Techniques de Programmation Algorithmique & Langage C

Manuel Munier

IUT des Pays de l'Adour - Mont de Marsan

Département GTR

2004-2005

Plan du cours

- Algorithmique & Python
  - variables, opérateurs, expressions
  - séquence d'instructions
  - conditionnelle (`if`)
  - boucles (`while`, `for`)
  - fonctions
- Langage C
  - types, instructions C, compilation, fonctions, tableaux, pointeurs, listes, arbres, fichiers, récursivité,... bref, de quoi s'occuper ;-)

Partie n°2

Langage C
Norme ANSI

Quelques liens

- http://www.multimania.com/dancel
- http://www.inf.enst.fr/~charon/CFacile
- http://www.ltam.lu/Tutoriel_Ansi_C
- http://www.esil.univ-mrs.fr/~tourai/main/Enseignement.html
- http://www-ipst.u-strasbg.fr/pat/program/index.htm
- http://www-igm.univ-mlv.fr/~dr/Cours.html

Introduction

- Historique
  - 1971 : D.Ritchie et K.Thompson pour Unix
  - 1983 : début de normalisation par l'A.N.S.I.
  - 1988 : norme "C ANSI" approuvée en 1988
- Evolution
  - C++ : langage C enrichi des concepts objet
  - Java : langage orienté objets (GTR2)

Introduction

- Caractéristiques
  - Structuré, typé, "portable", puissant, flexible
  - À mi-chemin entre un langage de haut niveau et un assembleur
  - Langage réduit mais complété par des bibliothèques de fonctions normalisées
  - Utilisable sur matériel et système quelconque
  - Nombreux domaines d'application

Généralités

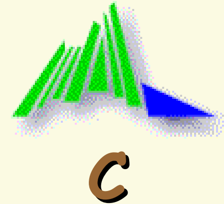
- Structure d'un programme C (3 niveaux)

Fichier	**Fonction**	**Bloc**

variables du fichier
 	
fonction_1
 	
…
 	
fonction_n
	en-tête de fonction
	
	bloc (corps de la fonction)
		{
		variables du bloc
		
		<*suite d'instructions*>
		
		}

- Une fonction particulière unique: `main`

Généralités



- Création d'un programme C

Éditeur de texte

fichier source

Pré-processeur C

fichier source modifié

Compilateur C

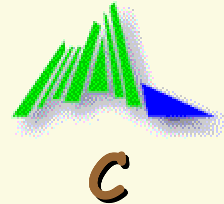
autres fichiers objet, bibliothèques standard

fichier objet

Édition de liens

fichier exécutable

Généralités



- Premier programme en C

```
#include <stdio.h>      /* fichiers d'en-têtes */
#include <math.h>

main()                  /* En-tête de la fonction principale */
{                       /* Début du corps de la fonction principale */
    float x;            /* Variable de bloc (de la fonction) */
    printf("Entrez le nombre dont vous voulez la racine carree\n");
    scanf("%f",&x);
    if (x<0.)
        printf("Le nombre %f ne possède pas de racine carree\n",x);
    else
    {
        float racx;     /* Variable locale à ce bloc */
        racx = sqrt(x);
        printf("Le nombre %f a pour racine carree %f\n",x,racx);
    }
    printf("Au revoir\n");
}   /* Fin du corps de la fonction principale */
```

Généralités

- Quelques règles d'écriture
  - Notion d'**identificateur**
  - Lettres, chiffres, caractère souligné
    - corrects : `iut_gtr2` `GTR54_14`
    - incorrects : `6_de_pique` `66`
  - Différence minuscules/majuscules
    - `GTR`, `gtr` et `Gtr` sont 3 identificateurs différents
  - 31 premiers caractères significatifs

Généralités

- Mots réservés (toujours en minuscules !)

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

Généralités

- Mise en forme
  - Séparateurs entre les mots (espace, return)
  - Indentation ➔ meilleure lisibilité
- Commentaires
  - Texte explicatif destiné au lecteur

```
/* ceci est un commentaire */

/* ceci est un commentaire
   plus long qui s'étend
   sur trois lignes */
```

Généralités

- Fichiers en-tête
  - Ensemble de déclarations et de définitions pour l'utilisation de fonctions prédéfinies

```
<assert.h> <float.h>  <math.h>   <stdarg.h> <stdlib.h>
<ctype.h>  <limits.h> <setjmp.h> <stddef.h> <string.h>
<errno.h>  <locale.h> <signal.h> <stdio.h>  <time.h>
```

- Directive d'inclusion

```
#include nom_fichier

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

Plan

- Instructions élémentaires
  - types,variables, opérateurs, expressions
  - affichage/lecture de données (`printf`/`scanf`)
- Instructions de contrôle
  - exécution conditionnelle (`if`,`switch`)
  - boucles (`for`,`while`,`do`…`while`)
- Tableaux, chaînes de caractères
- Fonctions
- Pointeurs
- Structures de données
- Fichiers

Types & Variables

- Notion de **type** (de données)
  - Pourquoi des types ?
  - A quoi fait référence un type ?
    - Signification de l'information
    - Règles de représentation technologique: codage
    - Ensemble de définitions et d'opérateurs
  - Taille d'un type
    - Nombre d'octets nécessaires pour représenter les valeurs possibles

Types & Variables

- Types simples
  - Donnée élémentaire: entier, réel, caractère
  - Type identifié par un seul mot-clé

Type	Mot-clé
nombre entier	`int`
nombre réel	`float`
caractère	`char`

- Types dérivés
  - Construits à partir d'un ou plusieurs autres types (tableaux, tuples,…)

Types & Variables

- Déclaration de variable

  [<*id_classe_stockage*>] <*id_type*> <*id_variable*> ;

  - Exemples

```
int compteur;     /* variable entière */
float moyenne;    /* variable réelle */
char reponse;     /* variable caractère */
static int i;     /* entier i avec
                     classe de stockage */
```

Types & Variables

- Déclaration étendue
  - `int i,j,k,t;`
- Initialisation à la déclaration
  - `int i = 1;`
  - `float moy = 0.0;`
  - `char rep = 'O';`
  - `int i, j=8, k, t=0;`

Types & Variables

- Les types entiers
  - `int`
  - `short   /* abréviation de short int */`
  - `long    /* abréviation de long int */`

Type	Taille habituelle	Borne inférieure	Borne supérieure
short	2 octets	-32 768	+32 767
int	2 octets ou 4 octets	-32 768 -2 147 483 648	+32 767 +2 147 483 647
long	4 octets	-2 147 483 648	+2 147 483 647

- Les entiers non signés
  - `unsigned`

Types & Variables

- Notation des constantes entières
  - Notation décimale
    - `56    +467    -140`
    - `1896534L  /* constante entier long */`
    - `55228U    /* constante entier non signé */`
  - Notation octale
    - `0177`
    - `027`
  - Notation hexadécimale
    - `0x34AB`
    - `0xFFFF`

Types & Variables

- Rapidement…
  - Déclarer des variables entières

```
int a,i,j;
short cpt;
unsigned int stock;
long lim_inf = -2563424L, lim_sup;
```

  - Lire des variables entières

```
scanf("%d",&a);           /* un int */
scanf("%d%d",&i,&j);      /* 2 entiers i et j */
scanf("%hd",&cpt);        /* un short */
scanf("%ud",&stock);      /* un non signé */
scanf("%ld",&lim_sup);    /* un long */
```

Types & Variables

- Rapidement…
  - Déclarer des variables entières

```
int a,i,j;
short cpt;
unsigned int stock;
long lim_inf = -2563424L, lim_sup;
```

  - Afficher des variables entières

```
printf("%d",a);            /* un int */
printf("%hd",cpt);         /* un short */
printf("%ud",stock);       /* un non signé */
printf("%ld",lim_sup);     /* un long */
printf("I vaut %d et J vaut %d",i,j);
```

Types & Variables

- Les types flottants
  - **float** /* simple précision (erreur < à 10^{-6}) */
  - **double** /* double précision (erreur < à 10^{-10}) */
  - **long double** /* précision étendue */
- Notations de constantes réelles
  - Notation décimale
    - 56. -140.125 .14 /* doubles par défaut */
    - 326.46**F** /* simple précision */
    - -125365474.**L** /* flottant long */
  - Notation scientifique
    - 14.25E4 (14.25×10^4)
    - -527e-22

Types & Variables

- Rapidement…
  - Déclarer des variables réelles

```
float note,montant;
double mesure;
long double dist = 2563424L;
...
```

  - Lire et afficher des variables réelles

```
...
scanf("%f",&note);       /* lire un float */
printf("%f",montant);    /* afficher un float */
...
```

Types & Variables

- Le type caractère: `char`
  - Codé sur un octet
  - Considéré comme un type entier
- Notations des constantes
  - Caractères imprimables:
    - `'a' 'B' '+' '$'`
  - Caractères spéciaux:
    - notation particulière telle que `'\n'` ou `'\t'`
  - Notations octales et héxadécimales:
    - `'A' '\x41' '\101'`
  - Caractère NUL noté `'\0'`

Types & Variables

- Rapidement…
  - Lire et afficher un caractère: 1ère méthode

```
char rep;
...
scanf("%c",&rep); fflush(stdin);
printf("%c",rep); fflush(stdout);
```

  - Lire et afficher un caractère: 2ème méthode (on considère ici le caractère comme un entier)

```
int rep;
...
rep = getchar();    /* lecture */
putchar(rep);       /* édition */
```

Types & Variables

- Les chaînes de caractères
  - Plus vraiment un type simple
  - Suite de caractères (tableau) terminée par un caractère '\0'
- Notations
  - `"Bonjour"   "Bonjour\n"   "\t\tBonjour\n"`
  - `"B"   " "   ""`
- Édition de constantes chaînes
  - `printf ("Bonjour\n");`
  - `puts("Bonjour");`

Types & Variables

- Un type spécial: le type indéfini `void`
  - Ex: quand une fonction de renvoie pas résultat

- Et les valeurs logiques ?
  - Pas de type booléen en C
  - Convention:
    - FAUX ⇔ zéro (valeur nulle du type)
    - VRAI ⇔ non zéro (autres valeurs du type)

Types & Variables

- Les constantes littérales
  - Directive du pré-processeur

    `#define <nom_symbolique> <constante>`
  - Notations
    - `#define PI 3.14159`
    - `#define FAUX 0`
    - `#define VRAI 1`
    - `#define LIMITE 326`
  - Utilisation

```
void main()
{ float rayon = 3.54;
  float circonference = 2 * PI * rayon;
  ...
```

Types & Variables

- Les constantes littérales (suite)
  - En fait, la directive **#define** indique au pré-processeur de **remplacer** chaque occurrence du nom symbolique par la valeur indiquée

- Les variables à valeur constante

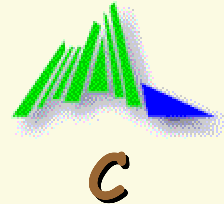
```
const float e = 2.7183;
```

  - Le mot-clé **const** interdit toute modification ultérieure de la valeur de la variable

Opérateurs

- Généralités
  - Notion d'**opérateur**
    - ➔opérateurs unaires, binaires
  - Notion d'**expression**
    - ➔combinaison d'opérateurs, de constantes, d'identificateurs et/ou d'expressions
  - Comment une expression est-elle évaluée ?
  - Et les expressions mixtes ?
    - ➔combinaison d'opérandes de types différents

Opérateurs



- Les opérateurs arithmétiques

Symbole	Nombre d'opérandes	Opération	Type opérande(s)	Type résultat
+	unaire	**valeur positive**	entier réel	entier réel
-	unaire	**valeur opposée**	idem	idem
+	binaire	**addition**	entier réel entier,réel	entier réel réel
-	binaire	**soustraction**	idem	idem
*	binaire	**multiplication**	idem	idem
/	binaire	**division**	idem	idem
%	binaire	**modulo (reste)**	entier (uniquement)	entier

Opérateurs

- Expressions mixtes: règles de conversion
  - Conversion d'**ajustement de type**

    `int → long → float → double → long double`

  - **Promotion numérique** (conversion systématique)

    `char → int`

    `short → int`

Opérateurs

- Les opérateurs relationnels

Symbole	Opération
==	égal à
!=	différent de
<	inférieur strictement à
<=	inférieur ou égal à
>	supérieur strictement à
>=	supérieur ou égal à

FAUX → 0
VRAI → 1

- Les opérateurs logiques

Symbole	Opération
!	NON
&&	ET
||	OU

FAUX → 0
VRAI → 1

Opérateurs

- Table de vérité des opérateurs logiques

Opérande 1	Opération	Opérande 2	Résultat
0	&&	0	0
0	&&	Non Zéro	0
Non Zéro	&&	0	0
Non Zéro	&&	Non Zéro	1
0	||	0	0
0	||	Non Zéro	1
Non Zéro	||	0	1
Non Zéro	||	Non Zéro	1
	!	0	1
	!	Non Zéro	0

FAUX → 0
VRAI → 1

Opérateurs

- Particularités des opérateurs logiques en C
  - Dans le cas d'un **ET**, l'évaluation des opérandes s'arrête dès qu'un des opérandes est évalué à **FAUX**
    - **faux ET** *qqch* ➔ **faux**
  - Dans le cas d'un **OU**, l'évaluation s'arrête dès qu'un opérande est évalué à **VRAI**
    - **vrai OU** *qqch* ➔ **vrai**

Opérateurs

- L'opérateur d'affectation simple
  - Notion de **lvalue** (left value)
  - Syntaxe d'une affectation

    `<lvalue> = <expression>`

  - Principe d'évaluation
    - on évalue l'expression à droite du signe =
    - le résultat (une valeur) est recopié dans la lvalue
  - Dans le cas d'expression mixte

    ➔ conversion systématique (`char` → `int`, etc…)

Opérateurs

- Les opérateurs d'affectation élargie
  - la notation

    `<lvalue> <op>= <expression>`
  - est équivalente à

    `<lvalue> = <lvalue> <op> (<expression>)`
  - avec
    - `<op>` ∈ { +, -, *, /, %, >>, <<, &, |, ^ }

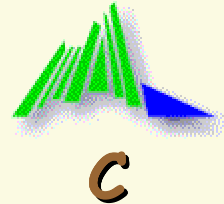
+=	-=	*=	/=	%=	>>=	<<=	&=	|=	^=

```
k += 4;              /* équiv. à k=k+4 */
cumul += montant;    /* cumul = cumul+montant */
r *= (c+3);          /* r = r * (c+3) */
```

Opérateurs

- Quelques opérateurs spéciaux
  - post in(dé)crémentation
    - `<lvalue> ++        /* post incrémentation */`
    - `<lvalue> --        /* post décrémentation */`
  - pré in(dé)crémentation
    - `++ <lvalue>        /* pré incrémentation */`
    - `-- <lvalue>        /* pré décrémentation */`
  - Fonction: ajouter (soustraire) 1 à une lvalue
    - post... ➔ après l'évaluation
    - pré... ➔ avant l'évaluation

Opérateurs



- Quelques opérateurs spéciaux
  - Exemple de post incrémentation

```
a=5; b=3; c=0;    /* a vaut 5, b vaut 3, c vaut 0 */
a++;              /* a vaut 6, b vaut 3, c vaut 0 */
c=b+(a++);        /* a vaut 7, b vaut 3, c vaut 9 */
```

  - Exemple de pré incrémentation

```
a=5; b=3; c=0;    /* a vaut 5, b vaut 3, c vaut 0 */
++a;              /* a vaut 6, b vaut 3, c vaut 0 */
c=b+(++a);        /* a vaut 7, b vaut 3, c vaut 10*/
```

Opérateurs

- L'opérateur de conversion de type (**cast**)

  (<*type_cible*>) <*opérande_expression*>

  - Exemple:

```
...
int n=7, p=2, q_entier;
float q_reel;
...
q_entier = n/p;              /* q_entier vaut 3 */
q_reel = (float)(n/p);       /* q_reel vaut 3.0 */
q_reel = (float)n/p;         /* q_reel vaut 3.5 */
```

Opérateurs

- L'opérateur **sizeof**
  - Renvoie la taille en octets...
    - `sizeof <objet>          /* taille d'une variable */`
    - `sizeof (<id_type>)    /* taille d'un type */`
  - Exemples:

```
...
double x;
int a;
...
a = sizeof x;          /* a vaut 8 */
a = sizeof(double); /* a vaut 8 */
a = sizeof(char);    /* a vaut 1 */
```

Opérateurs

- L'opérateur **séquentiel**

```
<expression_1>, <expression_2>
```

  - Exemple:
    - `i++, j+=k;   /* est équivalent à i++; j+=k; */`

- L'opérateur **conditionnel**

```
<expr_1> ? <expr_2> : <expr_3>
```

  - Exemple:
    - ```
      max = (a>b) ? a : b;  /* max reçoit le plus grand
                               de a et de b */
      ```
    - `(montant>3000.) ? montant*=0.05 : montant*=0.03;`
      ```

Opérateurs (priorités)

Catégorie	Opérateurs	Associativité
référence	( ) [ ] -> .	→
unaires	+ - ++ -- ! * & (cast) sizeof	←
multiplicatifs	* / %	→
additifs	+ -	→
décalage	<< >>	→
relationnels	< <= > >=	→
relationnels	== !=	→
manip. bits	&	→
manip. bits	^	→
manip. bits	|	→
logique	&&	→
logique	||	→
conditionnel	? :	→
affectations	= += -= *= /= %= &= ^= |= <<= >>=	←
séquentiel	,	→

priorité décroissante

Opérateurs & Expressions

Instructions	x	y	z	expr. gauche
`int x,y,z;`				
`x = -3 + 4 * 5 - 6;`				
`x = 3 + 4 % 5 - 6;`				
`x = -3 * 4 % 6 + 5;`				
`x *= 3 + 5;`				
`x *= (y = (z = 4));`				
`x = y == z;`				
`x == (y = z);`				
`x=0; y=-1; z=0;`				
`x = x && y || z;`				
`z = x++ - 1;`				
`z+=-x++++y;`				
`x = z>0 && x<y;`				

Opérateurs & Expressions

Instructions	x	y	z	expr. gauche
`int x,y,z;`				
`x = (((-3) + (4 * 5)) - 6);`	11			11
`x = 3 + 4 % 5 - 6;`				
`x = -3 * 4 % 6 + 5;`				
`x *= 3 + 5;`				
`x *= (y = (z = 4));`				
`x = y == z;`				
`x == (y = z);`				
`x=0; y=-1; z=0;`				
`x = x && y || z;`				
`z = x++ - 1;`				
`z+=-x++++y;`				
`x = z>0 && x<y;`				

Opérateurs & Expressions

<table>
<tr><th>Instructions</th><th>x</th><th>y</th><th>z</th><th>expr. gauche</th></tr>
<tr><td>int x,y,z;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = (((-3) + (4 * 5)) - 6);</td><td>11</td><td></td><td></td><td>11</td></tr>
<tr><td>x = ((3 + (4 % 5) - 6);</td><td>1</td><td></td><td></td><td>1</td></tr>
<tr><td>x = -3 * 4 % 6 + 5;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x *= 3 + 5;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x *= (y = (z = 4));</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = y == z;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x == (y = z);</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x=0; y=-1; z=0;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = x && y || z;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>z = x++ - 1;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>z+=-x+++++y;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = z>0 && x<y;</td><td></td><td></td><td></td><td></td></tr>
</table>

Opérateurs & Expressions

<table>
<tr><th>Instructions</th><th>x</th><th>y</th><th>z</th><th>expr. gauche</th></tr>
<tr><td>int x,y,z;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = (((-3) + (4 * 5)) - 6);</td><td>11</td><td></td><td></td><td>11</td></tr>
<tr><td>x = ((3 + (4 % 5) - 6);</td><td>1</td><td></td><td></td><td>1</td></tr>
<tr><td>x = ((((-3) * 4) % 6) + 5);</td><td>5</td><td></td><td></td><td>5</td></tr>
<tr><td>x *= 3 + 5;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x *= (y = (z = 4));</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = y == z;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x == (y = z);</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x=0; y=-1; z=0;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = x && y || z;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>z = x++ - 1;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>z+=-x++++y;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = z>0 && x<y;</td><td></td><td></td><td></td><td></td></tr>
</table>

Opérateurs & Expressions

Instructions	x	y	z	expr. gauche
`int x,y,z;`				
`x = (((-3) + (4 * 5)) - 6);`	11			11
`x = ((3 + (4 % 5) - 6);`	1			1
`x = ((((-3) * 4) % 6) + 5);`	5			5
`x *= (3 + 5);`	40			40
`x *= (y = (z = 4));`				
`x = y == z;`				
`x == (y = z);`				
`x=0; y=-1; z=0;`				
`x = x && y || z;`				
`z = x++ - 1;`				
`z+=-x++++y;`				
`x = z>0 && x<y;`				

Opérateurs & Expressions

Instructions	x	y	z	expr. gauche
int x,y,z;				
x = (((-3) + (4 * 5)) - 6);	11			11
x = ((3 + (4 % 5) - 6);	1			1
x = ((((-3) * 4) % 6) + 5);	5			5
x *= (3 + 5);	40			40
x *= (y = (z = 4));	160	4	4	160
x = y == z;				
x == (y = z);				
x=0; y=-1; z=0;				
x = x && y || z;				
z = x++ - 1;				
z+=-x++++y;				
x = z>0 && x<y;				

Opérateurs & Expressions

<table>
<tr><th>Instructions</th><th>x</th><th>y</th><th>z</th><th>expr. gauche</th></tr>
<tr><td>int x,y,z;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = (((-3) + (4 * 5)) - 6);</td><td>11</td><td></td><td></td><td>11</td></tr>
<tr><td>x = ((3 + (4 % 5) - 6);</td><td>1</td><td></td><td></td><td>1</td></tr>
<tr><td>x = ((((-3) * 4) % 6) + 5);</td><td>5</td><td></td><td></td><td>5</td></tr>
<tr><td>x *= (3 + 5);</td><td>40</td><td></td><td></td><td>40</td></tr>
<tr><td>x *= (y = (z = 4));</td><td>160</td><td>4</td><td>4</td><td>160</td></tr>
<tr><td>x = (y == z);</td><td>1</td><td>4</td><td>4</td><td>1</td></tr>
<tr><td>x == (y = z);</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x=0; y=-1; z=0;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = x && y || z;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>z = x++ - 1;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>z+=-x++++y;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = z>0 && x<y;</td><td></td><td></td><td></td><td></td></tr>
</table>

Opérateurs & Expressions

<table>
<tr><th>Instructions</th><th>x</th><th>y</th><th>z</th><th>expr. gauche</th></tr>
<tr><td>int x,y,z;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = (((-3) + (4 * 5)) - 6);</td><td>11</td><td></td><td></td><td>11</td></tr>
<tr><td>x = ((3 + (4 % 5) - 6);</td><td>1</td><td></td><td></td><td>1</td></tr>
<tr><td>x = ((((-3) * 4) % 6) + 5);</td><td>5</td><td></td><td></td><td>5</td></tr>
<tr><td>x *= (3 + 5);</td><td>40</td><td></td><td></td><td>40</td></tr>
<tr><td>x *= (y = (z = 4));</td><td>160</td><td>4</td><td>4</td><td>160</td></tr>
<tr><td>x = (y == z);</td><td>1</td><td>4</td><td>4</td><td>1</td></tr>
<tr><td>x == (y = z);</td><td>1</td><td>4</td><td>4</td><td>0</td></tr>
<tr><td>x=0; y=-1; z=0;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = x && y || z;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>z = x++ - 1;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>z+=-x++++y;</td><td></td><td></td><td></td><td></td></tr>
<tr><td>x = z>0 && x<y;</td><td></td><td></td><td></td><td></td></tr>
</table>

Opérateurs & Expressions

Instructions	x	y	z	expr. gauche
int x,y,z;				
x = (((-3) + (4 * 5)) - 6);	11			11
x = ((3 + (4 % 5) - 6);	1			1
x = ((((-3) * 4) % 6) + 5);	5			5
x *= (3 + 5);	40			40
x *= (y = (z = 4));	160	4	4	160
x = (y == z);	1	4	4	1
x == (y = z);	0	4	4	0
x=0; y=-1; z=0;	0	-1	0	
x = x && y || z;				
z = x++ - 1;				
z+=-x++++y;				
x = z>0 && x<y;				

Opérateurs & Expressions

Instructions	x	y	z	expr. gauche
int x,y,z;				
x = (((-3) + (4 * 5)) - 6);	11			11
x = ((3 + (4 % 5) - 6);	1			1
x = ((((-3) * 4) % 6) + 5);	5			5
x *= (3 + 5);	40			40
x *= (y = (z = 4));	160	4	4	160
x = (y == z);	1	4	4	1
x == (y = z);	0	4	4	0
x=0; y=-1; z=0;	0	-1	0	
x = ((x && y) || z);	0	-1	0	0
z = x++ - 1;				
z+=-x++++y;				
x = z>0 && x<y;				

Opérateurs & Expressions

Instructions	x	y	z	expr. gauche
int x,y,z;				
x = (((-3) + (4 * 5)) - 6);	11			11
x = ((3 + (4 % 5) - 6);	1			1
x = ((((-3) * 4) % 6) + 5);	5			5
x *= (3 + 5);	40			40
x *= (y = (z = 4));	160	4	4	160
x = (y == z);	1	4	4	1
x == (y = z);	0	4	4	0
x=0; y=-1; z=0;	0	-1	0	
x = ((x && y) || z);	0	-1	0	0
z = (x++) - 1;	1	-1	-1	-1
z+=-x++++y;				
x = z>0 && x<y;				

Opérateurs & Expressions

Instructions	x	y	z	expr. gauche
`int x,y,z;`				
`x = (((-3) + (4 * 5)) - 6);`	11			11
`x = ((3 + (4 % 5) - 6);`	1			1
`x = ((((-3) * 4) % 6) + 5);`	5			5
`x *= (3 + 5);`	40			40
`x *= (y = (z = 4));`	160	4	4	160
`x = (y == z);`	1	4	4	1
`x == (y = z);`	0	4	4	0
`x=0; y=-1; z=0;`	0	-1	0	
`x = ((x && y) || z);`	0	-1	0	0
`z = (x++) - 1;`	1	-1	-1	-1
`z += ((-(x++)) + (++y));`	2	0	-2	-2
`x = z>0 && x<y;`				

Opérateurs & Expressions

Instructions	x	y	z	expr. gauche
`int x,y,z;`				
`x = (((-3) + (4 * 5)) - 6);`	11			11
`x = ((3 + (4 % 5) - 6);`	1			1
`x = ((((-3) * 4) % 6) + 5);`	5			5
`x *= (3 + 5);`	40			40
`x *= (y = (z = 4));`	160	4	4	160
`x = (y == z);`	1	4	4	1
`x == (y = z);`	0	4	4	0
`x=0; y=-1; z=0;`	0	-1	0	
`x = ((x && y) || z);`	0	-1	0	0
`z = (x++) - 1;`	1	-1	-1	-1
`z += ((-(x++)) + (++y));`	2	0	-2	-2
`x = ((z>0) && (x<y));`	0	0	-2	0

Plan

- Instructions élémentaires
  - types,variables, opérateurs, expressions
  - affichage/lecture de données (`printf`/`scanf`)
- Instructions de contrôle
  - exécution conditionnelle (`if`,`switch`)
  - boucles (`for`,`while`,`do…while`)
- Tableaux, chaînes de caractères
- Fonctions
- Pointeurs
- Structures de données
- Fichiers

Instructions

- 4 "catégories" d'instructions:
  - instruction **nulle**
    - syntaxe:

```
;
```

  - instruction **simple**
    - syntaxe:

```
<expression> ;
```

  - instruction composée (ou **bloc d'instructions**)
    - syntaxe:

```
{ <liste de déclarations>

  <liste d'instructions>
}
```

  - instruction de **contrôle** (de flux)

Instructions de contrôle

- Objectif (rappel): modifier la structure séquentielle du programme
  - répétition
  - prise de décision conditionnelle
  - déroutement inconditionnel
- 2 catégories
  - instructions structurées
  - instructions de branchement

Instructions de contrôle

- Classification:

	instructions structurées	instructions de branchement

boucles	`for`
	`while`
	`do … while`	

choix	`if [… else]` `else if`	
	`switch`	`switch`

déroutements		`break`
		~~`continue`~~
		`return`
		~~`goto`~~

Conditionnelle if

- Instruction **if**

```
if (<expression_booleenne>)
    <instruction>
```

  - Ex (instruction simple ou expression)

```
float note;
scanf("%f",&note);
if ( (note>=0.0) && (note<=20.0) )
    print("Note valide\n");
```

Conditionnelle if

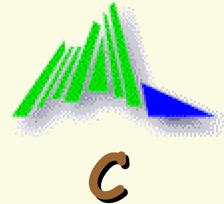
- Instruction if

```
if (<expression_booleenne>)
      <instruction>
```

  - Ex (instruction composée ou bloc)

```
...
delta = b*b - (4*a*c);
if (delta > 0.0)
      { float x1,x2;
        x1 = (-b - sqrt(delta)) / (2*a);
        x2 = (-b + sqrt(delta)) / (2*a);
        printf("%f%f\n",x1,x2);
      }
```

Conditionnelle `if`




```
/* Programme qui calcule, à partir d'un horaire donné
   (hh:mm:ss), l'horaire une seconde après (même format) */
#include <stdio.h>
main()
{ short h,m,s;
  printf("Entrer Heures/Minutes/Secondes\n");
  scanf("%hd %hd %hd",&h,&m,&s);
  s++;
  if (s==60)
    { s=0; m++;
      if (m==60)
        { m=0; h++;
          if (h==24)
            h=0;
        }
    }
  printf("%hd %hd %hd\n",h,m,s);
}
```

Conditionnelle if

- Instruction **if... else**

```
if (<expression_booleenne>)
    <instruction_1>
else
    <instruction_2>
```

  - Exemple

```
...
if (moyenne >= 10.0)
    printf("Etudiant RECU\n");
else
    printf("Etudiant en 2nde SESSION\n");
...
```

Conditionnelle if

- Instruction if... else
  - Règle (imbrication des if): un else se rapporte toujours au dernier if rencontré auquel aucun else n'a encore été attribué
  - Exemple

```
if (a<=b) if (b<=c) printf("Vrai\n");
     else printf("Faux\n");
```

  - Ce programme affiche:
    - Vrai si a≤b ≤ c
    - Faux si a ≤ b et b>c
    - rien dans le cas où a>b

Conditionnelle switch

- Passage du if... else au switch

```
/* Programme qui simule une calculatrice arithmétique */
#include <stdio.h>
main()
{ int a,b;  /* les 2 opérandes */
  char op;  /* le symbole de l'opérateur */
  printf("Entrer les deux opérandes entières:\n");
  scanf("%d %d",&a,&b);
  printf("Entrer un opérateur (+,-,* ou /):\n");
  scanf("%c",&op);
  printf("Le résultat est: ");
  if (op=='+') printf("%d\n",a+b);
  else if (op=='-') printf("%d\n",a-b);
       else if (op=='*') printf("%d\n",a*b);
            else if (op=='/') printf("%d\n",a/b);
                 else printf("opérateur '%c' inconnu !\n",op);
}
```

Conditionnelle switch

- Instruction switch
  - Syntaxe:

```
switch (<expression_entière>)
  { case <constante_1>: [<suite_inst_1>];
    case <constante_2>: [<suite_inst_2>];
    ...
    case <constante_n>: [<suite_inst_n>];
    [default: <suite_inst_par_defaut>]
  }
```

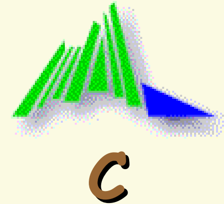
➔ Fonctionnement

Conditionnelle switch

- Instruction switch

```
/* Programme qui simule une calculatrice arithmétique */
#include <stdio.h>
main()
{ int a,b;  /* les 2 opérandes */
  char op;  /* le symbole de l'opérateur */
  ...
  switch (op)
  {
    case '+': printf("%d\n",a+b);
    case '-': printf("%d\n",a-b);
    case '*': printf("%d\n",a*b);
    case '/': printf("%d\n",a/b);
    default: printf("opérateur '%c' inconnu !\n",op);
  }
}
```

Conditionnelle switch



- Instruction switch

```
Entrer les deux opérandes entières:
3
4
Entrer un opérateur (+,-,* ou /):
+
Le résultat est: 7
-1
12
0
opérateur '+' inconnu !
```

Annotations: 7 → cas + ; -1 → cas - ; 12 → cas * ; 0 → cas / ; ! → default

- Pb: comment n'exécuter qu'un seul cas ?

Conditionnelle switch

- Instruction switch + break

```
/* Programme qui simule une calculatrice arithmétique */
#include <stdio.h>
main()
{ int a,b;  /* les 2 opérandes */
  char op;  /* le symbole de l'opérateur */
  ...
  switch (op)
  {
    case '+': printf("%d\n",a+b); break;
    case '-': printf("%d\n",a-b); break;
    case '*': printf("%d\n",a*b); break;
    case '/': printf("%d\n",a/b); break;
    default: printf("opérateur '%c' inconnu !\n",op);
  }
}
```

notez l'absence de bloc pour les différents cas

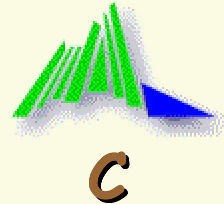
Conditionnelle switch

- Instruction **switch** + **break**
  - Le **break** interrompt l'exécution du **switch**
    - en fait, on n'a qu'un seul bloc d'instructions
    - les différents case peuvent être vus comme des « étiquettes »

  ➔ pas de délimiteur de bloc pour les cas

  - C'est la **<u>seule et unique</u>** utilisation du **break** qui vous est accordée !!!

Boucle **while**



- Instruction **while**

```
while (<expression_booléenne>)
    <instruction>
```

```c
/* Nombre de chiffres composant un nombre */
#include <stdio.h>
main()
{ int n,q,nbc;
  printf("Entrer un nombre entier:\n"); scanf("%d",&n);
  nbc = 1;
  q = n/10;
  while (q != 0)
    { q = q/10;
      nbc++;
    }
  printf("Le nombre %d est composé de %d chiffre(s)\n",n,nbc);
}
```

Boucle while

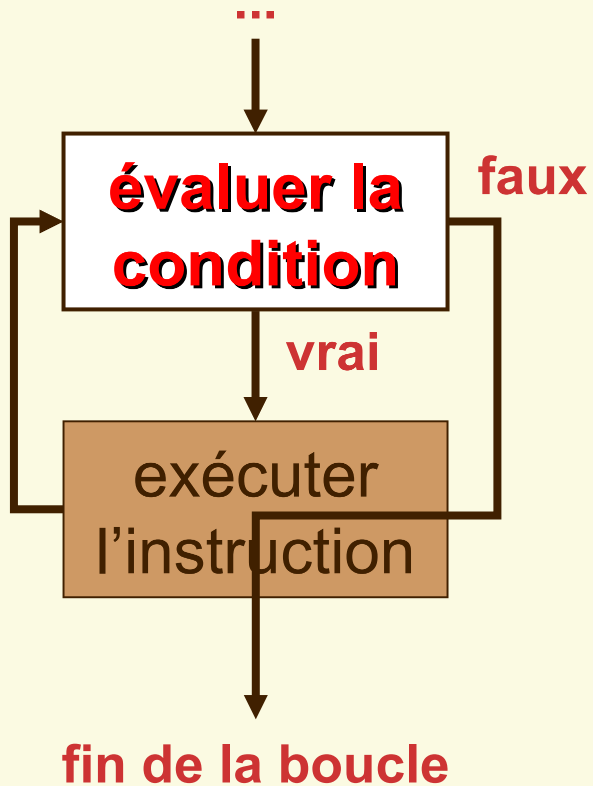
- Variante: instruction do...while

```
do
    <instruction>
while (<expression_booléenne>)
```

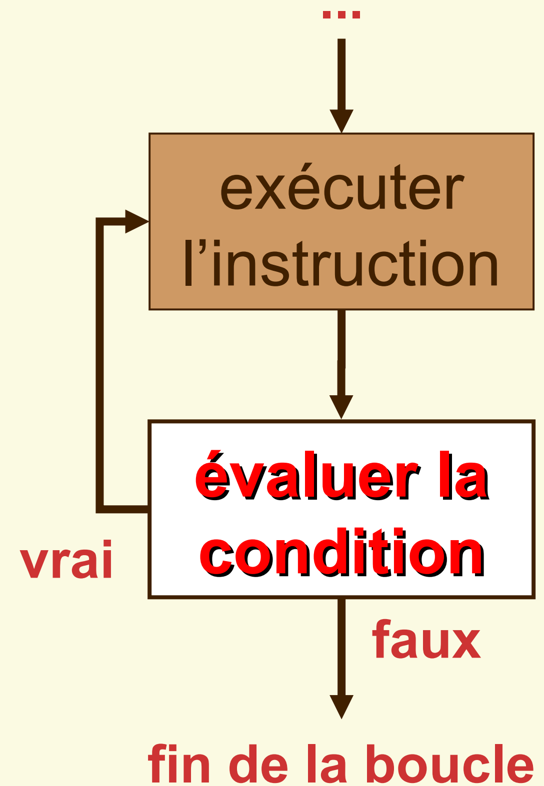
```
/* Contrôle de la validité d'une valeur numérique */
...
do
{
  printf("Entrer la note de l'étudiant:\n");
  scanf("%f",&note);
} while ( (note < 0.0) || (note > 20.0) );
...
```

Boucle while

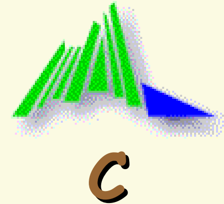
- while



- do...while



Boucle `for`



- Instruction `for`

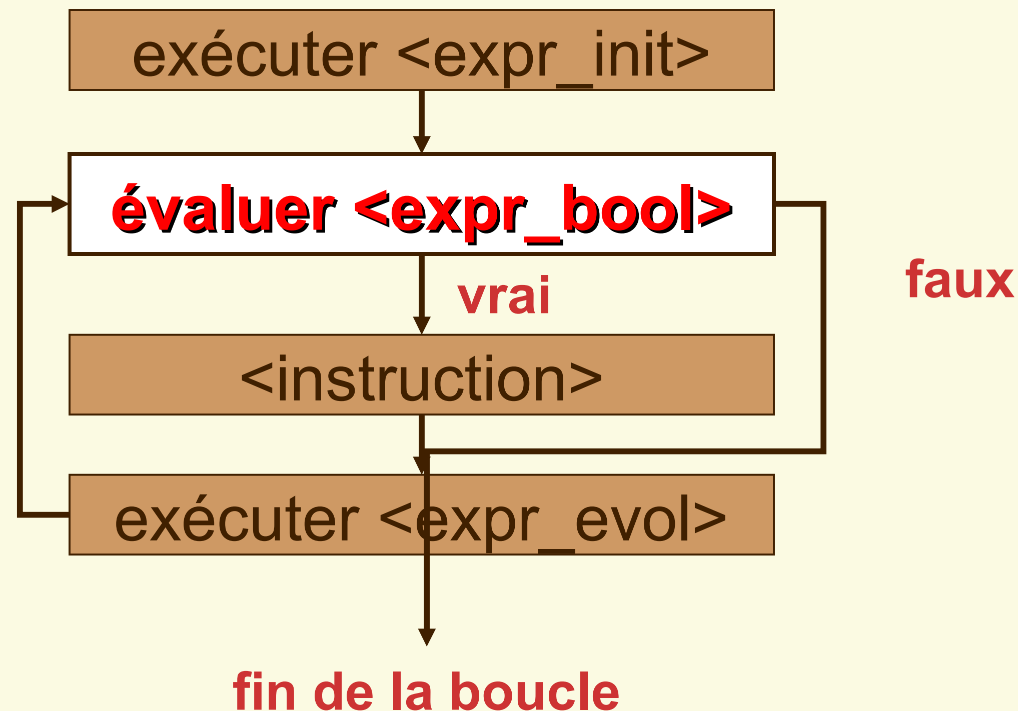
```
for (<expr_init>;<expr_bool>;<expr_evol>)
    <instruction>
```

  - `<expr_init>` : **initialisation de la boucle**
    - exécutée une seule fois avant d'entrer dans la boucle
  - `<expr_bool>` : **condition booléenne**
    - on continue tant que cette condition est évaluée à vrai
  - `<expr_evol>` : **modification des itérateurs**
    - une fois que le corps de la boucle a été exécuté, on évalue cette expression pour mettre à jour les itérateurs de la boucle avant de réévaluer la condition

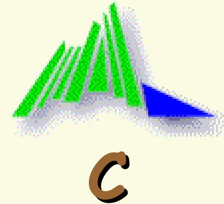
Boucle for

- Instruction for

```
for (<expr_init>;<expr_bool>;<expr_evol>)
    <instruction>
```



Boucle for




```c
/* Moyenne des notes */

#include <stdio.h>

main()
{
  int i,n;
  float note,somme=0.0;
  printf("Entrer le nombre de notes:\n");
  scanf("%d",&n);
  for (i=1; i<=n; i++)
    {
      printf("Note %d:\n",i);
      scanf("%f",&note);
      somme = somme+note;
    }
  printf("La moyenne des %d note(s) est %f\n",n,somme/n);
}
```

Boucle for

- Exemple

```
/* somme des n premiers entiers */
somme = 0;
for (i=1; i<=n; i++)
  somme = somme+i;
```

- Formes particulières
  - regroupement

```
/* dans la partie initialisation */
for (i=1,somme=0; i<=n; i++) {...}

/* dans les parties initialisation et évolution */
for (i=1,somme=0; i<=n; somme+=i,i++) {...}
```

Boucle for

- Formes particulières
  - absence

```
/* partie initialisation absente */
/*   => on suppose i initialisé avant le for */
for (; i<=n; i++) {...}

/* partie évolution absente */
/*  (ici, condition et évolution sont regroupées) */
for (i=1,somme=0; i++<=n;) {...}

/* corps vide */
for (i=1,somme=0; i<=n; somme+=i++);

/* boucle infinie */
for (;;);
```

Instructions prohibées

- Instruction `break`
  - Permet de sortir immédiatement de la boucle la plus interne

```
/* exemple de break dans une boucle for */
...
printf("Vous avez 10 essais pour trouver le nombre caché\n");
for (i=1; i<=10; i++)
  { scanf("%d",&nb);
    if (nb == 963) break;
  }
if (i>10)
  printf("PERDU...\n");
else
  printf("GAGNE en %d essais\n",i);
...
```

Instructions **prohibées**

- Même programme sans `break`

```
/* comment se passer d'un break */
...
printf("Vous avez 10 essais pour trouver le nombre caché\n");
i=1;
do
{
  scanf("%d",&nb);
  i++;
} while ( (i<=10) && (nb!=963) );

if (nb!=963)
  printf("PERDU...\n");
else
  printf("GAGNE en %d essais\n",i);
...
```

Instructions **prohibées**

- Instruction `continue`
  - Interrompt l'exécution du corps de la boucle et passe directement à l'évaluation de la condition

```
/* on ne veut traiter que les valeurs positives */
...
for (somme=0.0,i=1; i<=n; i++)
  {
    scanf("%d",&nb);
    if (nb<=0) continue;
    somme += sqrt(nb);
  }
printf("Résultat = %f\n",somme);
...
```

Instructions **prohibées**

- Même programme sans `continue`

```
/* comment se passer d'un continue */
...
for (somme=0.0,i=1; i<=n; i++)
  {
    scanf("%d",&nb);
    if (nb>0)  /* condition opposée */
      {
        /* ce qu'on voulait éviter avec le continue */
        somme += sqrt(nb);
      }
  }
printf("Résultat = %f\n",somme);
...
```

Instructions **prohibées**

- Instruction `goto`
  - On n'en parlera même pas; vous risqueriez de casser la machine en l'utilisant ;-)

Plan

- Instructions élémentaires
  - types,variables, opérateurs, expressions
  - affichage/lecture de données (`printf/scanf`)
- Instructions de contrôle
  - exécution conditionnelle (`if,switch`)
  - boucles (`for,while,do…while`)
- Tableaux, chaînes de caractères
- Fonctions
- Pointeurs
- Structures de données
- Fichiers

Tableaux

- Rappel
  - Un tableau permet de mémoriser un ensemble de valeurs de même type avec une seule variable
- Vocabulaire
  - taille: nombre **maximum** d'éléments (de valeurs)
    - ➔à vous de gérer le nombre réel d'éléments (une variable entière par ex)
  - rang d'un élément: sa position dans le tableau (indice)

Tableaux

- Syntaxe: déclarateur `[]`

```
<id_type_elts>  <id_tab>[<cste_taille>];
```

  - Ceci déclare une variable tableau
    - dont le nom est `<id_tab>`,
    - pouvant contenir `<cste_taille>` éléments,
    - chaque élément étant du type `<id_type_elts>`
  - Allocation **statique** (octets contigus)

```
<cste_taille> * sizeof(<id_type_elts>)
```

Tableaux

- Exemples
  - Déclarations correctes

```
/* tableau de 5 entiers (int) */
int tab[5];

/* tableau de 12 entiers courts (short) */
#define NBMOIS 12
short tmois[NBMOIS];
```

  - A ne jamais faire !!!

```
int n;
float tnotes[n];  /* à cet instant n vaut 0 ! */
...
printf("Nombre de notes:\n");
scanf("%d",&n);
```

Tableaux

- Accès aux éléments
  - Syntaxe: opérateur [ ]

    <id_tableau>[<indice>]

  - Illustration:

les éléments sont numérotés à partir de 0

tab		
23	←	tab[0]
48	←	tab[1]
57	←	tab[2]
12	←	tab[3]
14	←	tab[4]
	←	tab[5]

débordement d'indice ➔ sortie du tableau

- Le i^{ème} élément est à l'indice i-1
- Chaque élément fonctionne comme une variable

Tableaux

```c
#include <stdio.h>

main()
{
  int   n;               /* nombre d'étudiants au contrôle */
  int   i;               /* compteur de notes (indice du tab.) */
  int   nbSup;           /* nbre de notes sup. à la moyenne */
  float somme;           /* somme de toutes les notes */
  float moy;             /* moyenne de toutes les notes */
  float tabNotes[72];    /* tableau contenant les notes */

  /* Lecture du nombre de notes à saisir */
  printf("Entrer le nombre de notes à saisir: ");
  scanf("%d",&n);

  /* à suivre... */
```

Tableaux

```
/* Lecture des notes. Au fur et à mesure que celles-ci
   sont enregistrées dans le tableau, on en profite
   également pour faire la somme de toutes ces notes */
for (somme=0.0,i=0; i<n; i++)
  {
   printf("Entrer le note N°%d: ",i);
   scanf("%f",&tabNotes[i]);
   somme += tabNotes[i];
  }

/* Calcul et édition de la moyenne de la promo pour
   ce contrôle */
moy = somme/n;
printf("La moyenne est de %f\n",moy);

/* à suivre... */
```

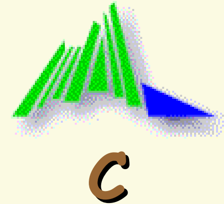
Tableaux

```
  /* Calcul et édition du nombre de notes supérieures
     à la moyenne du contrôle */
  for (nbSup=0,i=0; i<n; i++)
    if (tabNotes[i] >= moy)
      nbSup++;
  printf("Il y a %d notes sup. à cette moyenne\n",nbSup);

}   /* fin du programme (main) */
```

Tableaux

- Initialisation lors de la déclaration
  - Initialisation complète
    - `int tab[5] = {23, 48, 57, 12, 14};`
    - `short tmois[NBMOIS] = {21,28,31,30,31,30,31,31,30,31,30,31};`
  - Initialisation partielle
    - `int tab2[5] = {23, 48, 57};           /* 23, 48, 57, ??, ?? */`
    - `static int tab3[5] = {23, 48, 57};   /* 23, 48, 57, 0, 0 */`
  - Taille non précisée
    - `int tab4[] = {23, 48, 57, 12, 14};   /* taille à 5 */`
    - `int tab5[];                           /* INTERDIT */`

Tableaux



- Tableaux à plusieurs dimensions
  - Déclaration d'une matrice

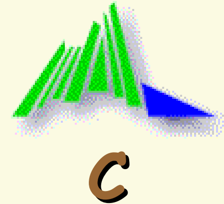
```
<id_type_elts> <id_tab>[<nb_lignes>][<nb_col>];
```

    - Exemple: `int mat[5][3];`
      - /* délaration d'un tableau de 15 entiers organisés en 5 lignes comportant chacune 3 éléments */

  - Accès à un élément

```
<id_tab>[<indice_ligne>][<indice_colonne>]
```

Tableaux



- Tableaux à plusieurs dimensions
  - Rangement en mémoire
    - tableau « à plat »
    - tous les éléments se suivent

mat[0][0]
mat[0][1]
mat[0][2]
mat[1][0]
mat[1][1]
mat[1][2]
...
mat[4][0]
mat[4][1]
mat[4][2]

  - Initialisation d'une matrice
    - ```
      int mat2[3][4]={ {12,23,-4,36},     /* 1ère ligne */
                       {33,-2,21,78},     /* 2ème ligne */
                       {-1,56,7,24} };    /* 3ème ligne */
      ```
    - ```
      int mat2[3][4]={12,23,-4,36,33,-2,21,78,-1,56,7,24};
      /* valeurs des 12 éléments consécutifs */
      ```

Chaînes de caractères

- Convention de représentation
  - « suite d'octets terminée par un \0 »
- Constantes chaînes

'I'	'U'	'T'	' '	'G'	'T'	'R'	'\0'

  - `sizeof("IUT GTR")` ➔ 8

Chaînes de caractères

- Variables chaînes
  - Principe
    - pas de type prédéfini

    ➔ on utilise un tableau de caractères pour y affecter une suite de caractères qui respecte la convention du \0 (NULL)
  - Déclaration (idem tableaux)
    - `char prenom[31];`
    - `char nom[];   /* interdit */`
  - Affectation globale interdite
    - `prenom = "Camille";   /* incorrect !!! */`

Chaînes de caractères

- Initialisation des variables chaînes lors de la déclaration
  - Vision **éléments**
    - `char prenom1[31]={'C','a','m','i','l','l','e','\0'};`

      ➔`prenom1` peut être manipulée en tant que chaîne; 31 octets sont réservés
    - `char prenom2[31] = {'C','a','m','i','l','l','e'};`

      ➔`prenom2` ne peut pas être manipulée en tant que chaîne (pas de \0 à la fin)
    - `char prenom3[] = {'C','a','m','i','l','l','e','\0'};`

      ➔`prenom3` peut être manipulée en tant que chaîne; 8 octets ont été automatiquement réservés

Chaînes de caractères

- Initialisation des variables chaînes lors de la déclaration
  - Vision **globale**
    - `char prenom4[31] = "Camille";`
      - ➔`prenom4` peut être manipulée en tant que chaîne; 31 octets sont réservés
      - ➔initialisation équivalente à celle de `prenom1`
    - `char prenom5[] = "Camille";`
      - ➔`prenom5` peut être manipulée en tant que chaîne; 8 octets ont été automatiquement réservés
      - ➔initialisation équivalente à celle de `prenom3`

Chaînes de caractères

- Utilisation des chaînes
  - Au niveau éléments

```
#include <stdio.h>

main()
{ char chaine[]="Lorraine: pays de la mirabelle";
  short i,cpt;     /* indice, compteur de 'a' */

  for (cpt=0,i=0; chaine[i]!='\0'; i++)
    if (chaine[i]=='a') cpt++;
  printf("Le caractère 'a' apparaît %d fois\n",cpt);
}
```

  - Au niveau global
    - ➔il faut utiliser les fonctions de manipulation de chaînes de caractères

Chaînes de caractères

- Fonctions standards
  - en début de prog.: `#include <string.h>`
  - respectent la convention du \0
  - Lecture
    - `gets(<chaîne>)`
    - `scanf("%s",<chaîne>)`
  - Affichage
    - `puts(<chaîne>)`
    - `printf("%s",<chaîne>)`
  - Longueur (ne compte pas le \0)
    - `strlen(<chaîne>)` ➔ `entier`

Chaînes de caractères

- Fonctions standard
  - **Comparaison**
    - `strcmp`(*<chaîne1>*,*<chaîne2>*)

    ➔ résultat: -1 si **chaîne1** < **chaîne2**
    0 si **chaîne1** et chaîne2 sont identiques
    1 si chaîne1 > chaîne2
  - **Recopie** (affectation)
    - `strcpy`(*<chaîne_dest>*,*<chaîne_source>*)

    ➔ recopie **chaîne_source** dans **chaîne_dest**
  - **Concaténation**
    - `strcat`(*<chaîne_dest>*,*<chaîne_source>*)

    ➔ ajoute **chaîne_source** à la fin de **chaîne_dest**

Plan

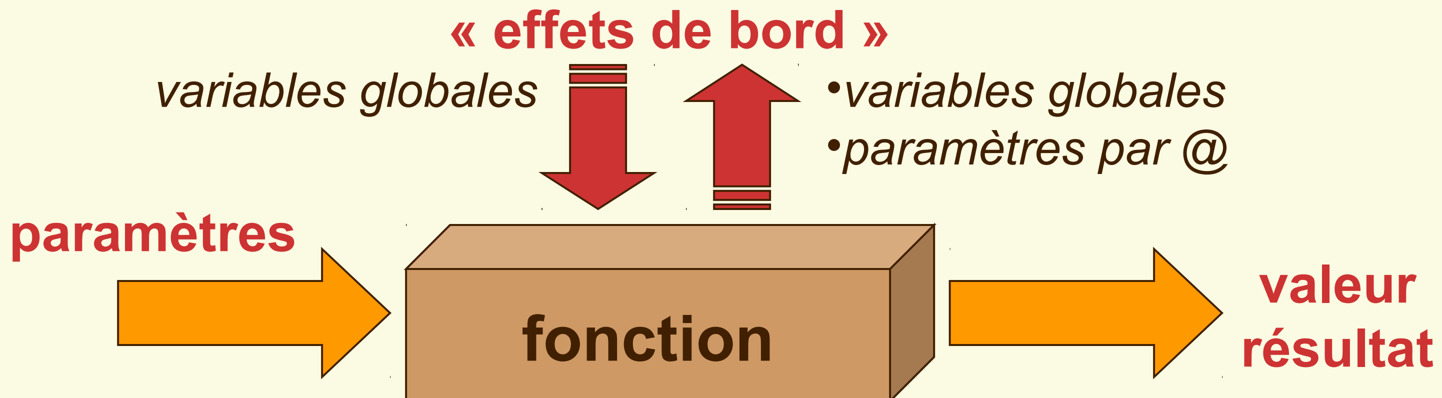
- Instructions élémentaires
  - types,variables, opérateurs, expressions
  - affichage/lecture de données (`printf/scanf`)
- Instructions de contrôle
  - exécution conditionnelle (`if,switch`)
  - boucles (`for,while,do…while`)
- Tableaux, chaînes de caractères
- Fonctions
- Pointeurs
- Structures de données
- Fichiers

Fonctions

- Idée: **programmation modulaire**
  - Indépendance fonctionnelle
    - ➔ unités de développement
    - ➔ unités de test
  - Répétitions d'actions identiques
    - ➔ unités réutilisables
    - ➔ unités paramétrables
  - Complexité
    - ➔ regroupement d'unités élémentaires

Fonctions

- Notion de fonction en C

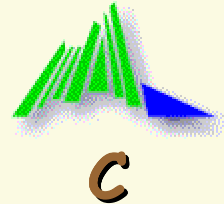


- Deux niveaux de manipulation
  - Utilisation
    - connaître le prototype, fichier en-tête, appel
  - Définition + utilisation
    - définir le prototype, la fonction, appel

Fonctions

```c
#include <stdio.h>  /* Fichiers d'en-têtes nécessaires pour */
#include <math.h>   /* les appels aux fonctions standards */

/* Déclaration des prototypes des fonctions utilisateur */
int discriminant(int a, int b, int c);
void afficher_racines(int a, int b, int c);
char continuer(void);

/* Définition de la fonction principale */
main()
{ char rep;
  int c1,c2,c3;
  do { puts("Entrez les 3 coefficients: ");
       scanf("%d %d %d",&c1,&c2,&c3);
       afficher_racines(c1,c2,c3); /* appel fonction util.*/
       rep = continuer();           /* appel fonction util.*/
     } while (rep == 'o');
}
```

Fonctions




```
/* Définition de la fonction qui retourne un caractère    */
/* représentant la réponse à un message en vue de répéter */
/* un traitement                                          */

char continuer(void)
{
  char c;
  puts("Voulez-vous continuer (o/n) ?");
  do { scanf("%c",&c);
       fflush(stdin);
     } while ( (c!='o') && (c!='n') );
  return(c);
}
```

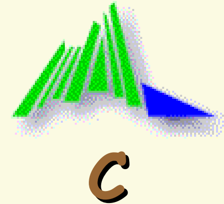
Fonctions

```
/* Définition de la fonction qui calcule et affiche les */
/* racines réelles d'un polynôme de degré inférieur ou  */
/* égal à 2 dont les coefficients sont formalisés par   */
/* les 3 paramètres a, b et c                           */

void afficher_racines(int a, int b, int c)
{ if (a==0)
    if (b==0)
      if (c==0)
        puts("Solution indéterminée\n");
      else
        puts("Solution impossible\n");
    else
      printf("Une seule racine: %f\n",(float)-b/c);
  else
    { /* à suivre... */
```

Fonctions

```
  /* suite */
  else
    { int delta = discriminant(a,b,c);    /* appel fct util */
      if (delta < 0)
        puts("Pas de racine\n");
      else if (delta == 0)
        printf("Une racine double: %f\n",(float)-b/(2*a));
      else
        { /* appels à une fonction standard de math.h */
          float x1 = (-b - sqrt((double)delta)) / (2*a);
          float x2 = (-b + sqrt((double)delta)) / (2*a);
          printf("2 racines: %f et %f\n",x1,x2);
        }
    }
} /* fin de la fonction afficher_racines */
```

Fonctions




```
/* Définition de la fonction qui calcule et retourne le  */
/* discriminant d'un polynôme de degré inférieur ou égal */
/* à 2 dont les coefficients sont formalisés par les 3   */
/* paramètres a, b et c                                  */

int discriminant(int a, int b, int c)
{
  return( (b*b) - (4*a*c) );
}

/* ------------------------------------------------- */
/* Fin de notre programme: toutes les fonctions dites */
/* « utilisateur » (i.e. non implémentées dans les    */
/* bibliothèques standards) ont été définies          */
/* ------------------------------------------------- */
```

Fonctions

- Déclaration et prototype de fonction
  - Prototype ⇔ fiche signalétique

    <id_type> <id_fonction>([<liste_décl_param>]

    - ***<id_type>***: type de la **valeur de retour** (résultat)
    - ***<id_fonction>***: **identificateur** (nom) de la fonction
    - ***<liste_décl_param>***: liste de déclaration des **paramètres** (`void` si aucun paramètre)
  - Positionnement des déclarations
    - hors de toute fonction ➔ portée globale
    - fonctions standards ➔ inclure le fichier .h adéquat

Fonctions

- Exemples de prototypes (signatures)
  - Classiques
    - `int factorielle(int n);`
    - `float moyenne(double tab[], long nbelt);`
    - `double sqrt(double x);  /* dans math.h */`
  - Sans valeur de retour
    - `void triCroissant(int tab[], long nbelt);`
  - Sans aucun paramètre
    - `float tirage(void);`
  - Corrects, mais non rigoureux
    - `float tirage2();`
    - `afficheMessage(char m[]);   /* int */`

Fonctions

- Définition de fonction
  - Implémentation ⇔ spécification du code
    - en-tête ➔ idem prototype
    - corps ➔ bloc d'instructions
  - Dans le corps de la fonction
    - informations visibles:
      - **paramètres**
      - variables déclarées dans le bloc (**locales**)
      - variables **globales** (déclarées hors de toute fonction)
    - une, plusieurs, voire aucune instruction `return`

Fonctions

- Utilisation d'une fonction
  - Paramètres **formels**
    - identificateurs précisés dans l'en-tête de la fonction
    - permettent de formaliser les arguments

```
void afficher_racines(int a,int b,int c)
/* a,b et c sont les paramètres formels de la
   fonction afficher_racines */
```

  - Paramètres **effectifs**
    - arguments précisés lors de l'appel
    - expressions réellement évaluées

```
afficher_racines(c1,c2,c3);
/* les variables c1,c2 et c3 constituent les
   paramètres effectifs (ou arguments) pour
   CET appel à la fonction */
```

Fonctions

- Échanges entre une fonction et son environnement

Technique	Sens de l'échange		Commentaires
	fct appelante → fct appelée	fct appelée → fct appelante	
valeur de retour (`return`)	**NON**	**OUI**	Valeur unique qui constitue le résultat de l'exécution de la fonction ***Moyen pratique mais limité***
arguments	**OUI** (directement)	**OUI** (indirectement)	***A privilégier*** (cf. passage de paramètres)
variable globale	**OUI**	**OUI**	***A éviter*** (cf. variables globales)

Fonctions

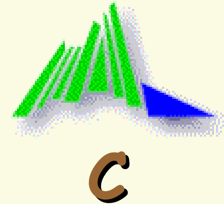
- Principe du passage par valeur

  ❶ A l'appel
  - **création** des paramètres formels
  - évaluation des arguments
  - recopie des valeurs des arguments dans les paramètres formels

  ❷ Lors de l'exécution
  - les calculs s'opèrent sur les paramètres formels

  ❸ A la fin de l'exécution
  - transmission de la valeur du résultat (via `return`)
  - **destruction** des paramètres formels

Fonctions



- Exemple

```
#include <stdio.h>

int plus_bidon(int a,int b);

main()
{ int x,y,som;
  printf("Entrer deux entiers: ");
  scanf("%d %d",&x,&y);                /* avant l'appel */
  som = plus_bidon(x,y);               /* appel */
  printf("Leur somme est: %d\n",som);  /* après l'appel */
}

int plus_bidon(int a,int b)
{ int c;
  c = a+b;                             /* pendant l'appel */
  return(c);
}
```

Fonctions

- Passage par valeur
  - **avant l'appel**

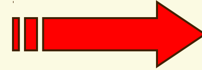
variables de la fonction ***main***

x	10
y	-5
som	???

variables de la fonction ***plus_bidon***

Fonctions

- Passage par valeur
  - **au moment de l'appel**

variables de la fonction ***main***			variables de la fonction ***plus_bidon***	
x	10		10	a
y	-5		-5	b
som	???			

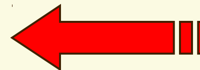
Fonctions

- Passage par valeur
  - **pendant de l'appel**

variables de la fonction ***main***			*variables de la fonction* ***plus_bidon***	
x	10		10	a
y	-5		-5	b
som	???		5	c

Fonctions

- Passage par valeur
  - fin de l'appel (return)

*variables de la fonction **main***			*variables de la fonction **plus_bidon***	
x	10		10	a
y	-5		-5	b
som	5		5	c

Fonctions

- Passage par valeur
  - après l'appel

variables de la fonction ***main***		variables de la fonction ***plus_bidon***	
x	10	10	a
y	-5	-5	b
som	5	5	c

Fonctions

- Remarques
  - Passage par valeur ⇔ copie des paramètres
    - ➔si une fonction tente de modifier un argument transmis par la fonction appelante, c'est la copie qui sera modifiée, pas l'argument lui-même
  - Éviter au maximum les variables globales
    - effets de bord préjudiciables
    - on verra plus loin comment une fonction peut retourner un résultat composé de plusieurs données (les structures)

Fonctions

- Variables **locales** vs variables **globales**

```
          GLOB1

main   { LOC1
                  }

fct1   { LOC2
                  }

fct2   { LOC3
         { LOC4
               }
                  }
```

LOC1

LOC2

LOC3

LOC4

GLOB1

Effet de bord: toutes les fonctions peuvent modifier les variables globales

Fonctions

$$C_p^n = \frac{n!}{p!\,(n-p)!}$$

```c
#include <stdio.h>

main()    /* sans fonction */
{
   int n,p,fn,fp,fnp;
   scanf("%d %d",&n,&p);

   /* calcul de n! */
   for (fn=1,i=1; i<=n; i++) fn*=i;

   /* calcul de p! */
   for (fp=1,i=1; i<=p; i++) fp*=i;

   /* calcul de (n-p)! */
   for (fnp=1,i=1; i<=(n-p); i++) fnp*=i;

   /* calcul et édition du résultat */
   printf("Nbre combinaisons = %d\n",fn/(fp*fnp));
}
```

Fonctions

$$C_p^n = \frac{n!}{p!(n-p)!}$$

```
#include <stdio.h>

int facto(int x);

main()   /* avec fonction factorielle */
{
  int n,p,cnp;
  scanf("%d %d",&n,&p);

  /* calcul et édition du résultat */
  cnp = facto(n) / (facto(p) * facto(n-p));
  printf("Nbre combinaisons = %d\n",cnp);
}

int facto(int x)
{
  int i,res;
  for (res=1,i=1; i<=x; i++) res*=i;
  return(res);
}
```

Fonctions

- Exercice N°1
  - Écrire une fonction C nommée **`estPremier`**
    - Paramètre: un entier strictement positif
    - Résultat: 1 si ce paramètre est un nombre premier
      0 si ce n'est pas un nombre premier
  - Écrire ensuite un programme de test complet

Fonctions

```
#include <stdio.h>

int estPremier(int n);

main()
{
  int nombre;

  do {
    printf("Entrez un entier strictement positif: ");
    scanf("%d",&nombre);
  } while (nombre <= 0);

  if (estPremier(nombre))
    printf("%d est un nombre premier\n",nombre);
  else
    printf("%d n'est pas un nombre premier\n",nombre);
}
```

Fonctions

```
/* On suppose que le nombre n passé en paramètre est > 0 */
int estPremier(int n)
{
  /* optimisation: s'arrêter à sqrt(n) */
  int resultat,i;

  resultat=1;
  i=2;
  while ( (i<n) && (resultat==1) )
    {
      if ( (n%i)==0 )
        resultat = 0;
      i++;
    }

  return(resultat);
}
```

Fonctions

- Remarque concernant les variables locales d'une fonction
  - Déclarée **normalement**, une variable locale est détruite quand la fonction se termine
    - ➔On ne peut pas garder sa valeur pour le prochain appel de cette fonction
    - ➔Elle aura été recréée (et donc réinitialisée) au début de ce nouvel appel

Fonctions

- Remarque concernant les variables locales d'une fonction
  - Si cette variable locale est déclarée **statique** (mot clé `static`), elle ne sera pas détruite à la fin de l'appel
    - ➔Elle est créée (et initialisée) lors du 1er appel à cette fonction
    - ➔Elle n'est pas détruite à la fin de l'appel ⇒ à l'appel suivant on retrouve la valeur qu'elle avait à la fin de l'appel précédent

Fonctions

- Remarque concernant les variables locales d'une fonction

```
void bidon(void)
{
  static int toto = 5;

  printf("toto = %d\n",toto);
  toto++;
}
```

  - 1er appel:
    - création de la variable toto avec la valeur 5
    - on affiche "toto = 5"
    - à la fin de cet appel, toto vaut 6
  - 2ème appel:
    - on récupère la variable toto qui valait 6
    - on affiche "toto = 6"
    - à la fin de ce second appel, toto vaut 7

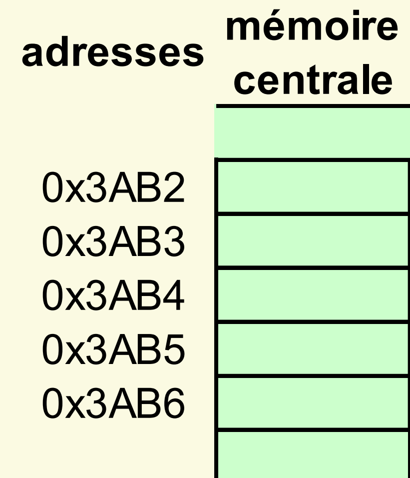
Plan

- Instructions élémentaires
  - types,variables, opérateurs, expressions
  - affichage/lecture de données (`printf/scanf`)
- Instructions de contrôle
  - exécution conditionnelle (`if,switch`)
  - boucles (`for,while,do…while`)
- Tableaux, chaînes de caractères
- Fonctions
- Pointeurs
- Structures de données
- Fichiers

Adresses & Pointeurs

- Adressage ?
  - mémoire centrale ⇔ suite d'octets ⇔ «tableau»
  - adresse ⇔ localisation ⇔ «indice»
  - valeur (d'une adresse) = fonction de la taille de l'espace d'adressage
    - Ex: espace de 64ko

    ⇒ adresses codées sur 4 octets

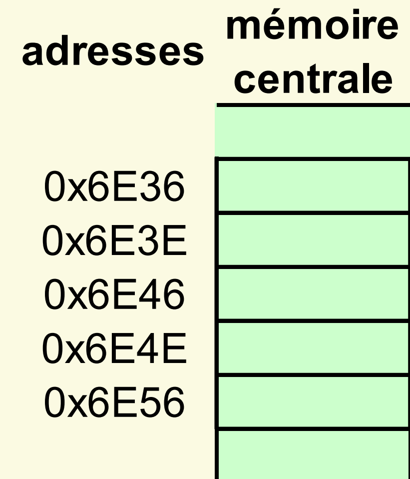


Adresses & Pointeurs

- Opérateur &
  - Permet de récupérer l'adresse d'une lvalue

```
...
double tab[5];
int i;
...
puts("Adresses des 5 éléments de tab:");
for (i=0; i<5; i++)
   printf("\tAdr elt %d: %X\n",i,&tab[i]);
...
```

```
Adresses des 5 éléments de tab
    Adr elt 0: 6E36
    Adr elt 1: 6E3E
    Adr elt 2: 6E46
    ...
```



type double
sur 8 octets

Adresses & Pointeurs

- Notion de pointeur
  - pointeur = objet dont la valeur est une adresse
    - variables et constantes pointeurs, « pointeur constant », constante **NULL**
  - type pointeur ? ⇒ plusieurs types dérivés
    - « pointeur sur <*un_type*> »
    - <*un_type*> est dit « type pointé »

Cette terminologie sous-entend qu'un pointeur contient une adresse à laquelle est implanté un objet d'un type bien particulier (celui désigné par <*un_type*>).

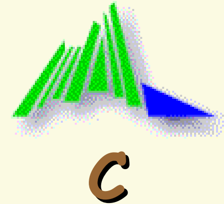
En C, un pointeur fait référence **A LA FOIS à une adresse en mémoire ET à un type (pointé)**.

Adresses & Pointeurs

- Variables pointeurs et déclarateur *
  - **Déclaration**: *<id_type> *<id_var_ptr>;*
  - Exemples
    - `float *p; /* p = variable pointeur sur float */`
    - `char *t;  /* t = pointeur sur caractère */`
    - `int *ptr; /* ptr = pointeur sur entier */`

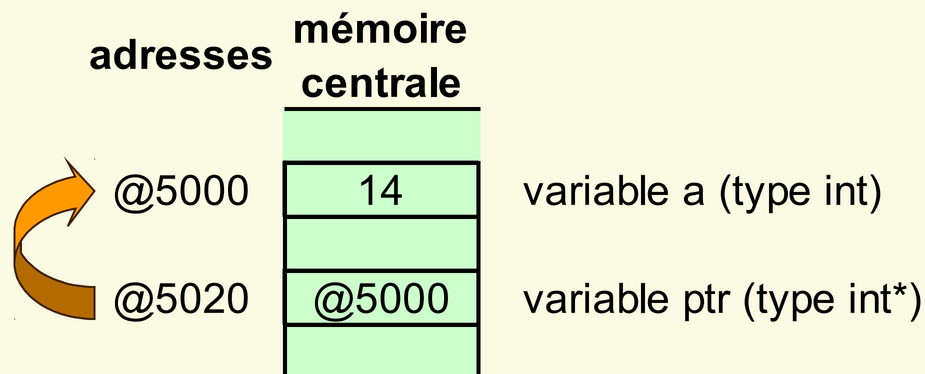
  Remarque: ces 3 variables ont la même taille en mémoire centrale (ce sont 3 adresses) !

Adresses & Pointeurs



- Variables pointeurs et déclarateur *
  - **Initialisation** lors de la déclaration

```
int a=14;    /* a = var entière initialisée à 14 */
...
int *ptr=&a; /* ptr est un pointeur sur entier
                initialisé par un pointeur constant
                qui est l'adresse de la variable a */
```



Adresses & Pointeurs

- Opérations sur les adresses
  - **Affectation** (pointeurs de même type)

```
float rayon, *ptr;
...
ptr = &rayon;
```

    - Remarque: seule une lvalue peut être affectée

```
&rayon = NULL;    /* INTERDIT !!! */
```

  - **Comparaison** (pointeurs de même type)

```
int *ptr1, *ptr2, *ptr;
...
if (ptr1>ptr2) ...
...
while (ptr!=NULL) ... /* idem while (!ptr) */
```

Adresses & Pointeurs

- Opérations sur les adresses
  - **Ajout/retrait d'un entier** à un pointeur

    `P + val` ⇔ `P + (val * sizeof(<type_pointé_par_P>))`

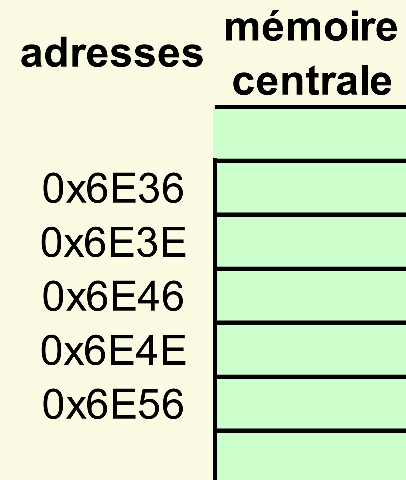
    ➔adresse du val^ième^ élément après celui pointé par P

    ➔notion de déplacement
  - **Opérateurs**
    - +, - : lvalue et constante
    - +=, -=, ++, -- : lvalue uniquement

Adresses & Pointeurs

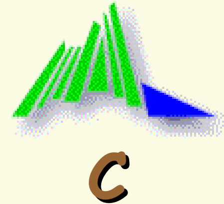
- Opérations sur les adresses
  - Ajout/retrait d'un entier

```
double tab[5];
double *ptr1,*ptr2;
...
ptr1 = &tab[0];    /* ptr1 vaut 6E36 */
ptr2 = &tab[0]+1; /* ptr2 vaut 6E3E,
                     i.e. &tab[1] */
ptr1++;            /* ptr1 vaut 6E3E */
ptr2+=2;           /* ptr2 vaut 6E4E,
                     i.e. &tab[3] */
```



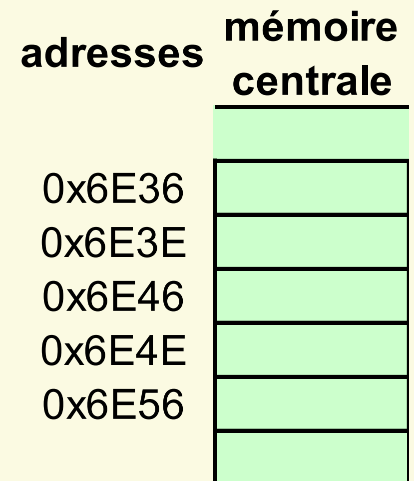
type double sur 8 octets

Adresses & Pointeurs



- Opérations sur les adresses
  - **Soustraction** de pointeurs de même type

```
double tab[5];
double *ptr1,*ptr2;
unsigned long diff;
...
ptr1 = &tab[4];   /* ptr1 vaut 6E56 */
ptr2 = &tab[1];   /* ptr2 vaut 6E3E */
diff = ptr1-ptr2; /* diff vaut 3 */
```



type double
sur 8 octets

Adresses & Pointeurs

- Opérations sur les adresses
  - Conversion d'un pointeur

```
void *ptr;
int tab[10];
...
ptr = (int*)&tab[0];
```

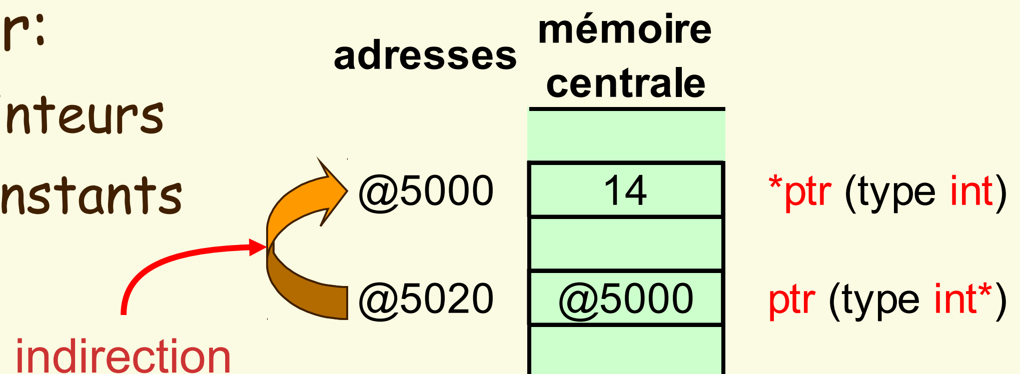
➔ Pour information uniquement

Adresses & Pointeurs

- Opérateur d'indirection *
  - Permet d'accéder à l'élément pointé

    `*<pointeur>`

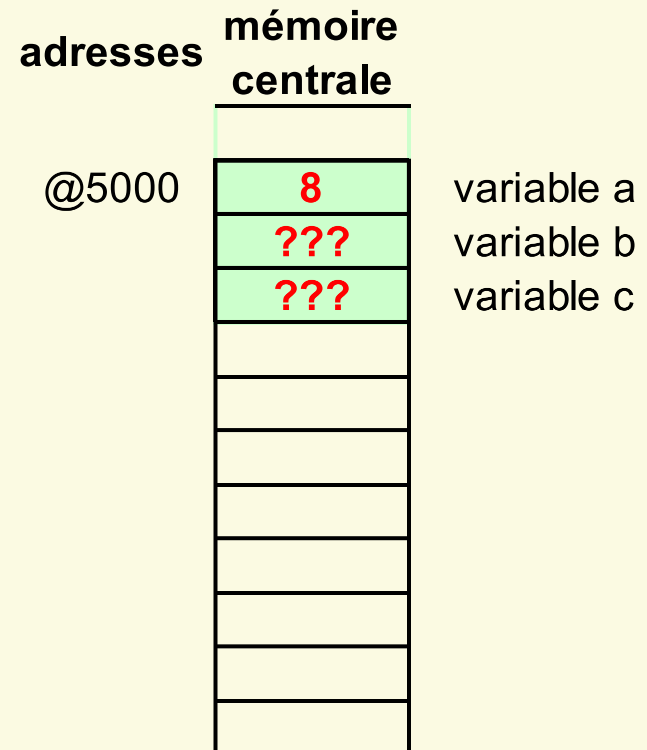
  - Type du résultat: type pointé
  - Applicable sur:
    - variables pointeurs
    - pointeurs constants



Adresses & Pointeurs

- Opérateur d'indirection *

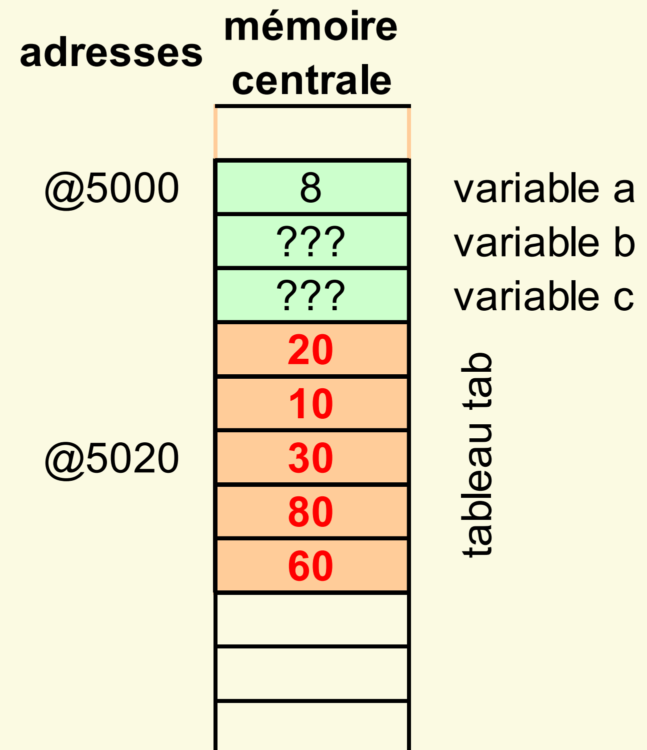
```
int a=8, b, c;
```



Adresses & Pointeurs

- Opérateur d'indirection *

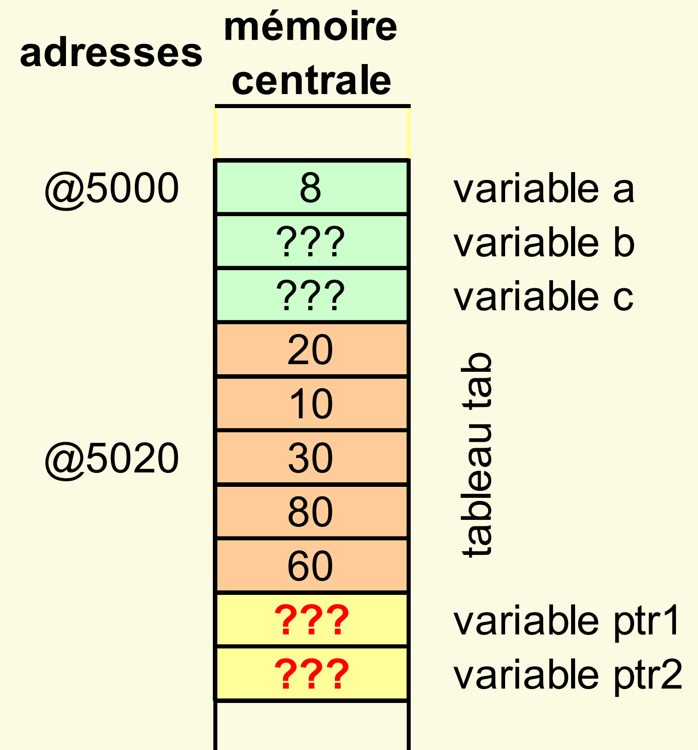
```
int a=8, b, c;
int tab[5]={20,10,30,80,60};
```



Adresses & Pointeurs

- Opérateur d'indirection *

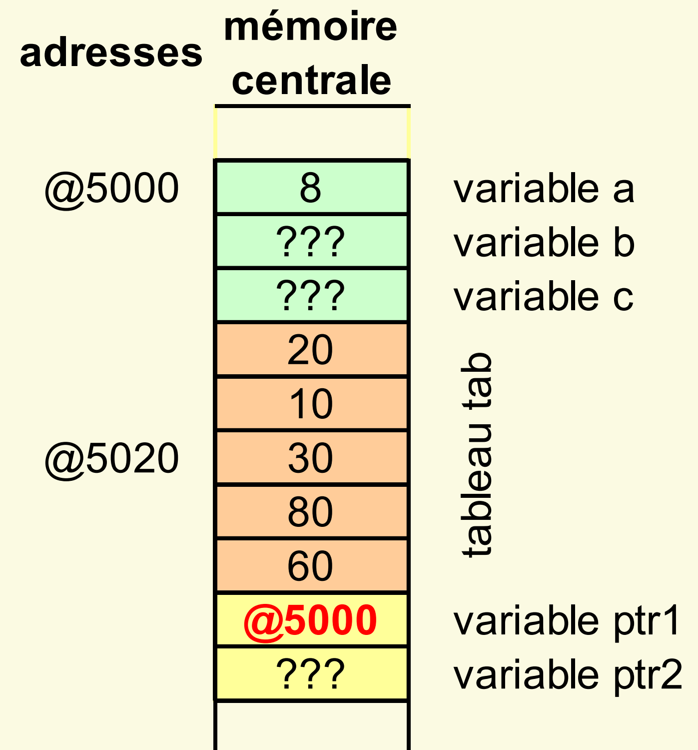
```
int a=8, b, c;
int tab[5]={20,10,30,80,60};
int *ptr1, *ptr2;
```



Adresses & Pointeurs

- Opérateur d'indirection *

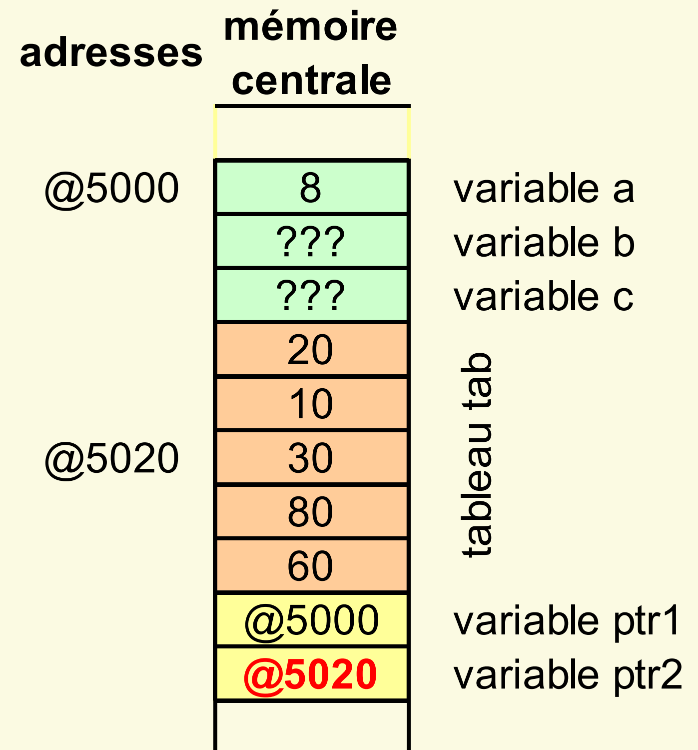
```
int a=8, b, c;
int tab[5]={20,10,30,80,60};
int *ptr1, *ptr2;
...
ptr1 = &a;
```



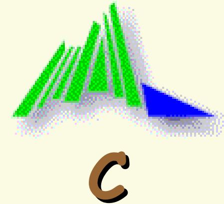
Adresses & Pointeurs

- Opérateur d'indirection *

```
int a=8, b, c;
int tab[5]={20,10,30,80,60};
int *ptr1, *ptr2;
...
ptr1 = &a;
ptr2 = &tab[2];
```

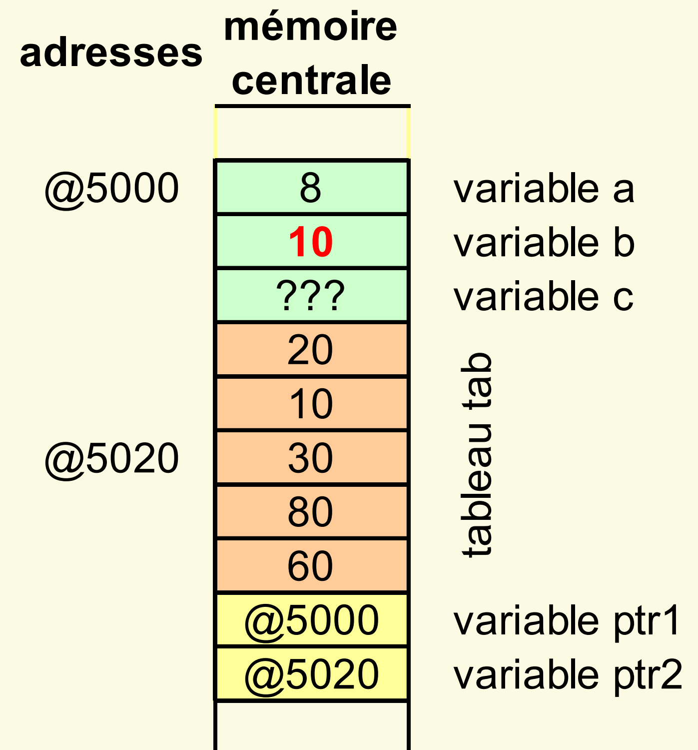


Adresses & Pointeurs

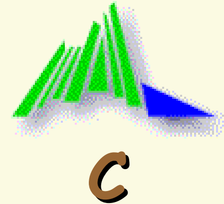


- Opérateur d'indirection *

```
int a=8, b, c;
int tab[5]={20,10,30,80,60};
int *ptr1, *ptr2;
...
ptr1 = &a;
ptr2 = &tab[2];
...
b = *ptr1 + 2;
```

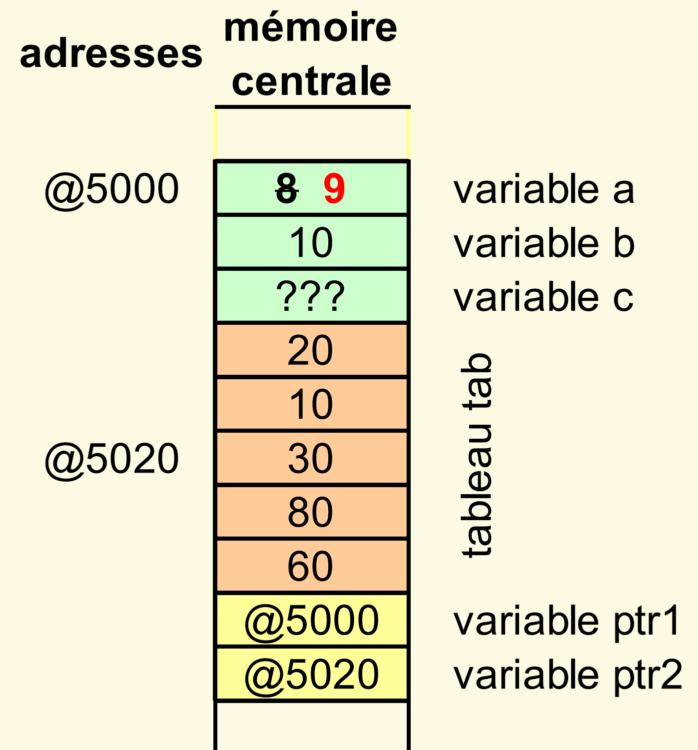


Adresses & Pointeurs

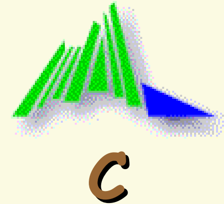


- Opérateur d'indirection *

```
int a=8, b, c;
int tab[5]={20,10,30,80,60};
int *ptr1, *ptr2;
...
ptr1 = &a;
ptr2 = &tab[2];
...
b = *ptr1 + 2;
(*ptr1)++;
```

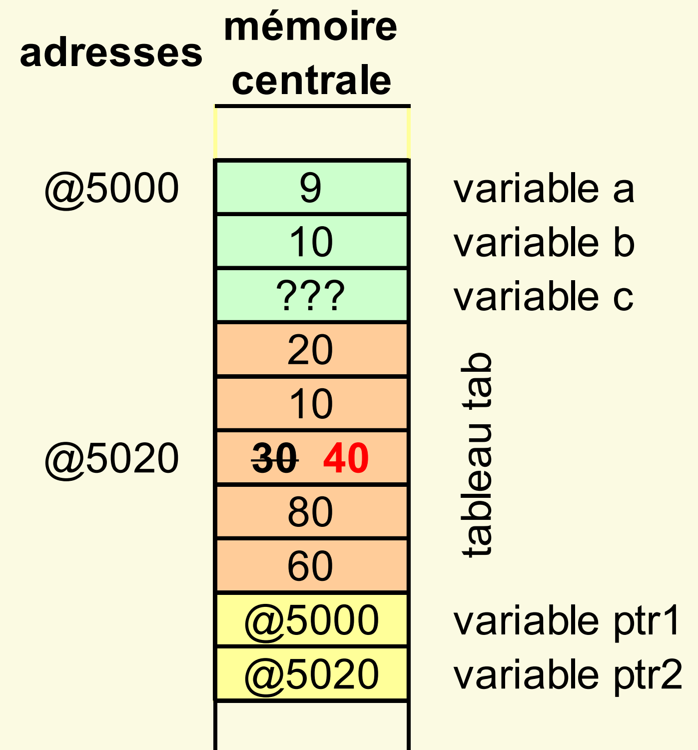


Adresses & Pointeurs



- Opérateur d'indirection *

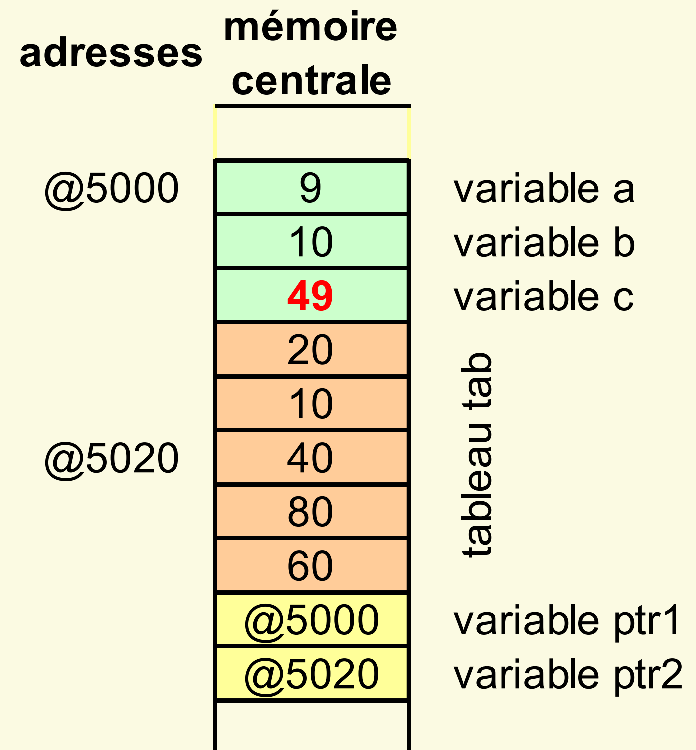
```
int a=8, b, c;
int tab[5]={20,10,30,80,60};
int *ptr1, *ptr2;
...
ptr1 = &a;
ptr2 = &tab[2];
...
b = *ptr1 + 2;
(*ptr1)++;
*ptr2 += b;
```



Adresses & Pointeurs

- Opérateur d'indirection *

```
int a=8, b, c;
int tab[5]={20,10,30,80,60};
int *ptr1, *ptr2;
...
ptr1 = &a;
ptr2 = &tab[2];
...
b = *ptr1 + 2;
(*ptr1)++;
*ptr2 += b;
c = *ptr1 + *ptr2;
...
```



Adresses & Pointeurs

- Récapitulatif & et *
  - Programme

```
#include <stdio.h>
main()
{ float y=5.38;
  float *ptr=&y;
  printf("y vaut %f\n&y vaut %x\n",y,&y);
  printf("ptr vaut %x\n&ptr vaut %x\n*ptr vaut %f\n",ptr,&ptr,*ptr);
}
```

  - Résultats

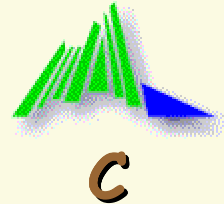
```
y vaut 5.38
&y vaut 5E24
ptr vaut 5E24
&ptr vaut 5E28
*ptr vaut 5.38
```

adresses	mémoire centrale	
0x5E24	5.38	y (type float)
0x5E28	0x5E24	ptr (type float*)

Pointeurs & Fonctions

- Trois liens entre pointeurs et fonctions:
  - Pointeurs sur fonction
    - non abordé (programmation très avancée)
    - idée: variables « contenant » des fonctions
  - Retour d'un pointeur
    - le résultat de la fonction est un pointeur
  - Modification d'arguments
    - ou comment peut-on modifier un argument malgré le passage par valeur

Pointeurs & Fonctions



- Fonction retournant un pointeur

```
#include <stdio.h>

char *member(char ch[100], char c)
{ int i;
  for (i=0; ch[i]!='\0' && ch[i]!=c; i++);
  return ( ch[i] ? &ch[i] : NULL );
}

main()
{ char chaine[100];
  char car, *pa;

  puts("Entrer une chaîne:"); gets(chaine);
  puts("Entrer un caractère:"); scanf("%c",&car); fflush(stdin);
  pa = member(chaine,car);
  if (pa) printf("%c appartient à la chaîne %s\n",car,chaine);
  else print("Aucune apparition de %c dans %s\n",car,chaine);
}
```

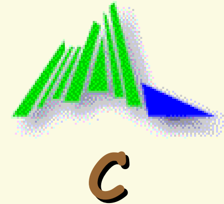
Pointeurs & Fonctions

- Modifier les arguments dans une fonction
  - Principe de base
    - les arguments sont **obligatoirement** passés par valeur
  - Problème
    - comment contourner ce mode de fonctionnement ?
  - Solution
    - ne pas passer la **valeur** de l'argument mais son **adresse**

Pointeurs & Fonctions

- Modifier les arguments dans une fonction
  - Règles à respecter
    - déclarer tout paramètre formel correspondant à un argument à modifier comme étant un pointeur
    - dans le corps de la fonction, se souvenir que ces paramètres sont des pointeurs
      - ➔ * pour obtenir leur valeur réelle
    - à l'appel, considérer que les paramètres effectifs sont des pointeurs
      - ➔ & pour transmettre leur adresse

Pointeurs & Fonctions

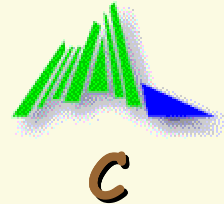


- Modifier les arguments dans une fonction

```
#include <stdio.h>

void permut(int a,int b);

main()
{ int x=10,y=-5;
  printf("Avant x=%d et y=%d\n",x,y);
  permut(x,y);
  printf("Après x=%d et y=%d\n",x,y);
}

void permut(int a,int b)
{ int aux;

  aux=a;
  a=b;
  b=aux;
}
```

a et b sont des copies des **valeurs** de x et de y ➔ pas d'effet de bord sur x et y

Pointeurs & Fonctions



- Modifier les arguments dans une fonction

```
#include <stdio.h>

void permut(int *a,int *b);

main()
{ int x=10,y=-5;
  printf("Avant x=%d et y=%d\n",x,y);
  permut(&x,&y);
  printf("Après x=%d et y=%d\n",x,y);
}

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

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

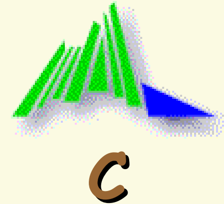
a et b contiennent les **adresses** de x et de y → x et y seront effectivement modifiés

Pointeurs & Fonctions

- Modifier les arguments dans une fonction
  - **avant l'appel**

	variables de la fonction **main**		variables de la fonction **permut**
@5600	10	x	
@5604	-5	y	

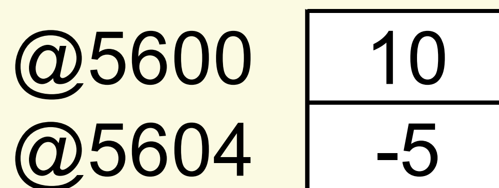
Pointeurs & Fonctions



- Modifier les arguments dans une fonction
  - **lors de l'appel**

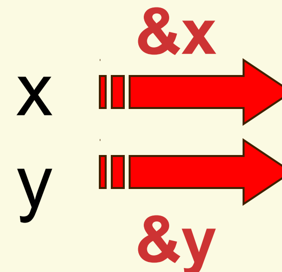
*variables de la fonction **main***

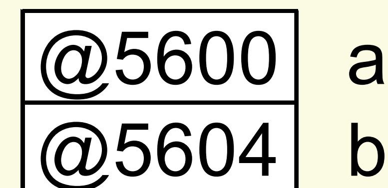
*variables de la fonction **permut***



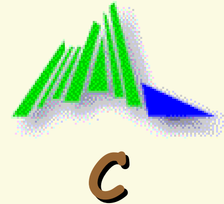
x

y





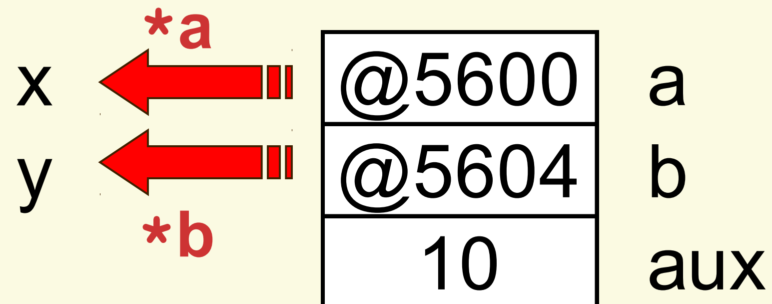
Pointeurs & Fonctions



- Modifier les arguments dans une fonction
  - **pendant l'appel**

variables de la fonction ***main***

variables de la fonction ***permut***



Pointeurs & Fonctions

- Modifier les arguments dans une fonction
  - **après l'appel**

*variables de la fonction **main***

@5600	-5	x
@5604	10	y

*variables de la fonction **permut*** (barrées d'une croix)

@5600	a
@5604	b
10	aux

Pointeurs & Fonctions

- Récapitulatif
  - On fait toujours du passage par valeur
    - au lieu de passer (par valeur) l'information elle-même on passe (par valeur) l'adresse à laquelle se trouve cette information
  - C'est exactement le fonctionnement de la fonction `scanf`
    - on doit bien passer l'adresse (`&`) de la variable dans laquelle on veut stocker l'information saisie
  - **Attention aux effets de bord !!!**

Pointeurs & Fonctions

- Exercice N°1
  - Écrire une fonction **`incrementer`** qui prend un entier en paramètre et augmente sa valeur de 1
  - Écrire ensuite un programme de test complet

Pointeurs & Fonctions

- Exercice N°2
  - Écrire une fonction `lireNote` qui lit deux valeurs (un réel et un entier) et les place dans ses deux paramètres
    - **1er paramètre**: note lue (comprise entre 0 et 20)
    - **2ème paramètre**: coefficient (compris entre 1 et 10)
    - **valeur de retour**: nombre d'erreurs de saisie (i.e. hors bornes) sur la note ou le coefficient
  - Écrire ensuite un programme de test complet

Pointeurs & Tableaux

- Nom d'un tableau ⇔ pointeur constant
  - Définition: <*un_type*> `Tab[100];`
    - ➔ `Tab` ⇔ `&Tab[0]`
  - Au niveau des adresses:
    - `Tab` ⇔ `Tab+0` ⇔ `&Tab[0]`
    - ➔ `Tab+i` ⇔ `&Tab[i]`
  - Au niveau des valeurs:
    - `*Tab` ⇔ `*(Tab+0)` ⇔ `*(&Tab[0])` ⇔ `Tab[0]`
    - ➔ `*(Tab+i)` ⇔ `Tab[i]`

Pointeurs & Tableaux

- Nom d'un tableau ⇔ pointeur constant
  - Définition: `<un_type> Tab[100];`
    - ➔ `*(Tab+i)` ⇔ `Tab[i]`
  - Remarques:
    - **`*Tab + i ≠ *(Tab+i)`**
      - cf. règles de priorité sur les opérateurs
    - **`Tab` n'est pas une lvalue**
      - donc `Tab=...` ou `Tab++` sont interdits !

Pointeurs & Tableaux

- Exemple
  - 1ère version: notation « tableau »
    - ➔ utilisation de l'opérateur d'indiciation `[]`

```
#include <stdio.h>

main()
{ int Tab[100];
  int i, som;
  ...
  /* calcul de la somme des éléments de Tab */
  for (i=0,som=0; i<100; i++)
    som += Tab[i];
  ...
}
```

Pointeurs & Tableaux

- Exemple
  - 2ème version: notation « pointeur »
    - ➔ utilisation de l'opérateur d'indirection *

```
#include <stdio.h>

main()
{ int Tab[100];
  int i, som;
  ...
  /* calcul de la somme des éléments de Tab */
  for (i=0,som=0; i<100; i++)
    som += *(Tab+i);
  ...
}
```

Pointeurs & Tableaux

- Exemple
  - 3ème version: notation « pointeur »
    - ➔ utilisation d'une variable pointeur

```
#include <stdio.h>

main()
{ int Tab[100];
  int *ptr, som;
  ...
  /* calcul de la somme des éléments de Tab */
  for (ptr=Tab,som=0; ptr<Tab+100; ptr++)
    som += *ptr;
  ...
}
```

Pointeurs & Tableaux

- Exemple
  - 4ème version: notation « pointeur »
    - ➔ **INCORRECTE** car Tab n'est pas une lvalue !

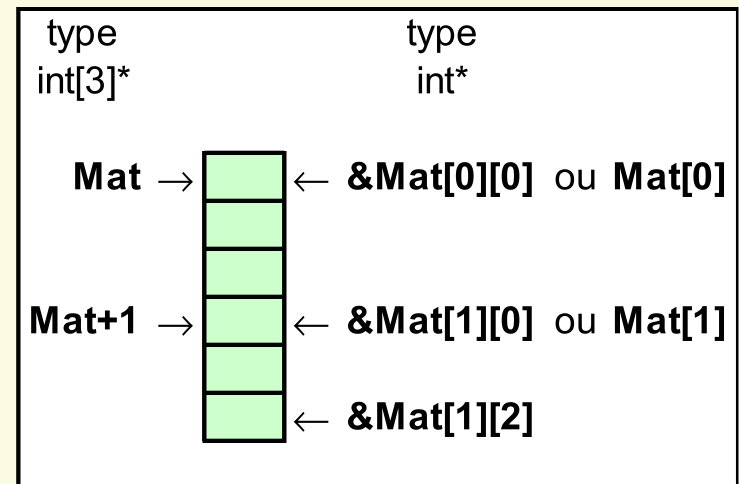
```
#include <stdio.h>

main()
{ int Tab[100];
  int som,*lim;
  ...
  /* calcul de la somme des éléments de Tab */
  for (lim=Tab+100,som=0; Tab<lim; Tab++)
    som += *Tab;
  ...
}
```

Pointeurs & Tableaux

- Tableaux à plusieurs dimensions
  - Définition: `int Mat[2][3];`
    - ➔ `Mat` ⇔ `&Mat[0][0]`
  - Au niveau des adresses:
    - `Mat+i` ⇔ `&Mat[i][0]`

    et

    - `Mat[0]` ⇔ `&Mat[0][0]`
    - `Mat[i]` ⇔ `&Mat[i][0]`



Pointeurs & Tableaux

- Tableaux en argument de fonction
  - C'est l'**adresse** qui est transmise **par valeur**
    - par indexation et/ou indirection on accède aux différents éléments du tableau

  → **modification possible des éléments !**

Pointeurs & Tableaux

- Tableaux en argument de fonction
  - Déclaration des fonctions
    - `void raz_tab1(int tab[100]);`
    - `void raz_tab2(int tab[]);`
      - la taille exacte du tableau n'est pas indispensable au compilateur
    - `void raz_tab3(int *tab);`
      - on formalise l'argument attendu comme étant l'adresse d'un entier

Pointeurs & Tableaux

- Tableaux en argument de fonction
  - Appel des fonctions

```
int tab1[100], tab2[100], tab3[100];
...
/* tab1 vu comme un int[100] */
raz_tab1(tab1);

/* tab2 vu comme un int[] */
raz_tab2(tab2);

/* tab3 vu comme un int* */
raz_tab3(tab3);
```

Pointeurs & Tableaux

- Tableaux en argument de fonction
  - Définition des fonctions
    - 1ère version: notation « tableau »

```
void raz_tab(...)
{ int i;
  for (i=0; i<100; i++) tab[i]=0;
}
```

    - 2ème version: notation « pointeur »

```
void raz_tab(...)
{ int i;
  for (i=0; i<100; i++) *(tab+i)=0;
}
```

Pointeurs & Tableaux

- Tableaux en argument de fonction
  - Définition des fonctions
    - 3ème version: utilisation d'une variable pointeur

```
void raz_tab(...)
{ int *ptr;
  for (ptr=tab; ptr<tab+100; ptr++) *ptr=0;
}
```

    - 4ème version: **CORRECTE** car tab est bien une lvalue

```
void raz_tab(...)
{ int *fin;
  for (fin=tab+100; tab<fin; tab++) *tab=0;
}
```

Pointeurs & Tableaux

- Tableaux en argument de fonction
  - Exemple 1

```
/* Fonction qui compte le nombre d'apparitions de l'entier val
   dans un tableau tab de enbelt valeurs entières */

int nbAppTab(int tab[], int nbelt, int val)
{
  int i,ct;

  for (ct=0,i=0; i<nbelt; i++)
    if ( *(tab+i) == val )
      ct++;

  return(ct);
}

/* Simple passage par valeur du nombre d'éléments */
```

Pointeurs & Tableaux

- Tableaux en argument de fonction
  - Exemple 2

```
/* Fonction qui insère un entier val à la fin d'un tableau tab
   de enbelt valeurs entières */

void insereTab(int tab[], int *nbelt, int val)
{
  *(tab + *nbelt++) = val;
}

/* Modification du nombre d'éléments par opération d'incrémentation:
   le paramètre formel nbelt est en fait l'adresse de l'argument
   correspondant */
```

Pointeurs & Tableaux

- Tableaux en argument de fonction
  - Cas des tableaux à plusieurs dimensions
    - **Autorisé**
      - `void f(int mat[][3]);`
      - `void f(int *mat, int nbelt);`
    - **Interdit**
      - `void f(int mat[][]);`

Pointeurs & Tableaux

- Chaînes de caractères
  - Constante chaîne ⇔ **pointeur constant**
  - Chaînes stockées dans des tableaux
    - ➔idem tableaux
  - Variables pointeurs sur caractères
    - ➔une adresse uniquement (pas de réservation pour les éléments)
  - En argument de fonction
    - ➔idem tableaux
      - `void fct_chaine(char tab[]);`
      - `void fct_chaine(char *tab);`

Pointeurs & Tableaux

- Chaînes de caractères
  - Au niveau des déclarations de variables

```
/* tableau de caractères non initialisé */
char mess1[31];

/* déclaration d'une variable pointeur seulement */
char *mess2;

/* déclaration d'un tableau avec initialisation par une constante
   chaîne; la taille du tableau peut être omise */
char mess3[] = "Message 3";

/* déclaration d'une variable pointeur initialisée à l'adresse
   de la constante chaîne */
char *mess4 = "Message 4";
```

Pointeurs & Tableaux

- Chaînes de caractères
  - Au niveau des instructions

```
/* ceci est INTERDIT */
mess1 = "Message 1";

/* recopie des caractères */
strcpy(mess1, "Message 1");

/* problème éventuel si vous n'avez pas pris la précaution de
   réserver manuellement la mémoire (cf. malloc vu plus loin)
   pour ce pointeur */
strcpy(mess2,"Toto");

/* ceci est autorisé */
mess2 = "Message 2";

/* INTERDIT: on ne peut pas modifier la constante "Message 2"... */
strcpy(mess2,"Toto");
```

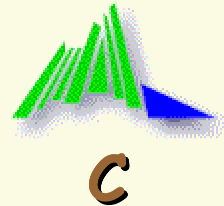
Entrées/Sorties standards

- Parenthèse: notions d'**entrée/sortie** (E/S)
  - **Sortie** ⇔ opération d'écriture
    - mémoire centrale ➔ unité périphérique
  - **Entrée** ⇔ opération de lecture
    - unité périphérique ➔ mémoire centrale

- Les E/S interconnectent 2 niveaux
  - Système d'exploitation: « bas niveau »
  - Langage de programmation: « haut niveau »

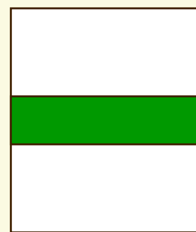
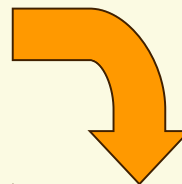
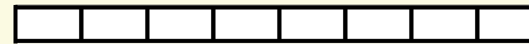
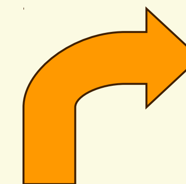
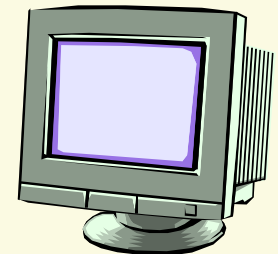
- Notion de **flot** (*stream*)
  - initialisé en lecture et/ou écriture entre le programme et une unité périphérique
  - **texte** (suite de lignes) ou **binaire** (codage inchangé)

Entrées/Sorties standards



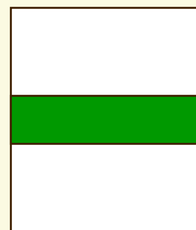
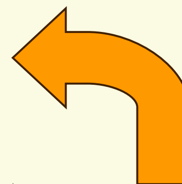
- Les entrées/sorties standards
  - Sortie standard

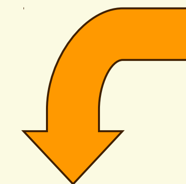
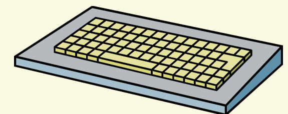
mémoire principale

  - Entrée standard

mémoire principale

Entrées/Sorties standards

- Les fonctions de sortie
  - Affichage d'un caractère

```
int putchar(int c)
```

```
int op = '*';
...
putchar(op);
```

  - Affichage d'une chaîne de caractères

```
int puts(const char *ch)
```

```
char prenom[31];
...
puts("Bonjour");
strcpy(prenom,"Roxanne");
puts(prenom);
```

Entrées/Sorties standards

- Les sorties formatées
  - Opération de formatage
    - conversion des données internes en chaînes
    - mise en forme sous le contrôle du format
  - Fonction d'affichage

    `int printf(char *format, expr1, expr2,..., exprn)`

  - Directive de mise en forme

    `%[drapeaux][gabarit][.précision][h|l|L]conversion`

Entrées/Sorties standards

- Les sorties formatées

caractère de conversion	type argument	type de l'affichage
c	*int, short, char*	caractère isolé affiché "en clair"
d	*int, short, char*	nombre décimal
o	*int, short, char*	nombre octal non signé (non précédé de O)
x, X	*int, short, char*	nombre hexadécimal non signé (non précédé de 0x) x affiche les lettres (abcdef) en minuscules X affiche les lettres (ABCDEF) en majuscules
u	*unsigned int* *unsigned short* *unsigned char*	nombre décimal non signé
s	*char **	chaîne de caractères respectant la convention du \0
f	*double, float*	notation décimale d'un réel
e, E	*double, float*	notation exponentielle d'un réel (minuscule, majuscule)
p	*void **	pointeur dont la représentation dépend de l'implémentation
%		affichage du caractère %

Entrées/Sorties standards

- Les fonctions d'entrée
  - Lecture d'un caractère

```
int getchar(void)
```

```
int op;
...
op=getchar();
```

  - Lecture d'une chaîne de caractères

```
char *gets(char *ch)
```

```
char prenom[31];
...
gets(prenom);
```

Entrées/Sorties standards

- Les entrées formatées
  - Opération de déformatage
    - extraire des données internes d'une chaîne
    - conversion en vue d'un stockage
  - Fonction de lecture

```
int scanf(char *format, adr1, adr2,..., adrn)
```

  - Directive de conversion

```
%[*][gabarit][h|l|L]conversion
```

Entrées/Sorties standards

- Les entrées formatées

caractère de conversion	type argument	type de l'affichage
c	*char **	un seul caractère si le gabarit est non précise ou vaut 1 n caractères si le gabarit vaut n (pas de \0 rajouté)
d	*int **	nombre décimal
o	*int **	nombre octal (précédé ou non de 0)
x	*int **	nombre hexadécimal (précédé ou non de 0x)
u	*unsigned int **	nombre décimal non signé
s	*char **	chaîne de caractères qui sera stockée en respectant la convention du \0 (prévoir la place mémoire)
f, e	*float **	notation décimale ou exponentielle d'un réel
p	*void **	pointeur dont la représentation dépend de l'implémentation
%		lecture du caractère %

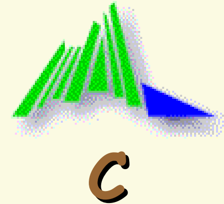
Plan

- Instructions élémentaires
  - types,variables, opérateurs, expressions
  - affichage/lecture de données (`printf/scanf`)
- Instructions de contrôle
  - exécution conditionnelle (`if,switch`)
  - boucles (`for,while,do…while`)
- Tableaux, chaînes de caractères
- Fonctions
- Pointeurs
- Structures de données
- Fichiers

Structures

- Principe et notions
  - **Structure** ⇔ ensemble de variables de types
    agrégation de variables en une seule entité identifiée de façon unique
    - **le modèle d'une structure** décrit le type des variables membres d'une structure ➔ c'est un **type**
    - **une instance d'une structure** a une existence réelle en mémoire déduite du modèle correspondant ➔ **variable** structurée (elle contient des valeurs)
  - **Champ** ⇔ identificateur d'une variable membre
    - chaque champ a son propre type (cf. modèle)
    - chaque champ a une valeur de ce type (cf. instance)

Structures



- Déclaration d'un modèle de structure

de la vision logique... ...à la déclaration du modèle

numInsc
numInsee
nom
prenom
adresse
moyenne

```
struct etudiant {
    long numInsc;
    char numInsee[14];
    char nom[30],
    char prenom[30];
    char adresse[100];
    float moyenne;
};
/* aucune réservation en mémoire */
```

  – selon la syntaxe C:

```
struct <id_type_struct> { <id_type_champ1> <id champ1>;
                          ...
                          <id_type_champn> <id champn>;
};
```

Structures

- Déclaration de variables structurées

```
struct <id_type_struct> <id_variable>;
```

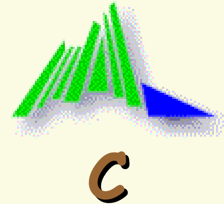
```
/* déclaration de trois variables de type étudiant */
struct etudiant unetu;
struct etudiant premier, dernier;

/* déclaration d'une structure de données et déclaration
   de trois variables */
struct date {
    short jour;
    short mois;
    short annee;
} today, tomorrow, yesterday;
```

Structures

- Initialisation lors de la déclaration

```
/* initialisation totale */
struct etudiant unetu = { 12563,
                          "1820564513025",
                          "MUNIER",
                          "Manuel",
                          "Mont-de-Marsan",
                          19.5 };

/* initialisation partielle */
struct etudiant unautre { 524,
                          "1810933201856",
                          "GALLON",
                          "Laurent",,};
```

Structures



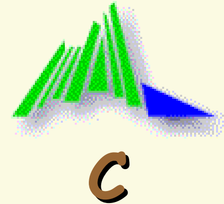
- Imbrication de structures

```
/* une personne en général */
struct personne {
    char numInsee[14];
    char nom[30];
    char prenom[30];
    char adresse[100];
};

/* un étudiant est une personne particulière */
struct etudiant {
    long umInsc;
    struct personne individu;
    float moyenne;
};
```

définition récurrente
INTERDITE

```
struct A { int i;
           struct A j; };
```

Structures



- Imbrication de structures

```
/* un employé est une personne particulière */
struct employe {
    long numSal;
    struct personne salarie;
    float salaire;
    struct date embauche;
};

/* déclaration d'une variable */
struct employe unemp = { 256,
                         { "2710531592048",
                           "TRONE",
                           "Paule",
                           "Toulouse" },
                         12800.0,
                         { 1, 1, 2002 }
                       };
```

Structures

- Utilisation « **globale** » d'une structure
  - Affectation globale: opérateur **=**
    - ceci n'est possible qu'entre deux variables déclarées à partir du même modèle

```
tomorrow = today;
```

    - il s'agit d'une recopie des valeurs champs par champs
  - Adresse d'une structure: opérateur **&**

```
/* déclaration d'une variable pointeur sur
   une structure du modèle date */
struct date *ptrdate;
...
ptrdate = &today;
```

Structures

- **Accès aux champs** d'une structure
  - Accès à la valeur d'un champ: opérateur .

    ```
    <id_variable> . <id_champ>
    ```

    - le type de cette expression est celui du champ
  - Exemples:

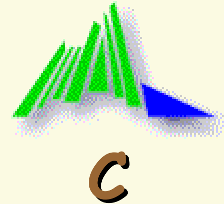
    ```
    unemp.salaire = 325000;
    tomorrow.jour = today.jour + 1;
    strcpy(unetu.prenom, "Bill");
    unemp.embauche = today;
    yesterday.mois--;
    scanf("%f", &unetu.moyenne);
    ```

Structures

- **Accès aux champs** d'une structure
  - Imbrication de structures:
    - si un champ est lui-même une structure, on peut à son tour lui appliquer l'opérateur . pour accéder à un de ses champs
  - Exemples:

```
gets(unemp.salarie.adresse);
unemp.embauche.mois = 2;
```

Structures



- Tableaux de structures
  - Déclaration

```
/* déclaration d'une var tableau de 100 employés */
struct employe tab_sal[100];
```

  - Accès aux éléments
    - au niveau global

```
tab_sal[i] = tab_sal[j];          /* indiciation */
*(tab_sal+i) = *(tab_sal+j);      /* indirection */
```

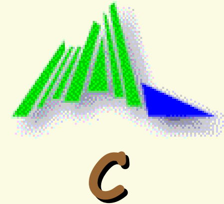
    - au niveau des champs

```
strcpy(tab_sal[i].salarie.nom, "DUPONT");
strcpy(*(tab_sal+i).salarie.prenom, "Pierre");
```

Structures

- Pointeurs et structures
  - Déclaration de pointeurs sur structure
    - `struct employe *ptr_emp;`
    - `struct employe *cour = tab_sal;`
  - Au niveau des adresses
    - `ptr_emp = &unemp;`
    - `cour++;`

Structures



- Pointeurs et structures
  - Déclaration de pointeurs sur structure
    - `struct employe *ptr_emp;`
    - `struct employe *cour = tab_sal;`
  - Au niveau des valeurs
    - au niveau global
      - `unemp = *cour;`
      - `*ptr_emp = tab_sal[8];`
    - au niveau des champs
      - `(*cour).salaire = 8000.0;`
      - `strcpy((*cour).salarie.nom, "DURAND");`

Structures

- Pointeurs et structures
  - Accès à la valeur d'un champ: opérateur ->

```
<adr_structure> -> <id_champ>
```

  - Exemples
    - le type de l'expression est celui du champ

```
(&unemp)->salaire = 32500.0;
ptrdate->mois--;
scanf("%f",&(cour->salaire));
(ptr_emp->embauche).mois = 2;
```

Structures

- Pointeurs et structures
  - Accès à la valeur d'un champ: opérateur ->

```
<adr_structure> -> <id_champ>
```

  - Équivalence de notation

```
struct date today;
struct date *ptr_today = &today;

    today.jour
⇔  ptr_today->jour
⇔  (&today)->jour
⇔  (*ptr_today).jour
```

Structures

- Structures auto-référentielles
  - **Définition:** Structure dont au moins un des champs est de type **pointeur sur cette structure**. Un tel champs peut contenir **l'adresse** d'une autre variable structurée de même type…

  → Structures de données dynamiques (liste, arbres, …)

```
struct noeudEtudiant {
    long numInsc;
    struct personne individu;
    float moyenne;
    char resultat;
    /* lien vers un autre étudiant */
    struct noeudEtudiant *ptr_etu;
};
```

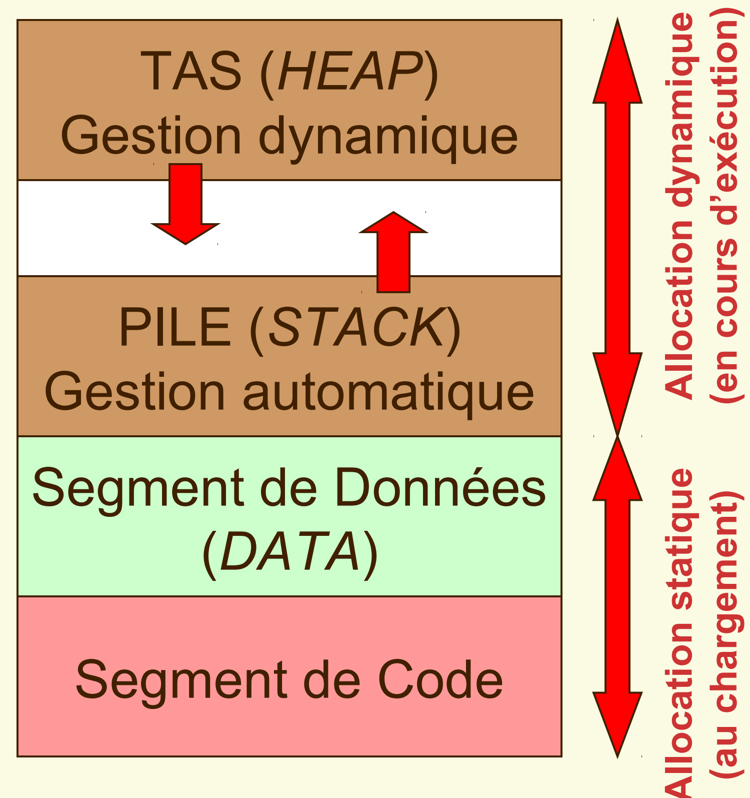
Structures

- Structures et fonctions
  - Passer une structure en argument:
    - passage par valeur
    - passage par adresse
      - modification possible de l'argument
      - plus performant (évite création + recopie)
  - Retourner une structure en résultat:
    - déclarer le type de retour de la fonction comme étant le type structuré

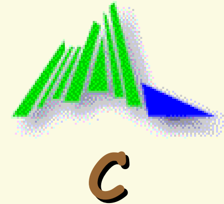
Gestion Dynamique

- Chargement et exécution d'un programme

1. Chargement en MC du fichier exécutable
   - Segment de code
   - Segment de données
2. Exécution du processus
   - Utilisation dynamique de la pile
3. Utilité du tas ?
   - Gestion dynamique (et programmée) de la mémoire



Gestion Dynamique



- En résumé:

Catégorie de données	Zone mémoire	Durée de vie
statiques	données (*data*)	processus
automatiques	pile (*stack*)	exécution du bloc
dynamiques	tas (*heap*)	prévue par le programmeur

- Outils de gestion dynamique:
  - Allocation `malloc`
  - Libération `free`

Gestion Dynamique

- Allocation dynamique dans le tas

```
void *malloc(size_t taille)
```

```
/* Allocation dynamique d'une zone de 100 caractères */

...
char *nom;
...
nom = malloc(100);
puts("Entrez votre nom:");
gets(nom);
...
```

Gestion Dynamique

- Allocation dynamique dans le tas

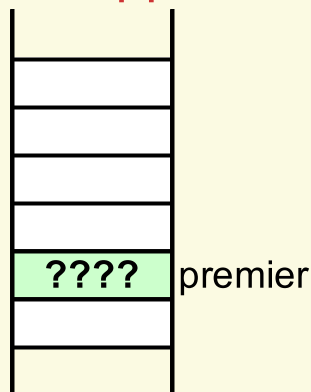
```
/* Allocation dynamique d'une zone de N réels */

...
int nb_temp;
float *temperatures, *ptr
...
puts("Entrez le nombre de températures:");
scanf("%d",&nb_temp);
temperatures = malloc(nb_temp*sizeof(float));
for (ptr=temperatures; ptr<temperatures+nb_temp; ptr++)
     {
       puts("Entrez une température:");
       scanf("%f",ptr);
     }
...
```

Gestion Dynamique

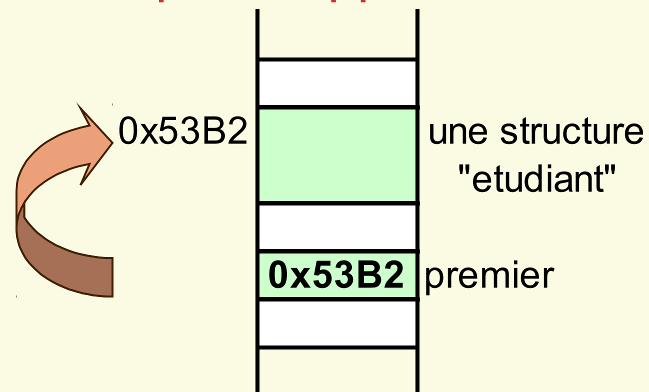
- Allocation dynamique: illustration

```
/* Allocation dynamique d'une variable structurée */
struct etudiant *premier;
...
premier = (struct etudiant)malloc(sizeof(struct etudiant));
/* Retypage de la valeur retournée (void*) par un cast */
```

Avant l'appel à malloc



Après l'appel à malloc



Gestion Dynamique

- Libération dynamique dans le tas

```
void free(void *adresse)
```

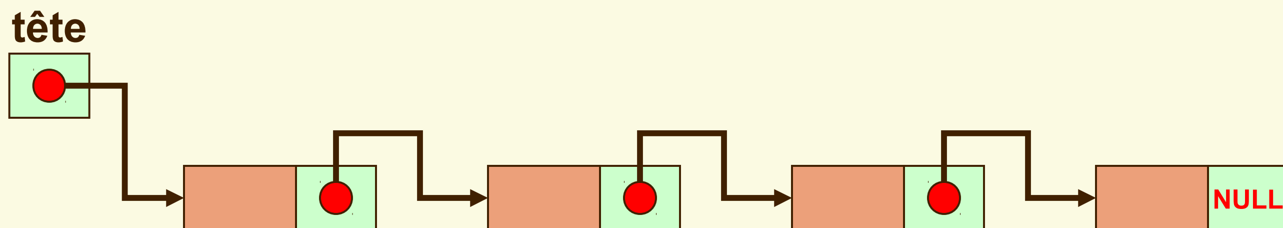
```
/* Libération des zones précédemment allouées */

...
free(nom);    /* libération des 100 caractères */
...
free(temperatures);    /* libération des N réels */
...
```

S.D. Dynamiques

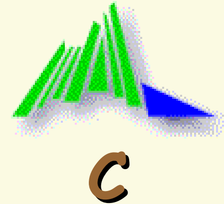
- Structures de données complexes
  - Relation d'organisation entre éléments de même nature
    - listes, files, piles, arbres, tas, graphes,... (tableaux ?)
- Structures de données dynamiques
  - **Structures auto-référentielles** (SDD)
  - Notion de **cellule** (ou boîte, nœud,...)
    - partie « **valeur** »
    - un (ou plusieurs) **liens** vers autre(s) cellule(s)
  - Struct. + pointeurs + gestion dyn.: **traitement**
  - **Modes de gestion**: politiques spécifiques

S.D. Dynamiques

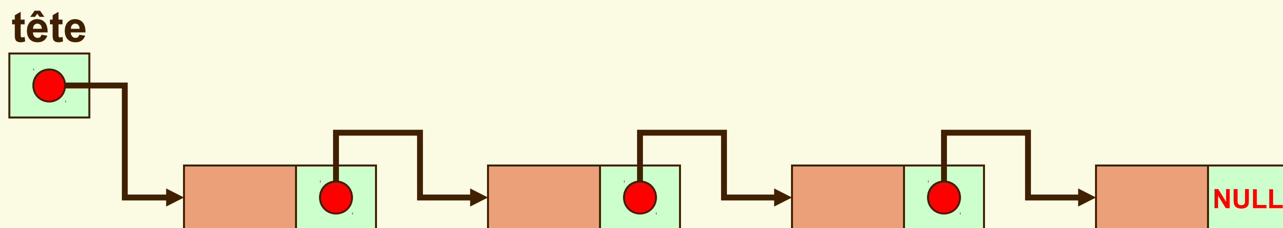
- Listes simplement chaînées
  - **Suite de cellules** liées entre elles par **un lien unique**
  - Une cellule = une **valeur** + un lien vers le **suivant**
  - Représentation logique



S.D. Dynamiques



- Listes simplement chaînées
  - Représentation logique



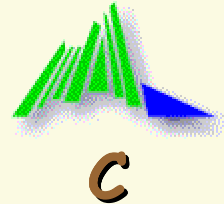
  - Description en C

```
/* modèle de la structure auto-référentielle */
struct listeSC { <type_elt> val;
                 struct listeSC *suivant; /* lien */
               };

/* variable repérant la liste */
struct listeSC *tete;
```

S.D. Dynamiques

- Listes simplement chaînées
  - Opérations sur les listes:
    - Fonctions à développer
    - Nouvelles classes d'algorithmes
  - Exemples:
    - initialiser
    - liste vide ?
    - insertion d'un élément
    - suppression d'un élément
    - édition des éléments de la liste
    - recherche d'un élément

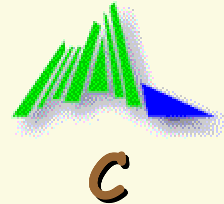
S.D. Dynamiques



- Listes simplement chaînées
  - Liste vide

```
/* modèle de la structure auto-référentielle */
struct listeSC { <type_elt> val;
                  struct listeSC *suivant; /* lien */
                };

/* construction d'une liste vide */
struct listeSC *listeConsVide()
{
  return NULL;
}

/* une liste est-elle vide ? */
int listeVide(struct listeSC *l)
{
  return (l == NULL);
}
```

S.D. Dynamiques



- Listes simplement chaînées
  - Insertion d'un élément en tête de liste

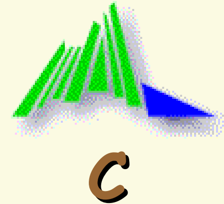
```
/* modèle de la structure auto-référentielle */
struct listeSC { <type_elt> val;
                 struct listeSC *suivant; /* lien */
               };

/* insertion d'un élément en tête de liste */
struct listeSC *listeCons(<type_elt> elt, struct listeSC *l)
{
  struct listeSC *nouveau;
  nouveau = (struct listeSC*)malloc(sizeof(struct listeSC));

  nouveau->val = elt;
  nouveau->suivant = l;

  return nouveau;
}
```

S.D. Dynamiques

- Listes simplement chaînées
  - Accès tête et queue d'une liste

```
/* modèle de la structure auto-référentielle */
struct listeSC { <type_elt> val;
                 struct listeSC *suivant; /* lien */
               };

/* valeur en tête de la liste */
<type_elt> listeTete(struct listeSC *l)
{
  return l->val;
}

/* queue de la liste, i.e. reste de la liste sans la tête */
struct listeSC *listeQueue(struct listeSC *l)
{
  return l->suivant;
}
```

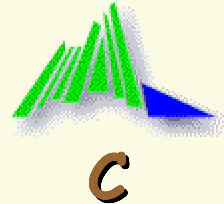
S.D. Dynamiques



- Listes simplement chaînées
  - Avec ces 5 fonctions d'accès aux listes on peut maintenant écrire des fonctions plus évoluées

```
void listeAfficher(struct listeSC *l)
{
  if (listeVide(l))
    printf("La liste est vide !!!\n");
  else
    {
      struct listeSC *temp = l; /* inutile car l passé par valeur */
      while (!listeVide(temp))
        {
          printf("...\n",listeTete(temp));      /* affichage tête */
          temp = listeQueue(temp);          /* on passe à la suite */
        }
    }
}
```

S.D. Dynamiques

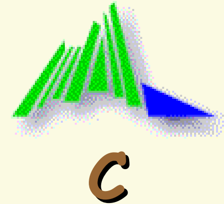


- Listes simplement chaînées
  - Bien évidemment, les structures de données récursives telles que les listes se prêtent à merveille aux algorithmes récursifs !

```
void listeAfficher(struct listeSC *l)
{
  if (!listeVide(l))
    printf("...\n",listeTete(l));    /* affichage de la tête  */
    listeAfficher(listeQueue(l));    /* affichage de la queue */
}
```

➔ **Attention à la condition d'arrêt de la récursivité !!!**

S.D. Dynamiques

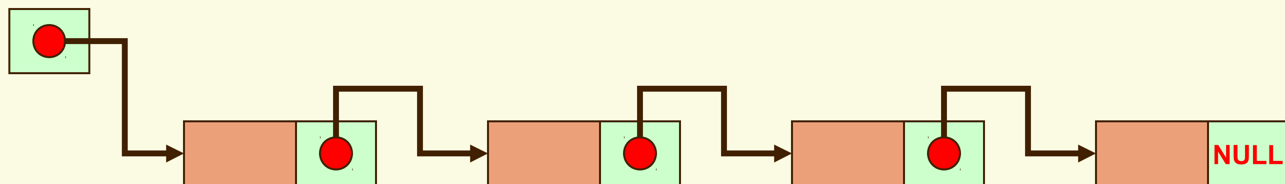


- Listes simplement chaînées
  - Utilisation de mot-clé **typedef**

```
/* modèle de la structure auto-référentielle */
struct listeSC { <type_elt> val;
                 struct listeSC *suivant; /* lien */
               };

/* définition d'un raccourci */
typedef struct listeSC* liste;

/* insertion d'un élément en tête de liste */
liste listeCons(<type_elt> elt, liste l)
{
  liste nouveau = (liste)malloc(sizeof(struct listeSC));
  nouveau->val = elt;
  nouveau->suivant = l;
  return nouveau;
}
```

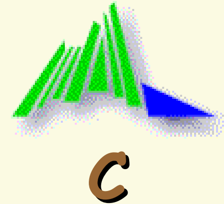
S.D. Dynamiques

- Mode de gestion **LIFO**: pile
  - Pile d'objets ➔ **Last In First Out**
  - Représentation logique

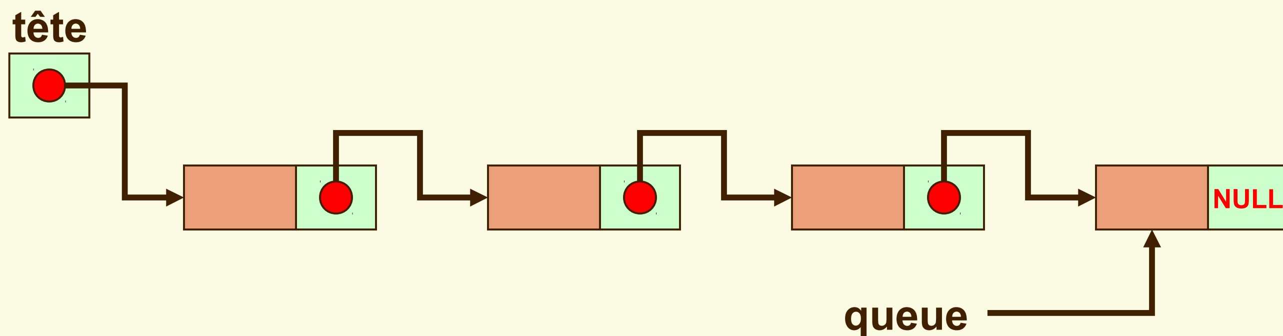


  - Opérations particulières
    - **empiler** (insérer au sommet)
    - **dépiler** (extraire au sommet)

S.D. Dynamiques



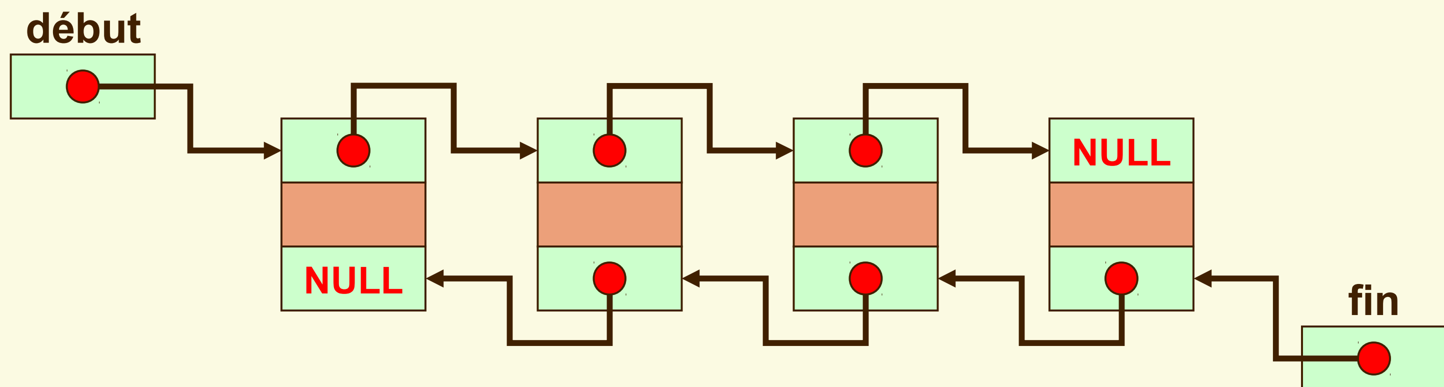
- Mode de gestion **FIFO**: file
  - File d'attente ➔ **First In First Out**
  - Représentation logique



  - Opérations particulières
    - **insérer** (toujours à la fin)
    - **extraire** (toujours au début)

S.D. Dynamiques

- Listes doublement chaînées
  - **Suite de cellules** liées entre elles par **deux liens**
  - Une cellule = **précédent** + **valeur** + **suivant**
  - Représentation logique

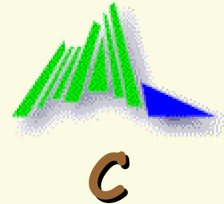


S.D. Dynamiques

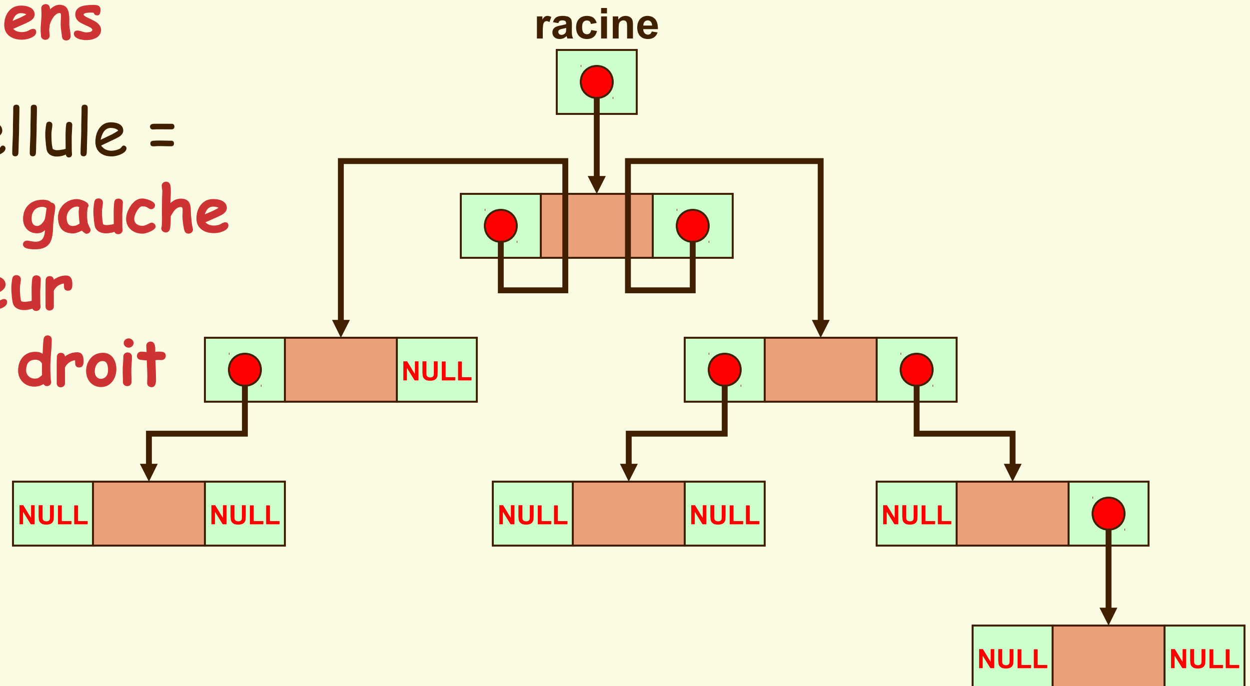
- Listes doublement chaînées
  - Description en C

```
/* modèle de la structure auto-référentielle */
struct listeDC {
    struct listDC *precedent; /* lien arrière */
    <type_elt> val;
    struct listeDC *suivant; /* lien avant */
};

/* variable repérant la liste */
struct listeDC *debut, *fin;
```

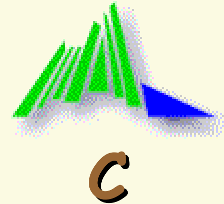
S.D. Dynamiques



- Arbres binaires
  - Hiérarchie de cellules liées entre elles par deux liens
  - Une cellule =
    fils gauche
    \+ valeur
    \+ fils droit



S.D. Dynamiques



- Arbres binaires
  - Description en C

```
/* modèle de la structure auto-référentielle */
struct arbre {
    <type_elt> val;
    struct arbre *gauche; /* lien fils gauche */
    struct arbre *droit;  /* lien fils droit */
};

/* variable repérant l'arbre */
struct arbre *racine;
```

S.D. Dynamiques

- Arbres binaires
  - Fonctions d'accès élémentaires

```
struct arbre *arbreConsVide()
```

```
int arbreVide(struct arbre *a)
```

```
struct arbre *arbreCons( <type_elt> racine,
                         struct arbre *fgauche,
                         struct arbre *fdroit)
```

```
<type_elt> arbreRacine(struct arbre *a)
```

```
struct arbre *arbreFilsGauche(struct arbre *a)
```

```
struct arbre *arbreFilsDroit(struct arbre *a)
```

Plan

- Instructions élémentaires
  - types,variables, opérateurs, expressions
  - affichage/lecture de données (`printf/scanf`)
- Instructions de contrôle
  - exécution conditionnelle (`if,switch`)
  - boucles (`for,while,do…while`)
- Tableaux, chaînes de caractères
- Fonctions
- Pointeurs
- Structures de données
- **Fichiers**

Fichiers

- Objectif: persistance de l'information
  - Nécessité de **conserver l'information** après traitement
    - ➔Ne pas saisir les données à chaque lancement du prog.
    - ➔Transférer des données d'un programme à un autre
  - **Mémoire secondaire** ≡ support de stockage non volatile
    - ➔Ne pas perdre de données quand on éteint la machine
    - ➔Mémoire de masse (disque dur, CD,…) ≠ RAM

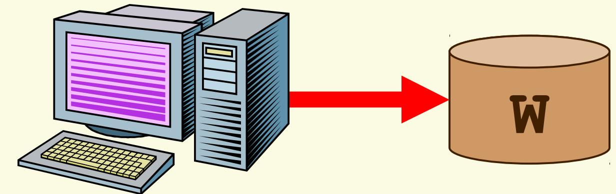
Fichiers

- Persistance de l'information
  - **Fichier** ≡ regroupement d'éléments d'information de même nature
  - **Répertoires** ➔ chemin d'accès ➔ localisation

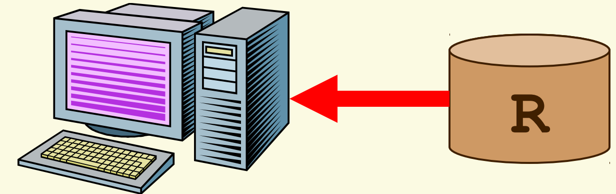
➔ **Liens entre programme et fichiers**

Fichiers

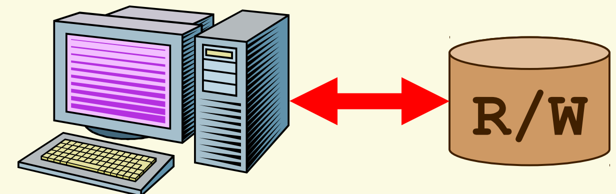
- 3 catégories d'opérations sur les fichiers:
  - Création
    - On crée un nouveau fichier pour y stocker des données



  - Consultation
    - On accède en lecture aux informations stockées



  - Modification
    - Adjonction, modification, suppression d'informations contenues dans un fichier



Fichiers

- Principales opérations sur les fichiers:
  - **Ouverture** (association vision logique/vision physique)
  - **Lecture** (mémoire secondaire ➔ mémoire principale)
  - **Écriture** (mémoire principale ➔ mémoire secondaire)
  - **Fermeture** (rupture du lien)

	création	consultation	mise à jour
ouverture	X	X	X
lecture		X	X
écriture	X		X
fermeture	X	X	X

Fichiers

- Un fichier est un **flot** ➔ 2 catégories
  - **Fichiers textes**
    - Suite de caractères découpée en lignes (`\n`)
    - Directement éditables
    - Programmes utilisateurs possibles
    - Lignes de longueur variable
  - **Fichiers binaires**
    - Codage interne des données
    - Non éditables
    - Programmes spécifiques
    - Suite d'octets ➔ le programmeur est responsable de la structuration logique ➔ enregistrements de longueur fixe

Fichiers

- Variable logique de fichier
  - **Variable pointeur** sur le type prédéfini `FILE`
    - **Descripteur** de fichier
    - Repère un flot associé au fichier
    - Utilisée comme **référence** au fichier dans toutes les opérations réalisées sur celui-ci
    - Valeur fixée au moment de l'ouverture
  - Exemples
    - `FILE *fp, *fpt;`
    - `FILE *fic1, *fic2, *fic3;`

Fichiers

- Ouverture d'un fichier
  - Prototype
    - `FILE *fopen(char *nom, char *mode)`
  - Mode
    - Mode d'ouverture : `w, r, a, w+, r+, a+`
      - `w : write`
      - `r : read`
      - `a : append`
    - Type de flot : t ou b
      - `t : text`
      - `b : binary`

Fichiers

- Modes d'ouverture d'un fichier

r Open text file for reading. The stream is positioned at the beginning of the file.

r+ Open for reading and writing. The stream is positioned at the beginning of the file.

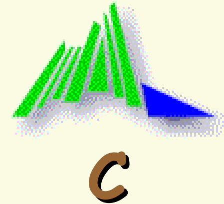
w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

a Open for writing. The file is created if it does not exist. The stream is positioned at the end of the file.

a+ Open for reading and writing. The file is created if it does not exist. The stream is positioned at the end of the file.

Fichiers



- Exemples d'ouverture de fichiers
  - ```
    /* ouverture en mode création d'un fichier binaire */
    if (fp=fopen("C:\TOTO\EMPL.DAT","w+b") == NULL)
      { puts("Erreur d'ouverture");
        exit(1); }
    else
      { /* traitement */ }
    ```
  - ```
    /* ouverture en mode MAJ d'un fichier texte */
    if (fpt=fopen("C:\TOTO\LETTRE.TXT","r+t") == NULL)
      { puts("Erreur d'ouverture");
        exit(1); }
    else
      { /* traitement */ }
    ```

Fichiers

- Lecture d'objets (ou blocs)
  - Prototype
    - ```
      size_t fread( void *adresse,
                    size_t taille,
                    size_t nbobj,
                    FILE *flot)
      ```
  - Exemple

```
/* lecture d'un objet de type employe (structure) */
if (fread((struct employe *)&un_emp,
          sizeof(struct employe), 1, fp) < 1)
  { puts("Erreur de lecture"); exit(1); }
...
```
```

Fichiers

- Écriture d'objets (ou blocs)
  - Prototype
    - ```
      size_t fwrite( void *adresse,
                     size_t taille,
                     size_t nbobj,
                     FILE *flot)
      ```
  - Exemple

```
/* écriture d'un objet de type employe (structure) */
if (fwrite((struct employe *)&un_emp,
           sizeof(struct employe), 1, fp) < 1)
  { puts("Erreur d'écriture"); exit(1); }
...
```
```

Fichiers

- Lecture/écriture d'un caractère
  - Prototypes
    - `int fgetc(FILE *flot)`
    - `int fputc(int c, FILE *flot)`
- Lecture/écriture d'une chaîne
  - Prototypes
    - `char *fgets(char *ch, int n, FILE *flot)`
    - `int fputs(char *ch, FILE *flot)`

Fichiers

- Lectures formatées
  - Prototype
    - `int fscanf(FILE *flot, char *ch_format, arg1, arg2,...,argn)`

```
int fscanf(FILE *flot,
           char *ch_format,
           arg1, arg2,...,argn)
```

- Écritures formatées
  - Prototype

```
int fprintf(FILE *flot,
            char *ch_format,
            arg1, arg2,...,argn)
```

Fichiers

- Fermeture d'un fichier
  - Prototype
    - `int fclose(FILE *flot)`
  - Exemple
    - `fclose(fp);`
    - `fclose(fpt);`
  - Remarque
    - Si vous oubliez de fermer un fichier dans lequel vous avez écrit des informations, vous risquez de perdre des données !

Fichiers

- Pointeur de fichier
  - Sert à repérer la position du prochain octet à lire ou à écrire
  - Est incrémenté automatiquement à chaque opération
- Fin de fichier: `int feof(FILE *flot)`

```
/* boucle de traitement séquentiel */
while (!feof(fp)) { /* 1 lecture et 1 traitement */ }
```

- Position courante: `long ftell(FILE *flot)`

Fichiers

- Positionnement en début de fichier

```
void rewind(FILE *flot)
```

- Positionnement par déplacement (relatif)

```
int fseek(FILE *flot, long depl, long orig)
```

  - avec
    - SEEK_SET (ou 0) correspond au début du fichier
    - SEEK_CUR (ou 1) précise que l'origine est la position courante actuelle
    - SEEK_END (ou 2) donne la fin de fichier comme origine

Fichiers

- Positionnement par déplacement (relatif)

```
int fseek(FILE *flot, long depl, long orig)
```

  - Exemple

```
/* positionnement sur le n-ième enregistrement
   d'un fichier interprété comme contenant des
   informations sur des employés */

fseek(fp, (n-1)*sizeof(struct employe), SEEK_SET);
```

Plan

- Instructions élémentaires
  - types,variables, opérateurs, expressions
  - affichage/lecture de données (`printf`/`scanf`)
- Instructions de contrôle
  - exécution conditionnelle (`if`,`switch`)
  - boucles (`for`,`while`,`do…while`)
- Tableaux, chaînes de caractères
- Fonctions
- Pointeurs
- Structures de données
- Fichiers