé : 0000 0111 ==> 0000 0000 0000 0111
    1000 0111 ==> 1111 1111 1000 0111
  - **propagation du bit de signe**
- Tout ce passe comme si les unsigned char prennent des valeurs entières comprises entre 0 et 255 et les signed char entre -128 et 127.

©Xavier Jeannin 2000-2005

Les unsigned char

- si n est entier :
  n = c;
  donnera une valeur entre 0 et 255 si c est unsigned char.
  donnera une valeur entre -128 et 127 si c est signed char.

- On peut ranger un entier (un bout seulement) dans un char, seuls les bits les moins significatifs sont recopiés, recopie exacte du motif binaire.

©Xavier Jeannin 2000-2005

Opérateurs de manipulation de bits

- Ces opérateurs ne portent que sur **les types entiers**.

Binaire	&	Et (bit à bit)
Binaire	|	Ou (bit à bit)
Binaire	^	Ou exclusif (bit à bit)
Binaire	<<	Décalage à gauche
Binaire	>>	Décalage à droite
Unaire	~	Complément à un NOT (bit à bit)

©Xavier Jeannin 2000-2005

Opérateurs bit à bit

- Tables de vérité :

Opérande 1	0	0	1	1
Opérande 2	0	1	0	1
Et &	0	0	0	1
Ou inclusif |	0	1	1	1
Ou exclusif ^	0	1	1	0

- L'opérateur ~ se contente d'inverser l'opérande.

©Xavier Jeannin 2000-2005

Opérateurs de décalage

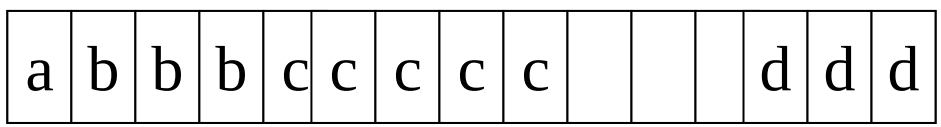
- << et >> permettent de réaliser des décalages sur le motif binaire du premier opérande et d'une amplitude définie par le second opérande.
  Par exemple : n << 2
  Les 2 bits de gauche sont perdus et 2 zéros apparaissent à droite.
- Pour le décalage droit, la valeur de remplissage dépend du signe de l'entier.

n << 2	signé	0111 0000 1000 0111 ==> 1100 0010 0001 1100
n << 2	signé	1111 0000 1000 0111 ==> 1100 0010 0001 1100
n << 2	non signé	0111 0000 1000 0111 ==> 1100 0010 0001 1100
n << 2	non signé	1111 0000 1000 0111 ==> 1100 0010 0001 1100
n >> 2	signé	0111 0000 1000 0111 ==> 0001 1100 0010 0001
n >> 2	signé	1111 0000 1000 0111 ==> 1111 1100 0010 0001
n >> 2	non signé	0111 0000 1000 0111 ==> 0001 1100 0010 0001
n >> 2	non signé	1111 0000 1000 0111 ==> 0011 1100 0010 0001

©Xavier Jeannin 2000-2005

Les champs de bits

- Derrière le deux points : on indique le nombre de bits, si on ne met pas d'identificateur, on saute le nb de bits correspondants.
- Seuls les types int et unsigned int peuvent apparaître dans les champs de bits.
- Dans l'exemple, mot.c désigne un entier compris entre -16 et 15
- Pas de norme pour indiquer le sens de remplissage.
- Taille max du champs 16 à 32 bits.
- Non portable.




```
Struct t_mot
    {unsigned  a : 1;
    unsigned  b : 3;
    int c : 5;
    int : 3;
    unsigned  d : 3;
    }mot;
```

Supplément sur les fonctions

Les arguments de la fonction main

- La fonction **main** ne dispose pas de véritable prototype. Son en-tête peut prendre deux formes :

  int main(void); /* forme usuelle */

  int main(int argc, char *argv[]); /* pour récupérer des arguments */

- #include <stdio.h>

```
int  main(int argc, char *argv[])
    {int i;
    printf("nom du programme %s \n", argv[0]);
    if(argc > 1)
      {for(i = 1; i < argc; i++)
                printf("argument %d  = %s \n", i, argv[i]);
      }
    }
```

©Xavier Jeannin 2000-2005

les fonctions à arguments variables

On déclare une fonction acceptant un nombre d'arguments variable :

float moyenne(int num, ...)

on transmet en général le nombre d'arguments.

Puis on récupère les arguments dans une variable de type **va_list**.

va_list arg_ptr;

Enfin, on extrait de la **va_list** les arguments un à un grâce à la fonction **va_arg**.

va_arg(arg_ptr, int);

on est obligé d'appeler **va_end** avant de quitter la fonction qui a fait le **va_start**.

va_end(arg_ptr);

note :

s'il n'y pas d'appel à va_end le comportement du pgrm est indéterminé, de même si il y a un appel de trop à va_end.

La fonction va_arg renvoie n'importe quoi lorsqu'il n'y a plus d'argument.

©Xavier Jeannin 2000-2005

```
float moyenne(int num, ...)
  { /* Déclarer une variable de type va_list. */
     va_list arg_ptr;
     int count, total = 0;

     va_start(arg_ptr, num);      /* Initialiser le pointeur vers les arguments. */
     for (count = 0; count < num; count++)  /* Récupérer chaque argument de
     la liste variable. */
        total += va_arg(arg_ptr, int);
     va_end(arg_ptr);              /* Donner un coup de balai. */


     /* Diviser le total par le nombre de valeurs à moyenner.
       Caster le résultat en float puisque c'est le type
       du résultat.  */
     return ((float)total/num);
  }
```

©Xavier Jeannin 2000-2005

Notion d'algorithmique et de structure de données

Référence :

- **Type de données et algorithme** edit. Ediscience international C. Froidevaux, M.-C. Gaudel, M. Soria
- **Eléments d'algorithmique** Edit. Masson D. Beauquier, J BERSTEL Ph. Chrétienne

Algorithme

- Un algorithme est un ensemble d'opérations de calcul élémentaires, organisés selon des règles précises dans le but de résoudre un problème

  Algorithme vient d'un mathématicien perse du 9ème siècle : Abu Ja'Far Mahommed Ibn Mûsa al-Khowa-rismî

- Exemple d'algorithmes
  - Une recette de cuisine
  - Un programme informatique
  - Tri, plus long chemin etc …
- Certains problèmes ne sont pas résolvables par des algorithmes
- Opérations élémentaires → non dépendantes des données
- Nombre fini d'opérations

Algorithme

- Complexité d'un algorithme en temps et en mémoire
  - Mesurée en temps de calcul / en nb d'opérations élémentaires
  - Jeu d'échec = 10^{19} coups, 10^{19} millisecondes = 300 millions d'années
  - Un polynôme s'exprimant en fonction du nb des données, par exemple : 1 tri élémentaire a une complexité proportionnelle au carré des éléments à trier.
  - Utilisation de la mémoire = nb instructions + nb données
- Surveiller les boucles,
  exemple de multiplication de matrice carrée : complexité ~ N^3

```
for(i = 1; i <= n ; i++)
    {for(j = 1; j <= n ; j++)
        {c[i][j] = 0;
         for(k = 1; k <= n ; k++)
                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
```

Algorithme

- Formalisme
  - **Si** *test* **alors** *action1* **sinon** *action2* **finsi**
  - **Tantque** *test* **faire** *action3* **fintantque**

Récursivité

- exemple :

```
long fac(int n)
    { if ( n  > 1) return (fac(n - 1) * n)) ;
      else return(1);
    }
```

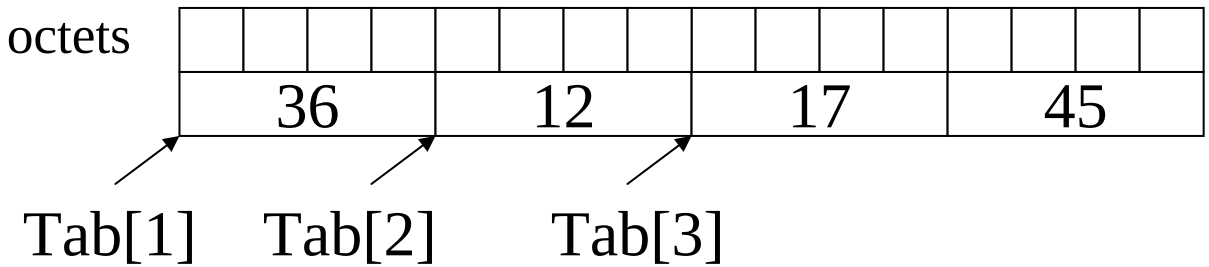
- Vérifier
  - Pas réapplication de la fonction à un ensemble de données plus grande
  - La terminaison de l'algorithme
- Permet l'expression concise d'un algorithme
- Tout algorithme récursif possède une version itérative

Structure de données

- Une structure de données est l'implémentation explicite d'un ensemble organisé de données, avec la réalisation des opérations d'accès, de construction et de modification.
- Un type de données abstrait est la description d'un ensemble organisé d'objets et des opérations de manipulation sur cet ensemble : accès, modification.
- En algorithmie, on utilise des types abstraits indépendants des langages
  - Expliciter comment les objets sont représentés et leurs méthodes
  - Parexemple :
    - Ensemble (liste contiguë Tableau)
    - Liste (liste contiguë Tableau, liste chaîné, liste doublement chaînée), Pile, File
    - Arbre
    - Graphe

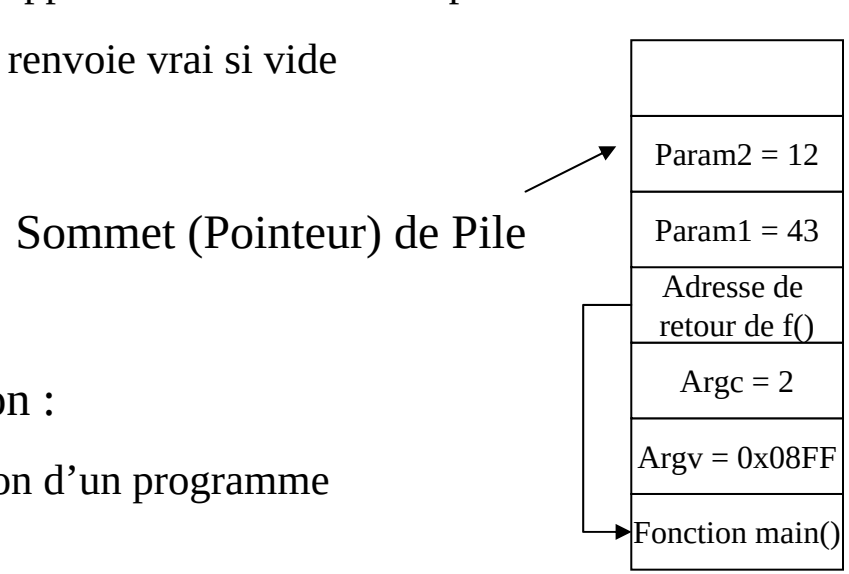
Structure de données

- Liste d'objets, un indice représente la place dans la liste
- On s'intéresse au deux côtés début et fin et aux opérations sur chacun des côtés
- Liste contiguë ou Ensemble : Tableau



Structure de données

- La pile et une liste où l'insertion et la suppression ne se fait que d'un coté
- Méthodes :
  - PileCreer renvoie une pile
  - Sommet(p : pile) renvoie le sommet de pile
  - Empiler(x : élément, p : pile) insère un élément dans la pile
  - Dépiler(p : pile) supprime un élément de la pile
  - PileVide(p : pile) renvoie vrai si vide



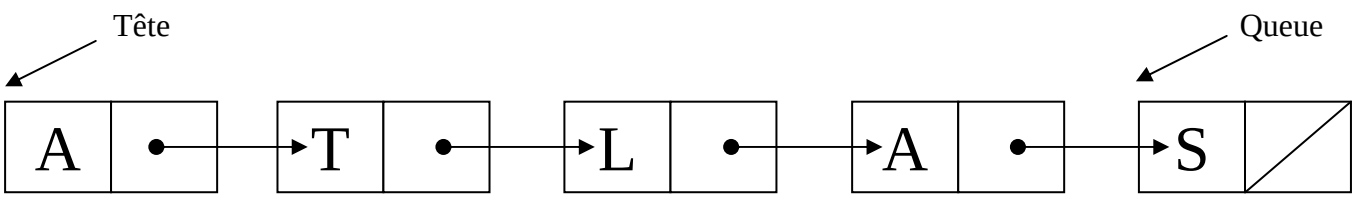

```
Int main(int argc, char *argv)
   {int i;
    printf(« bonjour »);
    f(i, 43);
    return 1;
   }
```

- Exemple d'utilisation :
  - La pile d'exécution d'un programme

©Xavier Jeannin 2000-2005

Structure de données

- La file est une liste où l'insertion se fait d'un côté et la suppression se fait de l'autre (FIFO First In First Out / LIFO Last In First Out)
- Méthodes :
  - FileCreer renvoie une file
  - Tête (f : file) renvoie la tête de la file / Queue (f : file) renvoie la queue de la file
  - Enfiler(x : élément, p : file) insère un élément dans la file
  - Défiler(f : file) supprime un élément de la file
  - FileVide(f : file) renvoie vrai si vide
- Implémentation : tableau, pointeur
- Exemple d'utilisation : FIFO Spool d'imprimante



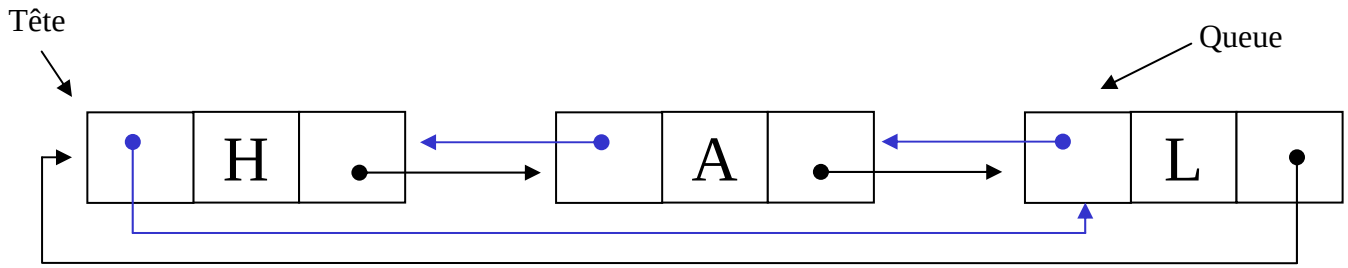
©Xavier Jeannin 2000-2005

Structure de données

- Une liste : l'insertion et la suppression se font non seulement aux extrêmités mais aussi à l'intérieur de la liste
- Méthodes :
  - FileCreer renvoie une file
  - Tête (f : file) renvoie la tête de la file / Queue (f : file) renvoie la queue de la file
  - InsérerAprés(x : élément, p : file, apres élément) insère un élément dans la file après un élément
  - InsérerAvant(x : élément, p : file, avant élément) insère un élément dans la file avant un élément
  - Suivant(x : élément, p : place) / Précedent(x : élément, p : place) renvoie l'élément suivant ou précedent
  - Supprimer(f : file, p :place) supprime un élément de la file
  - FileVide(f : file) renvoie vrai si vide

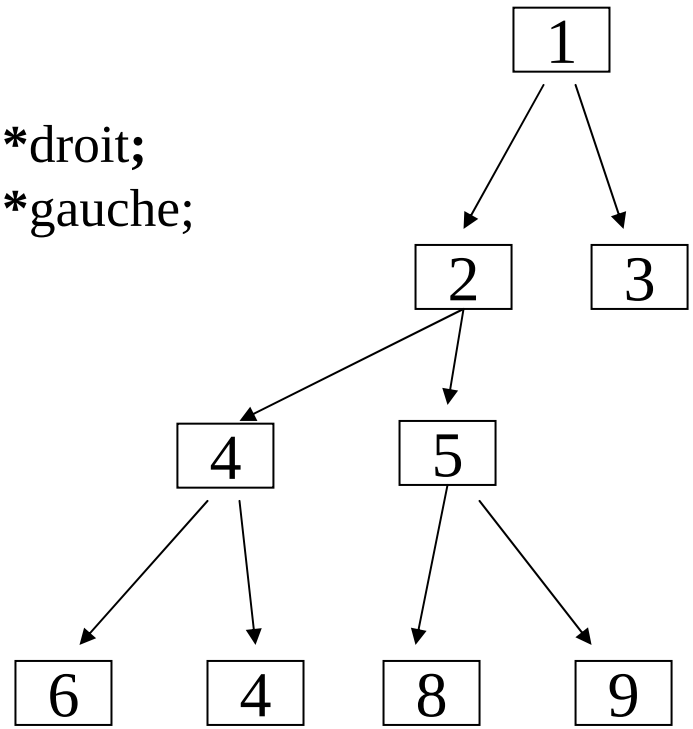
Structure de données

- Implémentation : pointeur, liste simplement ou doublement chaînée



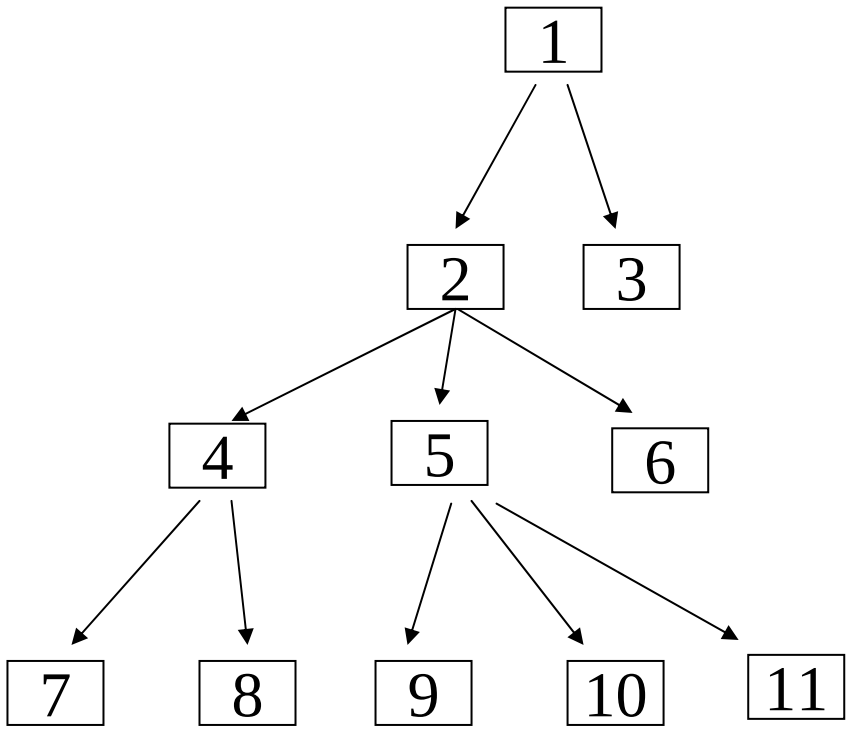
Structure de données

- Arbre binaire
- struct noeud
  {int val
  struct nœud ***droit;**
  **struct nœud ***gauche;
  };



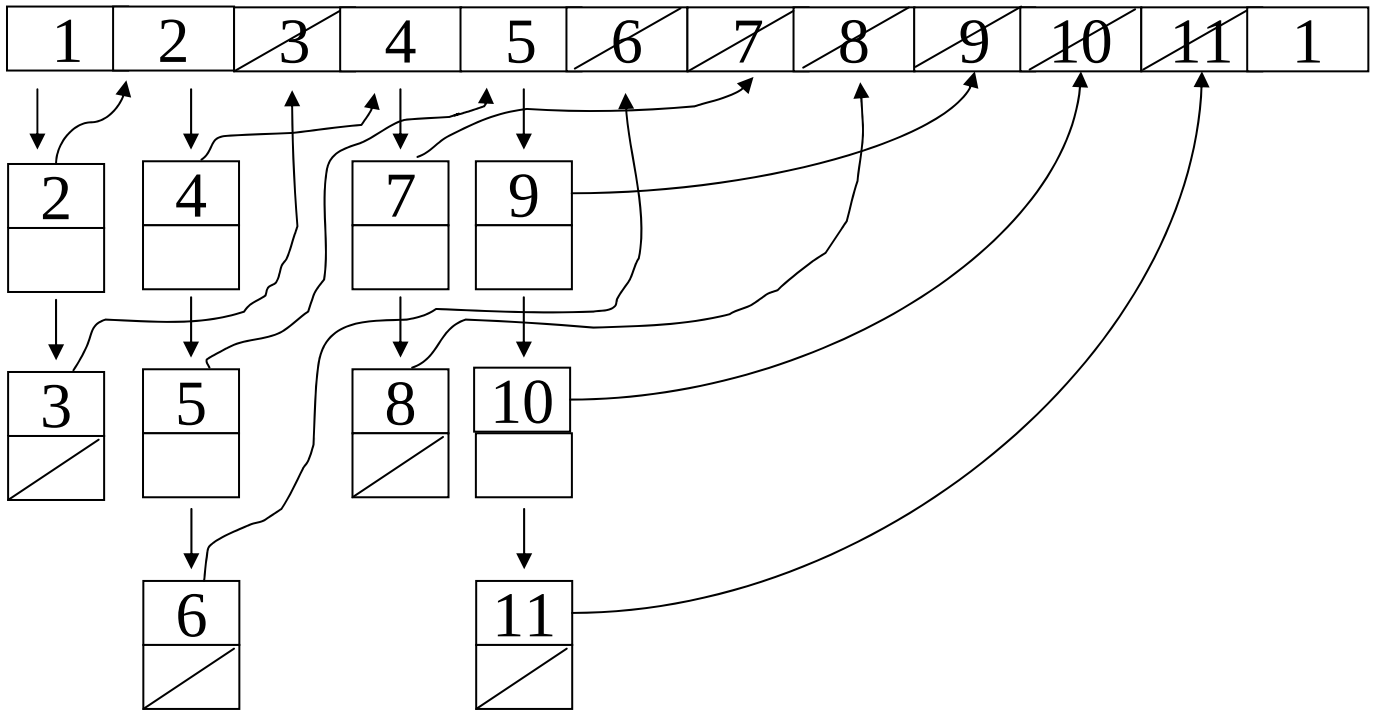
Structure de données

- Arbre quelconque



©Xavier Jeannin 2000-2005

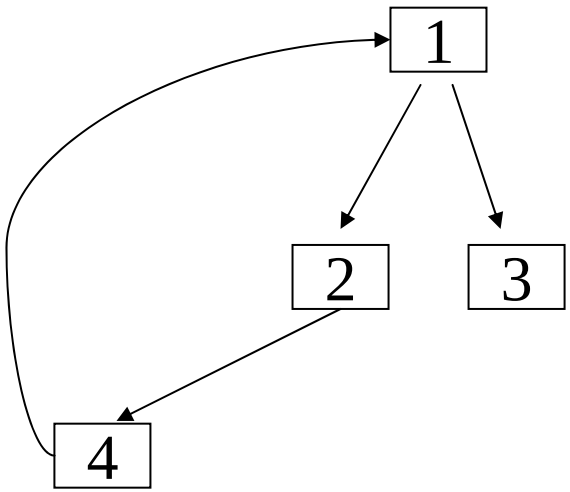
Structure de données



- Implémentation
  - 1 tableau de sommets
  - 1 liste chaînée par sommet

Structure de données

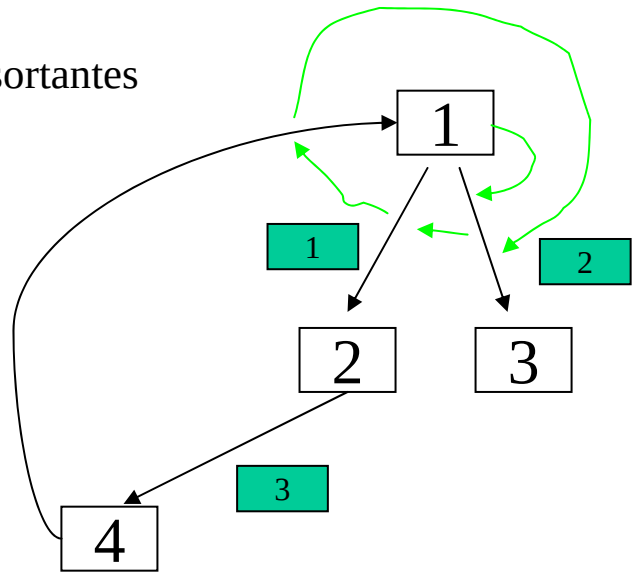
- Graphe est arbre avec des boucles



©Xavier Jeannin 2000-2005

Structure de données

- 1 tableau de sommets
- 1 tableau de premières arêtes sortantes
- 2 tableaux de demi-arêtes :
  - Arete[-3] = 2
  - Arete[3] = 4
- une liste chainée
  des demi-arêtes adjaçantes
  un sommet



1 arêtes

4 sommets

©Xavier Jeannin 2000-2005

Make
un "make" plutôt sympa

« tiré de la toile, citation perdue »

©Xavier Jeannin 2000-2005

Make

- Make permet d'automatiser des tâches sûres à effectuer pour le traitement d'un ensemble de fichiers (arborescence). On peut l'utiliser pour :
- Compiler et linker des programmes constitués de plusieurs fichiers et ceci avec n'importe quel compilateur fortran, C etc…
- Fabriquer des librairies.
- Archiver des programmes.
- Nettoyer une arborescence.
- Installer des programmes.

Make

- Il suffit de taper make sur la ligne de commande. L'utilitaire recherche et traite alors le fichier **makefile** (ou **Makefile**) présent dans le repertoire courant.

make

- Le fichier peut porter un autre nom que makefile ou Makefile, il doit alors être indiqué à l'aide de l'option -f

make -f make_mon_application

- Il y a bien d'autres options. L'option -n (``*no execution mode*'') permet de connaître les actions qui seraient effectuées mais sans les exécuter ce qui permet la mise au point du makefile.

Structure du fichier makefile

- Le fichier contient une suite de clauses de la forme:

cible: liste_de_dependances
    action_a_effectuer

- Attention, action_a_effectuer doit obligatoirement être précédée d'un caractère **tabulation**. Cette action_a_effectuer est réalisée par le make si :
  - La cible est un nom de fichier et sa date de mise à jour est antérieure à celle d'une des dépendances: action_a_effectuer est exécutée.
  - cible est un identificateur ou le nom d'un fichier qui n'existe pas: action_a_effectuer est exécutée.
- Concrètement, action_a_effectuer est une suite de commandes Unix.

Structure du fichier makefile

- Si il y a plusieurs clauses, make s'intéresse à la première cible du fichier. Il est aussi possible de préciser, sur la ligne de commande une liste de cibles à traiter.

```
make cible_1
make -f make_mon_application cible_2 cible_3
```

- make construit alors le graphe des dépendances pour la (les) cible(s) considérée(s) et traite récursivement tous les noeuds.
- # sont des commentaires.
- Il est possible d'insérer des messages qui seront écrits sur la <<sortie standard>> en utilisant la commande echo dans les action_a_effectuer et dernier point, make imprime les commandes exécutées sauf si elles sont précédées du caractère @ (d'où l'utilisation par la suite de @echo pour imprimer une fois seulement les messages).

Exemple 1

```
#
# fichier makefile pour le programme essai1
#
essai1 : essai1.c essai1.h
    gcc -o essai1 essai1.c
```

Exemple 2

```
#
# fichier makefile pour le programme essai2
#
essai2 : ess_type.o ess_util.o essai2.o
	@echo "edition des liens de essai2"
	gcc -o essai2 essai2.o ess_type.o ess_util.o
#
ess_type.o: ess_type.c ess_type.h
	@echo "compilation de ess_type"
	gcc -c ess_type.c
#
ess_util.o: ess_util.c ess_util.h
	@echo "compilation de ess_util"
	gcc -c ess_util.c
#
essai2.o: essai2.c ess_util.h ess_type.h
	@echo "compilation de essai2"

	gcc -c essai2.c
```

Les deboggers : GDB

exécution d'un programme pas à pas avec contrôle du cheminement, de la valeurs des variables.

GDB

- lancer l'application en mode <<débogage>> avec des points d'arrêt où l'on peut consulter les valeurs des variables, etc...et éventuellement relancer l'exécution pas à pas.
- l'application s'est terminée anormalement. Il y a alors génération d'un fichier core. dbx, en analysant ce fichier permet de connaître l'état du programme au moment où il s'est arrêté.

il convient de compiler les programmes avec l'option -g qui permet la génération des informations nécessaires au débogage:

gcc -g -o essai essai.c

on lancera ensuite, pour déboguer:

gdb essai

ou, si essai s'est terminé anormalement:

gdb core

- Les modules objets compilés avec l'option -g sont plus volumineux et le code généré plus lent à l'exécution. Il est préférable, une fois l'application mise au point de tout recompiler sans -g.

GDB

- quelques commandes de base:
  - `run` Pour commencer l'exécution du programme chargé par `dbx`.
  - `where` Pour savoir où le programme s'est arrété: numéro de ligne et <<trace des appels>>.
  - `print` Pour connaître les valeurs des variables.
  - `Stop` Pour insérer des points d'arrêt: suspendre temporairement l'exécution à un endroit précis.
  - `cont` Pour continuer l'exécution après un arrêt.
  - `step` Pour exécuter uniquement l'instruction suivante, entre dans les sous-routines.
  - `next` Pour exécuter uniquement l'instruction suivante sans entrer dans les sous-routines.
  - `Help` La liste des commandes.
  - `help cmd` Documentation de la commande `cmd`.
  - `quit` Pour sortir.

©Xavier Jeannin 2000-2005

GDB

- Il y a quelques commandes <<de base>> à connaître, les autres se découvrent au fur et à mesure des besoins grâce au <<`help`>>

- Utiliser gdb est un bon reflexe quand une application ne fonctionne pas, parfois meilleur (ou plus rapide) que de rajouter <<des printf>> partout dans le programme. Rappelons néanmoins qu'en C des instructions de débogage peuvent être insérées dans le source grâce à la compilation conditionnelle.

Notes sur les librairies : ar

• il existe des librairies statiques « libmath.a » ou dynamiques « libmath.so »
La convention de nom est lib« nom de la librairie ».a ou lib« nom de la librairie ».so

Exemple de création de librairie

- /* librairie libex.c*/

```
#include <stdio.h>
int  one()
        {
        printf(" premiere fonction " );
        }
int  two()
        {
        printf(" seconde fonction ");
        }
```

- /* include libex.h */

```
extern int  one()
extern int  two()
```

- Création de la librairie
  - gcc –c libex.c _Wall
  - ar rcs libxj.a libex.o (liste de .o …)

Exemple d'utilisation de la librairie

- /* programme main.c */

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

int  main(int argc, char *argv[])
        {one();
        Two();
        }
```

- Création de l'exécutable
  - gcc main.c –L/home/jeannin/src/ -lex –Wall
  - La convention de nommage implique que l'argument suivant –l se voit retirer son préfixe « lib » et son suffixe « .a » d'où « -lex »
  - On est obligé de completer le chemin de recherche de librairie «–L/home/jeannin/src/ »