mémo-C

- Objet : fiche mémo langage C
- Niveau requis :
  débutant, avisé

Référence : http://www.squalenet.net/fr/ti/tutorial_c/sommaire.php5

Introduction

installation gcc sous debian

```
apt-get install build-essential
```

Première compilation

- Création d'un fichier :

```
vim fichier.c
```

Code minimal :

```
#include<stdio.h>
#include<stdlib.h>
int main(void){    // peut s'écrire aussi: int main(int argc, char *argv[])
instructions;  // par exemple : printf("Hello world!\n");
return 0;
}
```

> **stdio.h** et **stlib.h** sont des **bibliothèques** (des fichiers source tout prêts).
> **main** est une **fonction**
> C'est toujours par la fonction main que le programme commence.
> Les instructions de la fonction se mettent entre accolade ouvrante **{** (début) et fermante **}** (fin). Une fonction est forcément délimitée par des accolades.
> Toute instruction se termine obligatoirement par un point-virgule « **;** »
> **return 0;** : Cette ligne indique qu'on arrive à la fin de notre fonction main et demande de renvoyer la valeur 0.

Exemple :

```
#include<stdio.h>
int main(void){
    printf("Hello World\n");
```

```
}
```

- \n : retour à la ligne
- \t : tabulation

printf :

- Syntaxe :

```
printf("")
```

```
printf("%type",variable)
```

ou

```
printf("%type", variable)
```

> La virgule est obligatoire.

```
printf("%type1 %type2",var1,var2)
```

> Les virgules sont obligatoires

- Compilation (se trouver dans le répertoire parent du fichier):

```
gcc main.c
```

- Il s'est créé un exécutable :

```
ls
```

```
a.out     main.c
```

- Exécuter :

```
./a.out
```

Ligne de commandes pour compilation et exécution

- Choisir le nom de l'exécutable :

```
gcc -o nom-exécutable nom fichier-à-compiler
```

```
gcc -o main.x main.c
```

```
ls
```

```
a.out  main.c   main.x
```

> -o : donner le nom “main.x” à l'exécutable ;
> main.x : nom choisi pour l'exécutable ;
> main.c : nom du fichier de code c qui va être compilé ;

- compilation et exécution :

```
gcc -o main.x main.c; ./main.x
```

- ./main.x : pour lancer l'exécutable issu de la compilation ;

Variables

Les variables sont typées. Deux grands types, numériques, caractères.

type	déclaration	récupération de valeur
int	int nombre=n (n est un nombre de 0 à 255)	“%d,nombre”
char	char caratere='x' (x est une lettre)	“%c,x”
char	char chaine[nombre-max]=“chaine”	“%s,chaine”

Nombre entier

- Variable nombre entier (int)

```
#include<stdio.h>
int main(void){
    int age=22;
    int taille=175;
    printf("valeur age: %d valeur taille: %d\n",age,taille);
}
```

- La dernière valeur d'une même variable est celle enregistrée.

```
#include<stdio.h>
int main(void){
    int age=22;
    age=23;
    int taille=175;
    printf("valeur age: %d valeur taille: %d\n",age,taille);
}
```

```
gcc -o main.x main.c; ./main.x;
```

```
valeur age: 23 valeur taille: 175
```

char : caractère, chaîne de caractères

Pour désigner un caractère imprimable, il suffit de le mettre entre apostrophes (par

ex. 'A' ou '$'). Les seuls caractères imprimables qu'on ne peut pas représenter de cette façon sont l'antislash et l'apostrophe, qui sont respectivement désignés par **** et **\'**.
Le point d'interrogation et les guillemets peuvent aussi être désignés par les notations **\?** et **\"**.
Les caractères non imprimables peuvent être désignés par **'\code-octal'** où code-octal est le code en octal du caractère.
On peut aussi écrire '\xcode-hexa' où code-hexa est le code en hexadécimal du caractère (cf. page X).
Par exemple, **'\33'** et **'\x1b'** désignent le caractère escape.

Toutefois, les caractères non-imprimables les plus fréquents disposent aussi d'une notation plus simple :

```
n     nouvelle ligne
t     tabulation horizontale
v     tabulation verticale
b     retour arrière
r     retour chariot
f     saut de page
a     signal d'alerte
```

Suivant les implémentations, le type char est signé ou non. En cas de doute, il vaut mieux préciser **unsigned char** ou **signed char**.

char : %c (un caractère)

```c
#include<stdio.h>
int main(void){
char lettre='x';
lettre='a';
printf("%c\n",lettre);
}
```

char : %s (chaîne de caractères)

%c	unsigned char	caractère
%s	char*	chaîne de caractères

```c
#include <stdio.h>
main()
{
  char c = 'A';
  char *chaine = "chaine de caracteres";
```

```
  printf("\nimpression de c: \n");
  printf("%c \t %d",c,c);
  printf("\nimpression de chaine: \n");
  printf("%s \t %.10s\n",chaine,chaine);
  printf("\t%.7s \t %.5s\t %.3s\n",chaine,chaine,chaine);
  printf("\n");
}
```

```
gcc -o char.x char.c; ./char.x
```

```
impression de c:
A    65
impression de chaine:
chaine de caracteres     chaine de
    chaine       chain     cha
```

- **Autre méthode** :

```
#include<stdio.h>
int main(void){
char chaine[6]="a b c";
printf("%s\n",chaine);
}
```

> Ne pas oublier de pointer un espace mémoire pour chaque caractère (ou plus) de la chaîne.
> les guillemets doubles autour de la valeur sont obligatoires.

Une des particularités du type char en C est qu'il peut être assimilé à un entier : tout objet de type char peut être utilisé dans une expression qui utilise des objets de type entier.
Par exemple, si c est de type char, l'expression c + 1 est valide. Elle désigne le caractère suivant dans le code ASCII.
Ainsi, le programme suivant imprime le caractère 'B'.

```
main()
{
  char c = 'A';
  printf("%c", c + 1);
}
```

Déclaration de plusieurs types de variables

- Soit le programme “declare.c” :

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

```
main(void){
  printf("Bonjour\n");
  char lettre='a';
  printf("valeur \"lettre\": %c\n",lettre);
  char chaine[10]="Hypathie";  // [10] pour déclarer une chaîne, il faut
indiquer le nombre max
  printf("valeur \"chaine\": %s\n",chaine);
  int age=22;
  printf("Valeur \"age\": %d\n",age);
 return 0;
}
```

```
gcc -o declare.x declare.c ; ./declare.x
```

```
Bonjour
valeur "lettre": a
valeur "chaine": Hypathie
Valeur "age": 22
```

Une seule instruction peut appeler plusieurs variables

```
#include<stdio.h>
int main(void){
char nom[10]="Hypathie"; // 10 max, "hypathie -> de 0 à 7"
char lettre='a';
printf("valeur nom : %s\nvaleur lettre : %c\n",nom, lettre);
}
```

```
gcc -o main.x main.c; ./main.x;
valeur nom : Hypathie
valeur lettre : a
```

Référence à une lettre d'une chaîne : la première est zéro

```
#include<stdio.h>
int main(void){
char nom[10]="Hypathie";
printf("valeur nom : %s\nvaleur lettre n°5 de nom : %c\n",nom, nom[5]);
}
```

```
gcc -o main.x main.c; ./main.x;
valeur nom : Hypathie
valeur lettre n°5 de nom : h
```

Remplacement d'un caractère d'une chaîne

```
#include<stdio.h>
int main(void){
char nom[10]="Hypathie";
nom[3]='A';  // On ne la re-type pas !
printf("lettre n°5 de nom : %s\n",nom);
}
```

```
lettre n°5 de nom : HypAthie
```

Récupération d'éléments d'une chaîne

```
#include<stdio.h>
int main(void){
int var[2];
var[0]=2;
var[1]=4;
var[2]=6;
printf("%d %d %d\n",var[0], var[1], var[2]);
}
```

```
2 4 6
```

déclarations de plusieurs variables (même type) sur une ligne

```
#include<stdio.h>
int main(void){
 int a, b, c;  // int a,b,c;
}
```

- équivalent de :

```
#include<stdio.h>
int main(void){
 int 1;
 int 2;
 int 3;
}
```

Tableau

```
#include<stdio.h>
int main(void){
int tableau1[3]={2,4,6}; //tab-de-0=2, tab-de-1=4, tab-de-3=6
printf("%d %d %d\n",tableau1[0],tableau1[1],tableau1[2]);
}
```

```
2 4 6
```

Arithmétique

```
#include<stdio.h>
int main(void){
int age=24;
age=age+1;
printf("%d\n",age+5);
}
```

```
30
```

> age=age+1 : modification de la valeur de la vaiable “age” ;
> mais le résultat de “age+5” n'est pas enregistré, la variable “age” a toujours la valeur 25.

Opérateurs de calcul :

- + addition
- - soustraction
- * multiplication
- / division (quotient)
- % modulo (reste de division euclidienne)

On peut utiliser des parenthèses pour définir des ordres de priorités dans un calcul.
x=(a+b)*2

Autre type de variables numériques

Pour les nombres flottant, la virgule est remplacée par un point.

Nom du type	Minimum	Maximum
signed char	-127	127
int	-32 767	32 767
long	-2 147 483 647	2 147 483 647
float	-1 x1037	1 x1037
double	-1 x1037	1 x1037
unsigned char	0	255
unsigned int	0	65 535
unsigned long	0	4 294 967 295

- affichage du contenu de variable :

Format	Type attendu
”%d“	int

Format	Type attendu
”%ld“	long
”%f“	float
”%f“	double

Les conditions

if else

- si plusieurs instructions, accolades :

```
#include<stdio.h>
int main(void){
déclarations1;
déclaration2;
...;
if(condition){
instruction1;
instruction2;
}
else{
instruction1;
instruction2;
}
}
```

> entre les parenthèses : condition
> entre accolades : instructions
> accolades dans le **`if` et dans le `else`**,
> accolades **nécessaires** dans le `if` et dans le `else`, si **plusieurs instructions** dans le `if` et dans le `else`; s'il n'y a qu'une instruction, on peut éviter les accolades.

- Avec qu'une instruction, pas besoin d'accolades :

```
#include<stdio.h>
int main(void){
déclaration1;
déclaration2;
if(condition)instruction;
else instruction;
}
```

- Exemples de if avec comparaison (==)

```
#include<stdio.h>
int main(void){
int var1=10;
if(var1==10){
printf("var1 est équivalent à %d\n", var1);
}
```

```
}
```

```
var1 est équivalent à 10
```

- sans accolades car qu'une instruction :

```
#include<stdio.h>
int main(void){
int var1=1;
if(var1==10) printf("var1 est équivalent à %d\n", var1);
else printf("var1 est équivalent à %d\n et non à 1\n", var1);
}
```

```
var1 est équivalent à 1
 et non à 1
```

Imbrication de if, else if, else

- Exemple : imbrication de if et if else sans accolades avec int

```
#include<stdio.h>
int main(void){
int taille=10;
if(taille==10)printf("\tMoyen\n");
else if(taille>10)printf("\tGrand\n");
else printf("\tPetit\n");
}
```

- Exemple 3 : idem avec char :

```
#include<stdio.h>
int main(void)
{
// afficher voyelle si 'x' == voyelle ou afficher consonne si 'x' ==
consonne
char var='x';
if(var=='a')printf("Voyelle\n");
else if(var=='e')printf("Voyelle\n");
else if(var=='i')printf("Voyelle\n");
else if(var=='o')printf("Voyelle\n");
else if(var=='u')printf("Voyelle\n");
else if(var=='y')printf("Voyelle\n");
else printf("Consonne\n");
}
```

- Imbrication de if et else avec accolades car plusieurs instructions :

```
#include<stdio.h>
int main(void)
{
 int taille=10;
```

```
 if(taille==10){
  printf("\tMoyen\n");
  printf("\n");
 }
 else
 {
  if(taille>10){
    printf("\tGrand\n");
    printf("\n");
    Moyen

  }
  else{
   printf("\tPetit\n");
   printf("\n");
   }
 }
}
```

```
    Moyen
```

Rappel if else avec le “shell”:

```
if [ "$USER" = "smolski" ];
then
  echo 'Salut chef !'
else
  echo "Bonjour $USER."
fi
```

Opérateurs relationnels

Autres opérateurs relationnels :

- **!** inverse
- **!=** ne correspond pas
- **<** inférieur à
- **>** supérieur à
- **>=** supérieur ou égal
- **<=** inférieur ou égal
- **==** correspond à

correspondance et non correspondance

- 0 = faux ; tout autre nombre = vrai :

```
#include<stdio.h>
int main(void){
int var1=1;
if(1){
printf("IF\n");
}
else{
 printf("ELSE\n");
}
}
```

```
IF
```

```
#include<stdio.h>
int main(void){
int var1=1;
if(0){
printf("IF\n");
}
else{
 printf("ELSE\n");
}
}
```

```
ELSE
```

- L'inverse de faux (de 0) :

```
#include<stdio.h>
int main(void){
int var1=1;
if(!0){
printf("IF\n");
}
else{
 printf("ELSE\n");
}
}
```

```
IF
```

(inverse de 0, c'est 1 ⇒ vrai)

On peut définir des ordres de priorités dans les conditions, avec les parenthèses.

- Inverse de l'inverse :

```
#include<stdio.h>
int main(void){
int var1=1;
if( !(var1!=1) ){ // !(var1==1) <-> var1!=1
printf("IF\n");
}
else{
 printf("ELSE\n");
}
}
```

```
IF
```

supérieurs inférieur

```
#include<stdio.h>
int main(void){
int var=5;
if(var <=4 ) printf("%d est inf. à 4\n",var);
else printf("%d n'est pas inf. à 4\n", var);
}
```

- Avec deux variables :

```
#include<stdio.h>
int main(void){
int var1=5;
if(var1 <=4 ) printf("var1 (%d) est inf. à 4\n",var1);
else if (var1 >= 4) printf("var1 (%d) est sup. ou ég. à 4\n", var1);
else if (var1 >= 5) printf("var1 (%d) est sup. ou ég. à 5\n", var1);
int var2=6;
if (var2 < 6) printf("var2 (%d) est inf. à 6\n",var2);
else if (var2 <= 6) printf("var2 (%d) est inf. ou ég. à 6\n", var2);
}
```

while, for, do while, ++

Boucle while

incrémentation

```
#include<stdio.h>
int main(void)
{
//faire prendre à i les valeurs de 1 à 9
```

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

Rappel syntaxe du shell:

```
while condition;
do
  commande;
done
```

Attention aux boucles infinies

```
#include<stdio.h>
int main(void)
{
  int i=1;
  while(1) //condition 1 -> condition vraie : exécution de printf infinie
  {
    printf("%d\n",i);
    i=i+1;
  }
}
```

Boucle for

- Syntaxe:

```
#include<stdio.h>
int main(void)
{
  int i;  //déclaration
  for(affectation;condition;incrémentation)
  {
    printf
  }
}
```

- Exemple :

```
#include<stdio.h>
int main(void)
```

```
{
  int i;  //déclaration
  for(i=0;i<10;i++)
  {
    printf("%d\n",i);
  }
}
```

Rappel : syntaxe du shell

```
for i in "les fleurs" "le lapin" "le chou" "la salade" "le staff DF";
do
  echo "J'aime $i."
done
```

do while

- Syntaxe :

```
#include<stdio.h>
int main(void)
{
  int i;
  do{
    printf("%d\n",i);
    i++;
  }while(i<10);
}
```

Spécificité de do while

la boucle “do while” exécute toujours au moins une fois l'instruction

- Comparaison avec autres types de boucle :

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

```
  i=0; // remise à zéro de i
  printf("\n"); //imprimer un retour à la ligne
  do{
    printf("%d\n",i);
    i++;
  }while(i<0);
}
```

```
gcc -o main.x main.c; ./main.x;

0
```

Conditions logiques

1 (ou autres chiffres) : vrai

```
#include<stdio.h>
int main(void)
{
 char lettre='a';
 if(lettre=='a') printf("IF\n");
}
```

```
IF
```

- De même :

```
#include<stdio.h>
int main(void)
{
 if(1) printf("IF\n");
}
```

```
IF
```

0 faux

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

```
else
```

Table de vérité

```
__OU logique (au moins l'un des deux est vrai) :__

0||0 -> 0
0||1 -> 1
1||0 -> 1
1||1 -> 1

__ET logique (l'un et l'autre est vrai) :__

0&&0 -> 0
0&&1 -> 0
1&&0 -> 0
1&&1 -> 1
```

condition logique sur variables

- OU

```
#include<stdio.h>
int main(void)
{
 char lettre='a';
 if(lettre=='a'||lettre=='b') printf("VRAI\n");
}
```

```
VRAI
```

- Autre exemple :

```
#include<stdio.h>
int main(void)
{
 char lettre='x';
if(lettre=='a'||lettre=='e'||lettre=='i'||lettre=='o'||lettre=='u'||lettre=='y'){
     printf("Voyelle\n");
 }
 else{
     printf("Consonne\n");
 }
}
```

```
Consonne
```

- ET (il faut deux variables) :

```

```

```
#include<stdio.h>
int main(void)
{
 char lettre='a';
 char lettre2='b';
 if(lettre=='a'&&lettre2=='b') printf("VRAI\n");
}
```

Consonne

```
VRAI
```

switch ; break et default

Syntaxe

```
switch(nom-variable){
cas1;
cas2;
...;
}
```

Exemples

```
#include<stdio.h>
int main(void){
 char lettre='a';
  switch(lettre){
   case 'a': printf(Voyelle\n);
   case 'z': printf(Consonne\n);
  }
}
```

```
Voyelle
Consonne
```

> “case” n'est pas une condition.
> La ligne du “case” définit seulement l'endroit où va commencer l'exécution.
> Dans le cas du “a”, il commence à “a” et va jusqu'à “z”.

```
#include<stdio.h>
int main(void){
 char lettre='z';
  switch(lettre){
   case 'a': printf(Voyelle\n);
   case 'z': printf(Consonne\n);
```

```
  }
}
```

```
Consonne
```

Avec break

```
#include<stdio.h>
int main(void){
 char lettre='a';
  switch(lettre){
   case 'a': printf(Voyelle\n);break;
   case 'z': printf(Consonne\n);
  }
}
```

```
Voyelle
```

Avec break et default

```
#include<stdio.h>
int main(void){
 char lettre='x';
  switch(lettre){
   case 'a': printf("Voyelle\n");break;
   case 'z': printf("Consonne\n");
   default : printf("Autre chose\n");
  }
}
```

```
Autre chose
```

- Autre exemple :

```
#include<stdio.h>
int main(void)
{
 char lettre='x';
 switch(lettre){
 case 'a':
 case 'e':
 case 'i':
 case 'o':
 case 'u':
 case 'y': printf("Voyelle\n");break;
 default: printf("Consonne\n");
 }
}
```

```
Consonne
```

- Autre syntaxe :

```
#include<stdio.h>
int main(void)
{
 char lettre='x';
 switch(lettre){
 case 'a':case 'e':case 'i':case 'o':case 'u':
 case 'y': printf("Voyelle\n");break;
 default: printf("Consonne\n");
 }
}
```

witch et plus rapide que if.

* commande time pour comparer:

```
gcc if.c; time ./a.out
```

Puis :

```
gcc switch.c; time ./a.out
```

break et continue

break permet de quitter n'importe quel programme !

```
#include<stdio.h>
int main(void){
 while(1){
  printf("1\n");
  break;
  printf("2\n");
  }
}
```

continue permet de repartir au début de la boucle !

* Autre boucle infinie :

```
#include<stdio.h>
int main(void){
 while(1){
  printf("1\n");
```

```
  continue;
  printf("2\n");
  }
}
```

scanf

Même usage que la commande Linux read.

Syntaxe

```
scanf("%type",&nom-variable);
```

Exemple

```
#include<stdio.h>
int main(void)
{
 char lettre; //déclaration, (on enlève l'initialisation; char lettre='y')
 printf("entre une lettre, svp :");
 scanf("%c",&lettre);
 switch(lettre){
 case 'a':
 case 'e':
 case 'i':
 case 'o':
 case 'u':
 case 'y': printf("Voyelle\n");break;
 default: printf("Consonne\n");
 }
}
```

Fonctions

Fonction "main" et fonction utilisateur

- Reprenons le code de base :

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

```
}
```

- On peut utiliser une fonction pour afficher :

Une fonction se déclare entre le "#include<stdio.h>" et le "int main(void){

```
#include<stdio.h>
void FONCTION(){
instruction
}
int main(void)
{
FONCTION();
}
```

> Ce qui est à gauche d'une fonction, c'est ce que retourne une fonction.
> On met `void` quand on veut qu'elle ne retourne rien; c'est-à-dire qu'il n'y aura pas d'argument à la fonction.

exemple

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

```
Bonjour
```

> On peut aussi inscrire : **`void bonjour(void){`**.

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

> ce qui commence par un "#" sont des instructions pour le compilateur
> "include" : inclure → inclure le fichier "stdio.h"
> c'est un fichier qui contient des commandes
> une commande comme printf se trouve dans le fichier "stdio.h"

Passer des éléments à une fonction

explications

- Sans fonction, inverser les valeurs de deux variables :

```
#include<stdio.h>
//void inverser(void){
//}
int main(void)
{
int a=1;
int b=2;
printf("a:%d\nb:%d\n",a,b);
//inverser();
int c=a;
a=b;
b=c;
printf("a:%d\nb:%d\n",a,b);
}
```

```
a:1
b:2
a:2
b:1
```

L'appel de la fonction exécute la fonction avec les valeurs de la dernière fonction

Les valeurs des variables déclarées dans la fonction “inverser”, ne sont pas enregistrées en mémoire, tant que la fonction “main” n'a pas appelé la fonction “inverser”.
Rappel : **la fonction “main” est toujours exécutée en premier**.

```
#include<stdio.h>
void inverser(int x, int y){
printf("\t\tx:%d\ty:%d\n",x,y);
int z=x;
x=3;
y=4;
printf("x:%d\ny:%d\n",x,y);
}
int main(void)
{
int a=1;
int b=2;
printf("\t\ta:%d\tb:%d\n",a,b);
inverser(a,b);
printf("\ta:%d\n\tb:%d\n",a,b);
}
```

```
        a:1    b:2 //exécution de ligne n°13
        x:1    y:2 //exécution de ligne n°3
x:3                        //exécution de ligne n°7
y:4                        //idem
    a:1                 //exécution de ligne n°15
```

```
    b:2                 //idem
```

> La fonction “inverser” a créé deux emplacements “int” en mémoire (“x” et “y”).
> Lors de l'exécution de la ligne n°3, la fonction “inverser” “récupère” les valeurs des emplacements “int” de la fonction “main”, dans l'ordre d'enregistrement.
> Lors de l'exécution de la ligne n°15 la fonction “main” utilise les valeurs des emplacements “int” de la fonction “inverser”.
> Cette “récupération”, n'en est pas exactement une.
>
> Rappel : dans une même fonction, quand on réaffecte une même variable, par une nouvelle valeur, l'ancienne valeur est écrasée, et c'est la dernier valeur qui est conservée en mémoire. Pour une même variable, chaque nouvelle affectation écrase en mémoire l'ancienne valeur, ainsi une même variable peut changer de valeur au cours du programme (“une variable varie”).
> Au niveau des fonctions, quand il y en a plusieurs, ce ne sont pas les noms des variables de chacune des fonctions qui sont conservés en mémoire, mais c'est le TYPE de variable et le nombre de variables de ce TYPE qui sont pointés en mémoire.
> À la fin de l'exécution d'une fonction, toutes les variables de cette fonction sont écrasés, pour l'exécution de la fonction suivante. Mais lors cet écrasement, les variables de la nouvelle fonction de même TYPE que celles de la fonction écrasée sont enregistrés au même endroit : “la valeur d'un type de variable”, joue le rôle de pointeur, d'une fonction à l'autre.

Créer une fonction "inverser"

- x et y de la fonction “inverser” utilisent les valeurs de a et b de la fonction main :

```
void inverser(int x, int y){
printf("\t\tx:%d\ty:%d\n",x,y);
}
int main(void)
{
int a=1;
int b=2;
printf("\t\ta:%d\tb:%d\n",a,b);
int c=a;
a=b;
b=c;
inverser(a,b);
}
```

```
        a:1     b:2
        x:2     y:1
```

- Il suffit donc de donner aux variables “int” de la fonction “inverser”, les mêmes noms que ceux des variables “int” du “main” dont elle récupère les valeurs :

```
void inverser(int a, int b){
printf("\t\ta:%d\tb:%d\n",a,b);
}
int main(void)
{
int a=1;
```

```
int b=2;
printf("\t\ta:%d\tb:%d\n",a,b);
int c=a;
a=b;
b=c;
inverser(a,b);
}
```

```
        a:1     b:2
        a:2     b:1
```

> Bien considérer que “a” et “b” de la fonction “inverser”, ne sont pas les “mêmes” variables dans la mémoire vive; il y a un emplacement mémoire pour chacune des variables “a” et “b”, de chacune des fonctions “inverser” et “main”.

- Ce qui revient au même (en plus propre) que :

```
void inverser(int a, int b){
printf("\t\ta:%d\tb:%d\n",a,b);
int c=a;
a=b;
b=c;
printf("\t\ta:%d\tb:%d\n",a,b);
}
int main(void)
{
int a=1;
int b=2;
inverser(a,b);
}
```

```
        a:1     b:2
        a:2     b:1
```

Les pointeurs

Définitions

Un pointeur est un objet Lvalue[1] dont la valeur est égale à l'adresse d'un autre objet.
On déclare un pointeur par l'instruction : **`type *nom-du-pointeur;`**, où “type” est le type de l'objet pointé.

Cette déclaration déclare un **identificateur**, nom-du-pointeur, associé à un objet dont la **valeur est l'adresse d'un autre objet de type “type”**.

L'**identificateur nom-du-pointeur** est donc en quelque sorte un **identificateur d'adresse**.
Comme pour n'importe quelle Lvalue

Une **Lvalue** est caractérisée par :

- **son adresse**, c'est-à-dire l'adresse-mémoire à partir de laquelle l'objet est stocké ;
- **sa valeur**, c'est-à-dire ce qui est stocké à cette adresse; sa valeur est modifiable.

Si le compilateur a placé la variable **`i`** à l'adresse 4831836000 en mémoire, et la variable **`j`** à l'adresse 4831836004, on a :

objet	adresse	valeur
i	4831836000	3
j	4831836004	3

- Deux variables différentes ont des adresses différentes.

L'affectation **`i = j;`** n'opère que sur les **valeurs** des variables.
Les variables i et j étant de type **`int`**, elles sont stockées sur 4 octets.
Ainsi la valeur de i est stockée sur les octets d'adresse 4831836000 à 4831836003.

- Pour **un pointeur sur un objet de type char**, la valeur donne l'adresse de l'octet où cet objet est stocké.
- **L'opérateur &** permet d'accéder à l'adresse d'une variable.

Toutefois **`&i`** n'est pas une “Lvalue” mais une “constante” : on ne peut pas faire figurer **`&i`** à gauche d'un opérateur d'affectation.
Pour pouvoir manipuler des adresses, on doit donc recourir un nouveau type d'objets, les **pointeurs**.

- **L'opérateur unaire d'indirection *** permet d'accéder directement à la valeur de l'objet pointé.

Ainsi, si **p** est un pointeur vers un entier **`i`**, **`*p`** désigne la **`valeur de i`**.

Exemple

```
#include<stdio.h>
main()
{
  int i = 3;
  int *p;

  p = &i;
  printf("*p = %d \n",*p);
}
```

```
*p = 3
```

On se trouve dans la configuration :

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Pour aller plus loin, voir https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/chapitre3.html

Créer une fonction "inverser" avec des pointeurs

```
#include<stdio.h>
void inverser(int *x, int *y){
int c;
c=*x; //"c" correspond à ce qui est pointé par "x" (valeur de "a": 1)
*x=*y; //ce qui est pointé par "x" ("a") correspond à ce qui est pointé par
"y" ("b" :2)
*y=c; // ce qui est pointé par "y" ("b") correspond à "c" (1)
}
int main(void)
{
int a=1;
int b=2;
printf("\t\ta:%d\tb:%d\n",a,b);
inverser(&a,&b); // la première variable pointée (*) dans la fonction
"inverser"
                 // prend la valeur de la première variable adressée (&),
ici a qui vaut 1
         // la deuxière variable pointée, prend la valeur de la deuxième
variable adressés, b=2
printf("\t\ta:%d\tb:%d\n",a,b);
}
```

```
        a:1     b:2
        a:2     b:1
```

Et le scanf !

- Il y a aussi un “et commercial” (&) dans le scanf.
  C'est aussi un pointeur.

```
#include<stdio.h>
int main(void)
{
 char lettre; //déclaration, (on enlève l'initialisation; char
lettre='y')
 printf("entre une lettre, svp :");
 scanf("%c",&lettre);
 switch(lettre){
 case 'a':
 case 'e':
 case 'i':
 case 'o':
 case 'u':
 case 'y': printf("Voyelle\n");break;
 default: printf("Consonne\n");
```

```
 }
}
```

- **&lettre** : Ce qu'on entre est enregistré à l'adresse de la variable “lettre” qui n'a pas été affectée d'une valeur.

tableau bis

Un tableau est pointeur qui pointe vers le premier élément du tableau.

tableau int

```
#include<stdio.h>
int main(void){
    int tab[10];
    tab[0]=10;
    tab[1]=20;
    tab[2]=30;
    printf("%d\t%d\t%d\n",tab[0],tab[1],tab[2]);
    printf("%d\n",*(tab));
    printf("%d\n",*(tab+1));
    printf("%d\n",*(tab+2));
    printf("%d\n",*(tab+3));
    printf("\n");
    *(tab+2)=60;
    printf("%d\n",tab[2]);
    printf("\n");
    tab[2]=55;
    printf("%d\n",*(tab+2));
    printf("%d\n",tab[2]);
}
```

```
10  20    30
10
20
30
0

60

55
```

> ***tab** : ce qui est pointé par tab
> **tab[0]** : premier élément du tableau
> ***(tab+2)** et **tab[2]** ont la même valeur, on peut modifier la valeur par le biais de **tab[2]** ou de ***(tab+2)**.
> ***(tab+1)** : premier élément du tableau + 4 octets (entier sur 4 octets), c'est-à-dire valeur du

> deuxième élément du tableau (le premier est `tab[0]` ou `*(tab+0)`)
> **`*(tab+2)`** : premier élément du tableau + 8 octets, c'est-à-dire valeur du troisième élément du tableau.
> etc.

tableau char

```
#include<stdio.h>
int main(void){
    char tab[27];
    tab[0]='a';
    tab[1]='b';
    tab[3]='c';
    tab[4]='d';
    tab[5]='e';
    tab[6]='f';
    tab[7]='g';
    tab[8]='h';
    tab[9]='i';
    tab[10]='j';
    tab[11]='k';
    tab[12]='l';
    tab[13]='m';
    tab[14]='n';
    tab[15]='o';
    tab[16]='p';
    tab[17]='q';
    tab[18]='r';
    tab[19]='s';
    tab[20]='t';
    tab[21]='u';
    tab[22]='v';
    tab[23]='w';
    tab[24]='x';
    tab[26]='y';
    tab[27]='z';
    printf("%c\t %c\t %c\t %c\t %c\t
%c\n",tab[0],tab[5],tab[9],tab[15],tab[21],tab[26]);
    *(tab+0)='A';
    tab[5]='E';
    printf("%c\t %c\n",tab[0],*(tab+5));
}
```

```
a    e    i    o    u    y
A    E
```

- Idem en mieux (pour mémoire dure) :

```
#include<stdio.h>
int main(void){
```

```
    char
tab[27]={'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q'
,'r','s','t','u','v','w','x','y','z'};
    printf("%c\t %c\t %c\t %c\t %c\t
%c\n",tab[0],tab[5],tab[9],tab[15],tab[21],tab[26]);
    *(tab+0)='A';
    tab[5]='E';
    printf("%c\t %c\n",tab[0],*(tab+5));
}
```

fonction bis

bibliothèque stdio.h

fonction int et commande sizeof

Un entier à une taille en octet.

```
#include<stdio.h>
int main(void){
    printf("%d\n",sizeof(int));
}
```

```
4
```

Cela permet de comprendre l'utilisation de la table ASCII.

On pourrait faire :

```
#include<stdio.h>
int main(void){
    printf("%d\n",sizeof(char));
}
```

```
1
```

> 1 octet = 8 bit
> 8 bits permettent de coder 255 caractères différents.
> char peut donc “contenir la table ASCII qui va pour le code décimal des caractères alphanumériques (+des caractères spéciaux) qui va de 0 à 255.

```
#include<stdio.h>
int main(void){
int lettre=97;
    printf("%c\n",lettre);
```

```
}
```

```
a
```

Pourquoi la fonction "main" renvoie-t-elle à un "int" ?

Surtout qu'elle peut contenir des “char” ! Rappel : “void” : quand la fonction ne retourne rien.

```c
#include <stdio.h>

int somme(int x,int y){
    int result;
    result=x+y;
    return result;
}
int main(void){
    int addition;
    addition=somme(2,3);
    printf("%d\n",addition);
}
```

```
5
```

- ou en raccourci :

```c
#include <stdio.h>

int somme(int x,int y){
    int result;
    result=x+y;
    return result;
}
void main(void){
    int addition;
    int var1=2;
    int var2=3;
    addition=somme(var1,var2);
    printf("%d\n",addition);
}
```

> On affecte la variable “addition” de la fonction “somme” car elle a un “return”.
> On met **`int main(void){`** car il possible de faire renvoyer des information par la fonction “main”.
> Quand elle ne renvoie rien on peut aussi mettre **`void main(void){`**.
>
> **C'est au système d'exploitation que le “int” renvoie des informations**.

Se servir du "int" dans la fonction "main"

```
#include <stdio.h>

int somme(int x,int y){
    int result;
    result=x+y;
    return result;
}
int main(void){
    int addition;
    int var1=2;
    int var2=3;
    addition=somme(var1,var2);
    printf("%d\n",addition);
        return 0;
}
```

```
gcc -o fct.x fct.c; ./fct.x; echo $?
```

```
5
0
```

> 0 est le code de retour.

Bibliothèque string.h

fonction strcpy

Elle permet d'utiliser la fonction **strcpy**. Cette fonction “strcpy” permet la manipulation des chaînes de caractères.

```
strcpy(nom-de-la-variable-declaree,"nlle-valeur");
```

```
#include <stdio.h>
#include <string.h>
int main(void){
    char nom[5]="toto";
    strcpy(nom,"tata");
    printf("%s\n",nom);
}
```

```
tata
```

⇒ Voir le fichier “string.h” pour voir quelle autre fonction il contient.

fonction strcat

Elle permet de rajouter à la suite.
“cat”, comme “concaténer” !

```
#include <stdio.h>
#include <string.h>
int main(void){
    char nom[15]="bonjour";
    strcat(nom," toto");
    printf("%s\n",nom);
}
```

```
bonjour toto
```

for dans chaîne de caractères

```
#include <stdio.h>
#include <string.h>
int main(void){
    char nom[25]="Hello";
    strcat(nom," Toto, are you ok ?");
    int i;
    for(i=0;i!=25;i++){
        printf("nom[%d]=%c\n",i,nom[i]);
    }
    printf("\n");

}
```

```
nom[0]=H
nom[1]=e
nom[2]=l
nom[3]=l
nom[4]=o
nom[5]=
nom[6]=T
nom[7]=o
nom[8]=t
nom[9]=o
nom[10]=,
nom[11]=
nom[12]=a
nom[13]=r
nom[14]=e
nom[15]=
nom[16]=y
nom[17]=o
nom[18]=u
```

```
nom[19]=
nom[20]=o
nom[21]=k
nom[22]=
nom[23]=?
nom[24]=
```

> L'espace réservé peut-être plus large que les éléments.

Fonction strcmp

Elle permet de comparer.

```
#include <stdio.h>
#include <string.h>
int main(void){
    char nom[10]="Hello b";
        printf("%d\n",strcmp(nom,"Hello"));
}
```

```
32
```

Les structures

Elles permettent d'utiliser plusieurs types de variable dans le “main”.

Syntaxe

Création d'une structure

- **Dans le “struct”** :
  On crée une structure constituée de plusieurs types de variable afin de pouvoir tous les utiliser dans un seul “main”.
  Pour ce faire,
  - on utilise la commande **`struct`**,
  - on nomme la structure qu'on crée,
  - enfin entre les accolades ouvrante et fermante, on annonce chacun des types de variables qui compose la structure.
  Ne pas oublier le point-virgule après l'accolade fermante de **`struct`**.
- **Dans le “main”**:
  on associe à chaque variable déclarée, un type de variable permis par la création de la structure.
  `var.type` : Le point permet “d'associer” à “var”, un type de variable annoncé dans le **`struct`**.
  C'est **`var.type`** qu'on affecte d'une valeur : **`var.type=valeur`**.
  C'est donc **`var.type`** qui permet d'appeler la valeur qu'on lui a affecté :
  `(”%type“,var.type);` .

```
#include<stdio.h>
#include<string.h>
struct nom-structure{
  type-chaine;
  type-caractère[n];
  type-entier;
};
int main(void){
  struct nom-structure var
  strcpy(var.type-chaine,"chaine de caractère");
  var.type-caractère='c';
  var.type-entier=n;

  printf("%s",var.type-chaine)
  printf("%c",var.type-caractère)
  printf("%d",var.type-entier)
}
```

Exemple

```
#include<stdio.h>
#include<string.h>
struct earthborn{
  char sex;
  char name[20];
  int age;
};
int main(void){
  struct earthborn person1;
   person1.age=78;
   person1.sex='M';
   strcpy(person1.name,"Paul Dupont");
   printf("age:%d\t sex:%c\t
name:%s\n",person1.age,person1.sex,person1.name);
  struct earthborn person2;
   person2.age=84;
   person2.sex='F';
   strcpy(person2.name,"Agath Montsoirie");
   printf("age:%d\t sex:%c\t
name:%s\n",person2.age,person2.sex,person2.name);
}
```

```
age:78   sex:M     name:Paul Dupont
age:84   sex:F     name:Agath Montsoirie
```

[1)]

left value : tout objet pouvant être placé à gauche d'un opérateur d'affectation.