

Cours Langage C

Université Sultan Moulay Sliman, Béni Mellal

FPK

Pr. Mohamed CHAKRAOUI

Références

☐ En ligne:

- <https://www.ltam.lu/cours-c/prg-c73.htm>

☐ Livre conseillé:

- Programmer en C Cloud Delannoy ...

Historique du C

Le langage C a été inventé au cours de l'année 1972 dans les Laboratoires Bell. Il était développé en même temps que UNIX par Dennis Ritchie et Ken Thompson. Ken Thompson avait développé un prédécesseur de C, le langage B, qui est lui-même inspiré de BCPL. Dennis Ritchie a fait évoluer le langage B dans une nouvelle version suffisamment différente, en ajoutant notamment les types, pour qu'elle soit appelée C.

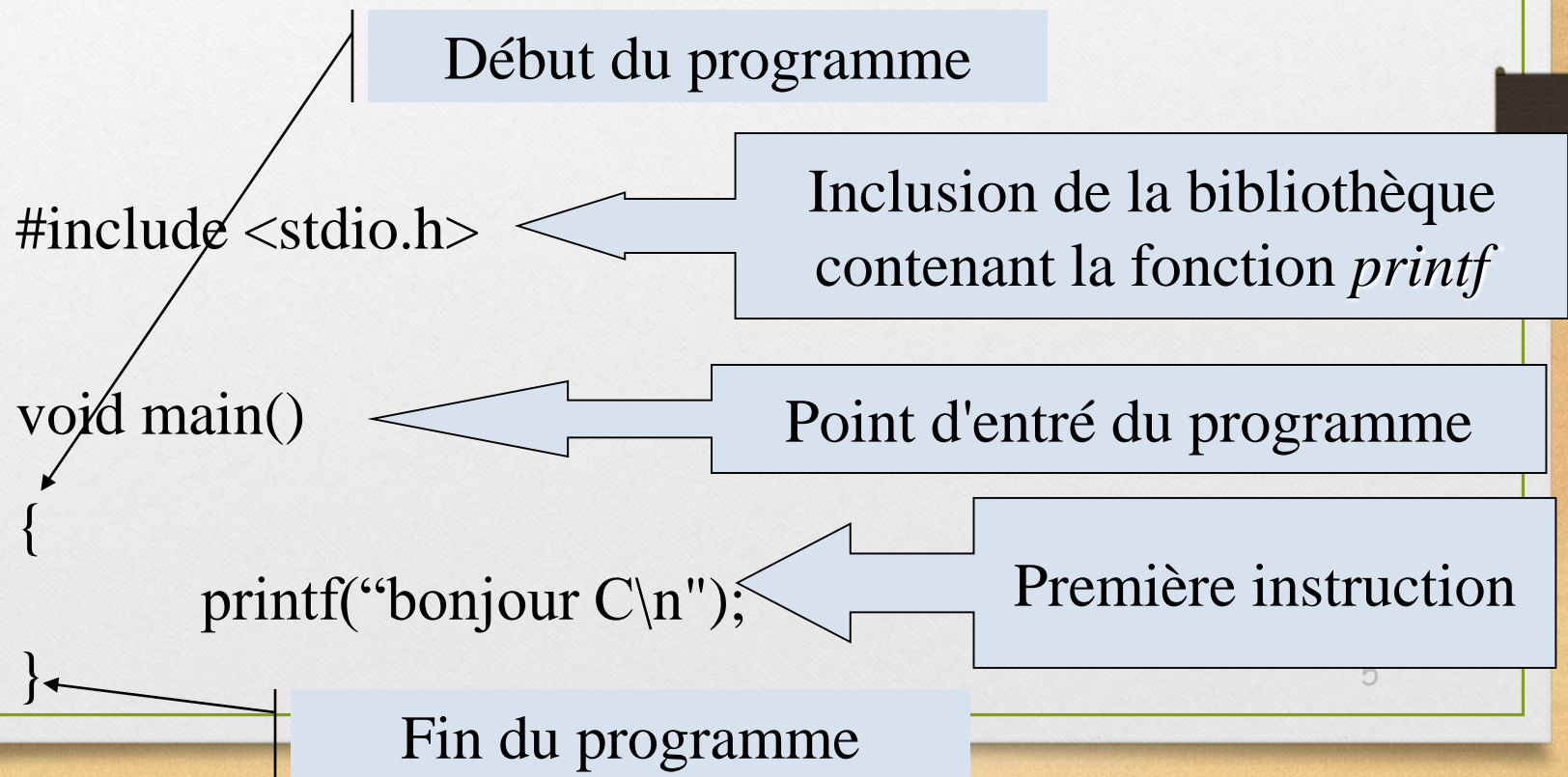
- Bien que C soit officiellement inspiré de B et de BCPL, on note une forte influence de PL/I (ou de PL360) ; on a pu dire que C était à Unix et au PDP-11 ce que PL/I fut pour la réécriture de Multics.
- Par la suite, Brian Kernighan aida à populariser le langage C. Il procéda aussi à quelques modifications de dernière minute.

Normalisation

- En 1983, l'Institut national américain de normalisation (ANSI) a formé un comité de normalisation (X3J11) du langage qui a abouti en 1989 à la norme dite **ANSI C** ou **C89** (formellement ANSI X3.159-1989). En 1990, cette norme a également été adoptée par l'Organisation internationale de normalisation (**C90**, **C ISO**, formellement ISO/CEI 9899:1990). ANSI C est une évolution du C K&R qui reste extrêmement compatible. Elle reprend quelques idées de C++, notamment la notion de prototype et les qualificateurs de type.

Introduction au Langage C

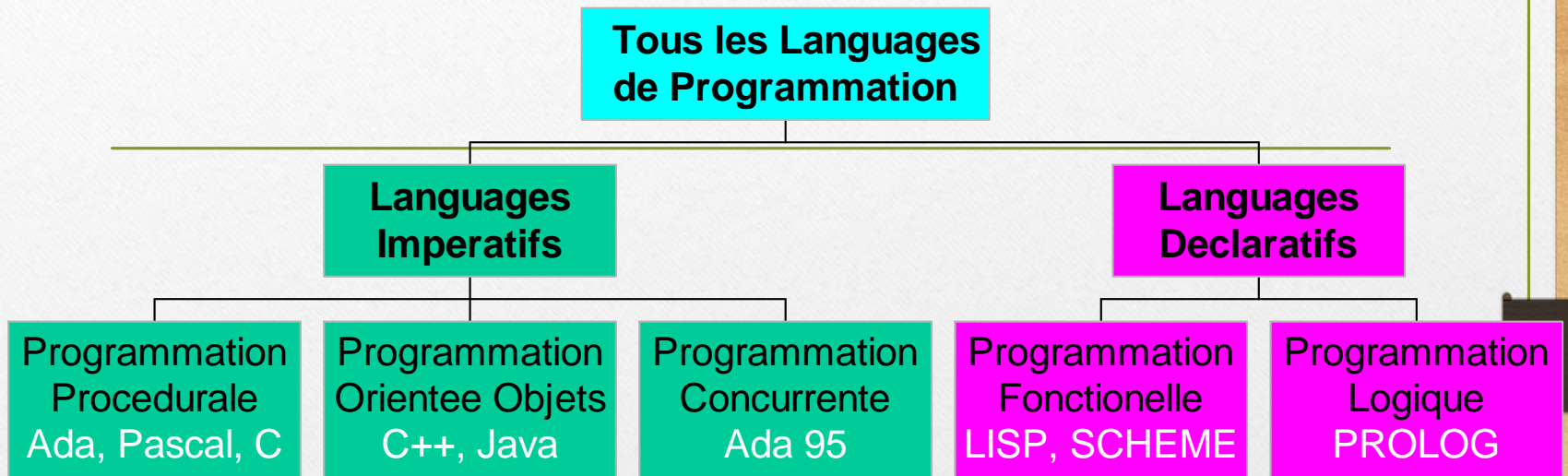
- Mon Premier Programme en C (bonjour C)



Caractéristique du C

- **Structuré**
- **Modulaire:** peut être découpé en modules qui peuvent être compilés séparément
- **Universel:** n'est pas orienté vers un domaine d'application particulier
- **Typé:** tout objet C doit être déclaré avant d'être utilisé
- **Portable:** sur n'importe quel système en possession d'un compilateur C

Classification



Langages Impératifs: Langages incluant des moyens pour le programmeur d'attribuer des valeurs à des locations en mémoire.

Langages Déclaratifs: Langages pour lesquels le programmeur réfléchit en terme de valeurs des fonctions et de relations entre entités diverses. Il n'y a pas d'attribution de valeurs aux variables.

Un long fleuve tranquille

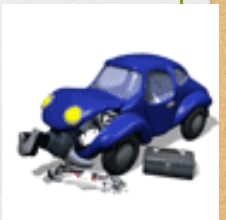


C
fichier

Compilateur C

Code
assembleur

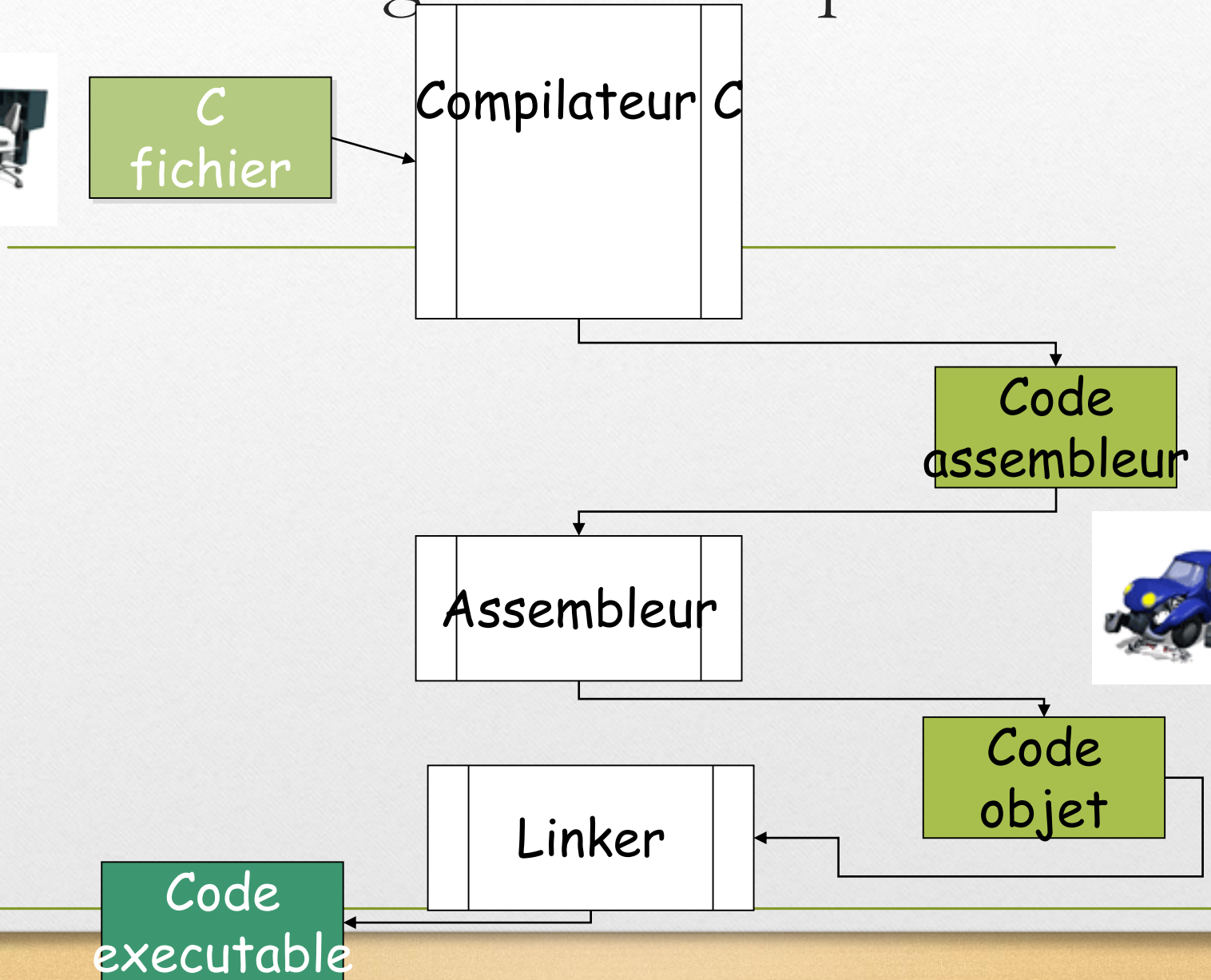
Assembleur



Code
objet

Linker

Code
executable



Compilation en Plusieurs phases

C'est un langage compilé. Cela signifie qu'un programme *C* est décrit par un fichier texte appelé fichier source. Ce fichier n'est pas exécutable par le microprocesseur, il faut le traduire en langage machine. Cette opération est effectuée par un programme appelé compilateur.

La compilation d'un programme *C* se décompose en 4 phases successives :

1. *Le traitement par le préprocesseur* : le fichier source est analysé par un programme appelé préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source, etc.).
2. *La compilation* : au cours de cette étape, le fichier engendré par le préprocesseur est traduit en assembleur, c'est à dire en une suite d'instructions qui sont chacune associées à une fonctionnalité du microprocesseur (faire une addition, une comparaison, etc.).
3. *L'assemblage* : cette opération transforme le code assembleur en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur. Le fichier produit par l'assemblage est appelé fichier objet *.o*).
4. *L'édition de liens* : un programme est souvent séparé en plusieurs fichiers source (ceci permet d'utiliser des bibliothèques de fonctions standard déjà écrites comme les fonctions d'affichage par exemple). Une fois le code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'édition de liens produit alors un fichier exécutable.

La commande *gcc* invoque tour à tour ces différentes phases. Des options de *gcc* permettent de stopper le processus de compilation après chaque des phases.

Fichier C (extension .c)

/* exemple de programme
- somme des nb de 1 à 10 et affichage
de la valeur*/

```
#include <stdio.h>
int main (void)
{
    int somme; int i;
    somme = 0;
    for (i = 1; i <= 3; i++)
    {
        somme = somme + i;
    }
    printf ("%d\n", somme);
}
```

En C le programme principal s'appelle toujours *main*

déclarations de variables de type entier (cases mémoire pouvant contenir un entier)

instruction d'affectation de valeur à la variable *somme*

instructions exécutées en séquence

l'instruction entre accolades est exécutée pour les valeurs de *i* allant de 1 à 3

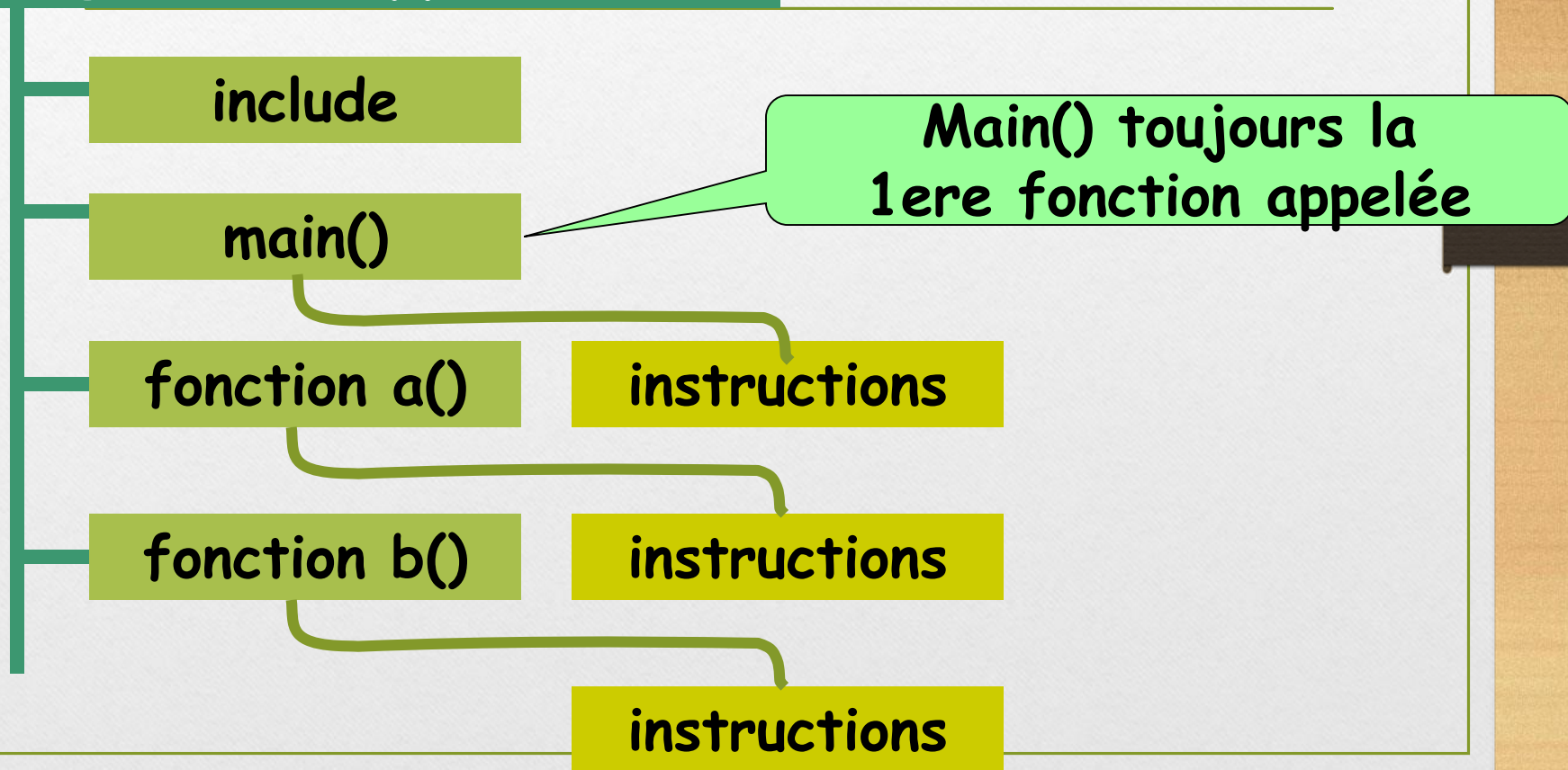
affiche à l'écran la valeur de l'entier contenu dans *somme*

Ecrire le programme suivant :

```
#include <stdio.h>
int main(void) {
    int a= 257, b = 381;
    if (a > 0 && b > 0) {
        printf(" PGCD(%d,%d)\n",a,b);
        while (a != b) {
            if (a < b)
                b = b-a;
            else
                a = a-b;
            printf("=PGCD(%d,%d)\n",a,b);
        }
        printf("=%d\n",a);
    }
    return 0;
}
```

Anatomie

Programme typique en C



Commandes simples de compilation

- Prétraitement, compilation et édition de liens :
- `gcc -Wall -g pgcd.o -lm -o pgcd`
- l'option `-Wall` demande une compilation avec des diagnostics sur la propreté du code
- l'option `-g` demande que la *table des symboles* soit ajoutée à l'exécutable
- l'option `-lm` demande de lier la librairie mathématique
- l'option `-o pgcd` demande que le résultat (l'exécutable) soit nommé `pgcd` au lieu de `a.out`
- Le programme est à lancer avec `./pgcd`

Définition

- Les variables contiennent les valeurs qui sont utilisées pendant l'exécution du programme. Les noms des variables sont des identificateurs quelconques. Les différents types de variables simples et les opérateurs admissibles seront discutés dans ce cours. Tout simplement une variable est une entité qui contient une valeur.

Introduction au Langage C

❑ Caractéristiques d'une variable

■ nom:

- ❑ Unique pour chaque variable
- ❑ Commence toujours par une lettre
- ❑ Différenciation minuscule-majuscule

■ type:

- ❑ Conditionne le format de la variable en mémoire
- ❑ Peut être soit un type standard ou un type utilisateur

■ valeur:

- ❑ Peut évoluer pendant l'exécution
- ❑ initialisation grâce à l'opérateur d'affectation par exemple

Introduction au Langage C

☐ Types de variable

// [...] signifie facultatif

- Char → caractères
- int → entiers
- short [int] → entiers courts
- long [int] → entiers longs
- float → nombres décimaux
- double → nombres décimaux de précision supérieure
- long double → nombres décimaux encore plus précis
- unsigned int → entier non signé

Introduction au Langage C

■ Déclaration d'une variable

Type nom_de_la_variable [= valeur]

■ Exemple:

- int nb;
- float pi = 3.14;
- char c = 'a';
- long i,j,k;
- double r = 6.2879821365;

Types d'instruction en C

- ✓ Déclarations des variables
- ✓ Assignations
- ✓ Fonctions
- ✓ Contrôle

Types de données et de variables

□ Déclaration des variables

- `int y;`
- `char yesno, ok;`
- `int ordered = 1, onhand = 0;`
- `float total = 43.132;`
- `char *cptr = NULL;`

Types de données et de variables: char

□ Type « char » ou « signed char »:

- ASCII sur 32 bits
 - de -2147483648 à 2147483647.

□ Type « unsigned char »:

- ASCII sur 32 bits
 - de 0 à 4294967295.

Types de données et de variables: short

□ Type « short » ou « signed short »

- en complément à 2 sur 32 bits
 - de -2147483648 à 2147483647.

□ Type « unsigned short »:

- binaire sur 32 bits
 - de 0 à 4294967295.

Types de données et de variables:

int

□ Type « int » ou « signed int »

- en complément à 2 sur 32 bits
 - de -2147483648 à 2147483647.

□ Type « unsigned int »:

- binaire sur 32 bits
 - de 0 à 4294967295.

Types de données et de variables:

long

- Les spécificateurs **%d**, **%i**, **%u**, **%o**, **%x** peuvent seulement représenter des valeurs du type **int** ou **unsigned int**. Une valeur trop grande pour être codée dans deux octets est coupée sans avertissement si nous utilisons **%d**.
- Pour pouvoir traiter correctement les arguments du type **long**, il faut utiliser les spécificateurs **%ld**, **%li**, **%lu**, **%lo**, **%lx**.
- Type « long » ou « signed long »
 - en complément à 2 sur 32 bits
 - de -2147483648 à 2147483647.
- Type « unsigned long »:
 - binaire sur 32 bits
 - de 0 à 4294967295.

Types de données et de variables:

long

Exemple:

- `long N = 1500000;`
- `printf("%ld, %lx", N, N); ==> 1500000, 16e360`
- `printf("%x, %x" , N); ==> e360, 16`
- `printf("%d, %d" , N); ==> -7328, 22`

Réel ou rationnels

- Les spécificateurs **%f** et **%e** peuvent être utilisés pour représenter des arguments du type **float** ou **double**. La mantisse des nombres représentés par **%e** contient exactement un chiffre (non nul) devant le point décimal. Cette représentation s'appelle la *notation scientifique* des rationnels.
- Pour pouvoir traiter correctement les arguments du type **long double**, il faut utiliser les spécificateurs **%Lf** et **%Le**.

- Exemple:

float N = 12.1234;

double M = 12.123456789;

long double P = 15.5;

➤ printf("%f", N); ==> 12.123400

➤ printf("%f", M); ==> 12.123457

➤ printf("%e", N); ==> 1.212340e+01

➤ printf("%e", M); ==> 1.212346e+01

➤ printf("%Le", P); ==> 1.550000e+01

Largeur minimale pour les entiers

□ Pour les entiers, nous pouvons indiquer la largeur minimale de la valeur à afficher. Dans le champ ainsi réservé, les nombres sont justifiés à droite.

■ Exemples (_ <=> position libre)

➤ `printf("%4d", 123);` ==> `_123`

➤ `printf("%4d", 1234);` ==> `1234`

➤ `printf("%4d", 12345);` ==> `12345`

➤ `printf("%4u", 0);` ==> `__0`

➤ `printf("%4X", 123);` ==> `__7B`

➤ `printf("%4x", 123);` ==> `__7b`

Largeur minimale et precision pour les r  ls

□ Pour les rationnels, nous pouvons indiquer la largeur minimale de la valeur    afficher et la pr  cision du nombre    afficher. La pr  cision par d  faut est fix  e    six d  cimales. Les positions d  cimales sont arrondies    la valeur la plus proche.

■ Exemples

➤ `printf("%f", 100.123);` ==> 100.123000

➤ `printf("%12f", 100.123);` ==> __100.123000

➤ `printf("%.2f", 100.123);` ==> 100.12

➤ `printf("%5.0f", 100.123);` ==> __100

➤ `printf("%10.3f", 100.123);` ==> ____100.123

➤ `printf("%.4f", 1.23456);` ==> 1.2346

Types de données et de variables

- ❖ Type « enum » (ordered list)
 - en complément à 2 sur 32 bits
 - de -2147483648 à 2147483647.

- ❖ Type « float »
 - format TMS320C30 sur 32 bits
 - de 5.9×10^{-39} à 3.4×10^{38} .

Types de données et de variables: double

□ Type « double »

- format TMS320C30 sur 32 bits
 - de 5.9×10^{-39} à 3.4×10^{38} .

□ Type « long double »

- format TMS320C30 sur 40 bits
 - de 5.9×10^{-39} à 3.4×10^{38} .

Types de données et de variables

- Type « pointer »
 - binaire sur 32 bits
 - de 0 à 0xFFFFFFFF.

Entrées sorties

□ Affichage de la valeur d'une variable

■ en C

❖ On utilise la fonction `printf(...)`;

❖ Exemple

- `int i = 8; int j = 10;`
- `printf("i vaut: %d\n", i, j);`

- `float r = 6.28;`
- `printf("le rayon = %f \n", r);`

scanf

□ scanf (dont le nom vient de l'anglais scan formatted) est une fonction de la bibliothèque standard du langage C. Déclarée dans l'entête <stdio.h>, cette fonction peut être utilisée pour la saisie de données formatées, qu'il s'agisse de lettres, de chiffres ou de chaînes de caractères.

□ Exemple

```
int i; float x;  
printf(" donnez une valeur de i \n");  
scanf("%d",&i);  
printf("svp saisir la valeur de x \n")  
scanf("%f",&x);  
printf("les valeurs saisies sont i =%d et x =%f \n",i,x);
```

Exercices

- Ecrire un programme C qui demande deux entiers et calcul leur somme, produit et division et les affiche séparément
- Ecrire un programme C qui demande un nombre et afficher sont carré et son double
- Ecrire un programme qui demande un caractère et l'affiche
- Télécharger devcpp
- Télécharger codeblocks

Entrées sorties

- %d: entier
- %f: réel
- %s chaine de caractères
- %u adresse
- %u: entier positif(utilisée pour les adresses)
- %p adresse en hexadécimal sur 8 bits(conseillé pour les pointeurs)
- %x adresse en hexadécimal

Spécificateurs de format pour printf

<i>SYMBOLE</i>	<i>TYPE</i>	<i>IMPRESSION COMME</i>
%d ou %i	int	entier relatif
%u	int	entier naturel (unsigned)
%o	int	entier exprimé en octal
%x	int	entier exprimé en hexadécimal
%c	int	caractère
%f	double	rationnel en notation décimale
%e	double	rationnel en notation scientifique
%s	char*	chaîne de caractères
%b	entier exprimé en hexadecimal	int

Indication de largeur maximale

- Pour tous les spécificateurs, nous pouvons indiquer la largeur maximale du champ à évaluer pour une donnée. Les chiffres qui passent au-delà du champ défini sont attribués à la prochaine variable qui sera lue !
- Exemple
- Soient les instructions:

```
int A,B;  
scanf("%4d %2d", &A, &B);
```
- Si nous entrons le nombre 1234567, nous obtiendrons les affectations suivantes:

```
A=1234  
B=56
```

le chiffre 7 sera gardé pour la prochaine instruction de lecture.

Nombre de valeur lues

- Lors de l'évaluation des données, scanf s'arrête si la chaîne de format a été travaillée jusqu'à la fin ou si une donnée ne correspond pas au format indiqué. scanf retourne comme résultat le nombre d'arguments correctement reçus et affectés.

- Exemple

La suite d'instructions

```
int JOUR, MOIS, ANNEE, RECU;
```

```
RECU = scanf("%i %i %i", &JOUR, &MOIS, &ANNEE);
```

réagit de la façon suivante (- valeur indéfinie):

saisit:		RECU	JOUR	MOIS	ANNEE
12 4 1980	==>	3	12	4	1980
12/4/1980	==>	1	12	-	-
12.4 1980	==>	1	12	-	-
12 4 19.80	==>	3	12	4	19

Instructions de base

- $+, -, *, /$ → opérateurs arithmétique de base
- $\%$ → reste d'une division entière
- $==$ → test d'égalité
- $!=$ → test de différence
- $<, >, <=, >=$ → test de comparaison
- $!$ → négation
- $||$ → ou logique pour évaluer une expression
- $\&\&$ → et logique pour évaluer une expression

Instructions de base

□ opérateurs de base

- $a = 2 + 3$
- $r = 3 \% 2$
- $a = (3 == 3)$
- $a = (6 == 5)$
- $a = (2 != 3)$
- $a = (6 <= 3)$
- $a = !1$
- $a = ((3 == 3) \ || \ (6 <= 3))$
- $a = ((3 == 3) \ \&\& \ (6 <= 3))$

Instructions de base

□ boucle pour

```
for(expr1;expr2;expr3)
{
    instructions
}
```

expr1: évaluée 1 seule fois en début de boucle

expr2: évaluée avant chaque itération. Si vrai alors les instructions de la boucle sont exécutées sinon la boucle est terminée

expr3: évaluée à la fin de chaque itération

Instructions de base

□ boucle pour

- exemple

// somme des 100 premiers entiers

```
int i,s;
```

```
s= 0;
```

```
for(i=0;i<=4;i = i+1)
```

```
{
```

```
    s = s+i;    // ou s+=i;
```

```
}
```

```
printf('la val de la somme %d', s);
```


Instructions de base

□ boucle tant que

```
while (condition)
{
    instructions;
}
```

Condition est évaluée avant chaque itération. Si le résultat est vrai alors les *instructions* sont exécutées sinon on sort de la boucle

□ Exemple

```
i = 0;
n = 20;
while (i < n)
{
    i++;
}
```

Instructions de base

□ boucle répéter

```
do
{
    instructions;
} while (expression)
```

expression est évaluée après chaque itération. Si le résultat est vrai alors les instructions sont exécutées sinon on sort de la boucle

□ Exemple

```
i = 0;
n = 20;
do
{
    i++;
} while(i < n)
```


Instructions de base

□ instruction conditionnelle simple *si alors*

```
if (expression)
{
    instructions;
}
```

expression est évaluée après chaque itération. Si le résultat est vrai alors les **instructions** sont exécutées sinon on sort de la boucle.

□ exemple

```
int i =5;
int n;
if (i<=20)
{
    n =0;
}
```

Instructions de base

- instruction conditionnelle simple *si alors sinon*

```
if (expression)
```

```
{  
    instructions1;
```

```
} else
```

```
{  
    instructions2;
```

```
} expression est évaluée après chaque itération. Si le résultat est vrai  
    alors les instructions1 sont exécutées sinon on exécute  
    l'ensemble instructions2
```

- exemple

```
int i =5;int n;
```

```
if (i<=20)
```

```
{
```

```
    n =0;
```

```
} else {n=5;}
```



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

```
main() {
```

```
    int N=10, P=5, Q=10, R;
```

```
    char C='S';
```

```
    N = 5; P = 2;
```

```
    Q = N++ > P || P++ != 3;
```

```
    printf ("C : N=%d P=%d Q=%d\n", N, P, Q);
```

```
    N = 5; P = 2;
```

```
    Q = N++ < P || P++ != 3;
```

```
    printf ("D : N=%d P=%d Q=%d\n", N, P, Q);
```

```
    N = 5; P = 2;
```

```
    Q = ++N == 3 && ++P == 3;
```

```
    printf ("E : N=%d P=%d Q=%d\n", N, P, Q);
```

```
    N=5; P=2;
```

```
    Q = ++N == 6 && ++P == 3;
```

```
    printf ("F : N=%d P=%d Q=%d\n", N, P, Q);
```

```
    N=C;
```

```
    printf ("G : %c %c\n", C, N);
```

```
    printf ("H : %d %d\n", C, N);
```

```
    printf ("I : %x %x\n", C, N);
```

```
    return 0;}
```

exercice

a) Sans utiliser l'ordinateur, trouvez et notez les résultats du programme ci-dessus.

b) Vérifiez vos résultats à l'aide de l'ordinateur.

Instructions de base

■ instruction conditionnelle multiple

```
switch (expression)
{
    case valeur1:
        instructions1;break;
    case valeur2:
        instructions2;break;
    :
    case valeur3:
        instruction3;break;
    default:
        instruction4;break;
}
```

`expression` est évaluée. Si le résultat vaut *valeur1*, alors **instruction1** est exécutée et on quitte le switch, sinon si *expression* vaut *valeur2*,, sinon on va dans *default* pour exécuter *instruction4*.

Instructions de base

□ instruction conditionnelle multiple

■ exemple

```
char Language='f';  
switch (Language)  
{  
    case 'f':  
        printf(" Français"); break;  
    case 'e':  
        printf("Anglais");   break;  
    case 'a':  
        printf(" Arab");     break;  
    default:  
        printf("Autres");    break;  
}
```

Instructions de base

□ l'instruction break

- permet d'interrompre prématurément une boucle et de se brancher vers la première instruction n'appartenant pas à la boucle

- exemple:

```
int i; int n=20;
for (i=0;i<10;i++)
{
    if (n==30){
        break;
    }
    n=n+2;
}
printf("%d \n", n);
```



Quand n vaut 30 alors la boucle est interrompue

Les tableaux

- Les tableaux statiques à 1 dimension
 - définition
 - Ensemble de variables de même type, de même nom caractérisées par un index.
 - déclaration
 - `type nom_tableau[dimension]` // dimension doit être une constante
 - exemples:
 - `char buffer[80];`
 - `int mat[10];`

Les tableaux

❑ Les tableaux statiques à 1 dimension

❖ accès aux éléments du tableau

- `Nom_tableau[indice]`

❖ exemples:

- `buffer[5] = 'c';`
- `mat[6] = 10;`

le premier élément commence à l'indice 0 !!

Les valeurs ne sont pas initialisées !!

Les débordements ne sont pas vérifiés

Les tableaux

- Les tableaux statiques à 2 dimensions et plus
 - définition
 - Il s'agit d'un tableau de tableaux
 - déclaration
 - `type nom_tableau[dim1][dim2]...[dimn]`
 - exemples:
 - `char buffer[20][80];`
 - `int mat[6][10];`
 - `char livres[100][60];`

Les tableaux

- Les tableaux statiques à 2 dimensions et plus
 - accès aux éléments
 - `nom_tableau[ind1][ind2]`
 - exemples:
 - `livre[30][15] = 'c';`
 - `mat[5][6] = 13;`
 - `Printf("%d",mat[5][6]);`

Les chaines(1)

- En C, il n'existe pas de type de variable pour les chaînes de caractères comme il en existe pour les entiers (**int**) ou pour les caractères (**char**). Les chaînes de caractères sont en fait stockées dans un tableau de char dont la fin est marquée par un caractère nul, de valeur 0 et représentée par le caractère '\0' ou '\x0' ou la valeur 0 directement. En mémoire la chaîne "Bonjour" est représentée ainsi :

B	o	n	j	o	u	r	\0
---	---	---	---	---	---	---	----

- Tout ce qui suit le caractère '\0' sera ignoré :

Les chaines(2)

```
char s[14] = "Hello\0World!" ;  
printf ("%s\n", s);
```

- ❖ Affichera seulement "Hello". Il ne faut donc pas oublier de réserver une place supplémentaire pour le caractère de fin de chaîne sinon on obtient un simple tableau de caractères et dans ce cas, son utilisation en tant que chaîne de caractère mène à un comportement indéfini.
- ❖ Il existe plusieurs manières de déclarer et d'initialiser une chaîne de caractères :
 - `char s1[11] = "languageC";`
 - `char s2[] = "languageC";`
 - `char *s3 = "languageC";`

Les chaines(7)

- ❖ La première méthode réserve une zone de 11 octets et la chaîne **"languageC"** y est stockée.
- ❖ La seconde méthode laisse le compilateur calculer la taille appropriée (11 octets dans notre cas).
- ❖ La dernière méthode ne réserve pas de mémoire, elle se contente de stocker l'adresse d'une chaîne de caractères qui peut se trouver dans un endroit de la mémoire non modifiable par le programme, dans ce cas toute modification de la chaîne s3 se conclura par un comportement indéterminé. il est donc conseillé d'écrire :
`const char *s3 = " programmation C";`

Les chaines(3)

- Initialisation avec une longueur explicite
- Comme pour n'importe quel tableau, l'initialisation se réalise à l'aide d'une liste d'initialisation. L'exemple ci-dessous définit donc un tableau de vingt-cinq char et initialise les sept premiers avec la suite de lettres « Bonjour ».

```
char chaine[25] = { 'B', 'o', 'n', 'j', 'o', 'u', 'r' };
```

- Étant donné que seule une partie des éléments sont initialisés, les autres sont implicitement mis à zéro, ce qui nous donne une chaîne de caractères valides puisqu'elle est bien terminée par un caractère nul. Faites cependant attention à ce qu'il y ait toujours de la place pour un caractère nul.

Les chaines(4)

- **Initialisation avec une longueur implicite**
- Dans le cas où vous ne spécifiez pas de taille lors de la définition, il vous faudra ajouter le **caractère nul** à la fin de la liste d'initialisation pour obtenir une chaîne valide.

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

Les chaines(5)

- **Avec une chaîne littérale**
- Bien que tout à fait valide, cette première solution est toutefois assez fastidieuse. Aussi, il en existe une seconde: recourir à une **chaîne de caractères littérales** pour initialiser un tableau. Une chaîne de caractères littérales est une suite de caractères entourée par le symbole " " Nous en avons déjà utilisé auparavant comme argument des fonctions printf et scanf.
- Techniquement, une chaîne littérale est un tableau de char terminé par un caractère nul. Elles peuvent donc s'utiliser comme n'importe quel autre tableau. Si vous exécutez le code ci-dessous, vous remarquerez que l'opérateur sizeof retourne bien le nombre de caractères composant la chaîne littérale (n'oubliez pas de compter le caractère nul) et que l'opérateur[] peut effectivement leur être appliqué

Les chaines(6)

```
#include <stdio.h>

int main(void)
{
    printf("%u\n", (unsigned)sizeof "Bonjour");
    printf("%c\n", "Bonjour"[3]);
    return 0;
}
```

Les chaines(8)

- Déclarer une chaîne de caractères constante est aussi une bonne habitude à prendre lors du passage d'une chaîne comme paramètre d'une fonction qui ne modifie pas cette dernière.

Si vous souhaitez utiliser une chaîne de taille variable, il faut utiliser les mécanismes d'allocation dynamique de la mémoire :

1. `char *s = malloc (sizeof (*s) * 256);`
`/* * Utilisation d'une chaine de 255 caractères maximum */`
2. `free (s);`

Exercices

- Écrivez un programme C qui compte le nombre d'occurrences d'un caractère `c` dans une chaîne `s`.
- Écrivez un programme qui demande une chaîne de caractères, la renverse sur elle-même exemple: ("toto!" a pour miroir "otot") et affiche l'adresse de cette chaîne
- Écrivez un programme qui recherche dans une chaîne chaque caractère `c` pour le remplacer par un caractère `r` et retourne l'adresse de la chaîne.

Ex. Chaines et le fonctions

- Ecrire une fonction qui comparer deux chaines
- Ecrire une fonction qui vérifie si les deux chaines sont identiques
- Ecrire une fonction qui cherche combien de caractère de chaine1 existe dans chaine2
- Ecrire une fonction qui cherche combien de caractère de chaine1 existe dans chaine2
- Ecrire une fonction qui cherche combien de caractère de chaine1 existe dans chaine2

Les fonctions

- **définition**

-
- Ensemble d'instructions pouvant être appelés de manière répétitive par son nom

- **déclaration**

```
type arg_retours nom_fonction( type arg1,type arg2, ...type argn)
{
    ensemble instructions
}
```

- arg_retours est l'argument renvoyé par la fonction (instruction return)
- nom_fonction est le nom de la fonction
- arg1 ...argn sont les arguments envoyés à la fonction

Les fonctions

■ règles d'utilisation

- L'ordre, le type et le nombre des arguments doivent être respectés lors de l'appel de la fonction
- L'appel d'une fonction doit être située après sa déclaration ou celle de son prototype (voir exemple transp suivant)
- Si la fonction ne renvoie rien alors préciser le type *void*

Les fonctions

- **Les fonctions**

- **exemple:**

```
int minimum(int a, int b);  
void main()  
{  
    int a,b;  
    printf("donnez les valeurs de a et b");  
    scanf(" %d",&a);  
    scanf(" %d",&b);  
    printf("%d", minimum(a,b));  
}  
int minimum(int a, int b)  
{  
    if (a < b) return a;  
    else return b;  
}
```

**Prototype de
la fonction
min**

**Programme
principal**

**Fonction min
et son bloc
d'instruction
s**

Les fonctions

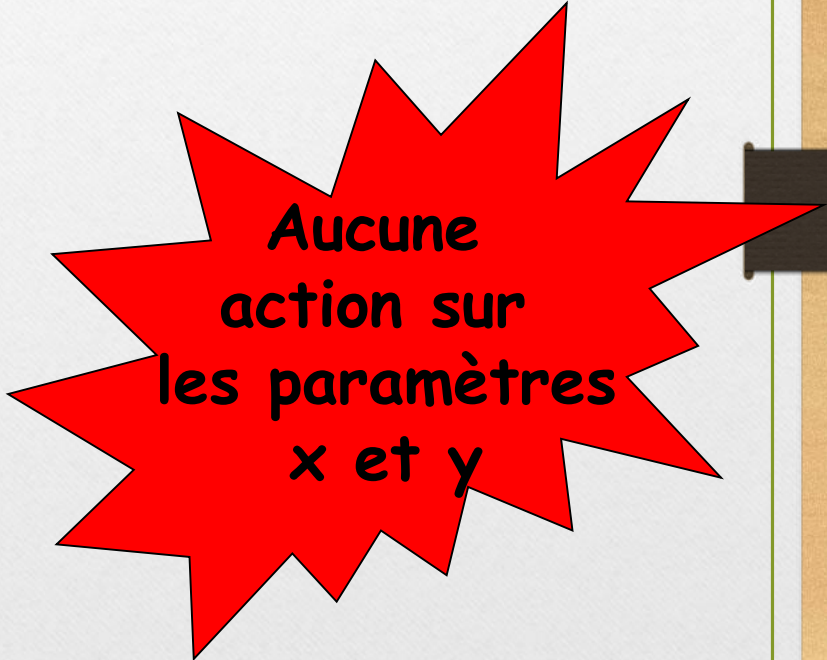
- Les paramètres se passent par valeur
 - Recopie en mémoire des paramètres dans des paramètres temporaires.
 - Toute modification des paramètres dans la fonction est sans effet sur les paramètres originelles
 - Quand on quitte la fonction, les paramètres temporaires sont effacés de la mémoire
 - Recopie dans le sous programme appelant de la valeur du paramètre retourné par la fonction *return*.

Les fonctions

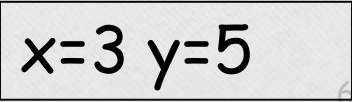
- exemple:

```
void permutter(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void main()
{
    int x=3;int y =5;
    permutter(x,y);
    printf( "x=%0d y=%0d ",x,y);
}
```



**Aucune
action sur
les paramètres
x et y**



x=3 y=5

La portée des fonctions et variables

- Pour clore ce chapitre, il nous faut impérativement découvrir la notion de portée des fonctions et des variables. Nous allons voir quand les variables et les fonctions sont accessibles, c'est-à-dire quand on peut faire appel à elles.
- Les variables propres aux fonctions
- Lorsque vous déclarez une variable dans une fonction, celle-ci est supprimée de la mémoire à la fin de la fonction :

```
int triple(int nombre)
{
    int resultat = 0; // La variable resultat est créée en mémoire

    resultat = 3 * nombre;
    return resultat;
} // La fonction est terminée, la variable resultat est supprimée de la mémoire
```


Les variables globales : à éviter

- Variable globale accessible dans tous les fichiers
- Il est possible de déclarer des variables qui seront accessibles dans toutes les fonctions de tous les fichiers du projet.
- généralement il faut éviter de le faire. Ça aura l'air de simplifier votre code au début, mais ensuite vous risquez de vous retrouver avec de nombreuses variables accessibles partout, ce qui risquera de vous créer des soucis.
- Pour déclarer une variable « globale » accessible partout, vous devez faire la déclaration de la variable en dehors des fonctions. Vous ferez généralement la déclaration tout en haut du fichier, après les `#include`

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

int resultat = 0; // Déclaration de variable globale

void triple(int nombre); // Prototype de fonction

int main(int argc, char *argv[])
{
    triple(15); // On appelle la fonction triple, qui modifie la variable globale resultat
    printf("Le triple de 15 est %d\n", resultat); // On a accès à resultat
    return 0;
}

void triple(int nombre)
{
    resultat = 3 * nombre;
}
```


Variable globale accessible uniquement dans un fichier

- La variable globale que nous venons de voir était accessible dans tous les fichiers du projet.
- Il est possible de la rendre accessible uniquement dans le fichier dans lequel elle se trouve. Ça reste une variable globale quand même, mais disons qu'elle n'est globale qu'aux fonctions de ce fichier, et non à toutes les fonctions du programme.
- Pour créer une variable globale accessible uniquement dans un fichier, rajoutez simplement le mot-clé static devant :

static int resultat = 0;

Variable statique à une fonction

- Si vous rajoutez le mot-clé static devant la déclaration d'une variable à l'intérieur d'une fonction, ça n'a pas le même sens que pour les variables globales.
- En fait, la variable static n'est plus supprimée à la fin de la fonction. La prochaine fois qu'on appellera la fonction, la variable aura conservé sa valeur.
- Par exemple :

```
int triple(int nombre)
{
    static int resultat = 0; // La variable resultat est créée la première fois que la fonction est appelée
    resultat = 3 * nombre;
    return resultat;
} // La variable resultat n'est PAS supprimée lorsque la fonction est terminée.
triple(4); ----> 12
triple(2); -----> 6
```


Variable statique

- Qu'on pourra rappeler la fonction plus tard et la variable resultat contiendra toujours la valeur de la dernière fois.
- Voici un petit exemple pour comprendre :

1
2
3
4

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

int incremente()
{
    static int nombre = 0;

    nombre++;
    return nombre;
}
```

Variable statique

- Exemple contraire

1
1
1
1

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

    nombre++;
    return nombre;
}
```

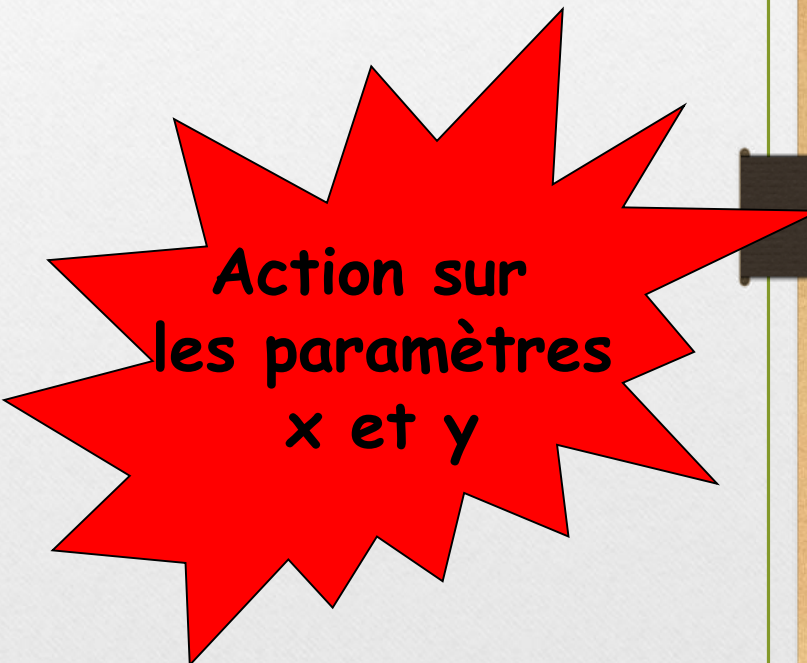

Les fonctions

- Passage d'arguments par adresse
 - On passe non pas la valeur de la variable mais son adresse
 - Il est donc possible par les pointeurs de modifier le contenu de l'adresse, donc la variable originelle
 - Permet de modifier plusieurs arguments à la fois dans une fonction
 - Permet de passer des tableaux en argument

Les fonctions

- exemple:

```
void permutter(int *pa, int *pb)
{
    int temp;
    temp = *pa;
    *pa = *pb;
    *pb = temp;
}
void main()
{
    int x=3;int y =5;
    permutter(&x, &y);
    printf("x=%d et y=%d", x,y);
}
```



**Action sur
les paramètres
x et y**

x=5 y=3

La Récursivité

- ▶ Les fonctions récursives, sont des fonctions s'appelant elles-mêmes, elles sont également un moyen rapide pour poser certains problèmes algorithmiques ; nous allons voir en détail comment elles fonctionnent.
- ▶ Prenons un problème simple, mais auquel vous n'avez peut-être pas pensé à utiliser la récursivité: le calcul d'une factorielle. Considérons $n!$ comme étant la factorielle à calculer, nous aurons ceci: $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$. Dans cette situation, nous pouvons déjà déterminer notre règle de sortie de notre fonction récursive: la valeur 1 qui symbolise la fin de la récursion !

La Récursivité

- ▶ En effet, il faut une condition de sortie pour la fonction, mais il faut être très vigilant quant au choix de la condition, vous devrez être sûr qu'elle soit validée à un moment ou à un autre sinon c'est comme si vous créez une boucle infinie sans condition de sortie !
- ▶ La règle de récursion que nous devons définir, est le calcul de la factorielle en elle même soit, si nous considérons notre exemple sur $6!$, cité précédemment, nous pouvons définir notre règle de cette manière: $n! = (n) (n-1) (n-2) \dots (1)$. Nous pouvons en déduire que nous allons faire des appels en décrémentant la valeur de n à chaque appel de la fonction jusqu'à ce que $n == 1$!

La Récursivité

- Nous pouvons considérer le sous programme suivant:

```
unsigned long factoriel (int n)
{
    if (n < 0) {
        exit (EXIT_FAILURE); }
    else if (n == 1 || n == 0)
    {
        return 1L;
    }
    return n * factoriel (n - 1);
}
```

La Récursivité

- Nous pouvons observer ici que le dernier **return** est en fait l'appel récursif et nous soustrayons 1 à chaque appel jusqu'à ce que $n == 1$ qui est, comme décrit plus haut, notre condition de sortie.
- Non, cela ne s'arrête pas là et c'est ici que nous allons voir le fonctionnement des fonctions récursives. Lorsque la fonction rencontre la condition de sortie, elle remonte dans tous les appels précédents pour calculer n avec la valeur précédemment trouvée !

La Récursivité

- Les appels des fonctions récursives sont en fait empilés (pile qui est une structure de données régie selon le mode LIFO: Last In First Out, Dernier Entré Premier Sorti). Chaque appel se trouve donc l'un à la suite de l'autre dans la pile du programme. Une fonction de ce type possède donc deux parcours: la phase de descente et la phase de remontée.
- Voyons ceci grâce à un petit schéma :

La Récursivité

- Nous voyons très bien la phase de descente et de remontée dans la pile des appels de la fonction récursive. Ce n'est qu'au moment de la remontée, donc également au moment où la condition de sortie est vraie, que les appels enregistrés sont dépilés au fur et à mesure de la remontée. Ici pour des petits calculs cela convient très bien, mais lorsqu'il s'agit de faire de plus profondes récursions, un autre type de fonction récursive existe, mais n'est pas forcément connue de tout le monde, c'est la récursivité terminale, que nous allons étudier dans le prochain chapitre.

Descente

N==6

N==5

N==4

N==3

N==2

N==1

Remontée

N==6

N==5

N==4

N==3

N==2

N==1

Condition de sortie N==1

Fonctions récursives terminales

- ▶ Avec ce que nous avons vu plus haut, imaginez un instant que vous devez calculer le factoriel d'un très grand nombre, par exemple, un calcul de probabilité sur le tirage des boules ! voire les combinatoires, vous savez que les calculs de factorielles y sont omniprésents, mais, je ne vais pas trop entrer dans les détails pour ceux qui ne sont pas très familiers avec ce genre de connaissances mathématiques.
- ▶ Nous allons simplement imaginer les inconvénients qu'une récursivité normale peut avoir sur des récursions plus profondes. Imaginez une grille de numéro, elle contient 49 numéros dont seulement 6 peuvent être tirés au sort. En restant dans un calcul simple pour ne pas trop nous dérouter du sujet, nous aurions une formule comme ceci pour calculer des probabilités :
 - ▶
$$C_{49}^6 = \frac{49!}{(49-6)! \cdot 6!}$$
- ▶ ci, nous devrions calculer le factoriel de 49 ce qui nous donne: $49! = 49 \times 48 \times 47 \dots 8 \times 7 \times 6!$; le résultat obtenu est d'une grandeur inimaginable !

Fonctions récursives terminales

- Mais cela n'est pas le sujet, c'est juste pour vous montrer que de cette manière, avec une récursivité normale, nous avons $(49-6) \times 2$ passages soit 43 appels empilés l'un après l'autre sans compter qu'il faut également remonter tous les appels ce qui nous fait un total de 86 passages. Vous pouvez vous imaginer la perte de temps sur des récursions encore plus profondes et qui risqueraient par ailleurs de faire exploser la pile ce qui conduirait irrémédiablement au plantage du programme !
- C'est là que peuvent intervenir les récursions terminales !

Fonctions récursives terminales

- c'est une récursion avec uniquement une phase de descente, sans remontée. Ceci est possible, car la dernière expression **return factoriel_terminale (...)** de notre fonction nous renvoie directement la valeur obtenue par l'appel récursif courant, sans qu'il n'y ait d'autres opérations à faire, ce qui n'est pas le cas dans notre fonction récursive simple, où l'on multiplie n par le retour de la fonction. En conséquence, les appels de la fonction n'ont pas besoin d'être empilés, car l'appel suivant remplace simplement l'appel précédent dans le contexte d'exécution.

Fonctions récursives terminales

```
unsigned long factoriel_terminal (int n, unsigned long result)
{
    if (n < 0)
    {
        exit (EXIT_FAILURE); }
    if (n == 1) {
        return result; }
    else if (n == 0) {
        return 1L; }
    return factoriel_terminal (n - 1, n * result);
}
```

Fonctions récursives terminales

Nous remarquons ici que nous avons pris un argument supplémentaire. C'est un passage obligatoire pour créer une récursivité terminale ; de cette manière, la dernière instruction est bel et bien la fin de l'appel courant de notre fonction et donc l'appel suivant peut prendre la place de la précédente car le résultat se trouve dans notre second argument.

Comme à l'accoutumée, observons un petit schéma pour mieux nous représenter le parcours de la fonction :

Descente

n == 6 result = 1
n == 5 result = 6
n == 4 result = 30
n == 3 result = 120
n == 2 result = 360
n == 1 result = 720

Condition de sortie: n == 1
Retour de la fonction = 720

Le schéma nous prouve bel et bien qu'il n'y a qu'une phase de descente et pas de remontée. De cette manière nous économisons l'utilisation de la pile du programme, et nous gagnons également du temps en exécution.

Dépassement de capacité

- Une des causes assez fréquentes quand vous travaillez sur de très grands nombres est le dépassement de capacité. C'est un phénomène qui se produit lorsque vous essayez de stocker un nombre plus grand que ce que peut contenir le type de votre variable.
- Il est d'usage de choisir un type approprié, même si vous êtes certains que le type que vous avez choisi ne sera jamais dépassé, utilisez tant que possible une variable pouvant contenir de plus grandes données. Ceci s'applique à tous types de données.
- Évitez le type **int** (**utiliser le type long**) si vous travaillez avec une fonction récursive, comme les exemples précédents pour le calcul de factorielles. Ce type est très petit et dans une fonction récursive il peut très vite arriver de le dépasser
➔ plantage.

Débordement de pile (Stack Overflow)

- Parmi les causes les plus rencontrées dans le plantage de programmes récurifs. Nous savons que les appels récurifs de fonctions sont placés dans la pile du programme, la pile a une taille assez limitée, car elle est fixée une fois pour toutes lors de la compilation du programme.
- Dans la pile sont non seulement stockées les valeurs des variables de retour, mais aussi les adresses des fonctions entre autres choses, les données sont nombreuses et un débordement de la pile peut très vite arriver ce qui provoque une sortie anormale du programme.

Stack Overflow

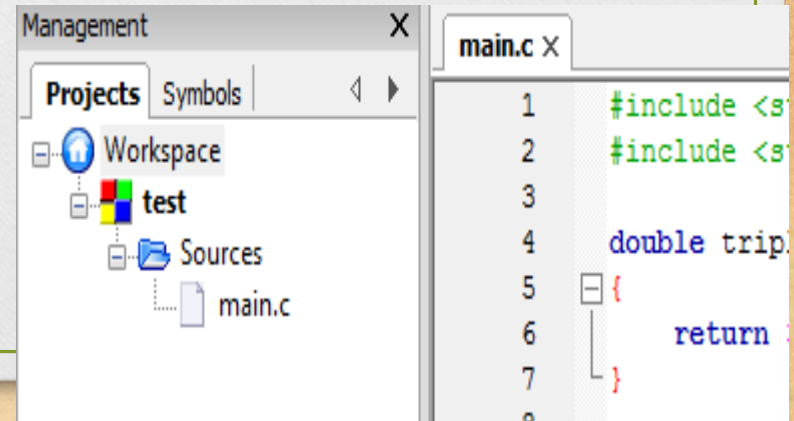
- Dans l'exemple de la fonction **factoriel**, il nous faut (en arrondissant) environ 135 000 appels récurifs pour faire exploser la pile.
- Une autre méthode existe cependant ! Si vous êtes presque sûr de dépasser ce genre de limites, préférez alors une approche itérative plutôt qu'une approche récursive du problème. Une approche récursive demande beaucoup de moyens en ressources, car énormément de données doivent être stockées dans la pile d'exécution alors qu'en revanche, une approche itérative telle une boucle **for** est bien moins coûteuse en termes de ressources et est bien plus sûre, sauf dans le cas d'un dépassement de capacité bien sûr !

programmation modulaire

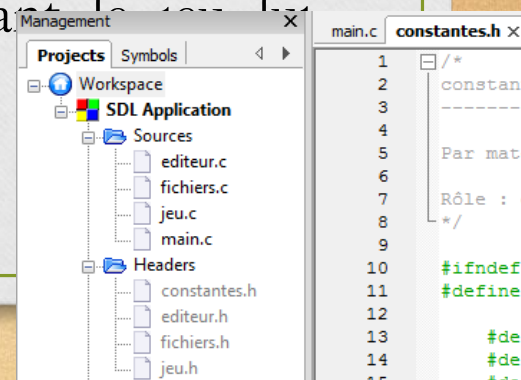
- Jusqu'ici nous n'avons travaillé que dans un seul fichier appelé `main.c`. Pour le moment c'était acceptable car nos programmes étaient tout petits, mais ils vont bientôt être composés de dizaines, que dis-je de centaines de fonctions, et si vous les mettez toutes dans un même fichier celui-là va finir par devenir très long !
- C'est pour cela que l'on a inventé ce qu'on appelle la programmation modulaire. Le principe est simple: plutôt que de placer tout le code de notre programme dans un seul fichier (`main.c`), nous le « séparons » en plusieurs petits fichiers.

Les prototypes

- Jusqu'ici, nous n'avions qu'un seul fichier source dans notre projet. Ce fichier source est main.c.
- Plusieurs fichiers par projet
- Dans la pratique, vos programmes ne seront pas tous écrits dans ce même fichier main.c. Bien sûr, il est possible de le faire, mais ce n'est jamais très pratique de se balader dans un fichier de 10 000 lignes. C'est pour cela qu'en général on crée plusieurs fichiers par projet.
- Un projet, c'est l'ensemble des fichiers source de votre programme. Regardez dans votre IDE, généralement c'est à gauche (fig. suivante).



- Comme dans l'image, il y a plusieurs fichiers: vous verrez plusieurs fichiers dans la colonne de gauche. Vous reconnaissez dans la liste le fichier `main.c` : c'est celui qui contient la fonction `main`. En général dans mes programmes, je ne mets que le `main` dans `main.c`. Pour information, ce n'est pas du tout une obligation, chacun s'organise comme il veut. Pour bien me suivre, je vous conseille néanmoins de faire comme moi.
- En général, on regroupe dans un même fichier des fonctions ayant le même thème. Ainsi, dans le fichier `editeur.c` j'ai regroupé toutes les fonctions concernant l'éditeur de niveau ; dans le fichier `jeu.c`, j'ai regroupé toutes les fonctions concernant le jeu lui-même, etc.



Fichiers.h et .c

- Il y a deux types de fichiers différents
 - Les.h, appelés fichiers headers. Ces fichiers contiennent les prototypes des fonctions.
 - Les.c : les fichiers source. Ces fichiers contiennent les fonctions elles-mêmes.
- En général, on met donc rarement les prototypes dans les fichiers.c comme on l'a fait tout à l'heure dans le main.c (sauf si votre programme est tout petit).
- Pour chaque fichier.c, il y a son équivalent.h qui contient les prototypes des fonctions.
- il y a editeur.c (le code des fonctions) et editeur.h (les prototypes des fonctions) ;
- il y a jeu.c et jeu.h;
- etc.

Fichiers.h et .c

- Il faut inclure le fichier.h grâce à une directive de préprocesseur.

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

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

```
#include "jeu.h"
```

```
void jouer(SDL_Surface* ecran)
```

```
{
```

```
// ...
```

- Notez une différence : les fichiers que vous avez créés et placés dans le répertoire de votre projet doivent être inclus avec des guillemets ("jeu.h") tandis que les fichiers correspondant aux bibliothèques (qui sont généralement installés, eux, dans le répertoire de votre IDE) sont inclus entre chevrons (<stdio.h>).

-
- ❑ La raison est en fait assez simple. Quand dans votre code vous faites appel à une fonction, votre ordinateur doit déjà la connaître, savoir combien de paramètres elle prend, etc. C'est à ça que sert un prototype : c'est le mode d'emploi de la fonction pour l'ordinateur.
 - ❑ Tout est une question d'ordre : si vous placez vos prototypes dans des.h(headers) inclus en haut des fichiers.c, votre ordinateur connaîtra le mode d'emploi de toutes vos fonctions dès le début de la lecture du fichier.
 - ❑ En faisant cela, vous n'aurez ainsi pas à vous soucier de l'ordre dans lequel les fonctions se trouvent dans vos fichiers.c. Si maintenant vous faites un petit programme contenant deux ou trois fonctions, vous vous rendrez peut-être compte que les prototypes semblent facultatifs (ça marche sans). Mais ça ne durera pas longtemps ! Dès que vous aurez un peu plus de fonctions, si vous ne mettez pas vos prototypes de fonctions dans des.h, la compilation échouera sans aucun doute.

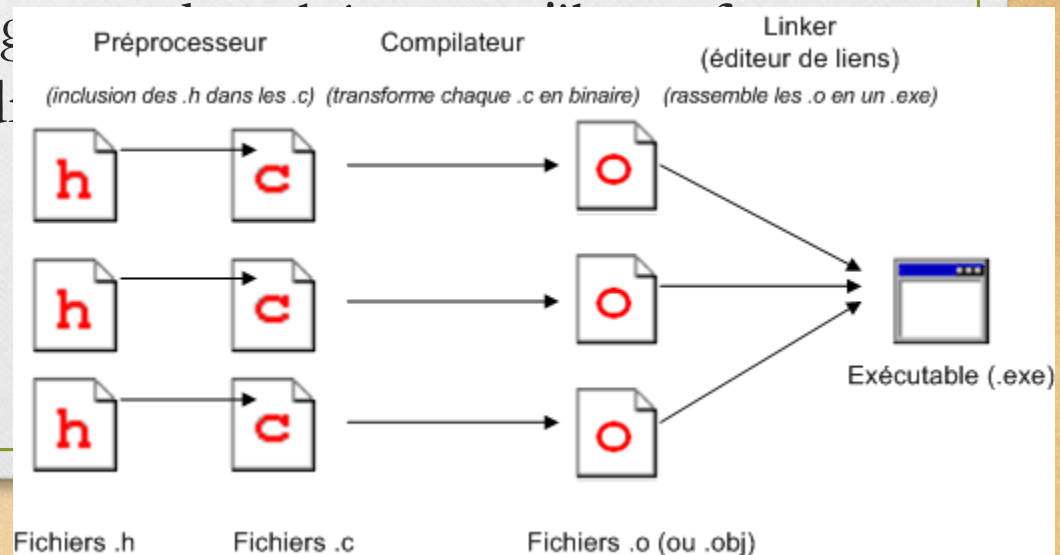
-
- Lorsque vous appellerez une fonction située dans fonctions.c depuis le fichier main.c, vous aurez besoin d'inclure les prototypes de fonctions.c dans main.c. Il faudra donc mettre un `#include "fonctions.h"` en haut de main.c.
 - Souvenez-vous de cette règle : à chaque fois que vous faites appel à une fonction X dans un fichier, il faut que vous ayez inclus les prototypes de cette fonction dans votre fichier. Cela permet au compilateur de vérifier si vous l'avez correctement appelée

Les include des bibliothèques standard

- Ils sont normalement installés là où se trouve votre IDE. Dans notre cas sous Code::Blocks, ils sont dans cet emplacement :
 - C:\Program Files\CodeBlocks\MinGW\include
- Il faut généralement chercher un dossier include.
- Là-dedans, vous allez trouver de très nombreux fichiers. Ce sont des headers (.h) des bibliothèques standard, c'est-à-dire des bibliothèques disponibles partout (que ce soit sous Windows, Mac, Linux...). Vous y retrouverez donc `stdio.h` et `stdlib.h`, entre autres.

La compilation séparée

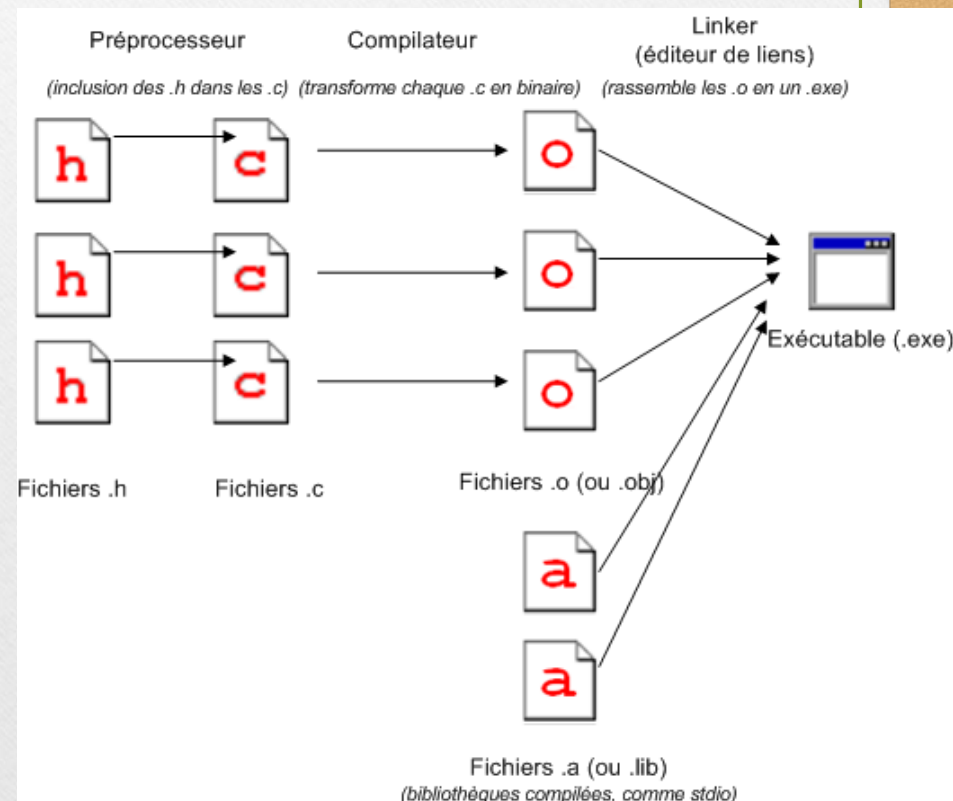
- Maintenant que vous savez qu'un projet est composé de plusieurs fichiers source, nous pouvons rentrer plus en détail dans le fonctionnement de la compilation. Jusqu'ici, nous avons vu un schéma très simplifié.
- La figure suivante est un schéma bien plus précis de la compilation. C'est le guide conseillé de comprendre



- Il s'agit d'un vrai schéma de ce qu'il se passe à la compilation. Détaillons-le.
- Préprocesseur : le préprocesseur est un programme qui démarre avant la compilation. Son rôle est d'exécuter les instructions spéciales qu'on lui a données dans des directives de préprocesseur ces fameuses lignes qui commencent par un #.
- Pour l'instant, la seule directive de préprocesseur que l'on connaît est `#include`, qui permet d'inclure un fichier dans un autre. Le préprocesseur sait faire d'autres choses, mais ça, nous le verrons plus tard. Le `#include` est quand même ce qu'il y a de plus important à connaître. Le préprocesseur « remplace » donc les lignes `#include` par le fichier indiqué. Il met à l'intérieur de chaque fichier.c le contenu des fichiers.h qu'on a demandé d'inclure. À ce moment-là de la compilation, votre fichier.c est complet et contient tous les prototypes des fonctions que vous utilisez (votre fichier.c est donc un peu plus gros que la normale)

-
- Compilation : cette étape très importante consiste à transformer vos fichiers source en code binaire compréhensible par l'ordinateur. Le compilateur compile chaque fichier.c un à un. Il compile tous les fichiers source de votre projet, d'où l'importance d'avoir bien ajouté tous vos fichiers au projet (ils doivent tous apparaître dans la fameuse liste à gauche).

- Édition de liens : le linker (ou « éditeur de liens » en français) est un programme dont le rôle est d'assembler les fichiers binaires.o. Il les assemble en un seul gros fichier : l'exécutable final ! Cet exécutable a l'extension.exe sous Windows. Si vous êtes sous un autre OS, il devrait prendre l'extension adéquate.



Les fichiers

- Le défaut avec les variables, c'est qu'elles n'existent que dans la mémoire vive. Une fois votre programme arrêté, toutes vos variables sont supprimées de la mémoire et il n'est pas possible de retrouver ensuite leur valeur. Comment peut-on, dans ce cas-là, enregistrer les meilleurs scores obtenus à son jeu ? Comment peut-on faire un éditeur de texte si tout le texte écrit disparaît lorsqu'on arrête le programme ?
- La solution consiste à lire et écrire dans des fichiers en langage C. Ces fichiers seront écrits sur le disque dur de votre ordinateur : l'avantage est donc qu'ils restent là, même si vous arrêtez le programme ou l'ordinateur.

Ouvrir et fermer un fichier

- Pour lire et écrire dans des fichiers, nous allons nous servir de fonctions situées dans la bibliothèque stdio que nous avons déjà utilisée.
- Oui, cette bibliothèque-là contient aussi les fonctions printf et scanf que nous connaissons bien ! Mais elle ne contient pas que ça : il y a aussi d'autres fonctions, notamment des fonctions faites pour travailler sur des fichiers.
- Assurez-vous donc, pour commencer, que vous incluez bien au moins les bibliothèques stdio.h et stdlib.h en haut de votre fichier.c:

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

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

- Pour ouvrir un fichier, que ce soit pour le lire ou pour y écrire.
 - On appelle la fonction d'ouverture de fichier `fopen` qui nous renvoie un pointeur sur le fichier.
 - On vérifie si l'ouverture a réussi (c'est-à-dire si le fichier existait) en testant la valeur du pointeur qu'on a reçu. Si le pointeur vaut `NULL`, c'est que l'ouverture du fichier n'a pas fonctionné, dans ce cas on ne peut pas continuer (il faut afficher un message d'erreur).
 - Si l'ouverture a fonctionné (si le pointeur est différent de `NULL`), alors on peut lire et écrire dans le fichier à travers des fonctions que nous verrons un peu plus loin.
 - Une fois qu'on a terminé de travailler sur le fichier, il faut penser à le « fermer » avec la fonction `fclose`.
- prototype de la fonction `fopen`:

`FILE* fopen(const char* nomDuFichier, const char* modeOuverture);`

- Cette fonction attend deux paramètres :
 - le nom du fichier à ouvrir ;
 - le mode d'ouverture du fichier, c'est-à-dire une indication qui ~~mentionne ce que vous voulez faire : seulement écrire dans le fichier, seulement le lire, ou les deux à la fois.~~
- Cette fonction renvoie un pointeur sur FILE! C'est un pointeur sur une structure de type FILE. Cette structure est définie dans stdio.h.
- Fopen renvoie un FILE*. Il est extrêmement important de récupérer ce pointeur pour pouvoir ensuite lire et écrire dans le fichier.
- Nous allons donc créer un pointeur de FILE au début de notre fonction (par exemple la fonction main)

```
int main(int argc, char *argv[])  
{  
    FILE* fichier = NULL;  
    return 0;  
}
```

- Notez que la forme de la structure peut changer d'un OS à l'autre (elle ne contient pas forcément les mêmes sous-variables partout). Pour cette raison, on ne modifiera jamais le contenu d'un FILE directement (on ne fera pas `fichier.element` par exemple). On passera par des fonctions qui manipulent le FILE à notre place.
- Maintenant, nous allons appeler la fonction `fopen` et récupérer la valeur qu'elle renvoie dans le pointeur fichier. Mais avant ça, il faut savoir comment se servir du second paramètre, le paramètre `modeOuverture`. En effet, il y a un code à envoyer qui indiquera à l'ordinateur si vous ouvrez le fichier en mode lecture seule, d'écriture seule, ou les deux à la fois.

- Voici les modes d'ouverture possibles.
 - "r": lecture seule. Vous pourrez lire le contenu du fichier, mais pas y écrire. Le fichier doit avoir été créé au préalable.
 - "w": écriture seule. Vous pourrez écrire dans le fichier, mais pas lire son contenu. Si le fichier n'existe pas, il sera créé.
 - "a": mode d'ajout. Vous écrirez dans le fichier, en partant de la fin du fichier. Vous ajouterez donc du texte à la fin du fichier. Si le fichier n'existe pas, il sera créé.
 - "r+": lecture et écriture. Vous pourrez lire et écrire dans le fichier. Le fichier doit avoir été créé au préalable.
 - "w+": lecture et écriture, avec suppression du contenu au préalable. Le fichier est donc d'abord vidé de son contenu, vous pouvez y écrire, et le lire ensuite. Si le fichier n'existe pas, il sera créé.
 - "a+": ajout en lecture / écriture à la fin. Vous écrivez et lisez du texte à partir de la fin du fichier. Si le fichier n'existe pas, il sera créé.

Ces modes d'accès ont pour particularités :

- ❑ Si le mode contient la lettre `r`, le fichier doit exister.
- ❑ Si le mode contient la lettre `w`, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, son ancien contenu sera perdu.
- ❑ Si le mode contient la lettre `a`, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier précédent.

Trois flots standard peuvent être utilisés en `C` sans qu'il soit nécessaire de les ouvrir ou de les fermer :

- ❑ `stdin` (standard input) : unité d'entrée (par défaut, le clavier) ;
- ❑ `stdout` (standard output) : unité de sortie (par défaut, l'écran) ;
- ❑ `stderr` (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran).

Il est fortement conseillé d'afficher systématiquement les messages d'erreur sur `stderr` afin que ces messages apparaissent à l'écran même lorsque la sortie standard est redirigée.

- Si vous avez juste l'intention de lire un fichier, il est conseillé de mettre "r". Certes, le mode "r+" aurait fonctionné lui aussi, mais en mettant "r" vous vous assurez que le fichier ne pourra pas être modifié, ce qui est en quelque sorte une sécurité.
- Si vous écrivez une fonction chargerNiveau (pour charger le niveau d'un jeu, par exemple), le mode "r" suffit. Si vous écrivez une fonction enregistrerNiveau, le mode "w" sera alors adapté.
- Le code suivant ouvre le fichier test.txt en mode "r+" (lecture / écriture) :

```
int main(int argc, char *argv[])  
{  
    FILE* fichier = NULL;  
    fichier = fopen("test.txt", "r+");  
    return 0;  
}
```

- Le pointeur fichier devient alors un pointeur sur test.txt.

- ~~Tester l'ouverture du fichier~~
- Le pointeur fichier devrait contenir l'adresse de la structure de type FILE qui sert de descripteur de fichier. Celui-ci a été chargé en mémoire pour vous par la fonction fopen().
 - soit l'ouverture a réussi, et vous pouvez continuer (c'est-à-dire commencer à lire et écrire dans le fichier) ;
 - soit l'ouverture a échoué parce que le fichier n'existait pas ou était utilisé par un autre programme. Dans ce cas, vous devez arrêter de travailler sur le fichier.
 - Juste après l'ouverture du fichier, il faut impérativement vérifier si l'ouverture a réussi ou non. Pour ce faire, c'est très simple : si le pointeur vaut NULL, l'ouverture a échoué. S'il vaut autre chose que NULL, l'ouverture a réussi.
 - On va donc suivre systématiquement le schéma suivant


```
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    fichier = fopen("test.txt", "r+");
    if (fichier != NULL)
    {
        // On peut lire et écrire dans le fichier
    }
    else
    {
        // On affiche un message d'erreur si on veut
        printf("Impossible d'ouvrir le fichier test.txt");
    }

    return 0; }
```

□ Une fois que vous aurez fini de travailler avec le fichier, il faudra le « fermer ». On utilise pour cela la fonction **fclose** qui a pour rôle de libérer la mémoire, c'est-à-dire supprimer votre fichier chargé dans la mémoire vive.

□ Son prototype est :

□ **int fclose(FILE* pointeurSurFichier);**

□ Elle renvoie un int qui indique si elle a réussi à fermer le fichier.
Cet int vaut :

0 : si la fermeture a marché ;

EOF: si la fermeture a échoué .EOF est un define situé dans stdio.h qui correspond à un nombre spécial, utilisé pour dire soit qu'il y a eu une erreur, soit que nous sommes arrivés à la fin du fichier. Dans le cas présent
Pour fermer le fichier, on va donc écrire :

fclose(fichier);

Différentes méthodes de lecture / écriture

1. Ecriture

- ❑ Maintenant que nous avons écrit le code qui ouvre et ferme le fichier, nous n'avons plus qu'à insérer le code qui le lit et y écrit.
- ❑ Écrire dans le fichier
 - ❖ Il existe plusieurs fonctions capables d'écrire dans un fichier. Ce sera à vous de choisir celle qui est la plus adaptée à votre cas.
 - ❖ **Voici les trois fonctions que nous allons étudier :**
 - fputc: écrit un caractère dans le fichier (UN SEUL caractère à la fois) ;
 - fputs: écrit une chaîne dans le fichier ;
 - fprintf: écrit une chaîne « formatée » dans le fichier, fonctionnement quasi-identique à printf.

fputc

□ Prototype:

`int fputc(int caractere, FILE* pointeurSurFichier);`

□ Elle prend deux paramètres.

- Le caractère à écrire (de type int, ce qui revient plus ou moins à utiliser un char, sauf que le nombre de caractères utilisables est ici plus grand). Vous pouvez donc écrire directement 'A' par exemple.
- Le pointeur sur le fichier dans lequel écrire. Dans notre exemple, notre pointeur s'appelle fichier. L'avantage de demander le pointeur de fichier à chaque fois, c'est que vous pouvez ouvrir plusieurs fichiers en même temps et donc lire et écrire dans chacun de ces fichiers. Vous n'êtes pas limités à un seul fichier ouvert à la fois.
- La fonction retourne un int, c'est un code d'erreur. Cet int vaut EOF si l'écriture a échoué, sinon il a une autre valeur.


```
int main(int argc, char *argv[])  
{  
    FILE* fichier = NULL;  
    fichier = fopen("test.txt", "w");  
    if (fichier != NULL)  
    {  
        fputc('A', fichier); // Écriture du caractère A  
        fclose(fichier);  
    }  
    return 0;  
}
```

fputs

- Cette fonction est très similaire à fputc, à la différence près qu'elle écrit tout une chaîne, ce qui est en général plus pratique que d'écrire caractère par caractère.
- fputc reste utile lorsque vous devez écrire caractère par caractère
- Prototype de la fonction :

char* fputs(const char* chaine, FILE* pointeurSurFichier);

- chaine: la chaîne à écrire. Notez que le type ici est const char*: en ajoutant le mot const dans le prototype, la fonction indique que pour elle la chaîne sera considérée comme une constante. En un mot comme en cent : elle s'interdit de modifier le contenu de votre chaîne. C'est logique quand on y pense : fputs doit juste lire votre chaîne, pas la modifier. C'est donc pour vous une information (et une sécurité) comme quoi votre chaîne ne subira pas de modification.
- Pointeur Sur Fichier: comme pour fputc, il s'agit de votre pointeur de type FILE* sur le fichier que vous avez ouvert.

La fonction renvoie EOF s'il y a eu une erreur, sinon c'est que cela a fonctionné.

Là non plus

```
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    fichier = fopen("test.txt", "w");
    if (fichier != NULL)
    {
        fputs(" Notre premier programme sur les fichiers\n vous avez
compris ?", fichier);
        fclose(fichier);
    }
    return 0;
}
```

fprintf

- ❑ fprintf peut être utilisée pour écrire dans un fichier. Elle s'utilise de la même manière que printf d'ailleurs, excepté le fait que vous devez indiquer un pointeur de FILE en premier paramètre.

```
int main(int argc, char *argv[])  
{  
    FILE* fichier = NULL;  
    int age = 0;  
    fichier = fopen("test.txt", "w");  
    if (fichier != NULL)  
    {  
        // On demande l'âge  
        printf("Quel age avez-vous ? ");  
        scanf("%d", &age);  
        // On l'écrit dans le fichier  
        fprintf(fichier, " votre age est : %d ans", age);  
        fclose(fichier);  
    }  
    return 0;  
}
```


2. Lire dans un fichier

- ❑ Nous pouvons utiliser quasiment les mêmes fonctions que pour l'écriture, le nom change juste un petit peu :
 - `fgetc`: lit un caractère ;
 - `fgets`: lit une chaîne ;
 - `fscanf`: lit une chaîne formatée.
- ❑ `fgetc`
 - ❑ Tout d'abord le prototype :
`int fgetc(FILE* pointeurDeFichier);`
 - ❑ Cette fonction retourne un `int` : c'est le caractère qui a été lu.
 - ❑ Si la fonction n'a pas pu lire de caractère, elle retourne `EOF`

```
int main(int argc, char *argv[])
```

```
{
```

```
    FILE* fichier = NULL;
```

```
    int c = 0;
```

```
    fichier = fopen("test.txt", "r");
```

```
    if (fichier != NULL)
```

```
    {
```

```
        // Boucle de lecture des caractères un à un à partir d'un fichier
```

```
        do
```

```
        {
```

```
            c = fgetc(fichier); // On lit le caractère
```

```
            printf("%c", c); // On l'affiche
```

```
        } while (c != EOF); // On continue tant que fgetc n'a pas retourné  
                        // EOF (fin de fichier)
```

```
        fclose(fichier);
```

```
    }
```

```
    return 0;
```

```
}
```


fgets

- ❑ Cette fonction lit une chaîne dans le fichier. Ça vous évite d'avoir à lire tous les caractères un par un. **La fonction lit au maximum une ligne** (elle s'arrête au premier `\n` qu'elle rencontre). Si vous voulez lire plusieurs lignes, il faudra faire une boucle.
- ❑ Voici le prototype de fgets:

```
char* fgets(char* chaine, int nbreDeCaracteresALire, FILE* pointeurSurFichier);
```
- ❑ Cette fonction demande un paramètre un peu particulier, qui va en fait s'avérer très pratique : le `nbrdecaracteresàlire`. Cela demande à la fonction fgets de s'arrêter de lire la ligne si elle contient plus de X caractères.
- ❑ `int getw(FILE *fichier);` idem avec un entier; le pointeur avance de la taille d'un entier.
 - ❑ Exemple: `int n ; n = getw(fichier) ;`

```
#define TAILLE_MAX 1000 // Tableau de taille 1000
```

```
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    char chaine[TAILLE_MAX] = "" // Chaîne vide de taille TAILLE_MAX
    fichier = fopen("test.txt", "r");
    if (fichier != NULL)
    {
        fgets(chaine, TAILLE_MAX, fichier); // On lit maximum TAILLE_MAX caractères
                                           // du fichier, on stocke le tout dans "chaine"
        printf("%s", chaine);              // On affiche la chaîne
        fclose(fichier);
    }
    return 0; }
```



```
#define TAILLE_MAX 1000
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    FILE* fichier = NULL;
```

```
    char chaine[TAILLE_MAX] = "";
```

```
    fichier = fopen("test.txt", "r");
```

```
    if (fichier != NULL)
```

```
    {
```

```
        while (fgets(chaine, TAILLE_MAX, fichier) != NULL) // On lit le fichier tant qu'on ne  
                                                             // reçoit pas d'erreur (NULL)
```

```
        {
```

```
            printf("%s", chaine); // On affiche la chaîne qu'on vient de lire
```

```
        }
```

```
        fclose(fichier);
```

```
    }
```

```
    return 0;}
```

```
while (fgets(chaine, TAILLE_MAX, fichier) != NULL)
```

-
- La ligne du while fait deux choses : elle lit une ligne dans le fichier et vérifie si fgets ne renvoie pas NULL. Elle peut donc se traduire comme ceci : « Lire une ligne du fichier tant que nous ne sommes pas arrivés à la fin du fichier ».

fscanf

-
- ☐ C'est le même principe que la fonction scanf.
 - ☐ Cette fonction lit dans un fichier qui doit avoir été écrit d'une manière précise.
 - ☐ Supposons que votre fichier contienne trois nombres séparés par un espace, qui sont par exemple les trois plus hauts scores obtenus à votre jeu :15 20 30.
 - ☐ Vous voudriez récupérer chacun de ces nombres dans une variable de type int.
 - ☐ La fonction fscanf va vous permettre de faire ça rapidement
 - ☐ Si aucune donnée ne peut être extraite, alors la valeur EOF vous sera retournée: si non, le nombre de paramètres correctement extraits vous sera renvoyé. Il est possible que seul une sous-partie des paramètres est pu être extraits (forcément les n premiers)

```
int main(int argc, char *argv[])  
{  
    FILE* fichier = NULL;  
    int score[3] = {0}; // Tableau des 3 meilleurs scores  
    fichier = fopen("test.txt", "r");  
    if (fichier != NULL)  
    {  
        fscanf(fichier, "%d %d %d", &score[0], &score[1], &score[2]);  
        printf("Les meilleurs scores sont : %d, %d et %d", score[0], score[1],  
score[2]);  
        fclose(fichier);  
    }  
    return 0; }
```


3. Se déplacer dans un fichier

- ❑ Chaque fois que vous ouvrez un fichier, il existe en effet un curseur qui indique votre position dans le fichier. Vous pouvez imaginer que c'est exactement comme le curseur de votre éditeur de texte (tel Bloc-Notes). Il indique où vous êtes dans le fichier, et donc où vous allez écrire.
- ❑ En résumé, le système de curseur vous permet d'aller lire et écrire à une position précise dans le fichier.
- ❑ Il existe trois fonctions à connaître :
 - ✓ `ftell`: indique à quelle position vous êtes actuellement dans le fichier ;
 - ✓ `fseek`: positionne le curseur à un endroit précis ;
 - ✓ `rewind`: remet le curseur au début du fichier (c'est équivalent à demander à la fonction `fseek` de positionner le curseur au début).

ftell

ftell position dans le fichier

- Cette fonction renvoie la position actuelle du curseur sous la forme d'un long:
`long ftell(FILE* pointeurSurFichier);`
- Le nombre renvoyé indique donc la position du curseur dans le fichier.

fseek: se positionner dans le fichier

- Le prototype de fseek est le suivant :
`int fseek(FILE* pointeurSurFichier, long déplacement, int origine);`

La fonction fseek permet de déplacer le curseur d'un certain nombre de caractères (indiqué par déplacement) à partir de la position indiquée par origine.

Le nombre de placement peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).

Quant au nombre origine, vous pouvez mettre comme valeur l'une des trois constantes (généralement des define) listées ci-dessous :

SEEK_SET: indique le début du fichier ;

SEEK_CUR: indique la position actuelle du curseur ;

SEEK_END: indique la fin du fichier.

Exemples:

Le code suivant place le curseur deux caractères *après* le début :

```
fseek(fichier, 2, SEEK_SET);
```

Le code suivant place le curseur quatre caractères *avant* la position courante :

```
fseek(fichier, -4, SEEK_CUR);
```

Le code suivant place le curseur à la fin du fichier :

```
fseek(fichier, 0, SEEK_END);
```

rewind: retour au début

- Cette fonction est équivalente à utiliser fseek pour nous renvoyer à la position 0 dans le fichier.

void rewind(FILE* pointeurSurFichier);

Renommer et supprimer un fichier

- ❑ rename: renomme un fichier ;
- ❑ remove: supprime un fichier.
- ❑ La particularité de ces fonctions est qu'elles ne nécessitent pas de pointeur de fichier pour fonctionner. Il suffira simplement d'indiquer le nom du fichier à renommer ou supprimer.
- ❑ Voici le prototype de la fonction rename :

```
int rename(const char* ancienNom, const char* nouveauNom);
```
- ❑ La fonction renvoie 0 si elle a réussi à renommer, sinon elle renvoie une valeur différente de 0.

Remove : supprimer un fichier

- Cette fonction supprime un fichier sans demander son reste :

```
int remove(const char* fichierASupprimer);
```

```
int main(int argc, char *argv[])
{
    rename("test.txt", "test_renomme.txt");
    return 0;}
int main(int argc, char *argv[])
{
    remove("test.txt");
    return 0;
}
```


Les fonctions locales à un fichier

- ❑ Quand on crée une fonction, celle-ci est globale à tout le programme. Elle est accessible depuis n'importe quel autre fichier.c.
- ❑ Il se peut que vous ayez besoin de créer des fonctions qui ne seront accessibles que dans le fichier dans lequel se trouve la fonction.
- ❑ Pour faire cela, rajoutez le mot-clé **static** devant la fonction :
 - ❑ `static int triple(int nombre)`
 - ❑ `{`
 - ❑ `// Instructions`
 - ❑ `}`
- ❑ Pensez à mettre à jour le prototype aussi :

`static int triple(int nombre);`

- ❑ Maintenant, notre fonction static triple ne peut être appelée que depuis une autre fonction du même fichier. Si vous essayez d'appeler la fonction triple depuis une fonction d'un autre fichier, ça ne marchera pas car triple n'y sera pas accessible.
- ❑ Résumons tous les types de portée qui peuvent exister pour les variables :
 - ✓ Une variable déclarée dans une fonction est supprimée à la fin de la fonction, elle n'est accessible que dans cette fonction.
 - ✓ Une variable déclarée dans une fonction avec le mot-clé static n'est pas supprimée à la fin de la fonction, elle conserve sa valeur au fur et à mesure de l'exécution du programme.
 - ✓ Une variable déclarée en dehors des fonctions est une variable globale, accessible depuis toutes les fonctions de tous les fichiers source du projet.
 - ✓ Une variable globale avec le mot-clé static est globale uniquement dans le fichier dans lequel elle se trouve, elle n'est pas accessible depuis les fonctions des autres fichiers.

Les structures

- Une structure est un assemblage de variables qui peuvent avoir différents types. Contrairement aux tableaux qui vous obligent à utiliser le même type dans tout le tableau, vous pouvez créer une structure comportant des variables de types long, char, int et double à la fois.
- Les structures sont généralement définies dans les fichiers.h, au même titre donc que les prototypes et les define. Voici un exemple de structure :

```
struct Nom_Structure
{
    type1 champ1;
    type2 champ2;
    type3 champ3;
    ....
};
```

- **Les structures de données**

- **intérêt**

- Rassembler des données hétérogènes caractérisant une entité pour en faire un type utilisateur.

- **exemple:**

- point dans l'espace → 3 entiers
 - nœud d'un arbre binaire → 2 adresses vers les fils, 3 entiers pour les données du nœud
 - noms et références des pièces mécaniques constituant une voiture → 600 tableaux de caractères + 600 entiers pour la référence

Structures

- **Les structures de données**
 - **Déclaration**
 - Il y a plusieurs méthodes possibles pour déclarer des structures.

```
struct nom_structure
{
    type1 nomchamps1;
    type2 nomchamps2;
    .
    .
    typeN nomchampsN;
};
```

- On peut déclarer des variables de type structure sans utiliser d'étiquette de structure, par exemple :

```
struct
{
    char nom[20];
    char prenom[20];
    int no_employe;
} p1,p2;
```

- déclare deux variables de noms p1 et p2 comme étant deux structures de trois membres, mais elle ne donne pas de nom au type de la structure. L'inconvénient de cette méthode est qu'il sera par la suite impossible de déclarer une autre variable du même type.

- On peut combiner déclaration d'étiquette de structure et déclaration de variables, comme ceci :

```
struct personne  
{  
    char nom[20];  
    char prenom[20];  
    int no_employe;  
} p1,p2;
```

- déclare les deux variables p1 et p2 et donne le nom personne à la structure. Là aussi, on pourra utiliser ultérieurement le nom struct personne pour déclarer d'autres variables :
- struct personne pers1, pers2, pers3;
- qui seront du même type que p1 et p2.
- **De ces trois méthodes c'est la première qui est recommandée, car elle permet de bien séparer la définition du type structure de ses utilisations.**

Initialisation d'une structure

- Là aussi il y'a plusieurs méthodes pour initialiser une structure parmi autres:
 1. `struct personne p = {"Jean", "Dupond", 7845};`
 2. `P.nom = "Jean";`

Structures

- **Les structures de données**
 - **exemple et utilisation**

```
struct pt
{
    int x;
    int y;
    int z;
    char nom;
};
```

```
main()
{
    struct pt p;
    printf("donnez la valeur du membre x");
    scanf("%d", &p.x);
}
```

Structures

- simplification de l'écriture

```
struct pt
{
    int x;
    int y;
    int z;
    char nom;
};
typedef struct pt point;

main()
{
    point p;
}
```

Notations
équivalentes

```
typedef struct pt
{
    int x;
    int y;
    int z;
    char nom;
} point;

main()
{
    point p;
}
```


-
- Exemple

Structures

- accès aux données

- Deux cas de figure
 - On dispose du nom de la variable
 - Accès par: NomVariable.NomChamps
 - exemple

```
main()
{
    point p1;
    p1.x = 8; p1.y = 9; p1.z = 10;
    printf("%d %d %d \n",p1.x, p1.y, p1.z);
}
```

Structures

- accès aux données

- Deux cas de figure
 - On dispose de l'adresse de la variable (pointeur)
 - Accès par: NomVariable->NomChamps
 - exemple

```
main()
{
    point *pp1, p1;
    pp1 = &p1;
    pp1->x = 8; pp1->y = 9; pp1->z = 10;
    printf("%d %d %d \n", pp1->x, pp1->y, pp1->z);
}
```


Exercice

- Ecrire un programme qui initialise et affiche un tableaux de dix points

Point(x,y,z)

Exemple

Etablissons une fiche pour chacun des items d'un répertoire. Simplifions le contenu de cette fiche; Disons qu'une fiche contiendra un nom et un numéro de téléphone.

- le nom sera un mot de longueur inférieure à 29 ;
- le numéro de téléphone sera un mot de longueur inférieure à 19.
- Nous allons demander le nom, puis le numéro de téléphone
- Nous considérerons qu'un nom commençant par '#' est une valeur sentinelle de fin d'initialisation. Notre fichier comprendra au plus 100 fiches.


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

```
struct item  
{  
    char nom[30];  
    char tel[20];  
};  
struct item fichier[100];  
int n;  
char name[30];  
void initialiser(void)  
{  
    n = 0;  
    printf("Nom : ");  
    gets(name);  
    while (name[0] != '#')  
    {  
        strcpy(fichier[n].nom,name);  
        printf("Numero de telephone : ");  
        gets(fichier[n].tel);  
        n++;  
    }
```

```
    printf("Nom : ");
```

```
    gets(name);
```

```
}
```

```
}
```

```
void afficher(void)
```

```
{
```

```
    int i;
```

```
    for (i = 0; i < n; i++)
```

```
        printf("%30s :  
        %20s\n",
```

```
        fichier[i].nom,
```

```
        fichier[i].tel);
```

```
    }
```

```
void main(void)
```

```
{
```

```
    initialiser();
```

```
    afficher();
```

```
}
```

Les structures

- Possibilité de l'affectation globale
 - Dans le langage C (de la norme ANSI seulement) on peut utiliser l'affectation globale du genre :
item1 = item2 ;
 - alors que ceci n'est pas possible dans un langage tel que PASCAL
- Initialisation lors de la déclaration

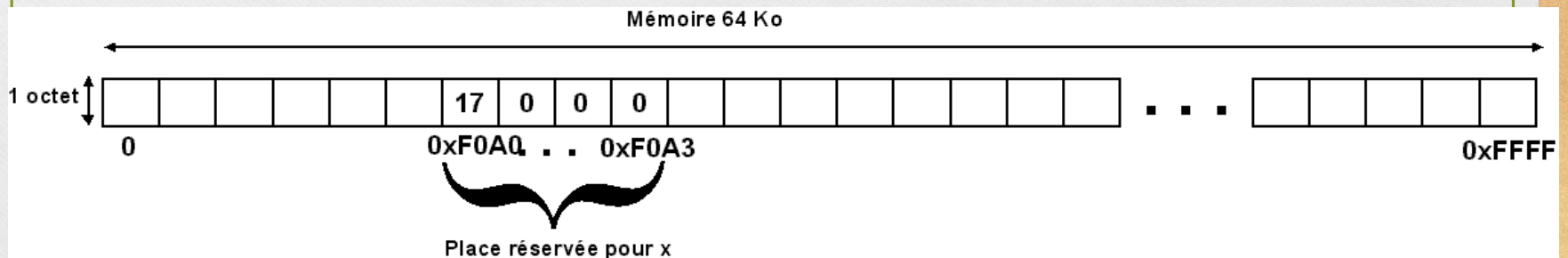
En langage C on peut initialiser une variable structurée lors de sa déclaration, comme pour les tableaux avec des accolades et des virgules. Par exemple:

```
struct personne{  
    char nom [30] ;  
    char prenom [20] ;  
    int age ;};  
struct personne item = {"Dubois", "Paul", 30};
```


Les Pointeurs

- Deux manières d'utiliser une variable
 1. Par son nom → adressage direct
 - compilateur réserve de la mémoire pour la variable
 - exemple:

`int x = 17;`



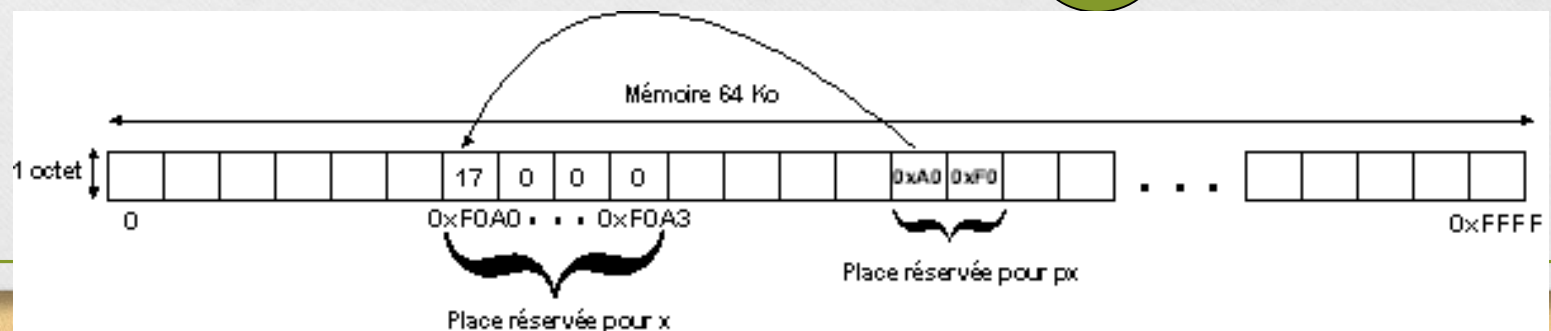
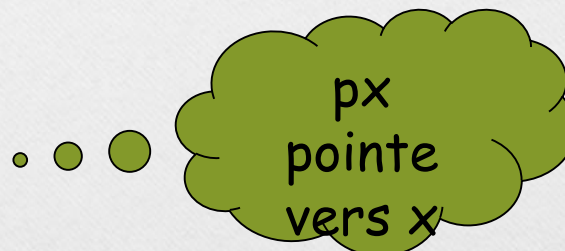
Les Pointeurs

2. Par son adresse → adressage indirect

- une variable de type pointeur contient l'adresse d'une autre variable
- Le lien entre pointeur et la variable pointée est géré par le programmeur
- exemple:

```
int x = 17;
```

```
int * px = &x;
```



Les Pointeurs

- déclaration d'un pointeur
 - `type * nom_du_pointeur`
- exemples
 - `char *buffer;`
 - `int *pf;`
 - `float *ppi;`



C'est au programmeur d'initialiser le pointeur

Les Pointeurs

- initialisation d'un pointeur
 - Il faut lui fournir l'adresse d'une variable existant en mémoire
 - Il doit pointer vers une zone mémoire réservée par le compilateur comme étant une variable
- utilisation de l'opérateur d'adressage **&**

```
int f;  
int *pf,*pg;  
pf = &f;  
  
pg =(int*)malloc(sizeof(int));  
.  
.  
free(pg);
```


Les Pointeurs

- utilisation de l'opérateur d'indirection *
 - travaille la valeur de la variable pointée
 - exemples

```
int i = 8;  
int *pi;  
pi = &i;  
cout << *pi << '\n';
```

8

```
*pi = 19;  
cout << i << '\n';
```

19

Pointeurs et tableaux

- Lien entre le nom d'un tableau à 1 dimension et les pointeurs
 - Nom du tableau = adresse du premier élément du tableau
`nom_tableau[i]` peut s'écrire `*(nom_tableau+i)`
 - exemples:
`char buffer[80];`
`*buffer = 'C';` accès au premier caractère
`*(buffer + 5) = 'V';` accès au 6^{ème} caractère

Exemple

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

```
main()
```

```
{
```

```
char c = „C“;
```

```
int a = 50;
```

```
float f = 3.45;
```

```
printf(“Address of c = %u, Value of c = %c\n”, &c, c);
```

```
printf(“Address of a = %u, Value of a = %d\n”, &a, a);
```

```
printf(“Address of f = %u, Value of f = %f\n”, &f, f);
```

```
}
```

Output:

Address of c = 65529,

Value of c = C

Address of a = 65522,

Value of a = 50

Address of f = 65518,

Value of f = 3.450000

Address

Value

Variable name

65529

67

c

65522

50

a

65518

3.45

f

:

Memory


```
#include<stdio.h>

main()
{
int a = 50, b;
int *p;

p = &a;
b = *p;

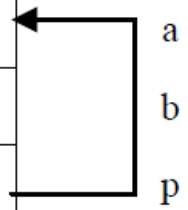
printf("Value of a = %d\n", a);
printf("Address of a = %u\n", &a);
printf("Value of p = %u\n", p);
printf("Value of *p = %d\n", *p);
printf("Value of b = %d\n", b);
printf("Address of b = %u\n", &b);
printf("Address of p = %u\n", &p);
}
```

Output:
Value of a = 50
Address of a = 65524
Value of p = 65524
Value of *p = 50
Value of b = 50
Address of b = 65522
Address of p = 65520

We can get the value of b in the following manner, $b = *p$ that means $b = *(65524)$ [since $p = 65524$] = 50 [value of address 65524]

Address	Value	Variable name
	:	
65524	50	a
65522	50	b
65520	65524	p
	:	

Memory



Type casting of Pointers

The pointer variables have their own data type. Unlike basic data types, the pointers do not support implicit type conversion. Therefore, pointer types can be converted to other pointer types using the explicit type casting mechanism.

- *Example:*

```
int a = 10;
```

```
float b = 12.3;
```

```
int *p;
```

```
float *q;
```

```
p = &a;
```

```
q = &b;
```

```
q = p; /* this is invalid */
```

```
p = q; /* this is invalid */
```

```
q = (float*)p; /* this is valid */
```

```
p = (int*)q; /* this is valid */
```

Null pointer

- ❖ A null pointer is a pointer value that points to no valid location. A null pointer is a constant pointer (often represented by address zero) that is compatible with any pointer. It is not compatible with function pointers. When a pointer is equivalent to `NULL` it is guaranteed not to point to any variable defined within the program.
- ❖ A null pointer value is an address that is different from any valid pointer. Assigning the integer constant `0` to a pointer assigns a null pointer value to it. The mnemonic `NULL` (defined in the standard library header file, `stdio.h`) can be used for legibility.

Example

```
int *p;
```

```
*p = 12;
```

Since p is not defined, therefore p may contain a garbage value (say, p = 65550). Now purposely or accidentally when a value assigning to *p then there are chances that modify that memory location (e.g. 65550) which is not allocated by the program. Therefore, you should use

```
int *p = NULL;
```

Dereferencing a null pointer is meaningless, typically resulting in a run-time error. Therefore, before the use of pointer we should check it with NULL value. All pointers can be successfully tested for equality or inequality to NULL which is logically equivalent to false.

In dynamic memory allocation, when `calloc()` or `malloc()` function fails to allocate memory block then they return NULL. Therefore, after the function call, it requires checking whether the memory block is allocated or not, before the use of the memory.

Void Pointer

A void pointer is a pointer, which may store the address of any type of variable. That means void pointer is a pointer to anything. The void pointer is also known as a type-less pointer or generic pointer.

Note that a variable of type void cannot be declared. However, the return type of a function may

be void.

```
void a;    /* this is invalid */
```

```
void *r;   /* valid */
```

The void pointers are used to store the address of any type of variable temporarily. Since the void pointer is a typeless pointer, the compiler has no information that how many bytes of data it will retrieve from the memory starting from stored address. Therefore, the indirection operator cannot be used with a void pointer.

Example

```
int a = 10;

float b = 12.3;

int *p;

float *q;

void *r;

p = &a;

q = &b;

printf(“%d”, *p);

r = p;

/* this is invalid */

printf(“%d”, *r);
```

```
r = p;
```

```
p = r;
```

```
or
```

```
r = q;
```

```
q = r;
```

where p, q, and r are declared as in the previous example.

The void pointer may be used as a lvalue or as a rvalue. Therefore, by using this concept, void pointer allows violating the basic rule of the type conversion of the pointer.

Generic Functions

The void pointer is used to write generic functions, which can accept any type of parameter. A group of functions that look the same, except the types of one or more of their arguments. A generic function allows defining a function to replace that group of functions.

Suppose we have a function that can be used to interchange the value of two integer type variables. Now we require another function to interchange the value of two floating-point type variables. For interchange the value of two long double type variables, we need one more function.

Therefore, for the different data type, we require to write different function.

Example:

```
/* Interchange the value of two
variables */
#include<stdio.h>
void swapi(int *a, int *b);
void swapf(float *a, float *b);
main()
{
    int i = 10, j = 20;
    float x = 12.3, y = 53.4;
    swapi(&i, &j);
    printf("i = %d, j = %d", i, j);
    swapf(&x, &y);
    printf("x = %f, y = %f", i, j);
}
```

```
/* Function to interchange two integer type
variables*/
void swapi(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
/* Function to interchange two floating-
point type variables*/
void swapf(float *a, float *b)
{
    float temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

→ *Example:*

```
/*Interchange the value of two variable using generic function */
```

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

```
void swap(void *a, void *b, int n);
```

```
main()
```

```
{
```

```
int i = 10, j = 20;
```

```
float x = 12.5, y = 53.5;
```

```
swap(&i, &j, sizeof(int));
```

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

```
swap(&x, &y, sizeof(float));
```

```
printf("x = %f, y = %f\n", x, y);
```

```
}
```

```
/* Generic function to interchange two variables */
```

```
void swap(void *a, void *b, int n)
```

```
{
```

```
char *p, *q, temp;
```

```
p = (char*)a;
```

```
q = (char*)b;
```

```
while(n>0)
```

```
{
```

```
temp = *p;
```

```
*p = *q;
```

```
*q = temp;
```

```
p++;
```

```
q++;
```

```
n--;
```

```
}
```

```
}
```

Output:

i = 20, j = 10

x = 53.500000, y = 12.500000

Dangling Pointer

Dangling pointers are pointers that do not point to a valid variable of the proper type. Dangling pointers are created when memory is deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. When the system reallocates the previously freed memory to another process and the original program dereferences the dangling pointer, then it may produce an unpredictable result, as the memory may now contain completely different data.

Example

```
{  
int *cp;  
{  
int c;  
cp = &c;  
}  
/* c is in out of scope, cp is now a dangling pointer */  
}
```


Example

A solution to the above is to assign NULL to cp immediately before the inner block is exited.

Another frequent source of dangling pointers is a combination of malloc () and free () library function calls: a pointer becomes dangling when the block of memory it points to is freed. As with the previous example, one way to avoid this is to make sure to reset the pointer to null after freeing its reference.

```
#include <stdlib.h>

{
int *cp = malloc (100);
free ( cp ); /* cp now becomes a dangling pointer */
cp = NULL; /* cp is no longer dangling */
}
```

Allowable Operations with Pointer

- ❑ There are only a few numbers of operations of pointer are allowed. The allowable operations of pointer as follows:
 - ❖ Increment and decrement operations with pointer variables.
 - ❖ Subtraction of two pointer variables.
 - ❖ Addition and subtraction of integer value with pointers.
 - ❖ Relational operations between two pointer variables.
 - ❖ Assignment Operation.

Arithmetic Operation with Pointer

- ❑ Addition and subtraction of integer value with pointer variable are permitted. However, other arithmetic operations, including multiplication, division are not permitted with pointers. Non-integer values such as floats or double value addition or subtraction with pointer variables are not accepted.
- ❑ The format for adding or subtracting an integer to a pointer is:
 - Pointer-variable + integer-value
 - Pointer-variable - integer-value

Increment and Decrement operations with Pointer

□ There are three possible mix pointer increment and indirection as follows:

❖ i) $*p++$

❖ ii) $*++p$

❖ iii) $++*p$

Where p is a pointer variable.

`*p++`

`*p++` returns the content at the location being pointed by `p` and then increment the pointer by one. The pointer `p` will point the next element. The decrement operator can be used with pointers, in the same manner, to move to the previous element. Note that the compiler may give some warning or error message if next memory location is not allocated by the program statically or dynamically.

Example:

Exercice

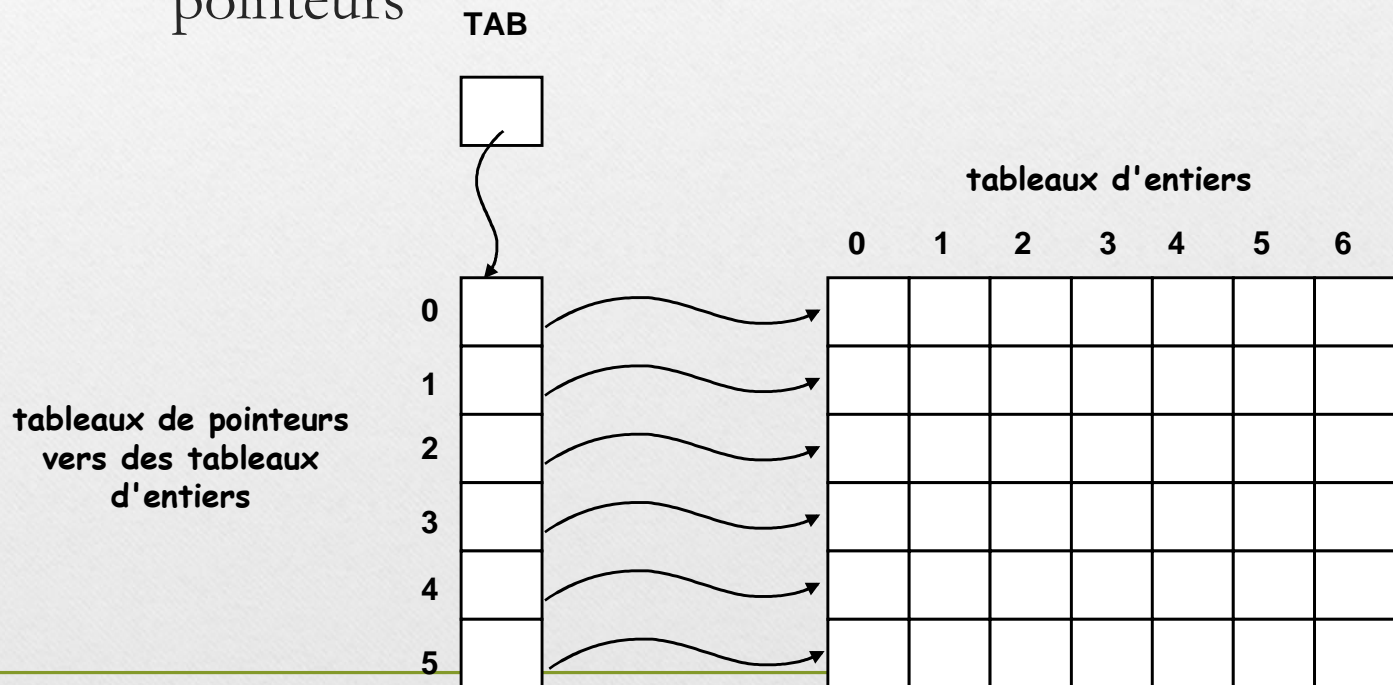
- Ecrire un programme qui declare et initialise un pointeur vers un tableau de 4 réels

Pointeurs et tableaux

- Lien entre le nom d'un tableau à 2 dimension et les pointeurs
 - un tableau 2d est un tableau de tableau de dimension 1
 - On peut donc dire que le nom d'un tableau 2dim est l'adresse d'un tableau d'adresse de tableaux de dimension 1
 - On dit aussi qu'il s'agit d'un pointeur de pointeur
- exemple:
`int TAB[6][7];`
`int **p;` déclaration d'un pointeur de pointeur
`p = TAB;` initialisation du pointeur de pointeur

Pointeurs et tableaux

- Lien entre le nom d'un tableau à 2 dimension et les pointeurs



Pointeurs et tableaux

- Initialisation dynamique d'un tableau
 - Tableau à une dimension

```
int * t;  
t = (int*)malloc(20*sizeof(int),
```

nom du
tableau

fonction
d'allocation
dynamique
de la
mémoire

nombre
d'éléments
dans le
tableau

fonction de
calcul
automatique
de la
taille d'un
entier en
octet

Pointeurs et tableaux

- Initialisation dynamique d'un tableau
 - Tableau à deux dimensions $n*m$
 - déclaration dynamique du tableau de n pointeurs et des n tableaux à une dimension de m éléments
 - quelques lignes de code sont nécessaires

Pointeurs et tableaux

- Initialisation dynamique d'un tableau
 - Tableau à deux dimensions $n \times m$

```
int **TAB,l;  
TAB = (int**)malloc(n*sizeof(int*));  
for (l=0;l<n;l++)  
{  
    TAB[l] = (int*)malloc(m*sizeof(int));  
}
```

désallocation:

```
for (l=0;l<n;l++)  
    free(TAB[l]);  
free(TAB);
```