Cours de C 1° année

Année 2001-2002

MBA, PBT, MCD, BDR, RFS, PMA

Professeurs

- M. Bastreghi (MBA) - coordinatrice
  mbastreghi@heb.be
- P. Bettens (PBT)
  pbettens@heb.be
- M. Codutti (MCD)
  mcodutti@heb.be
- B. Drion (BDR)
  bdrion@heb.be
- R. Fisset (RFS)
  roger.fisset@wanadoo.be
- P. Matsos (PMA)
  pmatsos@hotmail.com

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Présentation

- Contenu du cours
- Ce cours parmi les autres
- Modalités pratiques

Contenu du cours

- Qu'est-ce qu'un *langage de programmation* ?
- Historique des langages
- Historique du **C**
- Pourquoi le **C** ?
- Quel **C** va-t-on étudier ?

Langage de programmation

- Langage : « Ensemble de caractères, de symboles et de règles permettant de les assembler, utilisé pour donner des *instructions* à l'ordinateur. » (*Larousse*)
- Programme : « Séquence d'*instructions* et de données enregistrées sur un support et susceptible d'être traitée par un ordinateur » (*Larousse*)

Historique des langages

- Langage machine : ensemble de 0 et 1
- Langage d'assemblage : abstraction des instructions
- Langage de haut niveau (60'): abstraction des expressions
- Langage structuré (70'): abstraction des structures de contrôle
- Langage orienté objet (90'): abstraction des données

Historique du C

- **C** est un langage structuré
- Inventé par Kernighan & Ritchie
- Son sort fut lié à celui d'Unix
- Langage généraliste
- Lien étroit avec le système

Pourquoi le C ?

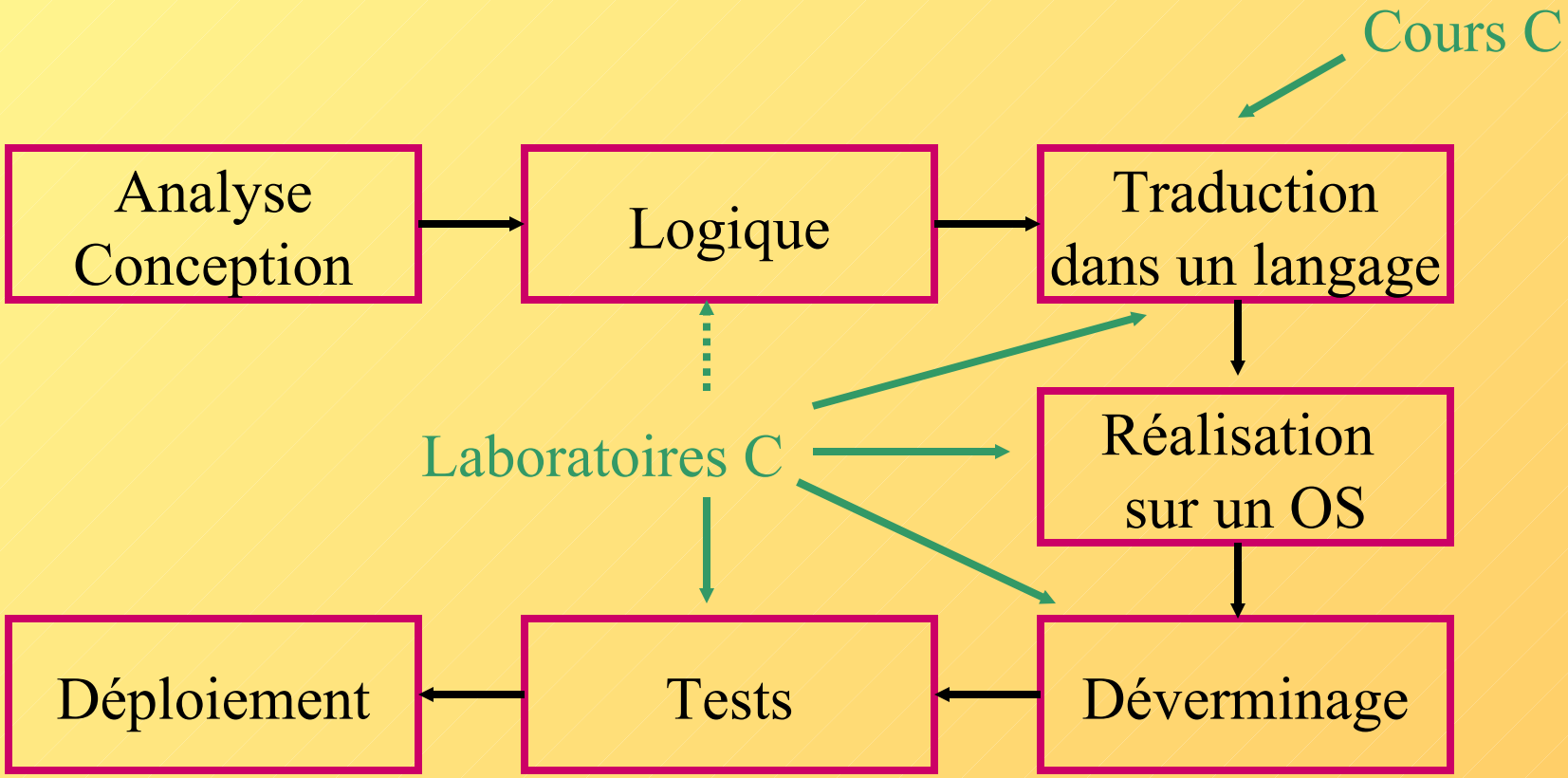
- Avantages
  - Répandu
  - Plaît au programmeur
  - Il a marqué toute une génération
  - Il a inspiré d'autres langages (C++, Java)
- Il a pourtant ses défauts
  - Trop permissif (typage faible par exemple)
  - Trop proche du bas niveau
  - Trop de cas particuliers

Quel C va-t-on étudier ?

- Norme ANSI-ISO de 1988
- On ne verra pas les particularités propres à un système (extensions Windows, Unix, …)
- On fera clairement la distinction entre ce qui est permis par le langage et ce qui est recommandé.

Liens verticaux

Phases du développement d'un projet



Liens horizontaux

Les autres langages

- Assembleur
- Java (Gestion)

Liens entre les années

- C++ en 2°
- Unix en 2°
- ***C*** se retrouve dans d'autres langages vus dans d'autres années

Modalités pratiques

- Rythme
- Supports & références
- Evaluations
- Attentes au niveau de la connaissance

Rythme

- Cours : 50 h à raison de
  - 3h / semaine au 1° semestre
  - 2h / semaine à la 1° moitié du 2° semestre

Sources & références

- Pages WWW concernant le cours
  - `http://home.tiscalinet.be/Marco-Codutti/ESI`
  - `\\Srv-etd\VERSETU\Distri\MCD\HomePage\index.html`
- Références
  - « La langage C; norme ANSI », Kernighan & Ritchie (2° édition, Dunod).
  - « Le livre du C premier langage », Delannoy (Eyrolles)

Evaluations

- Décembre
  - Interrogation, type QCM
  - 1/4 points
- Juin
  - Examen, *oral sur machine*
  - 3/4 points

Attentes

- Connaître la syntaxe du langage
- Savoir écrire un court programme sans faute
- Savoir écrire de plus gros programmes
- Savoir programmer *proprement*

Qualités d'un code source

- Maintenabilité !
  - lisibilité
    (commentaires, indentation, modularité)
- Robustesse
  - programmation défensive
    (test des cas "impossibles")
- Réutilisabilité
- Performance

La compilation

- Mon premier programme
- Compilation vs. interprétation
- Les phases de la compilation

Mon premier programme

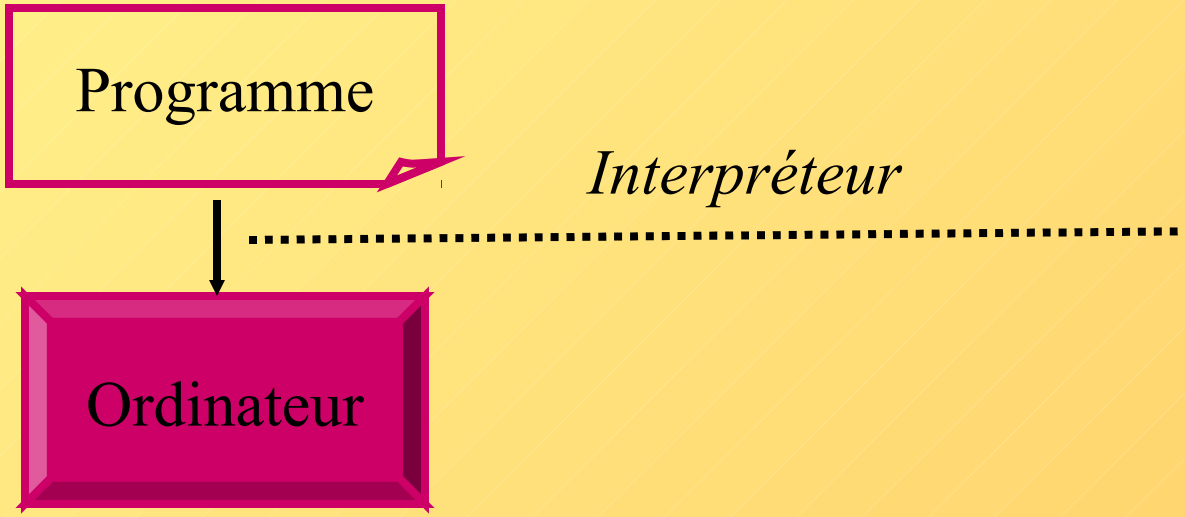
```
/* Mon premier programme */
/* Cours de C */
#include <stdio.h>
int main(void)
{
    printf("Bonjour\n");
    return 0;
}
```

Compilation vs interprétation

- Un ordinateur ne comprend que le langage machine (binaire)
- La traduction
  « autre langage » → « langage binaire »
  se fait
  - par compilation (cas du **C**)
  - par interprétation

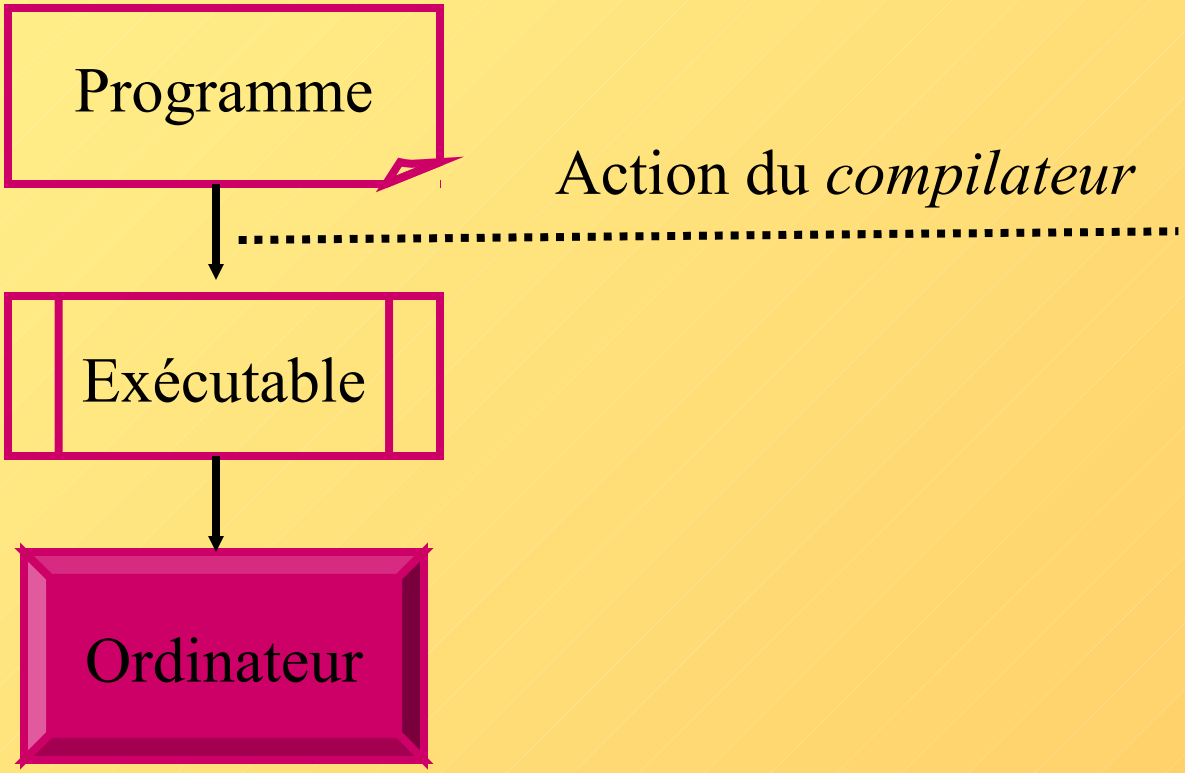
L'interprétation

Vue simplifiée



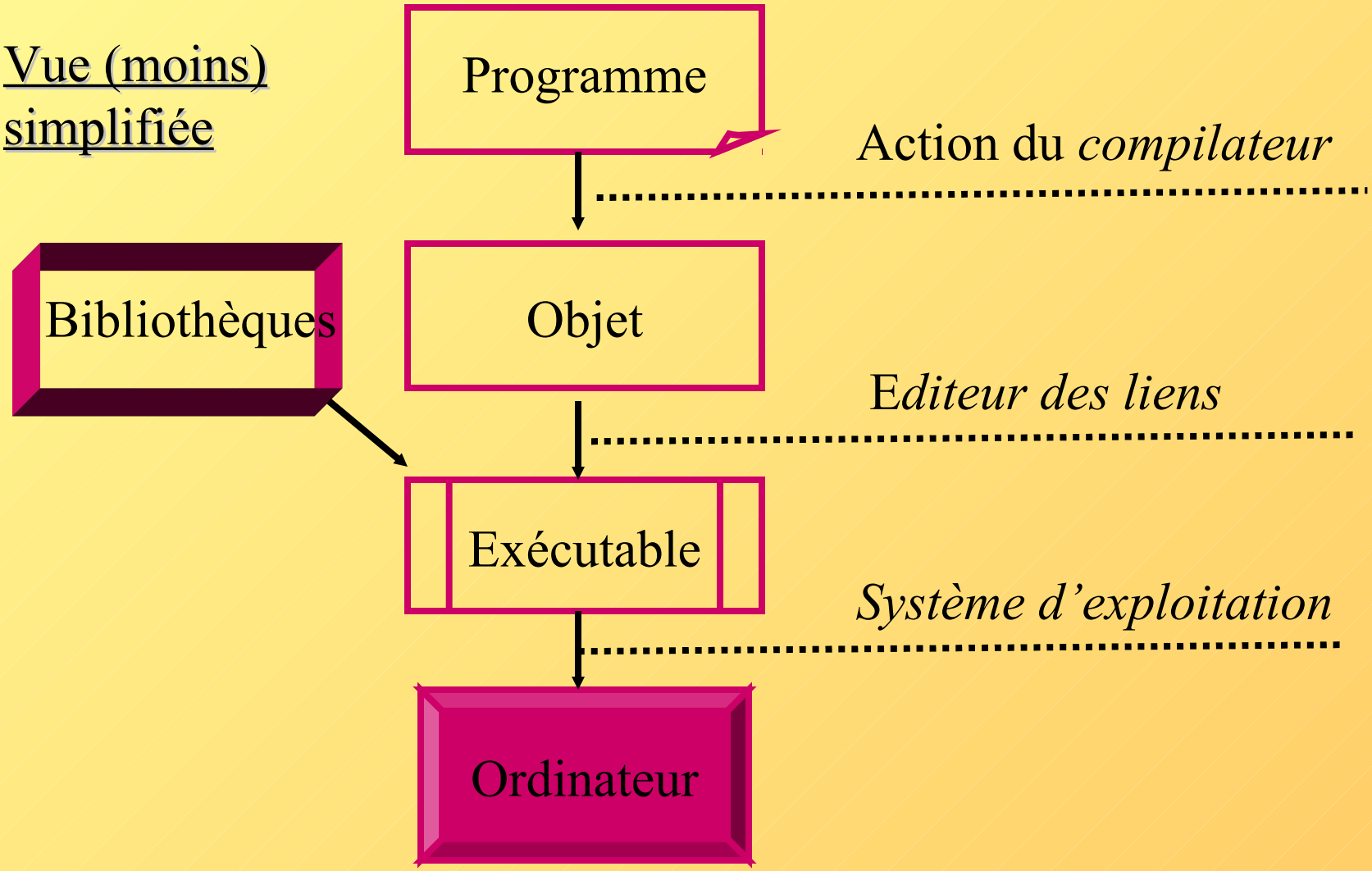
La compilation

Vue simplifiée



La compilation

Vue (moins) simplifiée



Vue d'ensemble

- Squelette d'un programme
- La mise en page
- Les commentaires

Squelette d 'un programme

- Inclusion de bibliothèques
- Définition de constantes
- Définition de types
- Déclaration des fonctions
- Fonction principale
- Autres fonctions

Exemples : Entête (Programme1.txt),
Fahrenheit (Programme2.txt)

Les commentaires

Syntaxe

- Entre /* et */
- Peut tenir sur plusieurs lignes
  Ex: /* début
  fin */
- Pas d'imbrication
  Ex: /* début /* suite */ hors com. */
- N'importe où tant qu'on ne coupe pas de mot
  Ex: a /*reçoit */ = 1;

Pourquoi commenter un code ?

- Un programme est lu souvent
- Pourquoi ?
  - pour comprendre ce qu'il fait
  - le corriger, compléter, modifier
- Par qui ?
  - par quelqu'un qui reprend le projet
  - par vous après une longue interruption
- Du texte est plus parlant que du code

Comment commenter un code

- En début de programme
  - quoi, qui, version, ...
- En début de chaque fonction
  - nom, paramètres, rôle, cas particuliers
- Avec les déclarations
  - expliciter le rôle des variables
- Dans le code
  - éclairer le sens d'un bout de code
- Exemple : Fahrenheit (Programme2.txt)

Comment commenter un code

- Exemple de mauvais commentaires

```
int f;          /* On déclare f entier */
f = 0;          /* On met 0 dans f */
```

- Pourrait être

```
int f;          /* f : température en  Fahrenheit*/
f = 0;          /* Température initiale = 0 */
```

- Mieux encore

```
int fahrenheit;     /* température à convertir */
fahrenheit = 0;     /* Température initiale = 0 */
```

Principe

- Le mise en page d'un code **C** est très libre au niveau du compilateur
- Seule règle : ne pas *couper* les *unités* (mots) du langage.
- Exemple : ceci est correct

```
#include <stdio.h>
int main(void){printf
("Bonjour\n");return 0;}
```

En pratique

- Pour la lisibilité, on s'impose plus de règles
  - une instruction par ligne
  - espaces pour « aérer » le code
  - indentation reflétant la structure
  - indentation si on coupe une instruction
  - utilisation massive des commentaires (pour autant qu'ils apportent de l'information !)

Indentation

- Indenter signifie décaler vers la droite une ou plusieurs lignes pour indiquer leur lien et leur dépendance vis-à-vis de lignes précédentes
- Exemple :

```
if (a==1)
{    /* Les 2 lignes suivantes font partie du if */
     printf("a=1\n");
     a = a + 1;
}
```

Indentation

- Il existe plusieurs *styles*. Voici les plus courants

```
if (a==1) {
      printf ( "a=1\n" );
      a = a + 1;
}

if (a==1)
{
      printf("a=1\n");
      a = a + 1;
}
```

```
if (a==1)
   {
   printf("a=1\n");
   a = a + 1;
   }
```

A vous de trouver le vôtre
MAIS
restez cohérents

Les types de base (1)

- L'entier
- Le flottant (pseudo-réel)
- Le caractère
- Le logique
- La chaîne

Introduction

- Il existe plusieurs types prédéfinis.
- Nous en présentons ici une sélection
- On présentera par exemple un seul des 3 types de réels.
- Nous présentons également des types non prédéfinis mais utiles (ceux qu'on retrouve en logique)

Principe

- Il existe plusieurs sortes d'entiers
  - taille différente
  - signé ou non
- Le plus courant est int
  (entier signé de taille moyenne)
- Le nombre d'octets en mémoire est dépendant du système

int

- Les littéraux
  - notation décimale : 12, -3, 47
  - notation octale : 014, -03, 057
  - notation hexadécimale : 0xC, -0X3, 0x2f
- Attention ! espaces interdits
  exemple : 10 000 est incorrect

Code portable

- Une commande comme
  int i; i = 1000000;
  peut fonctionner sur une machine et pas une autre (dépend de la taille !)
- Comment s'assurer d'un code portable ?
- La librairie limits.h définit les constantes donnant les valeurs extrêmes.
- Ex: INT_MIN, INT_MAX

Programme (Type1.txt)

sizeof

- Autre solution :
  utiliser l'opérateur sizeof
- sizeof (type) donne le nombre d'octets occupés par un type donné sur la machine en question.
- Ex: int t = sizeof(int);
  printf("int = %d octets ", t);

size_t

- En toute rigueur, sizeof ne retourne pas un entier mais une valeur de type size_t.
- Ce type cache un entier mais le norme ne précise pas lequel
- Cela peut avoir son importance lorsqu'on écrit cette valeur. Ceci peut ne pas marcher : printf("%d", sizeof(int));
- D'autres fonctions acceptent un argument ou retournent une valeur de type size_t.

Principe

- Il existe plusieurs sortes de flottants qui se différencient par la taille
- Le plus courant est double (flottant de taille moyenne)
- Le nombre d'octets en mémoire est dépendant du système

double

- Représente un flottant (numérique non entier) en double précision.
- Les littéraux
  - notation usuelle : 12.23, -3., .01
  - notation scientifique : 0.1223e2, -30E-1, 1E-2

Pseudo-réel

Tout n'est pas représentable

- On n'utilise pas le terme *réel* car tous les nombres réels ne sont pas représentables en machines.
  - non fini : 1/3 , Π
  - trop petit ou trop grand : 2E11234
  - trop de chiffres décimaux : 0.12345678913465789134649813246 5

Pseudo-réel

Tout n'est pas représentable

- Attention : même 0.1 n'est pas représentable de manière exacte dans la plupart des représentations de flottants (cf. norme ISO/IEEE)

Code portable

Obtenir un code portable

- Comme pour les entiers, une librairie, ici float.h, définit des constantes précisant les valeurs possibles.
- Plus compliqué que pour les entiers
- Ex: DOUBLE_MAX, DOUBLE _DIG, …

Principe

char

- Représente un *caractère* de la machine
- Occupe 1 octet en mémoire
- La configuration binaire de l'octet dépend du code utilisé (ASCII, EBCDIC,...)

Les littéraux

- Caractère entre ' ' : 'a', '1', '@'
- Caractères spéciaux :
  - '\n' fin de ligne
  - '\t' tabulation
  - '\'' apostrophe (''' serait ambigu)
  - '\\' barre ('\' serait ambigu)
  - '\"' guillemet
  - mais aussi \v, \b, \r, \f, \a, \?

Principe

- Attention : '3' ≠ 3
  - type ≠
  - représentation interne ≠
  - char c=3; est une erreur
  - int i= '3'; est une erreur

Principe

- En **C**, il n'**existe pas** de type **logique**
- On peut en définir un avec

  typedef enum {false, true} bool;
- Dès lors,
  - déclarer : bool var;
  - mettre à faux : var = false;
- Exemple : test de croissance d'entiers

Programme (Type2.txt)

Principe

- La chaîne **n'existe pas** en **C**
- On pourra la simuler via un *tableau de caractères* :
  Ex: char chaine [20]
  pour une chaîne de 19 (!) caractères
  (on verra pourquoi plus tard !)

Littéraux

- Il existe par contre un littéral *chaîne*
  Ex: "Je suis une chaîne"
- On utilise les caractères spéciaux :
  Ex: "\"Bonjour,\n\tle monde\""
  équivaut à la chaîne
  "Bonjour,
  le monde"

Littéraux

- Si la chaîne ne tient pas sur une ligne

```
"Je suis une chaîne un peu trop\
longue pour tenir sur une ligne"
```

ou encore

```
"Je suis une chaîne un peu trop"
"longue pour tenir sur une ligne"
```

Principe

- Attention : "A" ≠ 'A'
  - type ≠
  - représentation interne ≠
  - char c; c = "A" est une erreur

Les noms

- Où ?
- Syntaxe
- Conventions
- Restrictions

Où ?

- On utilise des noms pour
  - les variables
  - les constantes
  - les fonctions (modules de la logique)
  - les mots réservés du langage (for, if, ...)

- Caractères permis
  - les lettres : a, …, z, A, …, Z
  - les chiffres : 0, 1, …, 9
  - le caractère « souligné » : _
- Tout le reste est interdit
  - pas de lettre accentuée (vs Java)
  - pas de signe (- & * …)

Syntaxe

- Ne peut pas commencer par un chiffre
  ex: 1A, 423, 9_2 sont invalides
- minuscules ≠ majuscules
  ex: total ≠ Total ≠ TOTAL
- Exemples de noms valides
  salaire, nom_etudiant, i, nb1, MAX, Matrice, Entier2Flottant, …

Conventions

- Respect de « la notation hongroise »
  - variable : en minuscule
  - fonction : idem
  - type : commence par une majuscule
  - constante : en majuscule
- Pourquoi ? Permet de connaître l'utilisation d'un nom à sa simple lecture
  ex: max, max(…), Max, MAX

Conventions

- Si un nom est composé de plusieurs mots
  ex: total ventes (espace interdit)
  - soit on utilise le souligné (vieux C)
    ex: total_ventes, nom_professeur
  - soit on met en majuscule la première lettre des mots suivants (notation hongroise)
    ex: totalVentes, nomProfesseur
  - Simplement accolé les 2 mots est dangereux
    ex: nombrebis

Conventions

- Utiliser des noms explicites
  ex:totalVentes au lieu de totven
  nomClient au lieu de ncli
  - Avantages ? Plus lisible
  - Inconvénients ?
    - Plus long à taper (mauvaise raison)
    - lignes plus longues (→ parfois moins lisibles)
- Les abréviations courantes sont admises
  ex: nb, min, max

Conventions

- Ex : Comment comprendre ceci

sal = salh * nbh

A comparer à

salaire = salaire_horaire * nb_heures

Restrictions

- Le caractère _ est déconseillé en début de nom (réservé aux « noms internes »)
  ex: _nom1 (ok mais déconseillé)
- Pas de limite sur la taille des noms mais seuls les premiers caractères comptent
  - en général 31 caractères
  - sur certains systèmes, les noms « externes » doivent être plus courts

Restriction mainframe

MAINFRAME

- Pour les fonctions, les 8 premiers caractères comptent
  ex: calculerSomme ≡ calculerMoyenne

Comment gérer la restriction du mainframe ?

1) Soit réduire les noms
   ex: calcSomme ≠ calcMoyenne
   On perd en lisibilité

Restriction mainframe

Restriction du mainframe (suite)

2) Utiliser les #define

ex:

```
#define calculerSomme calcSomme
#define calculerMoyenne calcMoyenne
…
… calculerSomme(…) {…}
… calculerMoyenne(…) {…}
```

Plus long à taper mais on garde la lisibilité !

Programme (Nom1.txt)

Les variables

- Déclaration
- Assignation
- Conversion de type

Principe

Une variable est un nom

- représentant une zone mémoire *fixée*
- pouvant accueillir un type précis de valeurs
- dont le contenu peut changer tout au long du programme

Déclaration

- *déclaration* ≡

  type déclarateur [, *déclarateur*]* ;

  déclarateur ≡

  nom_var [= *valeur*]

- Exemples
  - int i;
  - char c = 'A';
  - int a, b = 1, d;

Où les déclarer ?

- Hors des fonctions : variables globales (pédagogiquement interdit)
- A l'intérieur d'un bloc ({…})
  - Souvent, le bloc d'une fonction
  - Au début, avant toute instruction
  - N'existe que dans ce bloc
- 2 variables de même nom :

  ok si dans des blocs ≠

Exemple de déclaration

```
int main(void)
{
    int i;
    i = 0;
    /* int j; interdit */
    return 0;
}
```

Exemple de déclaration

```
int main(void)
{
    int i;
    i = 0;
    {
        int j;
        j = 1;
    }
    /* j = 2; interdit: j n 'existe plus */
    return 0;
}
```

Exemple de déclaration

```
int main(void)
{
    int i;
    i = 0;
    {
        int i;
        i = 1; /* le i interne */
    }
    printf("%d\n",i); /* affiche 0 */
    return 0;
}
```

Valeur par défaut

- On va considérer que déclarer une variable ne lui donne pas de valeur par défaut
  (ce n'est pas tout-à-fait vrai car cela dépend du type de la variable : local/global
  mais on l'impose pédagogiquement)

Assignation

- Il faudra donc *assigner* une valeur à une variable avant de pouvoir l'utiliser.
  - Soit à la déclaration
  - Soit par assignation

Assignation

- Syntaxe

  nom_variable = expression;

- Expression est
  - un littéral
  - une variable
  - une expression générale avec des opérateurs
  - …

Conversion

- Lors d'une assignation, la variable et l'expression doivent être de même type
- Si ce n'est pas le cas, on peut demander une conversion de l'expression.
  ex : double d; d = (double) 2;
- Une conversion peut amener à une perte d 'information
  ex : int i; i = (int) 2.3;

Conversion

- On ne peut pas convertir tout en n'importe quoi
  ex: convertir une chaîne en entier n'a pas de sens (même si c'est accepté par C !!!)
- Une conversion explicite n'est pas toujours nécessaire car il y a parfois une conversion implicite mais on l'impose pédagogiquement.
  ex: d'entier à flottant

Les constantes

- Principe
- Les constantes du préprocesseur
- Les constantes du C
- Différences
- Pratique

Principe

- Une constante est un nom associé à une valeur
- Cette valeur ne pourra changer tout au long de la vie de la constante
- Deux sortes de constantes en **C**
  - les constantes du préprocesseur : #define
  - les constantes du **C** : const

Les constantes du préprocesseur

- Définies par

  #define NOM valeur

- Exemples

```
#define MAX 100
#define PI      3.1415
```

- Valable de la définition à la fin du fichier

Les constantes du C

- Ajouter const devant la déclaration
- Exemple
  const double PI = 3.1415;
- L'expression doit être constante (plus exactement, calculable au moment de la création de la constante)
- Même règles de visibilité que pour les variables (global, local, …)

Comparaison

- #define est compris par le préprocesseur, const par le compilateur
- #define est plus général
- #define est toujours global
- #define n'a pas de notion de type

Comparaison

- Une constante définie par const possède une grosse contrainte : elle ne peut être utilisée là où une constante est nécessaire (sic !)
  exemple : taille d'un tableau
  ⇒ intérêt limité

Pratique

- En C++, cette contrainte est enlevée et on préférera const mais en C #define sera souvent plus judicieux.

Les expressions numériques & caractères

- Les expressions entières
- Les expressions flottantes
- Les expressions caractères

Liste

Les expressions entières

- 5 opérateurs
  - Addition, Soustraction
  - Multiplication
  - Division entière, Modulo
- Des fonctions prédéfinies (stdlib.h)

Addition et soustraction

+ & -

- L'addition et la soustraction usuelle
- Opérateurs binaires & unaires
- En cas de dépassement de capacité ?
  Rien n'est défini par la norme !
  (*Ce sera vrai pour presque tout le reste*)
- Associativité : de gauche à droite
  ex: 3 - 2 - 1 signifie (3 - 2) - 1

Multiplication et division

* & /

- * : la multiplication usuelle
- / : la division entière
  ex: 7 / 2 vaut 3 et pas 3.5
- En cas de division par 0 ?
  Normalement, arrêt du programme

Exemple

```
/* Exemple d ’expression entière */
#include <stdio.h>
#define NB_LABOS  8
int main (void)
{
     int maxEleves ;
     maxEleves = 15 * NB_LABOS;
     printf ("Le nombre maximum   ");
     printf ("d’élèves est %d\n", maxEleves);
     return 0;
}
```

Le modulo

%

- Reste de la division entière
  ex: 7 % 2 vaut 1
- On a toujours : (a / b) * b + a % b ≡ a

Exemple

```
/* Exemple d'utilisation du modulo */
#include <stdio.h>
int main (void)
{
    int annee, anneeDansSiecle;

    scanf ( "%d", &annee );
    anneeDansSiecle = annee % 100;
    printf ( "%d", anneeDansSiecle );
    return 0;
}
```

Priorité et associativité

- Permet de déterminer l'ordre d 'évaluation
- Priorités et associativités

  + - (unaire) ⇐
  * / % ⇒
  + - ⇒

- Exemple : 3-7*10+9/2%4

  est compris comme 3 - (7*10) + ((9/2)%4)

La bibliothèque stdlib

stdlib.h

- Cette bibliothèque contient notamment quelques fonctions sur les entiers
  - int abs(int n) valeur absolue
- Ex : Valeur absolue (ExprNum1.txt)

Les expressions flottantes

- 4 opérateurs
  - Addition
  - Soustraction
  - Multiplication
  - Division
- Des fonctions prédéfinies (math.h)

Opérateurs algébriques

+, -, * & /

- Identiques aux versions pour les entiers
- Sauf / : division flottante
- Mêmes priorités et associativités que pour les entiers
- En cas de division par 0 ?
  Normalement, arrêt du programme

Exemple

```
/* Exemple : moyenne de 3 âges */
    #include <stdio.h>

    int main (void)
    {
        int age1, age2, age3;
        double ageMoyen;

        scanf ( "%d %d %d", &age1, &age2, &age3 );
        ageMoyen = (double) (age1 + age2 + age3) / 3.0;
        printf ( "%f", ageMoyen );
        return 0;
    }
```

La bibliothèque math

math.h

- Fonctions mathématiques de base
  - fabs(x) valeur absolue
  - sqrt(x) racine carrée ($x \geq 0$)
  - pow(x,y) x^y
  - exp(x) exponentielle : e^x
  - log(x) logarithme népérien : $\ln(x)$, $x \geq 0$
  - log10(x) logarithme en base 10 : $\log(x)$

La bibliothèque math

- Fonctions d'arrondi sur les double (le résultat est aussi un double)
  - ceil(x) le plus petit entier n ≥ x
  - floor(x) le plus grand entier n ≤ x
- Exemple :

```
ceil(3.3)  /* 4.0  */    floor(3.3) /* 3.0 */
ceil(-3.3) /*-3.0 */     floor(-3.3) /*-4.0 */
```

La bibliothèque math

- Fonctions trigonométriques
  (sur les double; angles en radians)
  - sin(x), cos(x), tan(x) sinus, ...
  - asin(x), acos(x), atan(x) arc sinus, ...
  - sinh(x), cosh(x), tanh(x) sinus hyperbolique, ...
- Exemple :

```
#define PI 3.141592654
sin( PI / 2 )   /* 1 */
acos( -1 )      /* PI */
```

Peut-on mélanger entiers & flottants ?

ex: 7 / 2.0

- Le langage le permet mais, **pédagogiquement, nous le refusons**
- Passer par des conversions explicites
  ex: (double) 7 / 2.0
- (*type*) est un opérateur aussi

Du plus prioritaire au moins prioritaire

opérateur	associativité
+ - (unaire) (*type*)	⇐
* / %	⇒
+ -	⇒

Principe

- Il n'existe pas d'opérateur propre aux caractères mais une bibliothèque (ctype.h) fournit des fonctions pour les manipuler

La bibliothèque ctype

- Test d'appartenance à une classe (retourne un booléen)
  - isdigit(c) c est un chiffre décimal
  - islower(c) c est une lettre minuscule
  - isupper(c) c est une lettre majuscule
  - isalpha(c) c est une lettre
  - isalnum(c) c est une lettre ou un chiffre décimal
  - isxdigit(c) c est un chiffre hexadécimal
  - isspace(c) c est un espace au sens large (espace, retour, tabulation, …)

La bibliothèque ctype

- Conversion de caractère (retourne un char)
  - toupper(c) convertit c en majuscule
  - tolower(c) convertit c en minuscule
- Dans les 2 cas, retourne c inchangé si ce n'est pas une lettre.

Les assignations

- Raccourcis d'écriture
- Incrémentation / décrémentation

Introduction

- Syntaxe

  nom_variable = expression;

- Expression est
  - un littéral
  - une variable
  - une expression générale avec des opérateurs
  - …
- Il existe des raccourcis d'écriture

Raccourcis

+=, -=, *=, /= & %=

- var += exp ≡ var = var + exp
- Idem pour les autres assignations
- Exemple :

```
int i;
i = 5;
i %= 2;        /* i vaut 1 */
i -= 2;        /* i vaut -1 */
```

Incrémentation, décrémentation

++ & --

- var ++ ≡ var += 1 ≡ var = var + 1
- var -- ≡ var -= 1 ≡ var = var - 1
- On peut aussi écrire ++ var & -- var
- Exemple :

```
int i;
i = 5;
i ++; /* i vaut 6 */
```

Incrémentation, décrémentation

Attention :

i++ & ++i ne sont pas tout-à-fait équivalents mais il n'y a pas de différence s'ils sont utilisés seuls, comme instruction.

Lecture & écriture (1° partie)

- Introduction
- Ecriture(printf)
- Lecture (scanf)

Introduction

- C dispose de fonctions standards pour écrire à l'écran et demander des informations au clavier.
- On voit ici 2 fonctions mais il en existe d 'autres
- On pourra étendre cela à d'autres périphériques (fichiers en général)

printf

Pour écrire un texte à l'écran.

```
printf( chaîne );
```

où chaîne est (presque) quelconque

Exemple :

```
printf( "Bonjour !");
printf( "J'ai dit : 'Bonjour'.");
```

printf

N’oublions pas les caractères « spéciaux »

Exemples :

```
printf( "J'ai dit : \"Bonjour \".");
printf( "Il a eu 0/20 ou 0\\20 ?");
```

Attention :

% doit être dédoublé car il joue un rôle particulier

```
printf( "Il faut 60%% pour réussir.");
```

printf

- Ne passe à la ligne que si c'est explicitement demandé

Exemples :

```
printf( "J'ai faim.");
printf( "très faim.");
```

écrira : J'ai faim.très faim.

printf

Exemples :

```
printf(  "Ceci apparaîtra "
         "sur la même ligne\n");
printf(  "Ceci \n sur 2 lignes.");
```

écrira :

```
Ceci apparaîtra sur la même ligne
Ceci
sur 2 lignes.
```

printf

Pour écrire un entier

```
printf( "%d", entier );
```

Exemples :

```
int i = 13;
printf( "%d", 7 );     /* 7 */
printf( "%d", i );     /* 13 */
printf( "%d", i+1 );   /* 14 */
```

printf

Il existe d'autres codes

- %d pour un int
- %f pour un double
- %c pour un char
- %s pour une chaîne (plus sûr)
- Rien pour le logique (bool)

printf

Exemples :

```
printf( "%f", 3.1415 );
printf( "%c", 'a' );
printf( "%c", '\n' );
printf( "%s", "Bonjour!\n" );
printf( "%s", "Il a eu 59% !" );
```

printf

- On peut mélanger texte et caractères de format.

```
printf( "PI vaut : %f.", 3.1415 );
/* PI vaut : 3.141500. */
```

(la fonction a un format d'affichage par défaut, par exemple : 6 chiffres décimaux)

printf

- On peut écrire plusieurs variables en une fois.

```
int a = 5, b = 7;
printf( "%d X %d = %d\n", a, b, a*b );
/* 5 X 7 = 35 */
```

printf

- ```
  int haut = 173;
  double poids = 68.5;
  printf(
      "Il mesure %d cm\n"
      "et pèse %f kilos\n", haut, poids
      );
  /*  Il mesure 173 cm

      et pèse 68.500000 kilos */
  ```
  ```

printf

Attention :

printf est beaucoup plus compliqué que cela (nombreux autres codes et options).

On y reviendra !

scanf

- Permet de lire des éléments formatés sur le clavier.
- On utilise le % pour indiquer le type de valeur qu'on veut lire.

scanf

Pour lire un entier

```
scanf( "%d", &var );
```

Exemples :

```
int i;
scanf( "%d", &i );
```

remarquer la présence du & car la fonction va modifier la valeur de i.

scanf

Il existe d'autres codes

- %d pour un int
- %lf pour un double
- %c pour un char
- %s pour une chaîne
- Rien pour le logique (bool)

scanf

- Exemples :

```
double poids; int taille;
char sexe, nom[20];
scanf( "%lf", &poids );
scanf( "%c", &sexe );
scanf( "%d", &taille );
scanf( "%s", &nom );
```

Pour une chaîne, on peut omettre le & :

```
scanf( "%s", nom );
```

Remarques

- Avant de lire une valeur (sauf un char), on passe les éventuels espaces (au sens large).
- La lecture d'un numérique s'arrête à la rencontre d'un caractère non valide
- La lecture d'une chaîne s'arrête à la rencontre d'un espace (au sens large)

- Exemples :

  avec l'exemple précédent, on peut taper

  68.50H 173Marcel

  ou 68.5H 173 Marcel

  ou 68.5H 173 Marcel

  ou 68.5H

  173

  Marcel

scanf

- MAIS PAS

  68.50 H 173Marcel

  ni 68.5H 173 Jean Michel

  ni 68.5H 1 73 Marcel

  ni H68.5 Marcel 173

scanf

- On peut lire plusieurs variables en une fois.

```
double poids;
int taille;
char sexe, nom[20];
scanf( "%f %c %d %s",
&poids, &sexe, &taille, &nom );
```

Attention :

scanf est beaucoup plus compliqué que cela (nombreux autres codes et options).

On y reviendra !

Cas du mainframe

Attention :

Sur le mainframe, nous ne travaillons pas en interactif. Le clavier et l'écran sont représentés par des cartes JCL

```
//GO.SYSIN DD *
  données
/*
//GO.SYSPRINT DD SYSOUT=Z
```

Les instructions simples & composées

- Les types d'instructions
- L'instruction simple
- L'instruction composée

Les types d'instructions

- La grammaire de **C** distingue 6 sortes d'instructions.

instruction ≡

instruction-expression

instruction-composée

instruction-de-sélection

instruction-d'itération

instruction-de-saut

instruction-étiquetée

l'instruction simple

- Appelée aussi instruction-expression
- Brique de base d'un code **C**.
- Elle se termine par un ;
- Sans être exhaustif
  - assignation
  - appel de fonction sans valeur de retour
  - l'instruction vide ;

l'instruction composée

- On dit aussi *bloc*
- Permet de regrouper plusieurs instructions
- On peut la retrouver partout où on peut avoir une instruction
- Cas connu : le corps d'une fonction
- Autres cas : corps de choix et boucles

l'instruction composée

- Syntaxe

instruction_composée ≡ { *déclaration* instruction** }

```
{
    déclaration_1
    …
    déclaration_N

    instruction_1
    …
    instruction_M
}
```

- N ≥ 0, M ≥ 0

l'instruction composée

- On peut y déclarer des variables
- Elles n'existent *que* dans ce bloc
- Elles peuvent avoir le même nom qu'une autre variable d'un autre bloc
- Ex : Blocs & localité (Instruction1.txt)

Relations et opérateurs booléens

- Les relations
- Les expressions booléennes

Principe

- 6 opérateurs : ==, !=, <, >, <=, >=
- *expression1 opérateur expression2*
- Le résultat est un booléen
- Exemples :
  - 3 <= 3 : true
  - (3 - 2) != (2 - 3) : true
  - i == j : dépend de i et j

Types valides

- Utilisables avec
  - Les entiers
  - Les flottants
  - Les caractères
  - Les booléens
- Attention : ne pas confondre

  i == j et i = j

Cas des caractères

- Comparer des caractères ?
  - Tester l'égalité ou la différence a un sens précis
  - Tester l'ordre dépend du code utilisé; peut varier d'un système à l'autre

    Dangereux.

Cas des logiques

- Comparer des booléens ?
  - Tester l'égalité ou la différence a un sens précis
  - Tester l'ordre
    - accepté par le compilateur
    - clairement défini et portable (faux < vrai)
    - pédagogiquement refusé

Du plus prioritaire au moins prioritaire

opérateur	associativité
+ - (unaire) (*type*)	⇐
* / %	⇒
+ -	⇒
< <= > >=	⇒
== !=	⇒

Principe

Les expressions booléennes

- 3 opérateurs
  - Négation (non)
  - Conjonction (et)
  - Disjonction (ou)

La négation

! : la négation

- Syntaxe : ! *exp_bool*
- ! true donne false
- ! false donne true
- Exemple : ! (2 < 3) est false

La conjonction

&& : la conjonction (et)

- Syntaxe : *exp_b1 && exp_b2*
-

&&	true	false
true	true	false
false	false	false

- Exemple : (2 < 3) && (2 <= 3) est true
- Attention : & existe mais avec d'autres sens

La disjonction

|| : la disjonction (ou)

- Syntaxe : *exp_b1* || *exp_b2*
-

||	true	false
true	true	true
false	true	false

- Exemple : (2 < 3) || (2 <= 3) est true
- Attention : | existe mais avec un autre sens

Du plus prioritaire au moins prioritaire

opérateur	associativité
! + - (unaire) (*type*)	⇐
* / %	⇒
+ -	⇒
< <= > >=	⇒
== !=	⇒
&&	⇒
||	⇒

Exemple

Exemple :

- Comment indiquer un tarif spécial pour les seniors (60 ou plus) et les étudiants (de moins de 26 ans)
- On a la variable entière age et la variable booléenne etudiant
- if (age >= 60 || age < 26 && etudiant)
  …

Instructions de sélection

- if
- if-else
- switch
- opérateur ?:

if

- N'effectuer une partie de code que si une condition est vérifiée.
- Syntaxe : *instruction_if* ≡
  if (*expr_bool*) *instruction*
- Si la valeur de l'expression est *vraie* l'instruction est exécutée puis on passe à la suite
- Sinon, on passe directement à la suite

if

- Style conventionnel si une seule instruction :

  if (*expression* **)**
  instruction

- Exemple : tester si nb négatif

```
if ( nb < 0 )
        printf( "Le nombre est négatif\n" );
```

- Style conventionnel si plusieurs instructions :

```
if ( expression )
{
        instruction 1
        …
        instruction n
}
```

if

- Exemple : tester si nb différent de 0

```
if ( nb != 0 )
{
        printf( "Le nombre %d", nb);
        printf( " est différent de 0 !\n");
}
```

- Attention : ceci est incorrect

```
if ( nb != 0 )
        printf( "Le nombre %d", nb);
        printf( " est différent de 0 !\n");
```

- Exemple : tester si nb nul

```
if ( nb == 0 )
        printf( "Le nombre est nul\n" );
```

- Attention : ne pas écrire

```
if ( nb = 0 )
        printf( "Le nombre est nul\n" );
```

qui sera accepté par le compilateur mais a un autre sens

- Attention : les parenthèses font partie de la syntaxe

```
if  nb = 0
    printf( "Le nombre est nul\n" );
```

est une erreur

if

- Trier 2 nombres nb1 et nb2 : on veut nb1< nb2

```
int nb1, nb2, t;
…
if (nb1 > nb2)
{
        t       = nb1;
        nb1     = nb2;
        nb2     = t;
}
```

```
int nb1, nb2;
…
if (nb1 > nb2)
{
        int t;

        t       = nb1;
        nb1     = nb2;
        nb2     = t;
}
```

if-else

- Effectuer 2 parties différentes en fonction de la véracité d'une condition.
- Syntaxe :

instruction_if_else ≡
if (*expression*) *instruction1*
else *instruction2*

- Si la valeur de l'expression est *vraie* l'instruction1 est exécutée
- Sinon, l'instruction2 est exécutée

if-else

- Style conventionnel :

```
if ( expression )
        instruction 1
else
        instruction 2
```

```
if ( expression )
{
        instruction 1
        …
        instruction n
}
else
{
        instruction 1
        …
        instruction m
}
```

if-else

- Style conventionnel :

```
if ( expression )
{
        instruction 1
        …
        instruction n
}
else
        instruction
```

```
if ( expression )
        instruction
else
{
        instruction 1
        …
        instruction m
}
```

if-else

- Exemple : tester le signe du nombre (version 1)

```
if ( nb < 0 )
        printf( "Le nombre est négatif\n");
else
        printf( "Le nombre est positif ou nul\n");
```

if-else

- Exemple : tester le signe du nombre (version 2)

```
if ( nb < 0 )
printf( "Le nombre est négatif\n");
else
if ( nb > 0 )
printf( "Le nombre est positif\n");
else
printf( "Le nombre est nul\n");
```

- Pas d'accolade car if-else est *une* instruction
  On peut les mettre pour la clarté

if-else

- Autre écriture

```
if ( nb < 0 )
        printf( "Le nombre est négatif\n");
else if ( nb > 0 )
        printf( "Le nombre est positif\n");
else
        printf( "Le nombre est nul\n");
```

- Présente les 3 cas au même niveau ce qui correspond mieux à la logique.

if-else

- Attention !

```
if ( nb1 < 0 )
        if ( nb2 < 0 )
                printf( "nb1 & nb2 >0\n");
else
        printf( "nb1 >= 0\n");
```

- FAUX ! Car le *else* se rattache toujours au if le plus proche (n'ayant pas encore de *else*)

```
if ( nb1 < 0 )
        if ( nb2 < 0 )
                printf( "nb1 & nb2 >0\n");
        else
                printf( "nb2 >= 0\n");
```

if-else

- Ou bien utiliser les parenthèses

```
if ( nb1 < 0 )
{
        if ( nb2 < 0 )
                printf( "nb1 & nb2 >0\n");
 }
 else
        printf( "nb1 >= 0\n");
```

if-else

- Pas de ; après un bloc
- Exemple : ceci est incorrect. Pourquoi ?

```
if (a < b)
{
printf ("ordonnés\n");
};
else
{
printf ("désordonnés\n");
}
```

switch

- Effectuer différents traitements en fonction de la valeur d'une variable
- Fonctionne avec des variables entières et caractères
- Permet de coder le « Choisir parmi » de la logique

switch

```
switch (expression)
    {
        case expression-constante :
                instruction 1
                ...
                instruction n
                break ;
        ...
        case expression-constante :
                instruction 1
                ...
                instruction m
                break ;
        default :
                instruction 1
                ...
                instruction p
                break ;
    }
```

switch

- Exemple :

  afficher les mois (Switch1.txt)

- La clause **default:** est facultative
- Que faire si plusieurs valeurs correspondent au même cas ?

  ex : nb de jours dans mois (Switch2.txt)

switch

- On teste n'importe quelle *expression entière*
  ex :

```
switch ( nbJours % 7 )
...
```

- Ca marche aussi avec les *caractères*
  ex:

Voyelles - consonnes (Switch3.txt)

switch

Attention :

Cette commande est plus complexe que cela. On y reviendra !

Principe

- Syntaxe : *exp_b ? exp1 : exp2*
- exp_b est une **expression** booléenne
- Si elle est true, le résultat est exp1
- Si elle est false, le résultat est exp2
- exp1 & exp2 sont des expressions quelconques de même type

Exemple

- Exemple :

```
int i, j;
scanf("%d", &i);
j = (i>0 ? i : -i );
```

- j sera la valeur absolue de i

Exemple

- Exemple :

```
int i, j, m;
scanf("%d%d", &i, &j);
m = ((i>0 ? i : -i ) > (j>0 ? j : -j )) ? i : j;
```

- m sera le nombre i ou j qui sera le plus grand en valeur absolue.

Du plus prioritaire au moins prioritaire

opérateur	associativité
! + - (unaire) (*type*)	⇐
* / %	⇒
+ -	⇒
< <= > >=	⇒
== !=	⇒
&&	⇒
||	⇒
?:	⇐

Différences entre ?: et if

- if est une instruction (pas de valeur)
- ? est un opérateur (valeur)
- Exemple :

```
j = (i>0 ? i : -i );
if (i>0)
        j = i;
else
        j = -i;
```

Instructions d'itération

- while
- do-while
- for

while

- Répéter une partie de code tant qu'une condition est vérifiée.
- Equivalent du « Tant que » en logique

while

instruction-while ≡
while (*expression*)
instruction

- Si l'expression est *vraie*, l'instruction est exécutée et on recommence une nouvelle fois l'exécution du while
- Si l'expression est *fausse*, l'instruction n'est pas exécutée et on passe à la suite

while

- Exemple : Calcul d'une moyenne

  Programme (Itération1.txt)

- L'instruction peut ne jamais être exécutée

  ex : while (false) …

- Création d 'une boucle infinie

  ex : while (true) …

while

- while (*condition*)
  instruction

L'instruction doit influencer la condition sinon boucle infinie.

Exemple :

```
while (nombre != -1)
{
    /* pas de modification de nombre */
}
```

do-while

instruction-do-while ≡
do *instruction*
while (*expression*);

- L'instruction est exécutée
- Puis, l'expression est évaluée
- Si elle est *vraie*, on recommence
- Si elle est *fausse*, on passe à la suite

do-while

do-while ≠ FAIRE-JUSQU'A (logique)!

- do-while :
  condition pour continuer
- FAIRE-JUSQU 'A :
  condition pour s 'arrêter

while vs do-while

Faut-il utiliser un *while* ou un *do-while* ?

- Un do-while exécute le corps au moins 1 fois.
-
```
do {
    printf("Entrez un entier positif : ");
    scanf("%d",&a);
    erreur = (n<=0);
    if (erreur) printf("Erreur !\n");
} while( erreur );
```

while vs do-while

- On aurait pu utiliser un while.
-
```
printf("Entrez un entier positif : ");
scanf("%d",&a);
while (n<=0)
{
    printf("Erreur !\n");
    printf("Entrez un entier positif : ");
    scanf("%d",&a);
}
```

for

- Permet de coder le pour de la logique
- Exemple :

  Ecrire les nombres pairs jusqu'à 20

  Logique : **pour** i **de** 2 **à** 20 **par** 2 **faire**

  écrire i

  fin pour

  C : `for ( i = 2; i <= 20; i = i + 2)`

  `printf( "%d\n", i);`

for

- Pas de pas par défaut
- Exemple : Compter jusqu'à 20

```
for ( i = 1; i <= 20; i++)
    printf( "%d\n", i);
```

- On appelle itérateur, la variable qui est modifiée à chaque étape.

for vs while

- for (<*début*>; <*test*>; <*pas*>)
  <instruction>

  est équivalent au code suivant

  <*début*>;
  while (<test>)
  {
  <instruction>
  <pas>;
  }

for vs while

- for (<*début*>; <*test*>; <*pas*>)
  <instruction>
- <*début*> est exécuté avant le processus d'itération (une seule fois)
- <*test*> est testé pour voir si on continue (même la toute première fois)
- <*fin*> est exécuté à la fin du corps de la boucle (avant de tester la condition)

for vs while

- Exemple

```
for ( i = 1; i < 10; i = i + 1)
    printf("%d\n",i);
```

est équivalent au code

```
i = 1;
while ( i < 10 )
{
    printf("%d\n",i);
    i = i + 1;
}
```

for vs while

Attention :

La commande for est plus complexe que cela. On y reviendra !

Les types de base (2)

- Les entiers
- Les flottants (pseudo-réel)

Principe

- 3 sortes d'entiers
  - short int
  - int
  - long int
- Se différencient par la taille

int

- Les littéraux
  - notation décimale : 12, -3, 47
  - notation octale : 014, -03, 057
  - notation hexadécimale : 0xC, -0X3, 0x2f
- Attention : espaces interdits
- Le nombre d'octets en mémoire est dépendant du système

long int & short int

long int

- On peut écrire long ou long int.
- Les littéraux : on ajoute l ou L à la fin
  ex: 12L, -3l, -0x3l, -1000000L

short int

- On peut écrire short ou short int
- Les littéraux : identique au type int
  ex: 12, -03, -0x3

Les entiers et le signe

- Par défaut, les entiers sont signés.
- Avec unsigned, l'entier est non signé.
- Le bit de signe est utilisé pour représenter plus de valeurs
- On peut indiquer explicitement signed
- On peut le combiner avec short et long
  Ex: unsigned short int, signed long.

Règles sur les tailles

- Norme :
  - Pas de nombre d'octets fixé pour les types short, int et long.
  - Mais doit respecter la contrainte

    short ≤ int ≤ long
  - De plus

    2 ≤ short, int 4 ≤ long

Règles sur les tailles

- Pour les non signés :
  2 x plus de valeurs dans les positifs
- MAINFRAME :
  - short : 2 octets ≈ ± 32 767
  - int : 4 octets ≈ ± 2 147 483 647
  - long : 4 octets ≈ ± 2 147 483 647

Code portable

Rappel :

- On peut se référer à la bibliothèque limits.h pour connaître les valeurs extrêmes sur la machine.
  Ex: SHRT_MIN, LONG_MAX
- On peut aussi utiliser sizeof pour connaître le nombre d'octets occupés par un type donné sur la machine

Code portable

- 3 sortes de flottants
  - float
  - double
  - long double
- Se différencient par la taille

double

- Représente un flottant (numérique non entier) en double précision.
- Les littéraux
  - notation usuelle : 12.23, -3., .01
  - notation scientifique : 0.1223e2, -30E-1, 1E-2
- Nombre d'octets en mémoire : DEPENDANT DU SYSTEME

float

- flottant en simple précision.
- Nombre d'octets en mémoire :
  DEPENDANT DU SYSTEME mais plus petit (donc moins précis) que double
- Les littéraux : idem avec f ou F à la fin.
  Ex: 12.23F, -3.f, .01F, 0.1223e2f

long double

- Représente un flottant en quadruple précision.
- Nombre d'octets en mémoire :
  DEPENDANT DU SYSTEME mais plus grand et plus précis que double
- Les littéraux : idem avec l ou L à la fin.

float ≤ double ≤ long double

Rappel :

- Tous les réels ne sont pas représentables en machine.
- Se référer à la bibliothèque float.h pour connaître la représentation sur la machine
- Utiliser sizeof pour connaître le nombre d'octets occupés par un type donné sur la machine

Lecture & écriture (2° partie)

- Introduction
- Ecriture(printf)
- Lecture (scanf)

Introduction

- Par défaut, on lit sur l'*entrée standard* et on écrit sur la *sortie standard*.
- Sous de nombreux systèmes, il s'agit du clavier et de l'écran.

printf

- Permet d'écrire des éléments formatés
- Déclaration : int printf(char [], …);
- Elle reçoit un nombre variable d'arguments (≥ 1)
- Le premier est toujours une chaîne de caractères qui sera écrite telle quelle sauf présence d'un %
- Exemple :
  printf("Bonjour.");

Introduction

- Le % indique qu'il faut, à cet endroit, écrire un des arguments suivants, dans l'ordre.
- Le signe % est accompagné d'autres caractères indiquant le type de l'argument à écrire, la taille à prendre, la précision, le cadrage, …
- Exemple : %d demande d'écrire un entier.

```
int i;
i = 10;
printf("i vaut %d\n", i);
```

Écrit: i vaut 10

Les entiers

Pour les entiers

d, i	entier décimal
o	entier octal non signé (sans le 0)
x, X	entier hexadécimal non signé (sans le 0x ou 0X)
u	entier décimal non signé

- Exemples :

```
printf("%d", 10);  /* 10 */
printf("%o", 10);  /* 12 */
```

Les entiers

Pour les entiers dans d'autres tailles

- On place un h devant pour indiquer un short
- On place un l devant pour indiquer un long
- Exemples :

```
printf("%ld", 10L);     /* 10 */
printf("%ld", 10);      /* Affichage incohérent */
printf("%hd", 10);      /* 10 */
```

Les flottants

Pour les flottants

f	flottant en notation décimale (float ou double)
e, E	flottant en notation scientifique
g, G	flottant dans la notation la plus appropriée

- On place un L devant pour indiquer un long double

Autres types

Autres

- c un caractère
- s une chaîne
- % le caractère %

- Exemples
  - printf("Augmentation de %d%%", 3);
    printf(" ce qui donne %g\n", -10010.34);
  - printf("Rapport de %d\\%d\n", 1, 3);

Nombre de paramètres

- Attention : Il faut respecter le nombre et le type des arguments passés à la fonction; aucun contrôle par le compilateur

  ex:

```
printf("%d", 1.3);
printf("%d%d", 1);
printf("%d", 1, 3);
```

Ecrire une chaîne

- Pour imprimer une chaîne (soit chaine),

  ```
  printf("%s", chaine);
  ```

  est plus sûr que

  ```
  printf(chaine);
  ```

  ex:

  si chaine vaut "Taux de 3% brut"

Format complet

- On peut encore préciser l'aspect du résultat
- L'écriture complète est *%[drapeaux][gabarit][précision][h|l|L] conversion*
  où
  - *drapeaux* : sont des options d 'affichage
  - *gabarit* : indique la taille de la zone à utiliser
  - *précison* : indique la « précision »
  - *h|l|L* : indique que la valeur à écrire est (short, long ou long double)
  - *conversion* : indique le type valeur à écrire (d, f , ...)
  - *[]* : indique que cette partie est facultative

Gabarit

- *%[drapeaux][gabarit][précision][h|l|L] conversion*
- *gabarit* est un entier
- Il donne la taille **minimale** du champ
  - si champ trop grand, on cadre à droite (espaces en tête)
  - si champ trop petit, on l'agrandit
- Exemples :

```
printf( "%3d", 12 );      /* ^12 */
printf( "%5s", "Oui" );   /* ^^Oui */
printf( "%3d", 1234 );    /* 1234 */
```

Précision

- *%[drapeaux][gabarit][précision][h|l|L] conversion*
- *précision* est un entier précédé du ‘ . ’.
- Il a une signification différente suivant le type de valeur
- Pour un flottant : nombre de chiffres décimaux à écrire (avec arrondi du dernier chiffre écrit)
- Par défaut, on a 6 chiffres
- Exemples :

```
printf( "%f", 3.1415 );      /* 3.141500 */
printf( "%.2f", 3.1415 );    /* 3.14 */
printf( "%.3f", 3.1415 );    /* 3.142 */
```

Précision

- Pour un entier : nombre de chiffres minimums à écrire (ajout de 0 en tête si nécessaire)
- Exemples :

```
printf( "%3d", 12 );  /* 012 */
printf( "%4.3d", 12 );        /* ^012 */
```

- Pour une chaîne : nombre maximum de caractères à écrire.
- Exemples :

```
printf( "%.3s", "Hello" );    /* Hel */
printf( "%5.3s", "Hello" );  /* ^^Hel */
```

Gabarit et Précision variables

- Les 2 options « gabarit » et « précision » peuvent être indiqués par « * » auquel cas la valeur est prise dans la liste des paramètres.
- Permet de paramétriser le code.
- Exemples :

```
printf( "%*d", 5, 234 );              /* ^^234 */
printf( "%*.*f", 8, 2, 3.1415 );      /* ^^^^3.14 */
#define N 10
char nom[N];
printf ( "%*s", N, nom );
```

Drapeaux

- *%[drapeaux][gabarit][précision][h|l|L] conversion*
- Les *drapeaux* possibles sont :
  - + : force la présence du + pour les nombres signés
  - (espace) : force la présence d'un espace à la place du signe +.
  - 0 : force le remplacement des espaces par des 0.
  - - : indique le cadrage à gauche (au lieu de la droite par défaut) pour tous les types

Drapeaux

- Exemples :

```
printf( "%5d", 314 );        /* ^^314 */
printf( "%05d", 314 );       /* 00314 */
printf( "%d", 314 );         /* 314 */
printf( "%+d", 314 );        /* +314 */
printf( "% d", 314 );        /* ^314 */
printf( "%-5d", 314 );       /* 314^^ */
printf( "%5s", "Oui");       /* ^^Oui */
printf( "%-5s", "Oui");      /* Oui^^ *
```

Exemples récapitulatifs

- Exemple : Impression de "Bonjour" avec différentes spécifications

```
:%s:        :Bonjour:
:%3s:       :Bonjour:
:%.3s:      :Bon:
:%-3s:      :Bonjour:
:%.9s:      :Bonjour:
:%-9s:      :Bonjour  :
:%9.3s:     :      Bon:
:%-9.3s:    :Bon      :
```

Exemples récapitulatifs

- Exemple : Impression de 3.1415

```
:%f:        :3.141500:
:%3f:       :3.141500:
:%.3f:      :3.142: /* ou :3.141: */
:%-3f:      :3.141500:
:%.9f:      :3.141500000:
:%3.9f:     :3.141500000:
:%9.3f:     :      3.142:
```

Exemples récapitulatifs

- Exemple : Impression de "3.1415"

```
:%-9s:      :3.1415    :
:%9.3s:     :      3.1:
:%-9.3s:    :3.1       :
```

scanf

- Permet de lire des éléments formatés sur l'entrée standard.
- Déclaration : int scanf(char [], …);
- Elle reçoit un nombre variable d'arguments (≥ 1)
- Le premier est toujours une chaîne de caractères qui indique ce qu'il faut lire
- A cet effet, on utilise le % comme pour le printf.

Introduction

- Le signe % est accompagné d'autres caractères indiquant le type de l'argument à lire et éventuellement son aspect
- Exemple : %d demande de lire un entier.

```
int i;
scanf("%d", &i);
```

- Attention : Remarquer la présence du & car il s'agit de modifier la valeur de la variable.

Les entiers

Pour les entiers

- d entier décimal
- i entier sous forme octale ou hexadécimale (précédé de 0, 0x ou 0X)
- o entier octal (avec ou sans le 0)
- x entier hexadécimal (avec ou sans 0x ou 0X)
- u entier décimal non signé

- On place un h devant pour indiquer un short
- On place un l devant pour indiquer un long

Les flottants

Pour les flottants

e, f, g flottant en notation décimale (float)

- On peut placer devant
  - un l indiquer que l'argument est un double (≠ printf() où il n'est pas nécessaire)
  - un L pour indiquer que l'argument est un long double

La chaîne

Pour les chaîne : s

- Attention : Le & est facultatif.
- Exemple

```
char chaine[20]
scanf("%s", chaine);
```

ou

```
scanf("%s", &chaine);
```

- A vous de choisir votre style.

Autres

- c un caractère
- % le caractère %

- Si la chaîne contient du texte (autre que %) celui ci doit se rencontrer tel quel en entrée.
- Exemples
  - scanf("%d %d", &i, &j);
  - scanf("%d + %d", &i, &j); /* ex: 2 + 3 */

Nombre de paramètres

- Attention : Il faut respecter le nombre et le type des arguments passés à la fonction; aucun contrôle par le compilateur

  ex:
```
int i; double d;
scanf("%d", &d);
scanf("%e", &d);
scanf("%d%d", &i);
scanf("%d", &i, &j);
```

Taille

Taille maximale du champ

- On peut intercaler un nombre entre le % et le format (%nC)
- Ce nombre donne la taille maximale du champ d'entrée
- ex:soit en entrée 2345.
  soit la déclaration int i,j;
  - scanf("%d %d", &i, &j); /* 2345 et rien */
  - scanf("%2d %2d", &i, &j); /* 23 et 45 */

Précisions sur la lecture

- Dans la chaîne de format, la présence d'un *caractère d'espacement* (espace, saut de ligne, tabulation, …) signifie : passer les (éventuels) *caractères d'espacements*

  ex: ces 3 formes sont équivalentes
  - scanf("%d %d", &i, &j);
  - scanf("%d %d", &i, &j);
  - scanf("%d \n %d", &i, &j);

Fin de chaîne

Précisions sur la lecture

- Une chaîne s'arrête à la rencontre d'un espace (sauf si taille spécifiée)
  ex: soit `Le cours de C` à lire

```
char chaine [20];
scanf("%s", chaine);        /* Lit : Le */
```

- Remarque : Pas de & pour une chaîne

Espaces

Précisions sur la lecture

- Lors de la lecture d'un nombre ou d'une chaîne, on passe les caractères d'espacements qui précèdent
  ex: ces 2 formes sont équivalentes
  - scanf("%d %d", &i, &j);
  - scanf("%d%d", &i, &j);

Espaces

- ex: soit le bout de code suivant

```
char s[20], c;
scanf(":%s%c:", s, &c);
printf(":%s:%c:", s, c);
```

  - avec `:Hello world:` on obtient
    `:Hello: :`
  - avec `:  Hello world:` on obtient
    `:Hello: :`
  - à comparer avec `scanf(":%s %c:", s, &c);`
    `/* :Hello:w: */`

Exercice

exercice: soit l'enregistrement contenant

- un nom sur 20 caractères
- l'âge (3 chiffres)
- le poids en kg (3 chiffres)
- un caractère indiquant le sexe (H ou F)
- le code postal de sa localité (4 chiffres)

ex: `Dumont,Pascal 032078H1080`

Comment lire tout cet enregistrement via un seul scanf ?

Exercice

ex: `Dumont,Pascal 032078H1080`

solution:

```
scanf("%20s%3d%3d%c%4d",
      s, &age, &poids, &sexe, &code);
```

Gérer les espaces

Comment gérer les espaces dans une chaîne ?

- On peut utiliser le format spécial %nc qui lit n caractères (espaces y compris) **mais** ne met pas de \0 à la fin
- Ou utiliser le format %[^\n] qui lit une chaîne en ne s'arrêtant qu'à la fin de ligne.
- ex: `Dumont, Pascal 032078H1080`

```
scanf("%20[^\n]%3d%3d%c%4d",
        s, &age, &poids, &sexe, &code);
```

Passer des valeurs

- On veut parfois passer des valeurs qui ne nous intéressent pas ?
- On utilise alors une * devant le code
- Exemple: scanf("%*d%d", &age);
  - lit un entier et n'en fait rien
  - lit un deuxième entier et l'assigne à la variable age

Valeur de retour

- Comment s'assurer du bon déroulement de la lecture ?
- scanf retourne le nombre d'arguments vraiment lus
- exemple:

```
nbLus = scanf("%d", &i);
if (nbLus < 1)
printf("Erreur!\n");
```

Conclusion

- Attention : Les fonctions printf et scanf possèdent encore plus d'options que celles vues (cf. référence)

Les fonctions

- Principes
- Mécanisme
- Portée, visibilité & durée de vie
- La fonction main

Principes

- Pourquoi découper un code en fonctions ?
- Comment découper un code en fonctions ?

Pourquoi découper un code ?

- Réutilisation
  - Code identique dans différents endroits du programme
  - Code déjà écrit « ailleurs »
- Scinder la difficulté
- Faciliter le déverminage
- Accroître la lisibilité
- Diviser le travail

Comment découper un code ?

- Une fonction
  - Résout un sous-problème bien précis
  - Est fortement documentée
  - Est la plus générale possible
  - Tient sur une page
  - N'a pas d'*effet de bord*

Mécanisme

- Définition
- Appel
- Utilisation
- Déclaration

Définition

- Syntaxe de la définition de fonction

définition-de-fonction ≡
type-valeur-de-retour nom-de-fonction
(*liste-de-paramètres-avec-types***)**
instruction-composée

Définition

- Exemple:

```
/* Additionne 2 entiers */
int somme( int nb1, int nb2 )
{
    int total;
    total = nb1 + nb2 ;
    return total;
}
```

Définition

- Un peu de vocabulaire:

```
/* Additionne 2 entiers */
int somme( int nb1, int nb2 )
{
    int total;
    total = nb1 + nb2 ;
    return total;
}
```

« signature »

« corps, texte »

- On parle indifféremment du type d'une fonction et de celui de sa valeur de retour

Définition

- Remarque : Une variable et une fonction peuvent avoir le même nom.

```
/* Additionne 2 entiers */
int somme( int nb1, int nb2 )
{
    int somme;
    somme = nb1 + nb2 ;
    return somme;
}
```

Définition

- Remarque :La position dans le programme source de la définition de fonction sera précisée plus loin dans l'exposé

Définition

- Si une fonction ne retourne rien, on indique void en place du type de retour
- Exemple :

```
/* Imprime un entête */
void entete( int numero )
{
   printf( "Laboratoire de C\n");
   printf( "Exercice n° %d\n", numero);
   return ;
}
```

Définition

- Si une fonction ne reçoit pas de paramètre, on indique void
- Exemple :

```
/* Imprime un entête */
void titre( void )
{
   printf( "Laboratoire de C\n");
   return ;
}
```

- Attention : void titre() a un autre sens

Définition

- Terminaison de l'exécution d'une fonction:
  - à la fin du bloc (pas de valeur retour)
  - à l'exécution d'une instruction return

- Syntaxe de l'instruction return

  instruction-return ≡

  return *[expression]*;

Définition

- On dit que la valeur et le type du return est la valeur et le type de son expression
- La fonction retournera la valeur du return
- Pas d'expression dans le return pour les fonctions ne retournant pas de valeur
- Le type de la valeur du return doit être le même que celui de la fonction

Définition

- On essaiera d'avoir un seul return, à la fin de la fonction
- Exemple :

Programme (Fonction1.txt)

Appel

- Une fonction est exécutée lors de l'appel de la fonction
- Syntaxe de l 'appel de fonction

  appel-de-fonction ≡

  nom-de-fonction **(***liste d'expressions***)**
- Exemples:
  - somme (i, 2)
  - somme (i+2, i/j)

Appel

- Un peu de vocabulaire:

« Paramètres effectifs »

```
somme( 3, 4 ); /* Appel */
```

« Paramètres formels »

```
int somme( int nb1, int nb2 )        /* Définition */
{
    int total;
    total = nb1 + nb2 ;
    return total;
}
```

Appel

- L'appel d 'une fonction entraînera successivement:
  - le passage des paramètres effectifs aux paramètres formels de la fonction
  - l'exécution du corps de la fonction
  - le passage de la valeur de retour de la fonction

Appel

- En **C**, le passage des paramètres se fait uniquement *par valeur* (en logique : *paramètre en entrée*)

- Le passage de paramètre *par valeur* signifie que les **VALEURS** des paramètres effectifs sont assignées aux paramètres formels correspondants

Appel

- On dit aussi que les valeurs des paramètres effectifs sont copiées dans les paramètres formels correspondants
- Le contenu des variables servant éventuellement de paramètre effectif est donc isolé de toutes modifications pouvant se produire dans le corps de la fonction

Appel

- Les paramètres effectifs donnés à l'appel doivent correspondre aux paramètres formels:
  - en nombre
  - en type
  - en ordre
- Exemple :
  - somme(3, 4, 5) /* Erreur */
  - somme('a', 'b')

Appel

- Si la définition ne prévoit pas de paramètres formels (void), on n'indique rien à l'appel
- Exemple:
  - titre()

Appel

- En **C**, le passage de la valeur de retour se fait uniquement *par valeur*

- Le passage de valeur de retour *par valeur* signifie que la **VALEUR** de l'instruction return constitue la valeur de l'appel de la fonction

Appel

- En **C**, on peut simuler le passage de *paramètre par adresse* (en logique : *paramètre en sortie*) par l'utilisation des pointeurs.
- Nous reviendrons sur cet aspect du mécanisme de passage de paramètre plus loin dans le cours après la présentation des pointeurs.

Appel

- Un appel de fonction est une expression.
- Le type de cette expression est celui de la fonction.
- Un appel de fonction peut être invoqué partout ou une expression du même type peut être utilisée.
- La valeur de cette expression est celle retournée par l'exécution de la fonction.

Du plus prioritaire au moins prioritaire

opérateur	associativité
()	⇒
! + - (unaire) (*type*)	⇐
* / %	⇒
+ -	⇒
< <= > >=	⇒
== !=	⇒
&&	⇒
||	⇒
?:	⇐
=	⇐

Utilisation

- Exemples:
  - somme(3, 4)

    expression entière, à 1 opérande, valant 7

  - int i = 2, j = 3;

    ...

    ..., somme(i + 2, j - 3), ...

    expression entière, à 1 opérande, valant 4

Utilisation

- Exemples (suite):
  - int i = 2, j = 3;

    ...

    ..., somme(i, j) * 2, ...

    expression entière, à 1 opérateur et 2 opérandes, valant 10

Utilisation

- Exemples(suite):
  - int i = 2, j = 3;
    ...
    ..., somme (somme(i, j), somme (j, i)), ...
    expression entière, à 1 opérande, valant 10
  - int s;
    ...
    s = somme(3, 4);
    expression entière formant une instruction simple, à 1 opérateur et 2 opérandes, valant 7

Utilisation

- Exemples(suite):
  - ...

    somme(3, 4);

    expression entière formant une instruction simple, à 1 opérande, valant 7, ce résultat est perdu
  - int s, i = 2, j = 3;

    ...

    s = i * j + somme(i + 2, j - 3);

    expression entière formant une instruction simple, à 3 opérateurs et 4 opérandes, valant 10

Déclaration

LE PROBLEME: contexte

- Un code source pouvant être réparti dans plusieurs fichiers, définition de fonction et appels correspondants peuvent se trouver dans des fichiers différents.
- Lorsque le compilateur analyse un code source, il le fait fichier par fichier indépendamment les uns des autres.

Déclaration

LE PROBLEME: contrainte

- Le compilateur doit pouvoir vérifier (fichier par fichier) la conformité de tout appel de fonction:
  - nombre d ’arguments
  - type des arguments
  - type de la fonction

Déclaration

LE PROBLEME: observation

- Ces informations se trouvent dans la signature d'une fonction

Déclaration

LE PROBLEME: stratégie de solution

- Tout appel de fonction doit être précédé dans la séquence de compilation d'un même fichier, au minimum, de la description de sa signature.

Déclaration

LE PROBLEME: solution du C

- Une *déclaration de fonction* précède tout appel correspondant dans la séquence de compilation d'un même fichier.
- La *définition de fonction* peut se faire dans n 'importe quel fichier du code source.
- Dans ce fichier, la *définition de fonction* peut cependant faire office de *déclaration de fonction*.

Déclaration

- Une *déclaration de fonction* sert à définir la signature d'une fonction

déclaration-de-fonction ≡
type-de-retour nom-de-fonction
(*liste-de-types***) ;**

Déclaration

- Exemples:
  - int somme(int , int);
  - void titre(void);
  - void entete(int);

Déclaration

- Une déclaration de fonction ne dispense pas d'une définition de fonction dans un des fichiers du code source
- Déclaration et définition de fonction doivent correspondre point à point
- On parle indifféremment de *déclaration de fonction* et de *prototype de fonction*

Déclaration

- Dans le contexte de l'école (code source sur un seul fichier) on recommande de
  - déclarer toutes les fonctions en début de code juste après les directives au précompilateur (sauf main)
  - définir d'abord la fonction main ensuite les autres fonctions (approche *top-down* vs approche *bottom-up*)

Déclaration

Exemple: Programme1

- La déclaration de fonction résout également le problème des appels croisés de fonction.

Déclaration

- Le même problème se pose pour les variables globales.
- L'usage de ces variables étant interdit pour des raisons pédagogiques, la solution à ce problème, retenue par C, ne sera exposée qu'en fin d'année lors de la présentation des variables globales.

Portée

- La zone de code source ou une variable existe s'appelle la *portée*.
- En C la *portée* d 'une variable est le bloc dans lequel elle est définie.
- La *portée* d 'un paramètre formel est le corps de sa fonction.

Portée

- Exemple:

```
int somme( int nb1, int nb2 )
{
   int total;
   total = nb1 + nb2 ;
   return total;
}
```

portée des variables nb1, nb2 et t

Portée

- Exemple:

```
{
    int nb1, nb2;
    if ( nb1 < nb2 )
    {
        int temp;
        temp = nb1;
        nb1 = nb2;
        nb2 = temp;
    }
}
```

portée de nb1 et nb2

portée de temp

Portée

- La définition d'une variable annonce ses propriétés et lui réserve de l'espace mémoire
- Lorsqu'on atteint la limite de sa portée, l'espace mémoire est récupéré; la variable est détruite

Visibilité

- Considérant les variables dont les *portées* recouvrent une instruction, une variable est *visible* de cette instruction si elle est la dernière définie parmi celles de même nom.
- Une instruction ne peut invoquer une variable que si elle lui est *visible*.

Visibilité

- Remarque:
  Une instruction ne pourrait invoquer une variable ne la recouvrant pas de sa *portée*; cette variable ayant déjà été physiquement récupérée ou n'ayant physiquement pas encore été créée lors de l'exécution de l'instruction.

Visibilité

- Remarque:
  Ne reste à régler pour les autres variables que les conflits d'homonymie par la règle de la dernière variable définie. En effet, deux variables de même nom peuvent exister dans des blocs différents.

Visibilité

- Pratiquement:

  Les variables *visibles* d'une instruction sont celles définies dans les blocs englobant cette instruction, les variables définies dans le bloc le moins englobant cachant les autres de même nom.

Visibilité

- Exemple:

```
{
   int nb1, nb2, temp;
   nb1 = 3; nb2 = 2; temp = 1;
   if ( nb1 < temp )
   {
      int temp;
      temp = nb1;  /* variables visibles de l 'assignation */
      nb1 = nb2;
      nb2 = temp;
   }
   printf("%d\n", temp); /* écrit 1 */
}
```

Visibilité

- Exemple:

```
{
   int nb1, nb2, temp;
   nb1 = 3; nb2 = 2; temp = 1;
   if ( nb1 < temp )
   {
      int temp;
      temp = nb1;  /* variables visibles de l 'assignation */
      nb1 = nb2;
      nb2 = temp;
   }
   printf("%d\n", temp); /* écrit 1 */
}
```

Visibilité

- Exemple:

```
{
   int nb, temp;
   if ( ... )
   {
      int temp;
      temp = nb;     variables visibles de l 'assignation
   } else
   {
      int temp;
      temp = nb;     variables visibles de l 'assignation
   }
}
```

Durée de vie

- Normalement, la durée de vie d'une variable se confond avec sa portée
- ex:

```
int somme( int nb1, int nb2 ) /* Additionne 2 entiers */
{
    int total;
    total = nb1 + nb2 ;
    return total;
}
```

A chaque appel nb1, nb2 et total sont crées puis détruits

Durée de vie

- L'option static accolée à une variable indique qu'elle ne doit pas être détruite
- L'initialisation lors de la déclaration n'est effectuée que la première fois
- ```
  void brol(void)
  {
      static int nb = 0;

      nb = nb + 1;
      printf("Appel numéro %d\n", nb);
      return ;
  }
  ```
  ```

Durée de vie

- Utile pour les fonctions qui doivent retenir des infos d'un appel à l'autre
- Cela ne modifie en rien la portée et la visibilité
-
```
void brol(void)
{
    static int nb = 0;

    nb = nb + 1;
    printf("Appel numéro %d\n", nb);
    return ;
}
/* nb ne peut pas être utilisée en dehors de brol() */
```

Durée de vie

- Vocabulaire : Les variables que l'on a rencontrées jusqu'à présent (sans static) s'appellent des variables automatiques

La fonction main

- La fonction main joue un rôle particulier.
- Il s ’agit de la fonction par où va débuter le programme; équivalent de l ’action en logique
- Un « programme » doit contenir une et une seule fonction main.
- Un « fichier source » ne contenant pas de fonction main sera compilé correctement mais devra être associé à une fonction main lors de l’édition des liens

La fonction main

- Comme toute fonction, main peut retourner une valeur : int main (void)
- Celle-ci est reçue par le système d ’exploitation (qui pourra la communiquer à l ’utilisateur)
- Souvent, la valeur
  - la valeur 0 indique un arrêt « normal » du programme
  - une autre valeur indique la rencontre d’un problème et pourra représenter le « numéro » de l ’erreur.

La fonction main

- La fonction main peut également recevoir des paramètres du système d ’exploitation (éventuellement reçue de l’utilisateur)
- Nous verrons ce mécanisme plus tard

La notion d'adresse

- Adresse d'une variable
- Le type « pointeur »
- Utilisation
- Exemples

Principe

- Une variable est stockée en mémoire
- Où exactement ? cela dépendra du genre de variable : locale, globale, dynamique. Cela ne nous importe pas ici.
- On peut demander l'adresse choisie pour stocker une variable.
- Cela permettra d'introduire la notion de pointeur.

Principe

- Si v est une variable,

  &v

  représente l'*adresse* de v.
- Attention :
  - &10 n'est pas permis
  - &(i+1) non plus

Pointeur

- On introduit un nouveau type de variable: le type pointeur.
- Une variable de type pointeur contient une adresse.
- Un pointeur pointe toujours vers une zone mémoire contenant un type précis d'information (int, char, ...)

Déclaration

- Pour déclarer un pointeur

  type *var;

  où type est un type quelconque
  var est le nom de la variable

- Exemples :

```
int *p;
char *pc;
```

Utilisation

- Soit la déclaration

```
int *p;
```

- p représente une adresse
  *p représente la valeur contenue à cette adresse

Utilisation

- Exemples

```
int v, *p;
v = 1;
p = &v;          /* p pointe vers v */
*p = 2;          /* v vaut 2 */
(*p)++;          /* v vaut 3 */
```

Utilisation

- & et * sont des opérateurs inverses

$$*\&v \equiv v$$

$$\&*p \equiv p$$

- Pour indiquer que p ne pointe vers rien pour le moment, on utilise NULL

```
int v, *p;
p = NULL;
*p = 1;          /* Erreur */
```

Écrire une adresse

- Il est possible d'« écrire » un pointeur via le code (%p)
- Exemple

```
int v,*ptr;
ptr = &v;          /* ptr pointe vers v */
printf("%p", ptr);
```

- Le format dépend du compilateur
- Cela n'a vraiment de sens qu'en phase de déverminage

Lire une adresse

- Il est également possible de « lire » un pointeur via le code (%p)
- Exemple

```
int v,*ptr;
scanf("%p", &ptr);
```

- Le format est compatible avec printf : on peut lire ce qui a été écrit avec printf.
- Cela n'a vraiment de sens qu'en phase de déverminage

Déclaration

- Il est possible de déclarer un pointeur sans préciser le type d'information pointée (on utilise void)

```
void *ptr;
```

- Dans ce cas, on ne peut pas utiliser (*) : *ptr provoque une erreur à la compilation.
- Cela pourra être utile dans certaines situations (nous en verrons l'une ou l'autre plus tard)

Pointeurs et fonctions

- Paramètres en sortie
- Lien avec les pointeurs

Rappel

- **C** : le passage de paramètre se fait uniquement *par valeur* (en logique, on dit : *paramètre en entrée*)

```
void brol (int i)
{
   i  = i + 1;
   return ;
}
```

```
int main(void)
{
    int i;
    i = 1;
    printf("%d\n",i);      /*1*/
    brol(i);
    printf("%d\n",i);      /*1*/
    return 0;
}
```

Paramètre en sortie

- On peut simuler le passage par adresse. En logique : *paramètre en (entrée-)sortie*

```
void brol (int *i)
{
   (*i) ++;
   return ;
}
```

```
int main(void)
{
    int i;
    i = 1;
    printf( "%d\n", i );   /*1*/
    brol( &i );
    printf( "%d\n", i );   /*2*/
    return 0;
}
```

Paramètre en sortie

- Ca marche même si les variables n'ont pas le même nom

```
void brol (int *i)
{
   (*i) ++;
   return ;
}
```

```
int main(void)
{
    int j;
    j = 1;
    printf( "%d\n", j );   /*1*/
    brol( &j );
    printf( "%d\n", j );   /*2*/
    return 0;
}
```

Paramètre en sortie

- Le paramètre effectif doit être une variable

  ex:

```
void brol (int *i)
{
   (*i) ++;
   return ;
}

int main(void)
{
    brol( &3 );
    return 0;
}
```

Paramètre en sortie

- La fonction reçoit, PAR VALEUR (une copie donc), l'adresse de l 'entier

```
void brol (int *i)
{
   (*i) ++;
   return ;
}
```

```
int main(void)
{
    int i;
    i = 1;
    printf( "%d\n", i );   /*1*/
    brol( &i );
    printf( "%d\n", i );   /*2*/
    return 0;
}
```

Les structures

- Principe
- Déclaration
- Utilisation
- Initialisation

Principe

- Type structuré pouvant regrouper plusieurs variables éventuellement de types différents sous un même nom.
- Quels sont les intérêts d'une telle pratique ?
  - Afin d'en faciliter la manipulation
  - Expliciter le lien entre des variables
  - Première étape vers l'abstraction des données
- On appelle champs ou membres les éléments de la structure.

Exemple

- Une date peut être vue comme étant composée de 3 éléments (le jour, le mois et l'année)
- On peut considérer le jour et l'année comme des entiers et le mois comme une chaîne (ou un entier).
- Ces 3 éléments se retrouveront souvent ensembles
- Les grouper explicitement montre mieux leur lien.

Déclaration

- Déclaration d'un type structuré

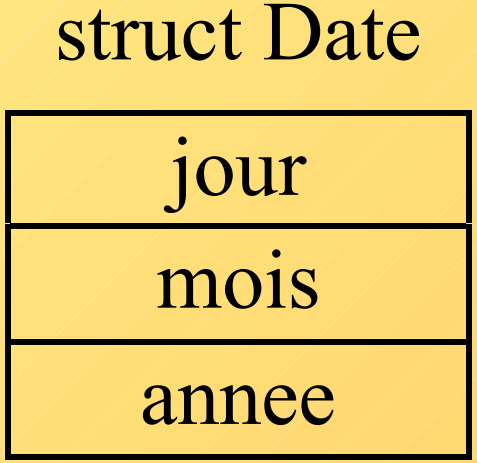
définition-de-type-structuré ≡
struct *[nom-de-type-structuré]*
{*[définition-de-membre]*[1]**}**
*[liste-de-variable]***;**

définition-de-membre ≡
*nom-de-type liste-non-vide-de-nom-de-membre***;**

Déclaration

Déclaration d'une structure (le type)

- struct Date
  {
  int jour;
  char mois[9];
  int annee;
  };



- Définit le type struct Date (comme int est un type)

Déclaration

- Déclaration d'une variable de ce type, variable structurée
- struct Date dateNaissance;

Déclaration

- On peut combiner déclaration du type et déclaration de la variable
-
```
struct Date
{
    int    jour;
    char mois[9];
    int    annee;
} dateNaissance;
```
- Attention : Pas du tout recommandé;

Où placer les déclarations ?

- Une déclaration de structure se placera en début de programme
(hors de toute fonction)
- Une variable structurée se déclarera comme toutes les autres variables **dans** les fonctions

Champ d'une structure

Référence à un champ d'une variable

- nom_stucture.nom_membre
- dateNaissance.jour = 12;
  dateNaissance.mois = "Novembre";
  dateNaissance.annee = 2001

DateNaissance

jour	12
mois	Novembre
annee	2001

Champ d'une structure

Attention : On indique bien le nom d'une variable structurée et pas le nom d'un type structuré

Exemple : Date.jour n'a pas de sens

Du plus prioritaire au moins prioritaire

opérateur	associativité
( ) [ ] .	⇒
! + - (unaire) (*type*) sizeof	⇐
* / %	⇒
+ -	⇒
< <= > >=	⇒
== !=	⇒
&&	⇒
||	⇒
?:	⇐

Restrictions sur les noms

- 2 champs de la même structure ne peuvent avoir le même nom
- 2 champs de structures différentes peuvent avoir le même nom
- 1 champ de structure peut avoir le même nom qu'une variable simple.

Initialisation à la déclaration

- struct Date date_naissance
  = { 12, "Novembre", 2001 };

date_naissance

jour	12
mois	Novembre
annee	2001

Assignation

- On peut assigner une structure dans sa globalité
- Exemple: avec

```
struct Date date1= { 12, "Novembre", 2001 };
struct Date date2;
```

- On a `date2 = date1;`

  est équivalent à

```
date2.jour = date1.jour;
date2.mois = date1.mois;
date2.annee = date1.annee;
```

Lire / écrire

- Il n'existe pas de code dans les fonctions standards (comme printf) pour lire/écrire une structure en une fois.
- Il faut donc lire/écrire les champs un à un.
- On pourra lire/écrire une structure dans sa globalité en binaire en temps que zone mémoire (cf. fread() / fwrite())_

Lire / écrire

- Exemple: avec

```
struct Date date1= { 12, "Novembre", 2001 };
struct Date date2;
```

On a

```
printf("%d%s%d", date1.jour, date1.mois,
date1.annee);
scanf("%d%s%d", &date2.jour, &date2.mois,
&date2.annee);
```

Structures complexes

- Les éléments d'une structure ne sont pas forcément des types de base.
  On peut avoir, par exemple
  - Des structures comprenant des structures
  - Des structures comprenant des tableaux
  - Des tableaux comprenant des structures
  - ...

Structures complexes

- On peut par exemple avoir des structures imbriquées

```
struct Personne
{
    char  nom[20+1];
    char  prenom[20+1];
    struct Date naissance;
};
struct Personne eleve;
```

- Indiquer que l'élève est né en 1983

```
eleve.naissance.annee = 1983;
```

Principe

Paramètres de fonctions

- Une structure peut apparaître aussi bien comme paramètre d'une fonction que comme valeur de retour
- Comme pour les autres types, le passage se fait *par valeur* (vs tableaux)
- Exemple : Programme (Structure1.txt)

Par adresse

- Comme pour les autres types, on peut passer l 'adresse de la structure pour imiter un paramètre « en sortie »
- C a introduit une simplification d'écriture
- Exemple :

```
struct Date *dateNaissance;
(*dateNaissace).jour
```

peut s'écrire

```
dateNaissace->jour
```

Par adresse

- Exemple :

```
void lireDate( struct Date *date)
{
        scanf("%d%s%d",
                &date->jour,
                &date->mois,
                &date->annee);
        return ;
}
```

La définition de type

- Principe
- Intérêts
- Fonctionnement
- Exemples

Principe

Qu'est-ce que c'est ?

- L'idée, non pleinement assumée en **C**, est de définir de nouveaux types de variables à partir de types existants.
- En **C**, cela se résume à donner un autre nom à un type déjà existant.

Intérêts

Quel est l'intérêt d'une telle pratique ?

- Raccourcir l'écriture
  (surtout valable pour les type complexes)
- Commentaire sur le contenu de variables
- Cacher les détails
- Détecter les incompatibilités de type
  (en théorie seulement)

Fonctionnement

- Syntaxe :

  définition_de_type ≡
  typedef ancien_type nouveau_type

- Exemples

```
typedef double    Poids;
typedef int       Age;

Poids poidsAvant, poidsApres;
Age agePere, ageMere;
```

Exemple

Détecter les incompatibilités de type

- Exemple :

```
typedef double Poids;
typedef double Taille;
Poids v1;
Taille v2;
v1 = v2;    /* N'a pas de sens logique */
```

- Attention : C ne produira pas d'erreur car il ne regarde que les types sous-jacents.

Principe

- typedef permet de simplifier l'écriture liée aux structures
- L'utiliser ou non sera une question de goût.

Fonctionnement

```
struct Date
{
      int     jour;
      char  mois[9];
      int     annee;
};
…
struct Date naissance;
```

```
typedef struct
{
      int     jour;
      char  mois[9];
      int     annee;
} Date ;
…
Date naissance;
```

Les fichiers

- Introduction
- Ouverture / Fermeture
- Lecture / Ecriture de haut niveau
- Fin de fichier
- Fichiers standards
- Lecture / Ecriture de bas niveau

Introduction

- Un fichier est un flot de données stocké sur un support informatique secondaire (disque dur)
- L'accès à ce flot de données se fait via des fonctions standards déclarées dans stdio.h
- On fait la distinction entre les fichiers textes et les fichiers binaires.
- On s'intéresse ici aux fichiers textes séquentiels.

Fichier texte vs fichier binaire

- Dans un fichier binaire, l ’information est représentée comme en mémoire
- Dans un fichier texte, elle est convertie sous forme « lisible ».
- Par exemple, un entier sera converti en décimal (ou en octal ou en hexadécimal)

Fichier texte vs fichier binaire

- Plaçons nous sur une machine ou les entiers sont représentés avec 2 octets et considérons l'entier 123.
- Sa représentation interne est (probablement) : 0x00 0x7B
- Sur un fichier binaire, on retrouverait ces 2 mêmes octets
- Sur un fichier texte, on retrouverait 3 octets correspondant aux codes de ' 1 ', ' 2 ' et ' 3 '.

Fichier texte vs fichier binaire

- Fondamentalement, cette distinction est peu pertinente en C ; un fichier texte peut être ouvert comme s'il était binaire et inversement.
- Au programmeur à savoir ce qu'il fait et à utiliser les bons ordres (relire en mode texte un fichier écrit en mode binaire a probablement peu de sens !)

Introduction

- En **C**, contrairement au cours de logique ou aux autres langages, il n'y a pas vraiment de notion de fichier structuré. On peut y lire/écrire des éléments hétéroclites. Au programmeur à savoir ce qu'il fait.

Introduction

- Pour accéder à un fichier
  - il faut d'abord l'ouvrir
  - on peut ensuite le lire et/ou y écrire (séquentiellement)
  - il faudra penser à le fermer après utilisation

Ouverture

```
#include <stdio.h>
FILE *fd;
fd = fopen( "nom_fichier", mode_acces);
```

- Pour ouvrir un fichier, il faut indiquer
  - le nom du fichier (dépend du système)
  - le *mode d'accès*
    - "r" : fichier en lecture; doit exister
    - "w" : fichier en écriture; (re)créé à l'ouverture
    - "a" : fichier en ajout; doit exister
    - ...

Ouverture

```
#include <stdio.h>
FILE *fd;
fd = fopen( "nom_fichier", mode_acces);
```

- fopen retourne un *descripteur de fichier*.
- Il sera utilisé par la suite pour accéder au fichier (lecture, écriture, fermeture)
- C'est une variable de type FILE * (définit dans stdio.h)

Ouverture

- Une ouverture peut parfois mal se passer (fichier inexistant, manque de permission, ...)
- Si l'ouverture est impossible, la fonction retourne NULL. Il faut toujours tester ce cas
- ```
  fd = fopen( "nom_fichier", mode_acces);
  if (fd == NULL)
  {
      printf("Erreur ouverture de fichier\n");
      exit(1);
  }
  ```
  ```

Ecriture

Ecriture dans un fichier

fprintf(fd, "format", …);

- Ex : fprintf(fd, "%d", 10);
  écrit 10 sur le fichier désigné par fd
- La syntaxe est la même que pour la fonction printf avec le descripteur de fichier en tête
- Le fichier doit avoir été ouvert en écriture

Lecture d'un fichier

nb = fscanf(fd, "format", …);

- Ex : fscanf(fd, "%d", &i);
  lit un entier sur le fichier désigné par fd et le place dans la variable i
- La syntaxe est la même que pour scanf
- Retourne le nombre d'éléments lus
  (permet de savoir si tout s'est bien passé)

Fermeture

Fermeture d'un fichier

```
fclose( fd );
```

- Attention : si on oublie de fermer un fichier avant que le programme ne se termine, il est possible de perdre de l'information.

Fin de fichier

- Lors d'une lecture de fichier, il est possible de savoir si on est arrivé en fin du fichier via la fonction : feof(fd);
  bool feof(FILE *);
- Retourne true si on est en fin de fichier

Fin de fichier

- Attention : en C, on est en fin de fichier après avoir tenté de lire au-delà du fichier.
- Exemples :

  Avec fin de fichier (Fichier1.txt)

  Sans fin de fichier (Fichier2.txt)

MAINFRAME

- On a 2 possibilités pour indiquer le nom du fichier dans fopen
  - Désignation directe du nom entre apostrophes
    fopen("'ANDR.DATA'", mode);
  - Appel à une carte JCL
    fopen("DD:FICH", mode);
    ```
    //GO.FICH DD DSN=ANDR.DATA,DISP=SHR
    ```
    cf. cours de Système pour les options
- On recommande la deuxième forme.

Code portable

- Tout bon programmeur aura la prétention d'avoir un code portable.
- Comment avoir un accès fichier le plus portable possible ?
- Via un #define.
  On limite ainsi les modifications.
- Exemple : Programme (Fichier3.txt)

Fichiers standards

- Au début du programme, 3 *fichiers* sont ouverts automatiquement
  - stdin : le fichier d'entrée standard (clavier)
  - stdout : le fichier de sortie standard (écran)
  - stderr : le fichier d'erreur standard (écran)
- L'OS définit à quoi ils correspondent mais on peut aussi le contrôler.
- Pas d'ouverture ou de fermeture

Fichiers standards

- Ainsi,

```
printf( "%d", n );
```

est strictement équivalent à

```
fprintf( stdout, "%d", n );
```

- Ou encore,

```
scanf( "%d", &n );
```

est strictement équivalent à

```
fscanf( stdin, "%d", &n );
```

Fichiers standards

- Sous le mainframe IBM, il s'agit du SYSIN et du SYSPRINT

```
//GO.SYSIN DD *
   données
/*
//GO.SYSPRINT DD SYSOUT=Z
```

- Les données sont indiquées comme si on les tapait au clavier

Accès bas niveau

- Il existe d'autres fonctions pour accéder aux fichiers
- Celles-ci ont une vision plus *basse* du fichier
- Il sera considéré comme une suite de caractères

Accès bas niveau

fgetc

- Lit un caractère sur un fichier
- Déclaration : int fgetc (FILE *fd);
- La fonction retourne EOF si on tente de lire au delà de la fin du fichier.
- EOF est défini dans <stdio.h>

Accès bas niveau

fgetc

- Attention : retourne un int et pas un char
- On ne peut le convertir en char que si ce n ’est pas EOF.
- Exemple :

  Compter lignes d’un fichier (Fichier5.txt)

Accès bas niveau

fputc

- Ecrit un caractère sur un fichier
- Déclaration : int fputc (int c, FILE *fd);
- Attention : reçoit un int et pas un char
- La fonction retourne le caractère écrit ou EOF si il y a eu un problème.
- Exemple : Copie de fichier (Fichier4.txt)

Accès bas niveau

Comparons fputc(car, fd) et fprintf(fd,"%c",car)

- Elles sont identiques
- Mais la deuxième est un peu plus lente car elle doit d'abord interpréter le format pour savoir quoi faire.

Idem pour fgetc(fd) et fscanf(fd,"%c",&car)

Accès bas niveau

- On dispose de macros équivalentes.
  - getc(fd) ≡ fgetc(fd)
  - putc(car, fd) ≡ fputc(car, fd)
- Comme ce sont des macros, c'est
  - légèrement plus rapide
  - légèrement plus dangereux (effets de bord possibles)

Accès bas niveau

- On dispose d'ordres équivalents pour les fichiers standard.
  - getchar() ≡ fgetc(stdin)
  - putchar(car) ≡ fputc(car, stdout)

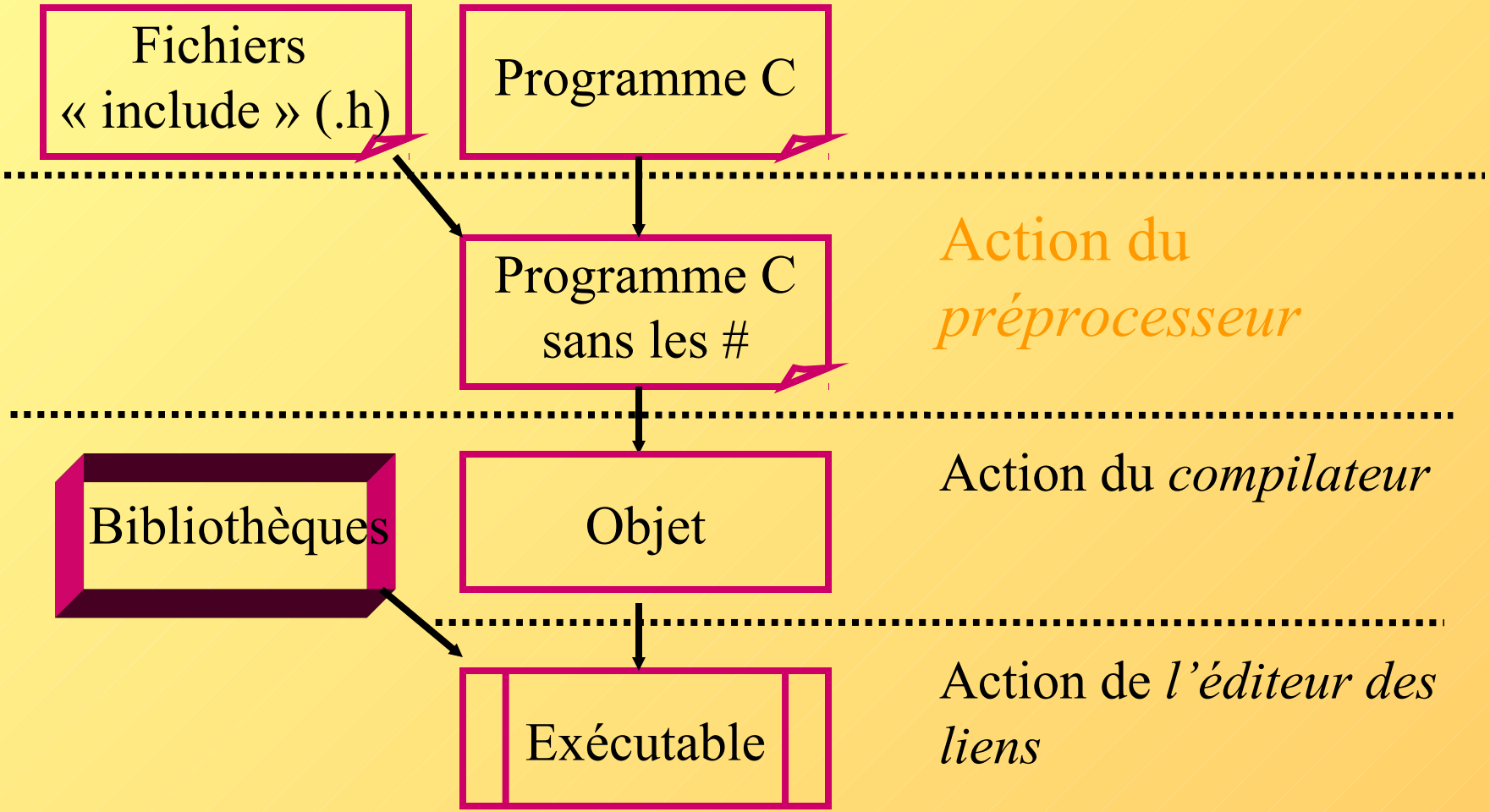
Le préprocesseur

- Le « Pré »-compilateur
- Les fichiers d'en-tête
- Les constantes symboliques
- Les macros
- La compilation conditionnelle

Introduction

- Le compilateur démarre un programme spécial qui prépare le source avant de le traduire en code objet.
- Le *préprocesseur* obtient ses instructions sous forme de directives présentes dans le source :
  - Ex. #include <stdio.h>

Introduction



Les directives

- Remarques sur les directives :
  - Commencent par #
  - Prennent toute la ligne
  - Pas de ;
  - N'importe où dans le source, mais généralement en tête
  - En vigueur dès l'instant où elles apparaissent jusqu'à la fin du fichier source

Les directives

- Le préprocesseur reconnaît les directives :

#include	#endif
#define	#ifdef
#undef	#ifndef
#if	#error
#elif	#line
#else	#pragma

Inclusion de fichiers

- Bibliothèque : Ensemble de fonctions écrites *ailleurs*
- On doit *inclure* leur fichier d'en-tête dans notre source avant de les utiliser :
  - bibliothèques *systèmes* :
    #include <nom.h>
    Ex: #include <stdio.h>
  - bibliothèques *utilisateurs* :
    #include "nom.h"

Définition de contantes

#define NOM [texte]

- Objectif : rendre les programmes plus ‘lisibles’
  - Ex: Texte de remplacement
- [texte] n’est pas obligatoire (cfr. Compilation conditionnelle)
- Convention : NOM en majuscule
- Effacée par : #undef NOM

Les macros

```
#define  NOM_MACRO(parm1, parm2,
   …,parmn)  texte
```

- Macro : suite d'instructions rassemblées sous un nom spécifique utilisable dans le source.
- Macros sans paramètres : ex.

```
#define  CLS      printf("\033[2J") /* efface l'écran */
#define  forever  for (;;)  /* boucle infinie */
```

- Exemple : Programme (préprocesseur1.txt)

Les macros

- Macros avec paramètres : ex.

```
#define  ADD(a,b)    (a) + (b)
{ int k, i, j;
   double z, x, y;
   …
   k=ADD(i,j);                  /* k=(i) + (j) */
   printf ("%d", ADD(1,1)); /* 2 */
   z=ADD(x,y);                  /* z=(x) + (y) */
}
```

Les macros

- Macros avec paramètres : il est conseillé de toujours utiliser des parenthèses autour des paramètres de macros. Ex. d'erreur :

```
#define     SUB(a,b)    a – b
```

Avec :

```
int  k, i=2, j=1;
k =  SUB(i, j +1);
```

Donnera :

```
k =  i  –  j  +1;        /* 2 – 1 + 1 = 2 ! */
```

Les macros

- Effets de bord : on ne peut pas utiliser des paramètres contenant des effets de bord
- ex. d'erreur

```
#define  CARRE(a)   ((a) * (a))
```

Avec : `int  i=2; long carre;`

```
carre = CARRE(--i);
```

Donnera :

```
carre = ((--i) * (--i));  /* (1) * (0) = 0 */
```

Les macros

- Effacement de macro :

  #undef CARRE /* sans les parms */

  Annule la définition de CARRE

- Macro ou appel de fonction ? Réponse :

  La macro vise à réduire le temps d'exécution d'un programme (pas de saut avant, vers les instructions dans le corps de la fonction appelée, pas de saut arrière en retour vers la fonction appelante).

Compilation conditionnelle

- Possibilité de compiler seulement certaines parties d'un programme si une certaine condition est TRUE
- Avec les directives :

  #if, #elif, #else, #endif, #ifdef,#ifndef
- et l'opérateur : defined
- Rem. : équivalences entre :

  #if defined (ABC) <-> #ifdef ABC

  #if !defined (ABC) <-> #ifndef ABC

Compilation conditionnelle

- Ex. de modèles de construction :

```
#if expr_constante1
   partie_1 du prog
#elif expr_constante2
   partie_2 du prog
   …
#else
   partie_3 du prog
#endif
```

```
#if defined (constante1)
   partie_1 du prog
#elif defined (constante2)
   partie_2 du prog
   …
#else
   partie_3 du prog
#endif
```

Compilation conditionnelle

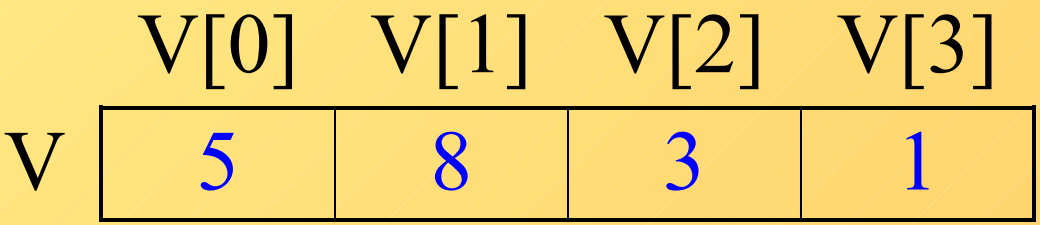
- Exemples : dans tous les fichiers d'en-tête des librairies standard de fonctions du C
  - Ex: #include <stdio.h>

Les tableaux

- Principes
- Mécanisme
- Tableaux en paramètre
- Tableaux à plusieurs dimensions

Principes

- Variable structurée pouvant regrouper plusieurs éléments de même nature sous un même nom afin d'en faciliter la manipulation
- Le nombre de ces éléments s'appelle la dimension du tableau
- Chaque élément est distingué par un numéro



Principes

Exemples

- Les points d'un élève à ses différents cours
- Les noms des élèves de 1° année
- La quantité de précipitation pour chaque jour de l'année
- Le nombre d'hamburgers vendus en 2000 par chaque Quick de Belgique

Contre-exemple

- Une date : 3 entiers (jour, mois, année) de natures différentes; on utilisera une structure

Principes

Que gagne-t-on par rapport à des variables séparées ?

- Ecriture simplifiée (ex: déclaration)
- S'adapter facilement à la modification de la taille (ex: nombre d'élèves)
- Effectuer facilement un même traitement sur chaque élément (ex: calculer le nombre d'hamburgers vendus par Quick en Belgique en 2000)

Mécanisme

- Déclaration
- Utilisation
- Initialisation
- Taille

Déclaration

définition-de-tableau ≡
descripteur-de-type nom-de-variable **[***dimension***] ;**

dimension ≡ *expression-constante*

- Exemples

```
int v [10];
unsigned long taille [20];
int w[n];       /* Non */
#define N 10
int x[N];       /* Oui */
```

Utilisation

- Un élément se référence en indiquant son nom suivi de son numéro entre crochets

  ex:

```
int v [10];
v[0] = 0;
v[1] = 2;
v[2] = 4;
v[3] = 6;
…
```

Utilisation

- Le numéro de l'élément peut être une expression entière

  ex:

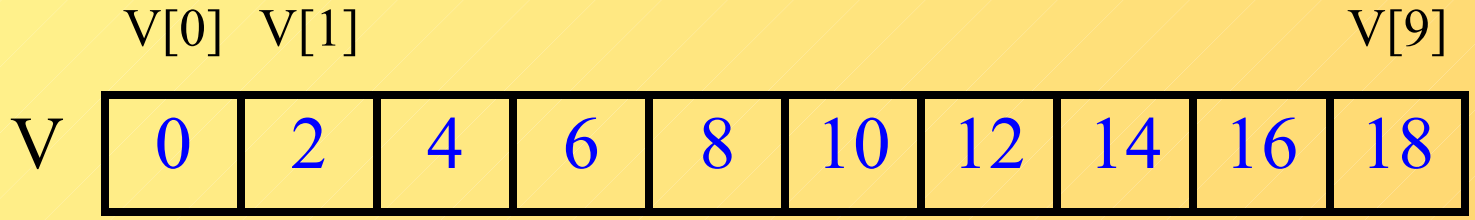
```
int v [10], i;
for (i=0; i<10; i++)
    v[i] = 2 * i;
```

- Rappel : La numérotation commence à 0

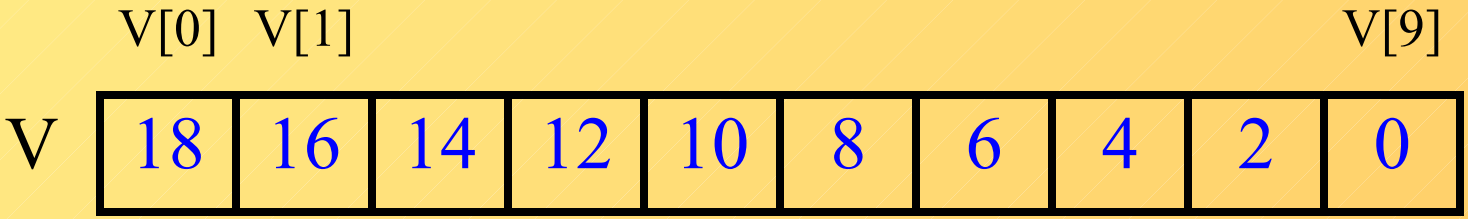
  → dans l'exemple, on va de v[0] à v[9]

Utilisation

- int v [10], i;
  for (i=0; i<10; i++)
  v[i] = 2 * i;



- for (i=0; i<10; i++)
  v[9-i] = 2 * i;



Utilisation

- Attention : Le compilateur ne vérifie pas que l'indice est valide
  ex: int v [10]; v[10] = 1; v[-2] = 2;
- Ne produira aucune erreur à la compilation
- A l'exécution on a un résultat imprévisible ou une erreur (accès à des zones mémoires interdites).

Initialisation

- On peut initialiser un tableau lors de sa déclaration

  ex:

  ```
  int v [10] = {0,2,4,6,8,10,12,14,16,18};
  ```

- Si on initialise, on peut oublier la taille

  ex:

  ```
  int v [] = {0,2,4,6,8,10,12,14,16,18};
  ```

  Elle est déduite du nombre d'éléments fournis

Initialisation

- Si on spécifie la taille, il faut donner le bon nombre d'éléments

  ex:

  - int v [10] = {0,2,4}; /* Dangereux */
  - int v [2] = {0,2,4}; /* Erreur */

Utilisation

- Il n'est pas possible d'assigner un tableau d'un coup. ex:

```
int v [5] = {0,2,4,6,8};
int w[5],
int i;
for ( i = 0; i < 5; i++ )     /* Et pas w = v */
     w[i] = v[i];
```

- De même, il n'est pas possible d'effectuer d'autres opérations globales sur le tableau (+, -, …) ou des impressions

Taille

Peut-on connaître le nombre d'éléments d'un tableau ?

- L'opérateur sizeof() donne le nombre d'octets occupés en mémoire par une variable (bibliothèque stddef.h)
- On peut spécifier le nom de la variable ou le type. ex: int i;

```
sizeof(i)   sizeof(int)     /* 2 ou 4 */
```

Taille

- Si on spécifie le nom du tableau, on a l'espace mémoire occupé par le tableau.
- D'où

```
int v [10], t;
t = sizeof(v) / sizeof(int);
printf( "Nb d élts : %d.\n ", t ); /* 10 */
```

- sizeof a le *look* d'une fonction mais c'est un opérateur

Du plus prioritaire au moins prioritaire

opérateur	associativité
( ) [ ]	⇒
! + - (unaire) (*type*) sizeof	⇐
* / %	⇒
+ -	⇒
< <= > >=	⇒
== !=	⇒
&&	⇒
||	⇒
?:	⇐

Tableau à plusieurs dimensions

- Déclaration : int v[2][4];

V[0][0]	V[0][1]	V[0][2]	V[0][3]
V[1][0]	V[1][1]	V[1][2]	V[1][3]

- v[2][4] est un entier
  donc v[2] est un entier tableau de 4 entiers
  donc v est un entier tableau de 2 tableaux de 4 entiers
  « donc » v est un entier tableau 2x4 d ’entiers
- Utilisation : v[1][2];

Tableau à plusieurs dimensions

- Initialisation :

```
int v[][4] = {{1,2,3,4},{5,6,7,8}};
```

1	2	3	4
5	6	7	8

- Attention : seule la taille de la première dimension peut être omise.

Arithmétiques des pointeurs

- Principe
- Addition et soustraction d ’un entier
- Soustraction de pointeurs
- Comparaison de pointeurs

Principe

- Un pointeur peut pointer vers un élément d 'un tableau
- Exemple :

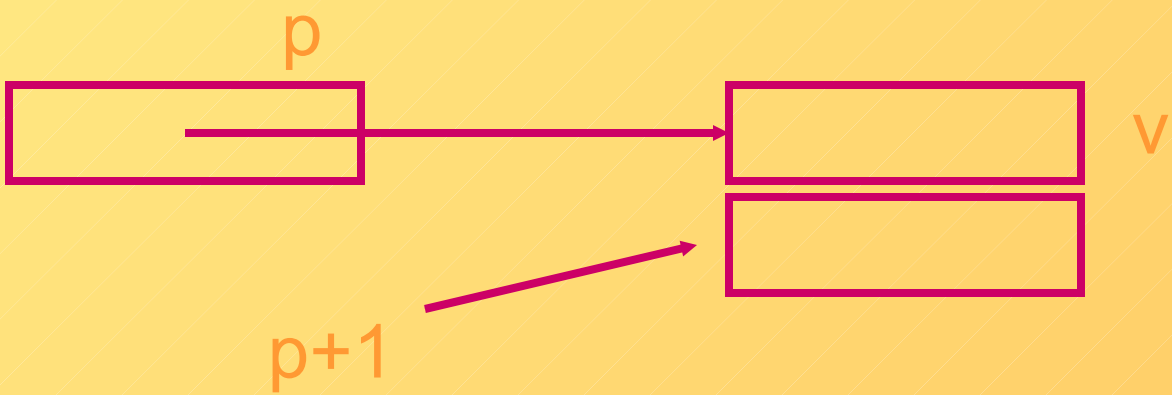
```
int t[10], *p;
p = &(t[5]);          /* p pointe vers t[5] */
```

- C permet certains « calculs » avec ces pointeurs.

Ajout d'un entier

- Si p est un pointeur, p+1 pointe vers la « zone mémoire » suivante.
- Cela dépend donc du type de valeur pointée par p.
- Exemple :

```
int v, *p;
p = &v;
```



Ajout d'un entier

- Cela n'a de sens que si p pointe vers un élément d 'un tableau.
- Dans ce cas, p+1 pointe vers l 'élément suivant dans le tableau
- Exemple :

```
int t[10], *p;
p = &(t[5]);    /* p pointe vers t[5] */
p = p +1;       /* p pointe vers t[6] */
*(p+2);  /* l 'élément t[8] */
```

Ajout d'un entier

- De façon générale, on peut ajouter ou soustraire un entier quelconque à un pointeur
- Opération définie si on reste dans le tableau
- Exemple :

```
int t[10], *p;
p = &(t[5]);      /* p pointe vers t[5] */
p = p - 5;        /* p pointe vers 1er élément */
p = p + 9;        /* p pointe vers dernier élt */
p = p + 3;        /* n 'a pas de sens défini */
```

Soustraction de pointeurs

- On peut également soustraire 2 pointeurs
- On obtient un entier
- Cela n'a de sens que si les pointeurs pointent vers un même tableau
- Exemple :

```
int t[10], *p, *q;
p = &(t[5]);    /* p pointe vers t[5] */
q = &(t[8]);    /* q pointe vers t[8] */
q - p;    /* vaut 3 */
```

Soustraction de pointeurs

- Attention :

```
int v1,v2, *p1, *p2;
p1 = &v1;        /* p pointe vers v1 */
p2 = &v2;        /* q pointe vers v2 */
p2 - p1;         /* non défini */
```

- La soustraction dépendra des zones mémoires utilisées par v1 et v2 ce qui peut varier à chaque exécution.

Comparer des pointeurs

- == et !=

  ont toujours un sens

- <, >, <= et >=

  n'ont de sens que pour des pointeurs vers un même tableau

Exemple :
```
int t[10], *p, *q;
p = &(t[5]);
q = &(t[8]);        /* p < q */
```

Remarque

Attention :

- L'utilisation à outrance des pointeurs amène rapidement à des programmes difficilement lisibles
- L'utilisation d'un pointeur ayant une valeur incohérente peut planter lamentablement le programme

Tableaux et pointeurs

- Principe
- Équivalences des notations

Principe

- En **C**, les tableaux n'existent pas vraiment
- Un tableau, n'est jamais qu'un pointeur vers une zone mémoire contiguë contenant des valeurs de même type.
- La notion introduite avec les tableaux ne sont que des équivalences d'écriture

Déclaration

- Une déclaration comme

  ```
  int t[10];
  ```

  signifie
  - réserver un espace contigu en mémoire pour contenir 10 entiers
  - t est un pointeur constant vers un entier
  - il pointe vers le premier élément du tableau (t ≡ &t[0])

Utilisation

- L'accès à un élément comme dans

  t[i];

  est syntaxiquement équivalent à

  (t[i] ≡ *(t+i))

- Exemple :

```
int t[] = {0,2,4,6,8};
/* t[3] équivalent à *(t+3) : vaut 6 */
```

Utilisation

Attention : Un nom de tableau est un pointeur constant; on ne peut pas le modifier

- Exemple :

```
int t[] = {0,2,4,6,8}, *p;
t = t +1;    /* Erreur à la compilation */
p = t +1;    /* OK */
```

Exemple

- Pour parcourir les éléments d'un tableau, on pourrait écrire

```
int t[] = {0,2,4,6,8}, *p, i;
p = t;
for (i=0; i<6; i++)
{
        printf("%d", *p);
        p++;
}
```

Tableaux à 2 dimensions

- Le même mécanisme s'applique pour un tableau à 2 dimensions

```
int t[][10];
```

- t est un tableau de « tableaux de 10 entiers »
- C'est donc l'adresse du premier élément, un tableau.
- En résumé, t est équivalent à &t[0]

Tableaux à 2 dimensions

- Une déclaration comme

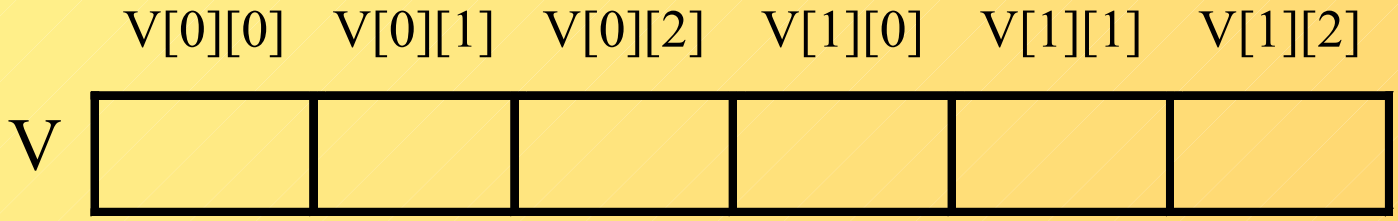
  ```
  int t[4][10];
  ```

  signifie
  - réserver un espace contigu en mémoire pour contenir 40 entiers
  - t est un pointeur constant vers un entier
  - il pointe vers le premier élément du tableau (t ≡ &t[0])

Tableaux à 2 dimensions

- En mémoire, les éléments sont stockés « ligne par ligne »

```
int v[2][3];
```



- v[i][j] est transformé en *(v+i*3+j)

Tableaux et fonctions

- Principe
- Paramètre de la fonction
- Valeur de retour de la fonction

Principe

- Les tableaux se comportent différemment que les autres types structurés en ce qui concerne les fonctions
- Passer un tableau en paramètre c'est passer son nom, c'est-à-dire un pointeur vers son premier élément
- Une fonction qui retourne un tableau, retourne en fait le pointeur vers son 1er élément.

Passer un tableau en paramètre

- Dans la signature d'une fonction:
  (int t[10] ≡ int t[] ≡ int *t)
- On recommande int t[].
- Cela signifie que la fonction recevra un pointeur vers un entier, qui sera assigné au paramètre formel t.

Passer un tableau en paramètre

- Implication : un tableau est toujours un paramètre en entrée-sortie.
- Exemple :

```
void f(int t[10]) {
        t[0] = 0;
        return ;
}
…
int v[10];
f(v);          /* Modifie v[0] */
```

Un tableau en paramètre n'a pas de taille

- Pas de taille : (int t[10] ≡ int t[])
  - La taille est ignorée par le compilateur
  - Avantage : une fonction peut recevoir des tableaux de tailles différentes
  - Inconvénient : il faudra souvent passer la taille en même temps que le tableau

Un tableau en paramètre n'a pas de taille

- Exemple

```
int somme (int t[], int taille)
{
        int somme, i;
        somme = 0;
        for (i=0; i<taille; i++)
                somme += t[i];
        return somme;
}
```

Un tableau en paramètre n'a pas de taille

- Attention : sizeof se comporte différemment

```
int somme (int t[])
{ ...
        sizeof (t);
        /* donne la taille d 'un pointeur */
        /* pas du tableau */
  ...
}
```

Un tableau en paramètre n 'est pas constant

- Exemple (non recommandable)

```
int somme (int t[], int taille)
{
int somme, i;
somme = 0;
for (i=0; i<taille; i++) {
somme += *t;
t ++;  /* Permis pour un paramètre */
}
return somme;
}
```

Passer un sous tableau

- Attention : rien n'empêche d'écrire

```
int main(void)
{
        int v[] = {0,1,2,3,4,5,6,7,8,9}, s;
        s = somme(v,10);        /* s = 45 */
        s = somme(v,5);         /* s = 10 */
        s = somme(v+3,7);       /* s = 42 */
        s = somme(v,20);        /* ??? */
        return 0;
}
```

Retourner un tableau

- Une fonction peut retourner un tableau mais uniquement avec la notation *.

```
int *f(void);        Ok
int [] f(void);      NON
int *f(void) [];     NON
```

- Elle retourne l'adresse du premier élément

Retourner un tableau

- Attention : piège à éviter

```
int *f (void)
{
        int t[] = {0,2,4,6,8};
        return t;
}
```

- retourne l'adresse d 'un tableau qui n 'existe plus !!!

Tableaux à 2 dimensions

- Pour un tableau à 2 dimensions, on a le même genre de notations.
- Mais seule la première dimension peut être omise.

```
int somme2(int t[][10], int n)
{
        ...
}
```

Tableaux à 2 dimensions

- Exemple complet

```
int somme(int t[][10], int n)
{
        int somme, i, j;
        somme = 0;
        for (i=0; i<n; i++)
                for (j=0; j<10; j++)
                        somme += t[i][j];
        return somme;
}
```

Tableaux à 2 dimensions

- Comment sommer les éléments d'un tableau à 2 dimensions quelconques ?

```
int v[4][10];
somme2(v, 4);              /* NON */
somme(v, 4*10);            /* type incompatible*/
somme(&v[0][0], 4*10);     /* OK */
```

- On voit le tableau à 2 dimensions comme un tableau à une seule dimension, de taille 4*10.

Chaîne de caractères

- Principe
- Constante chaîne
- Bibliothèque string.h

Principe

- Pas de type « chaîne » en C
- Une chaîne sera un tableau de caractères
  ex: char nom[10];
  nom est un tableau de 10 caractères
  → aussi une chaîne de 10 caractères.
- Il existe un littéral chaîne
  ex: "Bonjour"

Principe

- Attention : Dans tout littéral chaîne, il y a un caractère caché : '\0'
  ex: "Bonjour" est représenté

B	o	n	j	o	u	r	\0

- Ce caractère est utilisé pour déterminer la fin de la chaîne
- D'où la réservation d'un caractère en plus
  ex: char nom[8] = "Bonjour";
  ou char nom[7+1] = "Bonjour";

Principe

- Pour écrire une chaîne

```
char nom[19] = "Bonjour le monde !";
printf("%s\n", nom);
nom[7] = '\0';
printf("%s\n", nom);  /* Bonjour */
```

- De nombreuses fonctions utilisent ce caractère '\0' pour connaître la fin de la chaîne.

Principe

- On peut aussi utiliser la technique générale d'initialisation

```
char nom[] = { 'B','o','n','j','o','u','r','\0' };
```

- Attention : Ne pas oublier le '\0' à la fin

ex:
```
char nom[] = { 'B','o','n','j','o','u','r' };
    printf("%s\n", nom); /* imprévisible */
```

string.h

- Cette bibliothèque contient des fonctions de manipulation de chaînes.
- On explique ici les plus courantes

string.h

- Fonctions d'examen de chaîne.
  - strlen(s) longueur de s (sans \0)
    ex : strlen("Bonjour") vaut 7
  - strcmp(s1,s2)
    « Compare » s1 à s2; le résultat est
    < 0 si s1 < s2
    = 0 si s1 = s2
    > 0 si s1 > s2
  - strncmp(s1,s2,n)
    Compare les n premiers caractères

string.h

- Copie de chaîne.

  char dest[10], source[10];

  – strcpy(dest, source)

  copie source dans dest (avec \0)

  – strncpy(dest, source)

  copie au plus n caractères de source dans dest

- Attention : dest = source est interdit.

Principe

- typedef permet de simplifier l'écriture liée aux tableaux
- Exemples :

```
typedef char Chaine[20];
typedef int Table[10];
…
Chaine nom, prenom;
Table   cote;
```

Les unions

- Principes
- Mécanisme

Principe

- Comme la structure,
  une union est une entité structurée pouvant regrouper plusieurs éléments de types différents sous un même nom.
- A l'inverse d'une structure,
  ses membres n'existent pas simultanément mais s'excluent mutuellement

Avantage

- Seuls et uniques avantage
  - le gain de place
  - la multiple interprétation d ’une zone mémoire
- Cela avait son importance à l’époque où le **C** est apparu (mémoire chère)
- Cela perd de plus en plus de son intérêt

Déclaration

Déclaration du type correspondant à l'union

- En tout point pareil à la structure
- union Brol
```
{
    int         champ1;
    double      champ2;
};
```
- Déclaration d'une variable de ce type
- union Brol brol;

Utilisation

Référence à un membre d'une union

- nom_union.nom_membre
- Attention : il incombe au programmeur de s'assurer que l'union contient bien ce membre là à ce moment là
- Ex : ceci n'a pas de sens mais est admis

```
brol.champ1 = 10
printf("%g", brol.champ2);
```

Initialisation

Initialisation à la déclaration

- Uniquement le premier membre de l'union
- ex : union Brol brol = {12};

Assignation globale

- On peut assigner une union dans sa globalité
- Exemple:

```
union Brol brol1 = {12}, brol2;
brol2 = brol1;
```

Unions complexes

- On peut mélanger les unions avec tous les autres types
  - Unions comprenant des structures
  - Structures comprenant des unions
  - Des tableaux d'unions
  - …

Unions et typedef

- Même mécanisme que pour les structures

```
union Brol
{
    int     champ1;
    float   champ2;
};
…
union Brol brol;
```

```
typedef union
{
    int     champ1;
    float   champ2;
} Brol ;
…
Brol brol;
```

- Utiliser l'une ou l'autre est une question de goût.

Les énumérations

- Fonctionnement
- Intérêt

Principe

Problème

- Une carte à jouer est composée d'une couleur : Pique, Carreau, Cœur, Trèfle
- Représentée par un nombre
  ex: 0 = Pique, 1 = Carreau, …
- Utiliser les numéros n'est pas clair

```
int couleur;
if (couleur == 1) ...
```

Principe

Solution 1

- Utiliser des #define

```
#define PIQUE   0
#define COEUR   1
#define CARREAU 2
#define TREFLE  3
int couleur;
if (couleur == COEUR) ...
```

Solution 2

- Utiliser une énumération

```
enum CouleurCarte
  {PIQUE,COEUR,CARREAU,TREFLE};

enum CouleurCarte couleur;
if (couleur == COEUR);
```

- Avantage : vague notion de type (pour le lecteur uniquement)

Exemple

Exemple : un booléen

- enum bool {false, true};
  enum bool finFichier;
- Remarque : On se rapproche de la notation indiquée en début de cours; il ne reste plus qu ’à introduire le typedef.
- Exemple : Programme (Enumération1.txt)

Mécanisme

Ce qu'il y a derrière

- Lorsqu'on définit une énumération, le compilateur associe un nombre à chaque élément, en commençant par 0.
- Exemple : enum bool {false, true};
  On a false == 0, true == 1.

Exemple : les cartes

- On peut imposer une valeur
  - enum ValeurCarte {AS=1,VALET=11,DAME=12,ROI=13};
    enum ValeurCarte carte;
    if (carte==DAME) ...
- Par défaut, on continue l'énumération
  - enum ValeurCarte {AS=1,VALET=11,DAME,ROI};
    enum ValeurCarte carte;

Mécanisme

- enum n'empêche pas d'utiliser d 'autres valeurs.
  - enum ValeurCarte
    {AS=1,VALET=11,DAME=12,ROI=13};
    enum ValeurCarte carte;
    carte = 5;

Exemple : les mois

- Afficher le nombre de jours dans chaque mois.
  - Sans enum (Enumération2.txt)
  - Avec enum (Enumération3.txt)

Mécanisme

Attention !

- La norme n'impose pas le type entier utilisé pour réaliser l'énumération (int, short , ...)
- Dès lors, une instruction comme
  enum ValeurCarte carte;
  scanf("%d", &carte);
  n'est pas portable.

énumération et typedef

- Comme avec les autres types, on peut utiliser typedef pour simplifier l'écriture.
- enum bool {false, true};
  enum bool finFichier;
  devient
  typdef enum {false, true} bool ;
  bool finFichier;

Conclusion

- En C, Le type énuméré n'est qu'effleuré.
- Ils ne sont pas allés au bout de leur logique mais c'est déjà un début.

Les assignations

Principe

- Les assignations sont des expressions
- Leur évaluation a pour valeur le membre de gauche après assignation
- Exemple : a = 1
  - affecte à a la valeur 1
  - vaut 1
- Exemple : (a = 1) + 2
  a un sens (et probablement peu d'intérêt)

Instruction-expression

- Une expression suivie d'un ; devient une instruction
- Exemples : a = 1; mais aussi 1;
- La valeur de l'expression est inutilisée
- On comprend dès lors qu'on peut écrire

```
f ( … ) ;
```

même si cette fonction retourne une valeur

Instruction-expression

- Utilisation possible : l'assignation en chaîne

  Exemple :

  ```
  a = b = c = d = e = 0;
  ```

  met à 0 les 5 variables;

Instruction-expression

- Le mécanisme est identique pour les autres assignations
- Exemple :

  a += 2;

  - ajoute 2 à la variable a;
  - vaut la nouvelle valeur de a;
- Exemple :

  b = (a += 2); ≡ (a += 2); b = a;

Incrémentation - décrémentation

- Il existe une subtilité pour les opérateurs ++ et --
- i++ et ++i sont 2 expressions qui incrémentent i mais
  - avec i++ l'expression vaut la valeur de i avant incrémentation
  - avec ++i l'expression vaut la valeur de i après incrémentation

Incrémentation - décrémentation

- Exemple :

```
i = 1;
a = ++i;    /* i == 2, a == 2 */
b = i++;    /* i == 3, b == 2 */
```

- Règle de conduite : On acceptera
  - l'assignation multiple
  - La transformation d'expression en instruction

et c'est tout !

Priorités et associativités

opérateur	associativité
( ) [ ] .	⇒
! + - (unaire) ++ -- (*type*) sizeof	⇐
* / %	⇒
+ -	⇒
<< >>	⇒
< <= > >=	⇒
== !=	⇒
&	⇒
^	⇒
|	⇒
&&	⇒
||	⇒
?:	⇐
= += -= *= /= %=	⇐

De la vraie nature du char

Principe

C considère char comme un type entier

- Codé sur un octet
- L'octet contient le code du caractère
  ex: en ASCII, 'A' est représenté par 65
- En fait, 'A' signifie :
  « L'entier qui correspond à ce caractère dans le code utilisé »
  var_car = 65 ne serait pas *portable*.

Littéral

- Mais on peut aussi indiquer directement le code en octal '\0cc' ou en hexadécimal '\xcc '.
- exemples:

  '\041' est le caractère de code 33

  '\011' est le caractère de code 9

  '\x21' est le caractère de code 33

- Attention : code en général non portable. Ne se justifie que pour les caractères non représentables.

Calcul

On peut effectuer du calcul

- Exemples : (en ASCII)
  - 'A' + 1 vaut 66
  - '2' + '3' vaut 101 et pas '5' ou 5!
  - '5' - '2' donne l'écart entre les 2 caractères dans la table (ici 3)
- Attention : Rien n'assure que les lettres se suivent dans le code (ASCII oui, EBCDIC non)

Calcul

- Exemple classique d'utilisation.
  Pour transformer un chiffre, soit i, en son caractère, on note '0' + i.
- Contre-exemple. Pour obtenir la ième lettre de l'alphabet, 'a' + i -1 ne fonctionne pas sur mainframe où les lettres ne se suivent pas.
- Pour ce genre de manipulations, les tableaux sont plus lents mais sûrs à 100%.

Gain de place

On peut l'utiliser comme un numérique pur

- Peut se justifier si les valeurs tiennent sur un octet et qu'on veut gagner de la place
- Exemple :

```
struct Date
{
    char jour;
    char mois;
    int annee;
};
```

Écrire un char

On peut écrire un char

- comme un caractère

```
char car = 'A';
printf("%c", car );      /* A */
```

- ou comme un nombre

```
printf("%d", car );      /* 65 en ASCII */
```

char et signe

- unsigned char uc = 'è';
  uc contient la valeur 138 en ASCII.
- signed char sc = 'è';
  même contenu que pour uc mais vu ≠ (nombre négatif)
- choix par défaut: DEPENDANT DU SYSTEME.

char et signe

```
#include <stdio.h>
   int main()
   {
        unsigned char uc = 'è';
        signed char sc = 'è';
        printf("%d\n", uc);     /* 138 */
        printf("%d\n", sc);     /* -118 an ASCII */
        return 0;
   };
```

Ce qu'il y a derrière un booléen

Principe

- En **C**, il n'existe pas de type booléen
- Nous en avons introduit un avec

  typedef enum {false, true} bool;
- Pourquoi est-ce que cela fonctionne ?

Mécanisme

- En **C**, un booléen est simulé par un entier
- On a
  - faux si l'entier est = 0
  - vrai si l'entier est ≠ 0
- Ainsi, dans

  if (cond) …

  cond doit être un entier et la condition est vérifiée si cet entier est différent de 0

Lien

- Reprenons notre booléen

  typedef enum {false, true} bool;

- On a vu que les types énumérés cachaient des entiers
- De plus, C associe le nombre 0 au premier élément d'une énumération et 1 au second
- On a donc tout ce qui est nécessaire

Utilisation

- Déclaration : bool b;

  b sera en fait un entier

- Assignation

  b = false; (≡ b = 0;)

  b = true; (≡ b = 1;)

- Test

  if (b); (faux si b == 0 == false)

  (vrai si b == 1 == true)

Intérêt

Pourquoi passer par un type bool ?

- Plus lisible
- Erreur plus facile à détecter
  (à la lecture, pas par le compilateur)

A éviter

Écritures peu recommandables

- int nb; scanf("%d", & nb);
  while (nb) … /* while (nb != 0) */
  nb est entier; ne pas le traiter comme un booléen
- fd = fopen(…);
  if (fd) … /* if (fd != NULL) */
  fd est logiquement un FILE *; ne pas le traiter tantôt comme tel, tantôt comme un booléen

Exercices

- Que fait ceci ?

```
int n = 0;
if (n = 1) {printf("Hello\n");}
```

- et ceci ?

```
int n = 13;
if (0 < n < 5) {printf("Hello\n");}
```

Les conversions implicites

On dit aussi casting implicite

Principe

- C utilise un typage faible.
  - Toute variable a une type
  - Pas de barrière nette entre les types
- De nombreuses conversions sont effectuées implicitement.
- Exemple :
  double d; int i = 2; d = i;
  est accepté; équivalent à d = (double) i;

Principe

Quand y a-t-il conversion ?

- Quand il n'y a pas de perte d'information
  - char → short → int → long
  - float → double → long double

Principe

- Avec les opérateurs, on peut mélanger les types des opérandes
- *C* convertit les opérandes vers le type le plus général
- Exemple :

  3.0 / 2

  est équivalent à

  3.0 / (double) 2

L'instruction for

Principe

- Cette instruction permet de coder plus que le « *pour* » de la logique
- Syntaxe générale :

instruction-for ≡
for ($expr1_{opt}$; $expr2_{opt}$; $expr3_{opt}$ **)**
instruction

- On remarque que
  - les éléments entre parenthèses sont des expressions
  - les 3 éléments sont optionnels

Principe

- Syntaxe générale :

  instruction-for ≡

  for (*expr1*$_{opt}$; *expr2*$_{opt}$; *expr3*$_{opt}$ **)**

  instruction

- expr1: évalué avant d'entamer le for
- expr2 : évalué avant chaque exécution de l'instruction; si vrai on y va; si faux on passe à la suite. Si inexistant → vrai.
- expr3 : évalué après chaque exécution de l'instruction

Équivalence while

- for (<*expr1*>; <*expr2*>; <*expr3*>)
  <instruction>

  est équivalent au code suivant

```
<expr1>;
while (<expr2>)
{
    <instruction>
    <expr3>;
}
```

Exemples

- Exemples :

  for (; ;) … boucle infinie

  for (; 0 ;) … boucle jamais exécutée

  for (1; 2 ; 3) …

  for (-1; 0 ; 1) …

Opérateur ,

L'opérateur ,

- Cet opérateur binaire accepte 2 opérandes quelconques
- Le résultat est la valeur de l'opérande de droite
- Exemple :

```
a = (1, 2) ;
```

a reçoit 2

Opérateur ,

- L'intérêt pour le for est la possibilité de combiner des assignations
- Exemple :

```
for ( i = 0, j = 0; i < 10; i++, j += 2 )
    printf("%d, %d\n", i, j);
```

Priorités et associativités

opérateur	associativité
( ) [ ] .	⇒
! + - (unaire) ++ -- (*type*) sizeof	⇐
* / %	⇒
+ -	⇒
<< >>	⇒
< <= > >=	⇒
== !=	⇒
&	⇒
^	⇒
|	⇒
&&	⇒
||	⇒
?:	⇐
= += -= *= /= %=	⇐
,	⇒

Les instruction de branchement

- continue et boucles
- break et boucles
- break et switch
- exit()

Principe

- C fournit 2 instructions pour modifier le déroulement d'une instruction de répétition
- break quitte un bloc
- continue interrompt un bloc et le reprend

continue

- continue : le bloc est interrompu et on passe directement au test.
- Exemple : imprimer tous les nombres de 1 à 100 sauf les multiples de 7.

```
for ( i = 1; i <= 100; i++ )
{
if ( i % 7 == 0 )
continue;
printf("%d\n", i);
}
```

continue

- continue ne reprend que le bloc le plus interne.
- Exemple :

```
for (c=0; c<4; c++)
        for ( i = 1; i <= 100; i++ )
        {
                if ( i % 7 == 0 )
                        continue;
                printf("%d\n", i);
        }
}
```

break

- break : le bloc est interrompu et on passe à la suite.
- Exemple : sommer des nombres lus jusqu'à la valeur sentinelle (-1).

```
int somme = 0;
while( true )
{
scanf("%d", &nombre);
if ( nombre == -1 )
break;
somme += nombre;
}
```

break

- Exemple : autre écriture sans break

```
#define SENTINELLE -1
int somme = 0;
do {
        scanf("%d", &nombre);
        if ( nombre != SENTINELLE )
                somme += nombre;
} while( nombre != SENTINELLE );
```

break

- Ex : ou encore (écriture recommandée)

```
#define SENTINELLE -1
int somme = 0;
scanf("%d", &nombre);
while( nombre != SENTINELLE )
{
        somme += nombre;
        scanf("%d", &nombre);
}
```

continue

- break ne quitte que le bloc le plus interne.
- Exemple :

```
for (c=1; c<4; c++)
        for ( i = 1; i <= 100; i=i+c )
        {
                if ( i % 7 == 0 )
                        break;
                printf("%d\n", i);
        }
}
```

Break et switch

- On a vu que break est utilisé également avec l'instruction switch.
- Le comportement normal du switch est
  - trouver le cas adéquat
  - exécuter les instructions associées
  - continuer et exécuter les instructions suivantes
- break permet justement d'arrêter cela.

Break et switch

- Exemple :
  Reprenons le programme qui affiche le nom d'un mois dont on donne le numéro
  - Version déjà vue, avec break (Switch1.txt)
  - Idem sans break (Branchement1.txt)

exit()

- Un programme se termine lorsque la fonction main est terminée (rencontre d'un return ou exécution de la dernière instruction)
- On peut vouloir terminer un programme avant de façon abrupte; souvent suite à la rencontre d 'une situation d'erreur.
- La fonction void exit(int) sert à cela
- L'entier en paramètre est retourné au système (comme pour return)

Les opérations sur les bits

- Les opérateurs logiques
- Les opérateurs de décalage

Introduction

- Qu'est ce que c'est ?
  - Opérations directes sur les bits d'une donnée
- Sur quoi ça agit ?
  - Les entiers
- Pourquoi c'est introduit ?
  - Gain de place
  - Lien avec le système
- Est-ce à recommander ?
  - Non! car non portable

Les opérateurs logiques

- & : ET bit à bit
- | : OU inclusif bit à bit
- ^ : OU exclusif bit à bit
- ~ : complément à 1 (unaire)

& : ET bit à bit

- A ne pas confondre avec &&
- char c1=13; 00001101
  char c2=6; 00000110
  c1 & c2; 00000100 = 4
  c1 && c2; 00000001 = 1

& : ET bit à bit

- Attention avec les négatifs

  char c1 = -5, c2 = -6;

  - Complément à 1

    C1 = 11111010
    C2 = 11111001
    C1 & C2 = 11111000 = -7

  - Complément à 2

    C1 = 11111011
    C2 = 11111010
    C1 & C2 = 11111010 = -6

| : OU bit à bit

- A ne pas confondre avec ||
- char c1 = 13; 00001101
  char c2 = 6; 00000110
  c1 | c2; 00001111 = 15
  c1 || c2; 00000001 = 1

^ : OU exclusif bit à bit

- char c1 = 13; 00001101
  char c2 = 6; 00000110
  c1 ^ c2; 00001011 = 11

~ : complément à 1

- A ne pas confondre avec !
- unsigned char c1;

  c1 = 13; 00001101

  ~c1; 11110010 = 242

  !c1; 00000000 = 0

Liens avec l'assignation

- Il existe les versions combinées à l'assignation

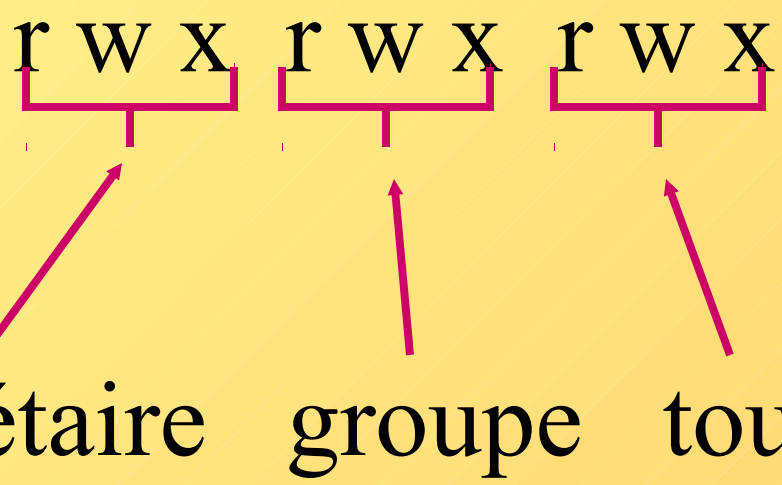
  &=, |=, ^=

- c = 7;

  c &= 3; /* c = c & 3 */

Exemple

- Gestion des permissions en Unix



- Exemple : 1 1 1 1 0 1 0 0 1

Exemple

- Imposer toutes les permissions

```
permission = 0751
```

- Ajouter certaines permissions

```
permission |= 0020
```

- Lire la permission

```
if(permission & 0020 == 0)
    printf("Pas permis !");
```

Exemple

- Enlever des permissions
  - permission &= 0776
  - ou permission &= ~1

Introduction

- << : décalage de bits à gauche
- >> : décalage de bits à droite
- Versions avec = : <<=, >>=

<< : décalage à gauche

valeur << nb

- décalage de nb positions vers la gauche
- remplissage avec des 0.
- unsigned char c;

  c = 17; 00010001

  c << 3; 10001000 = 136
- Remarque : 136 = 17 * 8

<< : décalage à gauche

valeur << nb

- Les bits à droite sont perdus
- char c;
  c = 81; 01010001
  c << 3; 10001000 = 136
- Remarque : 136 = (81 * 8) % 256

>> : décalage à droite

valeur >> nb

- décalage de nb positions vers la droite
- remplissage avec
  - des 0 sur certaines machines.
  - Le bit de signe sur d'autres
- unsigned char c;
  c = 17; 00010001
  c >> 3; 00000010 : 2
- Remarque : 2 = 17 / 8

Les opérations de décalage

>> : décalage à droite

- Hypothèse : complément à 1
- char c;
  c = -17; c = 11110110
  c >> 3;
  - 00011110 : 30
  - 11111110 : -254

Exemple récapitulatif

Écrire une fonction qui compte le nombre de bits à 1 dans un entier.

solution (Bits1.txt)

Les champs de bits

- Fonctionnement
- Intérêt

Principe

- Une carte à jouer est composée
  - d'une valeur : 1, 2, …, V, D, R
  - d'une couleur : Pique, …
- Représenté par une structure

```
struct Carte {
        char valeur;        /* 1..13 */
        char couleur;       /* 0..3 */
}
```

- Clair et utilisation facile

Principe

- On peut gagner de la place en utilisant un seul octet

- Exemple :

  char c = 011;

  c = 00 0010 01 : un 2 de la couleur 1

Principe

- On peut gérer cela avec des manipulations de bits

```
int couleur (char carte)
{
  return carte & 3;
}
int valeur (char carte)
{
  return (carte >> 2) & 15;
}
```

Principe

- C'est plus simple avec des champs de bits

```
struct Carte
{
    unsigned int valeur   : 4;  /* 4 bits */
    unsigned int couleur : 2;  /* 2 bits */
}
struct Carte carte;
carte.valeur;
carte.couleur;
```

Principe

Attention

- Aucun contrôle sur la représentation
- Ne pas utiliser d'opérations sur les bits
- Seule garantie, le compilateur essaie de gagner de la place.

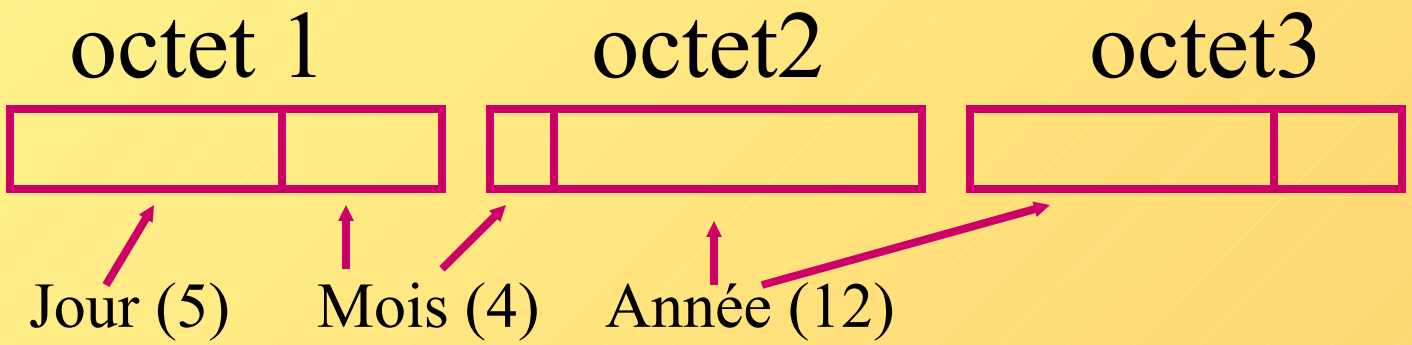
Exemple

- Autre exemple : la date

```
struct Date
{
    unsigned int jour       : 5;
    unsigned int mois       : 4;
    unsigned int annee      : 12;
}
```

Exemple

- Représentation interne possible



Allocation dynamique de mémoire

- Intérêt
- Allocation
- Récupération
- Précaution

Principe

- Normalement, les zones mémoires attribuées à des données sont définies par les variables
- int i;
  - alloue (lors de l'entrée dans son bloc) une zone mémoire pour un entier
  - l'appelle i.
  - Cette zone est libérée au sortir du bloc

Intérêt

- On peut allouer dynamiquement de l ’espace mémoire pour des données
- Avantages ?
  - Pour des tableaux de taille dynamique
  - Pour des structures auto-référencées comme les listes

Allocation

malloc(n);

- Cette fonction réserve un espace de *n* octets et retourne un pointeur vers cette zone
- Exemple :

```
double *p;
p = malloc (sizeof(double));
*p = 3.14;
```

Allocation

- Une allocation peut ne pas fonctionner (plus de mémoire)
- Dans ce cas, la fonction retourne NULL
- Il faut toujours tester ce cas
- Exemple :

```
double *p;
p = malloc (sizeof(double));
if (p==NULL) { /* problème */}
```

Désallocation

```
free( p );
```

- Libère la zone mémoire pointée par p et précédemment allouée
- Exemple :

```
double *p;
p = malloc (sizeof(double));
*p = 3.14;
free(p);
```

Bibliothèque

- Ces 2 fonctions ont leur déclaration dans la bibliothèque stdlib.h
- Exemple :

```
#include <stdlib.h>
double *p;
p = malloc (sizeof(double));
*p = 3.14;
free(p);
```

Précaution

- La mémoire reste allouée tant qu'on ne la libère pas

  Exemple :

```
int f (void) {
int *p;
p = malloc( sizeof(int) );
*p = 1;
return *p;
}
```

  Au sortir de la fonction, p est détruit mais pas la zone allouée ⇒ saturation possible

Tableaux de taille dynamique

Principe

- En **C**, la taille d'un tableau doit être donnée à la compilation (une constante donc)
- Cela pousse souvent à sur-dimensionner les tableaux
- L'allocation dynamique permet de s'en sortir à moindre coût

Exemple

```
int main (void)
{
    int *v, n, i;
    scanf("%d", &n);
    v = malloc( n * sizeof(int) );
    for ( i=0; i<n; i++)
        v[i] = 2*i;
    …
    free(v);
    return 0;
}
```

Listes

- Principe
- Structures auto-référencées
- Les listes en C

Principe

- Il est évidemment possible de définir et de manipuler en C des structures chaînées (listes)
- Voyons les quelques particularités et la traduction en C des algorithmes classiques
- Commençons par l'étude des structures auto-référencées.

Structures auto-référencées

- Un champ d'une structure peut-être quelconque (entier, tableau et même une structure)
- Toutefois, une structure ne peut se contenir elle même
-
```
struct Brol
{
    ...
    struct Brol brol;          /* Interdit */
}
```

Structures auto-référencées

- Cette limitation disparaît si on passe par un pointeur
- ```
  struct Brol
  {
      ...
      struct Brol *brol;  /* OK */
  }
  ```
- Tout simplement parce que la taille de la structure est fixe dans ce cas.
- Cela va permettre d'implémenter les listes
  ```

Définitions en C

- Voici la définition des types implémentant une liste
- Nous tentons de coller au plus près au cours de Logique
- ```
  struct Element {
      X    valeur;  /* X dépend du problème */
      struct Element *suivant;
  };

  struct Liste {
      struct Element *premier;
  };
  ```
  ```

Primitives

- Les primitives de la logique se traduisent ainsi

VALEUR(p)	p->valeur
SUIVANT(p)	p->suivant
AFFVAL(p,val)	p->valeur = val
AFFSUIV(p,q)	p->suivant = q
ALLOUER(Element)	malloc(sizeof(Element))
LIBERER(p)	free(p)
RIEN	NULL

Exemple

- En guise d'exemple, voyons la fonction qui compte le nombre d'éléments d'une liste d'entiers. D'abord les déclarations
- ```
  struct Element {
      int    valeur;
      struct Element *suivant;
  };

  struct Liste {
      struct Element *premier;
  };
  ```
  ```

Exemple

```
int longueur( struct Liste liste )
{
    int    nb;                  /* Nb d 'élément de la liste     */
    struct Element *p;          /* Element courant de la liste */
    nb = 0;
    p = liste.premier;
    while( p != NULL)
    {
        nb ++;
        p = p->suivant;
    }
    return nb;
}
```

Exemple

- Autre exemple : l'ajout d'un élément en tête.
  La tête de liste doit être modifiée -> on passe son adresse
- ```
  void ajoutTete( struct Liste *liste, int val )
  {
      struct Element *p;
      p = malloc( sizeof(struct Element) ); /* tester p ! */
      p->valeur = val;
      p->suivant = liste->premier;
      liste->premier = p;
      return ;
  };
  ```
  ```

Exemple

- Dernier exemple :
  supprimer le 1° élément et donner sa valeur.
  Retourne un logique pour indiquer si l'opération a pu se faire.

Exemple

```
bool enleveTete( struct Liste *liste, int *val)
{
        bool ok;
        struct Element *p;
        ok = (liste->premier != NULL)
        if (ok)
        {
                p = liste->premier;
                *val = p->valeur;
                liste->premier = liste->premier->suivant;
                free(p);
        }
        return ok;
}
```

Les fichiers binaires

- Ouverture
- Ecriture
- Lecture

Ouverture

- Rappelons qu'un fichier binaire s'ouvre avec une option spéciale

```
fopen( nom_fichier, "rb");      /* lecture */
fopen( nom_fichier, "wb");      /* écriture */
```

- Sur ce genre de fichier, on devrait lire/écrire des octets non formatés d'informations.
- Les fonctions adaptées sont fread et fwrite

Ecriture

```
size_t fwrite( void *adr,  size_t taille,
size_t nblocs, FILE *fd );
```

- On spécifie
  - l ’adresse de la zone mémoire à écrire
  - la taille des blocs
  - le nombre de blocs
  - le flux où écrire
- Elle retourne le nombre de blocs vraiment écrits

Ecriture

- Exemples
  - int n = 3; FILE *fd;
    fd = fopen(…,"wb");
    fwrite (&n, sizeof(int), 1, fd);
  - int t[] = {1,2,3,4,5}; FILE *fd;
    fd = fopen(…,"wb");
    fwrite (t, sizeof(int), 5, fd);
    /* ou */ fwrite (t, sizeof(t), 1, fd);

Lecture

```
size_t fread( void *adr,  size_t taille,
size_t nblocs, FILE *fd );
```

- On spécifie
  - l ’adresse de la zone mémoire à écrire
  - la taille des blocs
  - le nombre de blocs
  - le flux où écrire
- Elle retourne le nombre de blocs vraiment lus

- Exemples
  - int n; FILE *fd;
    fd = fopen(…,"rb");
    fread (&n, sizeof(int), 1, fd);
  - int t[5]; FILE *fd;
    fd = fopen(…,"rb");
    fread (t, sizeof(int), 5, fd);
    /* ou */ fread (t, sizeof(t), 1, fd);

Arguments de la fonction main

- Principe
- Mécanisme
- Exemple

Arguments de la fonction main()

Principe

- C fournit un mécanisme pour retourner une valeur au système (return de la fonction main)
- En sens inverse, il est possible de passer des informations au programme C au début de son exécution (on dira des arguments)

Arguments de la fonction main()

Mécanisme

- Nous avons vu la déclaration de main

  int main (void)

- Sous cette forme, on indique qu'on ne veut pas les arguments.
- L'autre forme est

  int main (int nbarg, char * argv[])

  où
  - nbarg : nombre d'arguments passés par le système
  - argv : table des arguments (des chaînes)

Remarques

- Par définition, le premier argument est toujours le nom du programme
  argv[0] est le nom de l 'exécutable
- C 'est parfois utilisé (en Unix notamment) pour avoir un même exécutable pour plusieurs comportements (distingués par le nom)

Arguments de la fonction main()

Exemple

- Pour afficher tous les arguments
- ```
  #include <stdio.h>
  int main(int nbarg, char * argv[])
  {
      int i;
      printf("Nom du programme %s\n", argv[0]);
      for( i=1; i<nbarg; i++)
        printf( "Argument n° %d : %s\n", i, argv[i]);
      return 0;
  }
  ```
  ```

Remarques

- Tous les arguments sont passés comme des chaînes, même les informations numériques.
- Une première opération sera souvent de convertir les chaînes en nombres (fonctions atoi(), atof(), sscanf(), …)

Arguments de la fonction main()

Passer les arguments

- Comment passe-t ’on les arguments au programme ?
- Cela dépend du système d’exploitation
- En DOS et en Unix, on fait suivre le nom de l’exécutable des arguments à lui passer
- Exemple (DOS) : `dir /p c:`
  La commande `dir` reçoit 3 arguments `"dir"`, `"/p"` et `"c:"`
- L’espace sert de délimiteur entre les arguments

Passer les arguments

- Sur mainframe, on les donne dans la carte EXEC.
- Le nom du programme est « GO »
- Exemple (mainframe) :

```
//CC EXEC EDCCLG,PARM.COMPILE='/SOURCE',
//          PARM.GO='Hello World'
```

- Le programme recevra 3 arguments `"GO"`, `"Hello"` et `"World"`

Fonctions à nombre variable d ’arguments

- Principe
- Mécanisme
- Exemple

Nombre variable d'arguments

Principe

- Certaines fonctions, comme printf acceptent un nombre variable d'arguments
- Avec ce que l'on sait, c'est impossible
- *C* introduit un mécanisme spécial pour traiter ce cas

Nombre variable d'arguments

Prototype

- Le prototype d'une fonction à nombre variable d'arguments est

  définition-de-fonction ≡
  type-valeur-de-retour nom-de-fonction
  (*liste-de-paramètres-avec-types* **, . . .)**

- Il y a donc une liste (non vide) de paramètres nommés « classiques » et la séquence **, . . .** qui indique la présence d'autres paramètres (éventuellement aucun) non nommés

Prototype

- Exemple :

  void brol(int n, ...)

  Cette fonction reçoit un premier argument entier n ainsi qu'une liste variable d'autres arguments
- On n'indique ni le nombre, ni le type de ces arguments.

Appel

- Lors de l'appel, le compilateur vérifie que les arguments nommés sont corrects, en type et ordre mais ne vérifie rien pour les autres
- Exemple :

```
brol(10)          /* OK */
brol(10, 1.2)     /* OK */
brol(10, 'a')     /* OK */
brol()            /* Erreur */
brol(1.2,10)      /* Erreur */
```

Nombre d'arguments

Comment connaître le nombre d 'arguments ?

- La fonction doit connaître d'une façon ou d 'une autre le nombre d'arguments reçus.
- Le plus courant, est qu'elle reçoive cette information via un des arguments nommés (pensez à printf et consorts)
- Exemple :

```
void brol(int nb, ...)
```

nb représente le nombre d'arguments variables

Liste des arguments

Comment connaître les arguments ?

- C fournit des macros pour cela. Voici un exemple qui affiche la liste de ses arguments variables (des entiers)

Nombre variable d'arguments

Exemple

```
#include <stdarg.h>
void brol(int nb, ...)
{
	int i, par;
	va_list adpar;
	va_start(adpar, nb);
	for( i=0; i<nb; i++)
	{
		par = va_arg(adpar, int);
		printf("%d : %d", i, par);
	}
	va_end(adpar);
}
```

Détails

- #include <stdarg.h>
  Les macros utilisées sont définies dans la bibliothèque stdarg.
- va_list adpar
  Il faut déclarer une variable de type va_list .
  Elle contiendra les informations nécessaires au parcours de la liste des arguments.

Détails

- va_start(adpar, nb);
  Initie le processus. Cette étape est nécessaire avant de pouvoir obtenir les arguments. On remarque qu 'on doit donner
  - la variable de type va_list
  - le nom du dernier argument nommé
- va_end(adpar);
  A la fin du processus, va_end est demandé par la norme. Sa non utilisation peut entraîner des problèmes lors des appels de fonctions suivants

Détails

- va_arg(adpar, int);
  L'accès à l'argument suivant (le premier lors de la première utilisation) se fait via va_arg. On doit fournir
  - la variable de type va_list
  - le type de l'argument à prendre
- Attention : la fonction doit donc connaître les types de ses arguments (cf. printf et consorts)

Le tri rapide quicksort

- Principe
- Fonction en paramètre
- Mécanisme
- Exemples

Principe

- Dans de nombreux problèmes, on est amené à trier des valeurs d'un tableau
- C fournit en standard une fonction qui le fait efficacement : qsort().
- Cette fonction utilise l'algorithme appelé Quicksort qui est très efficace pour des tableaux pas trop petits (à partir de quelques dizaines d'éléments)

Principe

- qsort() a été écrit de façon à pouvoir être utilisé dans la plupart des situations :
  - tri d'entiers, de chaînes, de flottants, …
  - tri ascendant, descendant, …
- En effet, le processus est similaire; seuls quelques détails changent. Essentiellement
  - les déclarations
  - la partie qui compare 2 valeurs
- Avant de poursuivre, faisons une digression sur les fonctions en paramètres

Fonction en paramètre

- Une fonction peut recevoir une autre fonction en paramètre. Cela permet de paramètriser des fonctions proches
- Exemple :

```
void brol(int t1[], int t2[], int taille, int f(int, int))
{
    int i;
    for (i=0; i<taille; i++)
        t1[i] = f( t1[i], t2[i] );
    return ;
}
```

Fonction en paramètre

- ```
  int plus(int a, int b) {return a+b;}
  int fois(int a, int b) {return a*b;}
  int mod(int a, int b) {return a%b;}
  …
  #define N 10
  int v[N],w[N];
  brol( v, w, N, plus);
  brol( v, w, N, mod);
  brol( v, w, N, fois);
  ```
  ```

Mécanisme

- Revenons à qsort
- Son prototype est :
- void qsort (
  void * table,
  size_t nelem,
  size_t taille,
  int (*fcompare)(const void*, const void*)
)
- Examinons les paramètres un à un.

Mécanisme

void * table : la table à trier.

- void * table ≡ void table []
- Cette écriture indique une table d'éléments de type non spécifié.
- Ce qui permet de recevoir tout type de table.
- Il faudra évidemment être prudent dans sa manipulation (mais c'est qsort qui s'en charge)

Mécanisme

size_t nelem : le nombre d'éléments du tableau

size_t taille : la taille des éléments de la table.

- Information nécessaire puisque la table est donnée sans son type.

Mécanisme

int (*fcompare)(const void* e1, const void* e1)

- La fonction de comparaison.
- On passe l'adresse de la fonction plutôt que la fonction (cela ne change rien)
- La fonction reçoit l'adresse de ses éléments dont le type n'est pas spécifié
- Elle retourne un entier comme strcmp
  - négatif si *e1 doit rester avant *e2
  - 0 si l 'ordre s 'ils sont égaux
  - positif si *e2 doit passer avant *e1

Exemple

- Prenons un exemple complet. Nous allons trier une table de 1000 entiers en ordre croissant.
- Première étape : écrire la fonction de comparaison
-
```
int intcmp (const void *e1, const void *e2)
{
    int *i1, *i2;
    i1 = (int *) e1; i2 = (int *) e2;
    return *i1-*i2;
} /* Attention : *e1 n 'est pas licite car void !*/
```

Exemple

- Deuxième étape : utiliser qsort.
-
```
#define N 1000
int main(void)
{
    int t[N];
    /* Donner des valeurs aux éléments */
    …
    /* Trier les éléments */
    qsort( (void *) t, N, sizeof(int), intcmp);
    return 0;
}
```

Manipuler le temps

- Principe
- Les types
- Les fonctions

Principe

- La bibliothèque time propose des fonctions pour gérer le temps. Elle permet de
  - connaître le moment d'exécution du programme
  - connaître le temps d'exécution du programme
  - manipuler et formater ces informations
- Nous décrivons ici une sélection de ces fonctions

Les types

- La bibliothèque time introduit 2 types
- time_t représente un temps (date + heure) et est utilisé en interne pour les calculs
- On n'a pas accès à sa structure et on ne doit donc pas y faire explicitement de calcul

Les types

- struct tm représente le même temps sous une forme connue et lisible.

```
struct tm {
int tm_sec;    /* 0 à 59 */
int tm_min;    /* 0 à 59 */
int tm_hour;   /* 0 à 23 */
int tm_mday; /* 1 à 31 */
int tm_mon;    /* 0 à 11 */
int tm_year;   /* année - 1900 */
int tm_wday;  /* Dimanche = 0 */
int tm_yday;  /* 0 à 365 */
int tm_isdst;  /* bool : heure hiver */
}
```

Les fonctions

- time_t time(NULL)
  retourne le temps au moment de l'appel à cette fonction
- struct tm *localtime(time_t)
  convertit le temps en une structure accessible
- char *ctime(time_t)
  convertit le temps en une chaîne de caractères lisible par un humain
  (ex: Jeu 16 Jan 19:15:32 2002)

Le pseudo-aléatoire

- Principe
- Mécanisme

Principe

- Un ordinateur ne peut pas choisir un nombre au hasard.
- On simule le hasard via un algorithme qui génère des nombres qui ont l'air aléatoires (on parle de pseudo-aléatoire)
- C fournit 2 fonctions pour obtenir des nombres pseudo aléatoires (rand() et srand() définies dans la bibliothèque stdlib)

Mécanisme

int rand(void)

- Retourne un nombre *pseudo-aléatoire* entre 0 et RAND_MAX
  (défini par l'implémentation mais > 32766)
- A partir de là, pour obtenir un nombre aléatoire entre 1 et N, on peut écrire

rand()%N+1

- Attention : 2 exécutions du même programme vont produire les mêmes nombres

Mécanisme

void srand(unsigned int graine)

- Initie une nouvelle série pseudo-aléatoire à partir de la valeur de graine.
- Exemple : srand(99)
- Attention : un appel à cette fonction en début de programme avec une constante amènera à la génération de la même suite pseudo-aléatoire à chaque exécution du programme.

Mécanisme

- Pour obtenir une nouvelle suite à chaque exécution, il faut donner une *graine* différente
- On peut la demander à l'utilisateur
- Ou peut utiliser le temps.
- Exemple :

```
time_t t; struct tm tm;
t  = time(NULL);        /* Temps depuis 1/1/1970 */
tm = *localtime(&t);   /* Converti en HHMMSS */
srand( tm.tm_sec + tm.tm_min*60
                 + tm.tm_hour*3600 );
```

Compléments sur les entrées-sorties

- Entrées-sortie et interactivité
- Conversion de chaînes

Problème

- La fonction scanf() peut poser des problèmes dans des programmes interactifs.
- Exemple :

```
int main(void)
{
    int a,b;
    printf("Nombre 1 ? "); scanf("%d", &a);
    printf("Nombre 2 ? "); scanf("%d", &b);
    printf("La somme vaut : %d\n", a+b);
    return 0;
}
```

Problème

- Ce programme demande de respecter l'ordre des questions et des réponses, ce que ne fait pas le scanf().
- Exemple :
  Nombre 1 ? 3 4
  Nombre 2 ? La somme vaut : 7
- Car un \n à la même valeur qu'un espace
- Il faudrait pouvoir éliminer le reste non utilisé d'une ligne.

gets

- Un début de solution est apporté par gets().

  char *gets(char* chaine)

- Cette fonction lit une ligne de données (seul le \n sert donc de délimiteur).
- La ligne est rangée dans ‘ chaine ’.
- Elle retourne ‘ chaine ’ ou NULL si une erreur s’est produite (fin des données par exemple)
- Il existe l’équivalent pour les fichiers : fgets()

Conversion de chaînes

- Ce n'est qu'un début de solution car on ne lit que des chaînes.
- Pour convertir cette chaîne en un nombre, plusieurs solutions.
- 1) Utiliser des fonctions explicites de conversion : atoi, atof , … définies dans stdlib.
- Exemple : atoi(chaine)
  'chaine' commence par un nombre entier; la fonction retourne ce nombre

Lire une chaîne

- 2) Les fonctions sscanf(), sprintf() , offrent l 'équivalent des fonctions de lecture et d'écriture MAIS dans une chaîne.
- Exemple : sscanf(chaine, "%d", &a)
  'chaine' commence par un nombre entier; la fonction assigne ce nombre à ' a '.
- Exemple : sscanf(chaine, "%d%d", &a, &b)
  'chaine' commence par 2 entiers séparés par un espace; la fonction les extrait et les assigne à ' a ' et ' b '

Résumé

- Le programme du début peut s'écrire
- ```
  int main(void)
  {
      int a,b;
      char s[80];

      printf("Nombre 1 ? ");
      gets(s); sscanf(s, "%d", &a);
      printf("Nombre 2 ? ");
      gets(s); sscanf(s, "%d", &b);
      printf("La somme vaut : %d\n", a+b);
      return 0;
  }
  ```
  ```