PROGRAMMATION C

POLYTECH'MONTPELLIER
IG1

Anne Laurent
laurent@lirmm.fr

Présentation du cours

- 10,5 heures de cours
- 21 heures de TD/TP
- Soit :
  - 7 cours
  - TD/TP : séances de 1,5h (TD et TP) puis 3h TP
- Contrôle des connaissances :
  - Partiel (2/3 de la note finale)
  - Projet (1/3 de la note finale)

Bibliographie

- Kernighan et Ritchie, *Le langage C*, 2nd édition, Masson, 1997.
- Garetta, *C : Langage, bibliothèque, applications*, Interéditions
- Sedgewick, *Algorithmes en langage C*, Interéditions.
- Sites Web : INT Evry
- ...

Historique

- Initialement écrit pour supporter le système UNIX par Dennis Ritchie et Brian Kernighan. Début des années 70. (Bell)
- Plus de 90% du noyau UNIX écrit en C
- Utilitaires systèmes écrits en C (shell)
- Normalisation en 1989 par l'ANSI (American National Standard Institute) → C ANSI
- Le C n'est pas lié à UNIX
- Successeur orienté objet : C++

I. Présentation générale de C

- Langage de haut niveau ...
- ... mais proche de la machine :
  - Permet l'accès aux données manipulées par l'ordinateur (bits, octets, adresses)
- Programmation modulaire. Compilation séparée (plusieurs fichiers sources plusieurs fichiers objet)
- Langage faiblement typé

Structure générale d'un programme C

- Un programme C est formé :
  - d'un ensemble de fonctions dans lequel doit figurer une fonction main. Équivalent ADA d'une fonction C : procédure ou fonction
  - éventuellement d'un ensemble de variables globales
  - éventuellement de directives au préprocesseur C
- Fonctions écrites par le programmeur ou issues de bibliothèques
- Unité de compilation : fichier source

Compilation : Obtention d'un exécutable

- En deux étapes :
  - gcc -Wall -ansi -c prog.c
  - gcc -o prog prog.o
- En une étape :
  - gcc -Wall -ansi -o prog prog.c
- -Wall : option de compilation pour avoir les *warnings* du compilateur
- -ansi pour compiler en C ANSI
- -c ne lance que la compilation mais pas l'édition de liens
- -o : si vous n'utilisez pas cette option, l'exécutable se nomme a.out

Modularité

- Programme C : constitué de différents fichiers sources destinés à être compilés de manière séparée et à subir une édition de lien commune
- Trois parties :
  - Fichiers entêtes (suffixe .h)
  - Fichiers de définitions de fonctions C (suffixe .c)
  - Le fichier .c contenant le début du programme : fonction main

Fonction

- Interface :
  - Type et nom de la fonction
  - Types et noms des paramètres
- Bloc (corps de la fonction) :
  - Accolade ouvrante
  - Définitions des variables locales
  - Instructions
  - Accolade fermante
- Instruction :
  - bloc ou expression suivie d'un « ; » ou instruction de contrôle (test, boucle ...)

II. Types et variables

- Rôles de la définition d'une variable :
  - Définition du domaine de valeur de cette variable
  - Définition des opérations possibles sur cette variable
  - Association du nom de variable à une adresse mémoire
  - Initialisation à une valeur compatible avec le domaine de valeurs
- Déclaration :
  - type nom

Types de base

- void : type vide
- int : type entier.
  - Taille : long/short int
  - Signe : unsigned/signed int
  - int = signed int
- char : entier sur 8 bits. de -128 à 127. table ASCII
- Float : flottants
  - Simple précision : float
  - Double précision : double
  - Très grande précision : long double

Taille des types

- Espace occupé en mémoire : dépend de la machine
- char : 1 octet
- short int : 1 mot mémoire. 16 bits
- long int : 2 mots mémoire. 32 bits
- float : 1 mot mémoire
- double : 2 mots mémoire

Portée des variables

- Variables déclarées dans un bloc :
  - Créées à l'entrée du bloc
  - Visibles dans le bloc et les sous-blocs imbriqués
  - Détruites à la sortie du bloc
- Variables déclarées au début du fichier (à l'extérieur d'un bloc) :
  - Créées au début du programme
  - Visibles partout
  - Détruites à la fin du programme

Validité des variables

- 4 classes de variables :
  - AUTOMATIC (par défaut)
  - EXTERN
  - STATIC
  - REGISTER

Détail

- AUTOMATIC : variable déclarée à l'intérieur d'une fonction. Durée de vie = durée de vie de la fonction
- EXTERN : variables globales à toutes les fonctions. Précise les variables définies dans un autre fichier et utilisées dans ce fichier source. Pas d'allocation de mémoire.
- STATIC :
  - variable locale à une fonction MAIS qui existe quand même quand la fonction n'est pas appelée
  - variable globale MAIS mais pas utilisable dans un autre fichier
- REGISTER : avertir le compilateur que la variable sera très utilisée. Si possible, la variable sera placée dans un registre processeur.

Constantes symboliques

- Pour donner une nom à une constante on peut utiliser la directive #define traitée par le préprocesseur C
- #define Nom texte_remplacement
- #define VRAI 1
- #define FAUX 0
- Pas de booléen en C !
- Pas de vérification syntaxique
- Convention : noms des constantes en majuscules

Pas de ;

III. Opérateurs et expressions

- Affectation : opération normale
- Opérateurs unaires
- Opérateurs binaires
- Opérateurs ternaires
- Expression : suite syntaxiquement correcte d'opérandes et d'opérateurs
- Une expression ramène toujours une valeur (même non utilisée)

Opérateurs unaires

- & opérateur d'adresse. Retourne l'adresse de la variable suivant le &
- * opérateur d'indirection sur une adresse. Permet d'accéder à une variable à partir d'une adresse.
- -- opérateur de décrémentation. *var*-- versus --*var*
- ++ opérateur d'incrémentation. *var*++ versus ++*var*
- Sizeof opérateur donnant la taille en octets
- ! not logique (inversion d'une condition)
- - moins unaire (inversion du signe)
- ~ complément à un

Fausse si =0. vrai sinon.
!0 vaut 1 !9 vaut 0

```
int i, j=0;
i = j++ ;
i = ++j ;
```

Opérateurs binaires d'affectation

- Arithmétique +=, -=, *=, /=, %=
- Masquage &=, |=, ^=
- Décalage >>=, <<=
- Exemples :
  - i += 10 ; i = i + 10 ;
  - i += j + 25 ; i = i + j + 25 ;
  - i <<= 4 ; i = i << 4 ; décale i à gauche de 4 positions

Opérateur ternaire

- Met en jeu trois expressions (exp1, exp2, exp3)
- exp1?exp2:exp3
- On commence par évaluer exp1
  - Si sa valeur n'est pas nulle (condition vraie)
  - Alors la valeur retournée est celle de exp2
  - Sinon c'est celle de exp3
- Exemple :
  - z = (a>b)?a:b ;

Précédence des opérateurs

1. Parenthésage
2. Opérateurs unaires
3. Changements de types
4. Multiplicatifs * / %
5. Additifs + -
6. Décalages << >>
7. Comparaisons < <= > >=
8. Egalités == !=
9. Et bit à bit &
10. ou exculsif bit à bit ^
11. Ou bit à bit |
12. Et logique &&
13. Ou logique ||
14. Condition
15. Opérateurs binaires d'affectation
16. Succession

Exercice

Ecrire un programme qui lit un nombre entier et l'écrit accompagné du label 'P' ou 'I'. Proposer au moins deux solutions.

Fonction de lecture sur l'entrée standard :

```
#include <stdio.h>
int i;
scanf("%d",&i);
```

Correction 1

```
#include <stdio.h>
int main(void)
{
int i;
char res;
printf("Saisissez un nombre entier : ");
scanf("%d",&i);
res=(i%2 == 0)?'P':'I' ;
printf("Le nombre %d est %c", i, res);
return (1);
}
```

Correction 2

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

printf("Saisissez un nombre entier : ");
scanf("%d",&i);
printf("Le nombre %d est %c", i, (i%2 == 0)?'P':'I');
return (1);
}
```

Correction 3

```
#include <stdio.h>
int main(void)
{
int i;
char res;
printf("Saisissez un nombre entier : ");
scanf("%d",&i);
res=(i & 1 == 0)?'P':'I' ;
printf("Le nombre %d est %c", i, res);
return (1);
}
```

IV. Instructions de contrôle

- Instructions conditionnelles :
  - if
  - switch
- Instructions répétitives :
  - while
  - for
  - do while

Instructions conditionnelles : if

- Syntaxes du if :
  - if (expression) instruction
  - if (expression) instruction1 else instruction2
- Exemple :
  - if (i % 2 == 0) printf("pair ") else printf("impair");
- Imbrication des if. Le else va avec le if le plus proche non terminé (

```
if (i == 1) inst1
else if (i == 2) inst2
else if (i==3) inst3
...
```

Utiliser des blocs :

```
if (cond1){
    if (cond2)
        inst1
}
else
    inst2 ;
```

Pas de ;
une seule
instruction entre la
condition et le *else*

EXERCICE

- Quelle différence entre

```
if(c1) if(c2) i1; else i2;
```

- Et

```
if (c1) {if (c2) i1;} else i2;
```

1. si c1 et c2 alors i1, si c1 et pas c2 alors i2, si pas c1 alors (quel que soit c2) rien.

2. si c1 et c2 alors i1, si c1 et pas c2 alors rien, si pas c1 alors i2.

Instructions conditionnelles : switch

- But : éviter les imbrications if
- Syntaxe :

```
switch (expression) {
case value1 : inst10
                Inst11
                [break;]
case value2 : inst20
...
[default : inst30
                inst 31]
}
```

- *break;* : en cas d'absence, continue jusqu'au bout les tests
- Non obligatoire après le default mais cohérence de l'écriture
- Exemple :

```
switch (i & 1)
{
    case 0 : printf("%d est PAIR",i); break;
    case 1 : printf("%d est IMPAIR",i); break;
}
```

Boucle WHILE

- *while (exp) inst*

```
while (exp)
{
...
}
```

Sans accolades si une seule instruction :
*while (i<j) i = 2 * i ;*

- exp : test de continuation
- inst : corps de la boucle
- Le corps de la boucle est exécuté tant que le test de continuation est vrai → peut ne jamais être exécuté

Boucle DO WHILE

- *do inst while (exp);*
- inst : corps de la boucle
- exp : test de continuation
- Le corps de la boucle est exécuté jusqu'à ce que le test de continuation soit faux → ...

Boucle FOR

- Trois expressions exp1, exp2, exp3
- *for (exp1;exp2;exp3) inst*
- Exemple :

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

- Absence des expressions : *for(;;) inst*

exp1 : début

exp2 : fin

exp3 : exécution à chaque itération

Exercice

Exprimez la construction

for (exp1;exp2;exp3) inst

avec un while

for (exp1;exp2;exp3) inst

Équivaut strictement à

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

Break et continue

- Pour les instructions for, while, do et switch :
  - *break* provoque l'abandon de la structure et le passage à l'instruction écrite immédiatement derrière
- Pour les instructions for, while et do :
  - Continue provoque l'abandon de l'itération courante et le cas échéant le passage de l'itération suivante
- Boucles imbriquées : c'est la boucle la plus profonde qui est concernée par break et continue

Exemple

- Avec la construction for
- Utilisation de break pour éviter des boucles infinies :

```
for (;;){
    printf("donnez un nombre (0 pour sortir) : ");
    scanf("%d",&n);
    if (n == 0)
        break;
    ...
}
```

EXERCICE

Ecrire un programme qui calcule la somme des n premiers cubes d'entiers avec les 3 boucles possibles.

Ecrire un programme qui calcule le plus grand terme de la série $\sum_{i=1}^{n} i^3$ qui est inférieur à 1000.

V. Types dérivés des types de base : Structures, Pointeurs et tableaux

- Tableaux
- Structures
- Enumérations
- Pointeurs

Tableaux

- Cas simple :
  - *type_de_base nom [nb_elements_optionnel] ;*
  - Exemple : *unsigned long table[10];*
- Premier élément : indice 0
- Sur l'exemple précédent :
  - *table[0], table[1], ..., table[**9**]*
- Expression optionnelle : de type entier et constante (le compilateur doit pouvoir évaluer la taille du tableau en mémoire)

Tableaux à plusieurs dimensions

- Dimensions entre crochets
- Syntaxe :
  - type nom_tab [dim1][dim2]...[dimn]
- Exemple :
  - *int tab[5][10];*
- Accès aux composantes :
  - *printf("%d\n", tab[0][1]);*

Attention

- Impossible d'affecter un tableau à un autre
- Pas de contrôle de dépassement de bornes
- Arrêt sur segmentation fault

Chaînes de caractères

- Tableau de caractères
- Dernier caractère : code nul \0
- Constante délimitée par les "
- char mess[]="bonjour";
- Tableau de 8 caractères (\0 dans mess[7])

Fonctions de manipulation des chaînes de caractères

- *int* ***strlen****(chaîne)* donne la longueur de la chaîne (\0 non compris)
- *char *****strcpy****(char *destination,char *source)* recopie la source dans la destination, rend un pointeur sur la destination
- *char *****strncpy****(char *destination,char *source,int longmax)* idem strcpy mais s'arrête au \0 ou longmax (qui doit comprendre le \0)
- *char *****strcat****(char *destination,char *source)* recopie la source à la suite de la destination, rend un pointeur sur la destination
- *char *****strncat****(char *destination,char *source,int longmax)* idem
- *int* ***strcmp****(char *str1,char*str2)* rend 0 si str1= =str2, <0 si str1<str2, >0 si str1>str2. Idem strncmp

Structures

- Constituants (champs) de types différents
- Champs désignés par un identificateur
- Syntaxe :
  - *struct nom_type {déclaration des champs} ;*

```
struct <identificateur>{
    <type><liste_identificateurs>;
    ...
    <type><liste_identificateurs>;
};
```

  - *struct nom_type {déclaration des champs} liste_variables ;*
- Accès à une composante : *nom_variable.nom_champ*

Définition de types

- Mot-clé typedef
- Syntaxe générale :
  - *typedef type nouveau_type*
- Définition d'un nouveau nom et non d'un nouveau type
- Utilisé pour la lisibilité du code
- Exemple :
  - typedef struct {int jour, mois, annee} date;

Exemples

- Structure personne

```
struct personne{
    char nom[20];
    int age;
};
```

- Déclaration
  - *struct personne p1, p2 ;*
- Accès aux champs :
  - *p1.age = 30;*
  - *printf("%s a %d ans\n",p1.nom,p1.age) ;*

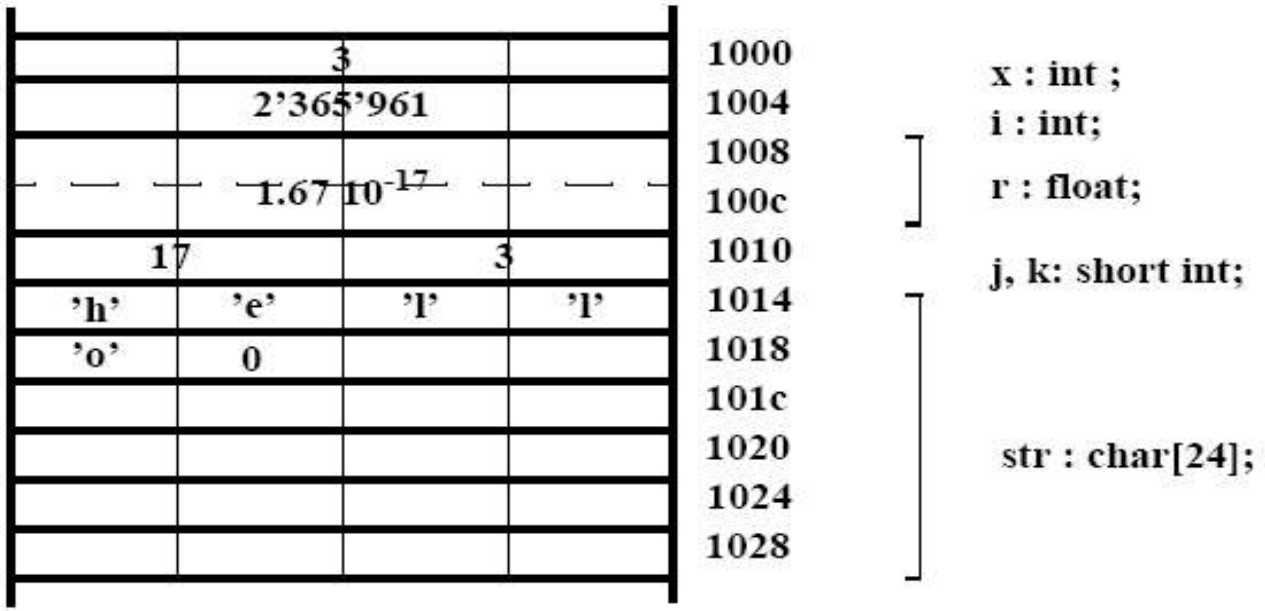
Enumérations

- Liste finie de nombre d'entiers
- Syntaxe de déclaration :
  - *enum nom_enum {liste_enum};*
- Exemple :
  - *enum jour_semaine{lundi, mardi, .., dimanche};*
  - *lundi* vaut 0, mardi 1, etc.
  - *lundi + 2* représente la même valeur que mercredi
- On peut préciser explicitement les valeurs :
  - *enum echap {LIGNE='\n',TAB='\t'};*
- Ou que certaines puis les autres sont déduites par incrémentation

Pointeurs : rappels

- **Mémoire** visualisée comme un tableau d'octets
- **Variable** : série de cases consécutives (nombre de cases selon le type de la variable)
- **Pointeur** : adresse d'un groupe de cases mémoire

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int x, i;
    float r;
    short int j,k;
    char str[24];
    x=3;
    strcpy(str,"Hello");
    x=x+1;
    return (0);
}
```



Pointeurs en C

- Manipulation des **adresses** des objets
- **Opérateur &** sur un objet de type quelconque : fournit l'adresse de cet objet en mémoire
- **Opérateur * :** accès à la valeur stockée dans une case mémoire dont on connaît l'adresse
- Déclaration d'un pointeur sur un objet de type <type> : *<type> *p;*
- **Pointeur NULL** : le pointeur n'est pas initialisé et ne contient pas d'adresse utilisable
- **Type void** : pointeur de n'importe quel type MAIS l'opérateur * ne peut pas s'appliquer. *void *ptr;*

Pointeurs et tableaux

- Déclaration d'un tableau : *<type> t[x];*
- Réservation en mémoire d'un bloc de x objets du type <type>
- t ≡ &(t[0])
- La valeur d'une variable de type tableau est l'**adresse du premier élément du tableau**
- **t* vaut *t[0]*

Tableaux de caractères et chaînes de caractères

- Chaîne de caractères : espace mémoire avec des caractères dont le dernier est '\0'
- Si '\0' absent alors tableau de caractères et non chaîne de caractères
  - fonctions de manipulation de chaînes ne marcheront pas convenablement
- *char tab[]=”xx”; ≠ char *point=”xx”;*
- Lecture sur l'entrée standard :

```
char tab[100];
scanf(”%s”,tab);
```

Tableaux de pointeurs

- Pointeurs de pointeurs
- <type> *pt[n];
- Essentiellement utilisés comme tableaux de chaînes de caractères
  - Tableaux de 10 chaînes de caractères :

*char *ch[10];*

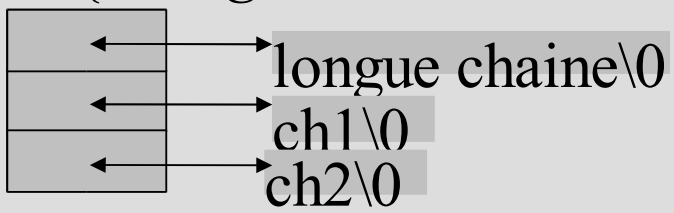
- Pointeur sur une chaîne : pointeur sur un pointeur de caractères.
- *char *ch[10];* et *char **ch;*

Tableaux de pointeurs et tableaux multidimensionnels

- Tableaux multidimensionnels préférables aux tableaux de pointeurs
- Sauf chaînes de caractères :
  - Tableau multidimensionnel de type char : tableau de chaînes de longueur fixe
    - *char tab1[][14] = {"longue chaine","ch1","ch2"} ;*

longue chaine\0 ch1\0 ch2\0

  - Tableau de pointeurs char : possibilité de chaînes de longueur différentes
    - *char *tab2[] = {"longue chaine","ch1","ch2"} ;*

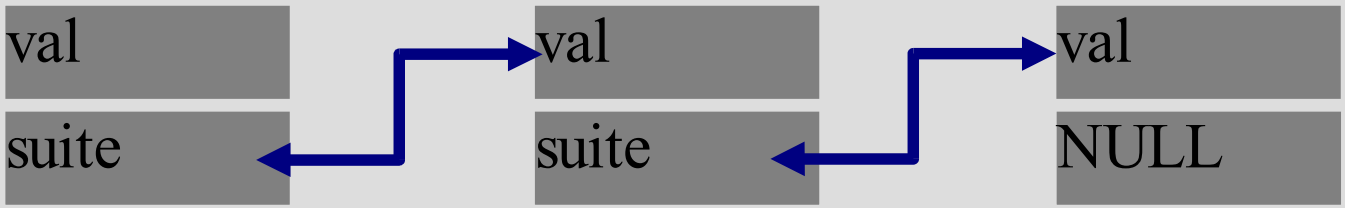


Structures chaînées

- Liste chaînée :

```
struct liste{
        int val;
        struct liste *suite;
    }
```

```
struct liste{
        int val;
        struct liste
          suite;
    }
```



Opérations sur les pointeurs

- pi et qi deux pointeurs
- Si pi pointe sur la variable x alors $*pi$ équivaut à x
- **Affectation entre pointeurs.** $qi = pi;$ avec qi et pi de même type : qi pointe aussi sur x
- **Calculs d'adresse par incrémentation** : $pi{+}{+}$ incrémente l'adresse du nombre d'octets correspondant au type de pi (augmentation de 4 pour les entiers)

Instructions malloc et calloc

- Allocation mémoire
- ***void * malloc(size_t size);*** réserve un espace mémoire de size octets et renvoie l'adresse du début de la zone. Renvoie NULL si impossible de réserver cet espace
- ***void * calloc(size_t size);*** même opération avec initialisation des octets à la valeur 0
- **Conversion de type** avant d'affecter le résultat à un pointeur typé
- Utilisation de ***sizeof*** pour obtenir la taille en octets d'un type

Exemple

```
#include <stdio.h>
#include <stdlib.h>
void main(void){
    int *p, i, n;

    scanf("%d", &n);
    p=(int *) malloc(n * sizeof(int));
    for (i=0;i<n;i++) p[i] = i ;
    printf("p[0]=%d,p[1]=%d,...\n",p[0],p[1]);

}
```

Pour malloc, free, calloc

Intérêt de l'allocation dynamique

Conversion de type

malloc et tableaux de pointeurs

- Espace mémoire non alloué par la déclaration d'un tableau de pointeurs

```
char *ch[10];
ch=(char **)malloc(10*sizeof(char *));
```

- Attention à ne pas provoquer de *segmentation fault*

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

```
int *t[10];
int i;
for (i=0;i<10;i++){
        t[i]=(int *)malloc(sizeof(int));
         *(t[i])=i;
}
```

Instruction free

- Tout espace alloué dynamiquement doit être libéré par l'utilisateur
- Utilisation de la fonction free
- *void * free(void *p);*
- Pas de modification du contenu des octets

Instruction realloc

- Permet de modifier la taille d'une zone précédemment allouée
- *void * realloc(void *p, size_t size);*
- size donne la nouvelle taille du bloc
- Recopie du contenu de l'ancienne zone (dans les limites imposées par les tailles des deux zones)

Exercices

Comment définir une structure d'arbre binaire ?

VI. FONCTIONS

- Rappels :
  - définie par son entête, suivie d'un bloc d'instructions
  - entête : *type_retourné nom_fonction(liste_arguments)* (pas de ;)
  - La liste d'arguments est typée :

    float toto(int a, float b) {bloc}
- Récursivité : une fonction peut s'appeler elle-même

Passage de paramètres

- Par valeur :
  - Évaluation de la valeur puis recopie sur la pile.
  - Les modifications ne sont pas transmises à l'appelant
- Par adresse :
  - Paramètre = pointeur
  - Attention : un nom de tableau est traité comme l'adresse du premier élément
  - Si un tableau est passé en paramètre, la fonction peut accéder et modifier tout le tableau

Exemple : Qu'affiche ce programme ?

```
#include <stdio.h>
void echanger(int a, int b){
        int temp=a;
        a=b;
        b=temp;
          printf("%d, %d\n", a, b);
}
int main(void){
        int nb1,nb2;
        scanf("%d %d",&nb1, &nb2);
        echanger(nb1,nb2);
        printf("%d, %d\n", nb1, nb2);
        return (0) ;
}
```

Passage par référence

- Moyen de modifier une variable de la fonction appelante
- Indication de l'adresse mémoire où sont stockées les valeurs des variables

```
void echanger(int *px, int *py)
{
    int temp=*px;
    *px=*py;
    *py=temp;
}
```

Appel de la fonction :

```
int a,b;
echanger(&a,&b);
```

Tableau en paramètre

- Pas nécessaire d'indiquer le nombre d'éléments
  - *void f(int tab[]);*
- Équivalent à
  - *void f(int *tab);*

La fonction main

- Transmission d'arguments au programme :

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

- argc : nombre d'arguments
- argv : tableau des arguments
- argv[0] : nom de l'exécutable
- atoi : conversion des paramètres vers le type int

EXERCICE

Programmez la fonction factorielle et insérez la fonction dans un programme permettant de passer un entier en paramètre et d'afficher la factorielle :

- de manière itérative
- de manière récursive

Correction : fonction factorielle

- Fonction itérative :

```
int factorielle(int i)
{
  int result;
  for(result=1;i>1;i--) result*=i;
  return(result);
}
```

- Fonction récursive :

```
int factorielle(int i)
{
    if (i>1)
        return(i*factorielle(i-1));
    else return(1);
}
```

Correction : main

```
#include <stdio.h>
#include <stdlib.h>
int factorielle(int i){...}
int main(int argc, char *argv[]){
    int fact ;
    if (argc != 2){
        printf("Usage : %s nombre\n",argv[0]);
        return (-1);
    }
    fact = factorielle(atoi(argv[1]));
    printf("Factorielle de %d : %d\n",atoi(argv[1]),fact);
    return(0);
}
```

VII. Entrées/Sorties

- Réalisées par des **fonctions de la bibliothèque standard** *stdio.h*
- **Formatage des données** et **mémorisation** des données dans une mémoire tampon
- **Flots de données** (data streams) : suites d'octets
- Flots **textes** : suite de caractères ASCII structurée en lignes terminées par '\n'
- Flots **binaires** : données enregistrées sans conversion au format ASCII

Flots standards

- 3 flots standards : stdin, stdout, stderr
- Connectés par défaut au lancement du programme à l'écran pour les sorties, au clavier pour les entrées
- Peuvent être explicitement redirigés vers un fichier ou un périphérique
- Dans un programme, flot déclaré de type FILE*

Fichiers

- 2 types de fonctions de manipulation de fichiers
- Fonctions de niveau 1 :
  - très proches du SE.
  - accès direct aux informations non bufferisées.
  - manipulation sous forme binaire sans formatage.
- Fonctions de niveau 2 :
  - basées sur les fonctions de niveau 1.
  - E/S bufferisées.
  - manipulation binaire ou formatée.

Opérations sur les fichiers (niveau 2)

- **Déclaration** d'un pointeur de fichier : *FILE *mon_fich;*
- **Ouverture** : *FILE * fopen(char *nom, char *mode);*
  - Ouvre le fichier nom. Rend un pointeur sur le flot correspondant ou NULL si échec
  - Valeur mode :
    - “r”, “w”, “a”, “r+”, “w+”, “a+”
    - mode binaire : ajouter b au mode (“r+b”)
- **Fermeture** : *fclose(FILE *f);*
- **Contrôle du tampon** : *int fflush(FILE *f);*
- **Fin de fichier** : *int feof(FILE *f);*

Opérations sur les fichiers : lecture et écriture textuelles

- ***fscanf***(*FILE *f, const char *format, liste_adresses);*
- ***fprintf***(*FILE *f, const char *format, liste_expressions);*
- *int* ***fgetc***(*FILE * f);* ***getchar***()≡*fgetc*(*stdin*);
- *int* ***fputc***(*int c, FILE * f);* ***putchar***(*c*)≡*fputc*(*c, stdin*);
- *char **fgets*(*char *ch, int lgmax, FILE *f*);
- ***gets***(*char *ch*);
- *int* ***fputs***(*char *ch, FILE *f*);
- *int* ***puts***(*char *ch*);

Mode binaire

- Lecture et écriture :
  - *size_t **fwrite**(const void *ptr, size_t t, size_t nobj, FILE *f);*
  - *size_t **fread**(void *ptr, size_t t, size_t nobj, FILE *f);*
- Positionnement :
  - *int **fseek**(FILE *f, long offset, int mode);*
  - Mode : 0 par rapport au début du fichier, 1 par rapport à la position courante, 2 par rapport à la fin du fichier
  - *long **ftell**(FILE *f);*
  - *int **rewind**(FILE *f);*

EXERCICE 1

Ecrire un programme permettant de vérifier qu'un fichier dont le nom est passé en paramètre existe.

CORRIGE

```
#include <stdio.h>
int main(int argc, char *argv[]){
    FILE *in;
    char c;
    if (argc != 2){
        printf(”nbre de parametres incorrect\n”); return(1);
    }
    if ((in=fopen(arg[1],”r”))= =NULL){
        printf(”fichier inexistant\n”); return (2);
    }
    return(0);
    }
```

EXERCICE 2

Ecrire un programme permettant d'afficher le contenu d'un fichier texte dont le nom est passé en paramètre.

CORRIGE

```
#include <stdio.h>
int main(int argc, char *argv[]){
      FILE *f;
      int ok, c, i;
      if (argc != 2){
            printf(”nbre de parametres incorrect\n”); return(1);
      }
      f=fopen(argv[1],”rt”); c=fgetc(f);
      while(c!=EOF){
        putchar(c) ; c=fgetc(f);
      }
      ok = fclose(f); return(0);
      }
```

EXERCICE 3

Ecrire un programme permettant de réaliser une copie de fichiers (cp)

CORRIGE 1

```
#include <stdio.h>
int main(int argc, char *argv[]){
    FILE *in, *out;
    char c;
    if (argc != 3){
        printf(”nbre de parametres incorrect\n”); return 1;
    }
    if ((in=fopen(arg[1],”r”))= =NULL){
        printf(”fichier inexistant\n”);return 1;
    }
    if ((out=fopen(arg[2],”w”))= =NULL){
        printf(”impossible de creer le fichier\n”);return 1;
    }
    while((c=fgetc(in))!=EOF) fputc(c,out);
    return(0);
    }
```

CORRIGE 2

```
#include <stdio.h>

#define PAS_D_ERREUR        0    /* codes conventionnels */
#define ERREUR_OUVERTURE    1    /* à l'usage du système */
#define ERREUR_CREATION     2    /* d'exploitation */

FILE *srce, *dest;

main() {
     char tampon[512];
     int nombre;

     printf("source : ");
     gets(tampon);
     srce = fopen(tampon, "rb");
     if (srce == NULL)
          return ERREUR_OUVERTURE;

     printf("destination : ");
     gets(tampon);
     dest = fopen(tampon, "wb");
     if (dest == NULL)
          return ERREUR_CREATION;

     while ((nombre = fread(tampon, 1, 512, srce)) > 0)
          fwrite(tampon, 1, nombre, dest);

     fclose(dest);
     fclose(srce);
     return PAS_D_ERREUR;
}
```

EXERCICE 4

- Soit une structure permettant de stocker le nom de l'étudiant et ses 3 notes de C.
- Ecrivez un programme C permettant de
  - stocker les données saisies par un utilisateur dans un fichier.
  - relire ce fichier pour afficher les données saisies.

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{ struct art{ char nom[10]; int notes[3];  };
  typedef struct art Art;
  char c ='o'; Art elem; FILE *fich;
  fich = fopen("lefichier.dat","w+b");
  do{ printf("nom?");
      if ( fgets(elem.nom,10,stdin)== NULL) {
        printf("erreur"); exit;
      }
      scanf("%d%d%d", &elem.notes[0],
            &elem.notes[1], &elem.notes[2]);
      c=getchar();
      printf("encore? n pour stop, return pour continuer");
      c=getchar();
      fwrite((void *)&elem, sizeof(Art),1,fich);
      fflush(fich);
  }   while (c!='n');
  printf("fin de la création\n");
  /* relecture du fichier */ rewind(fich);
  fread((void *)&elem, sizeof(Art),1,fich);
  while (feof(fich) ==0){
    printf("%s -->%d,%d,%d\n",elem.nom, elem.notes[0],
                              elem.notes[1], elem.notes[2]);
    fread((void *)&elem, sizeof(Art),1,fich);  }
  fclose(fich);
}
```

VIII. Introduction à la compilation séparée

- Pour les programmes dépassant une certaine taille
- Regroupement des fonctions dans différents fichiers
- Un fichier source contenant la fonction main
- Appel de fonctions déclarées dans d'autres fichiers si déclaration préalable
- Variables et fonctions **EXTERN** : informer le compilateur que les définitions sont présentes dans d'autres fichiers. pas d'allocation de mémoire (définition vs. déclaration)

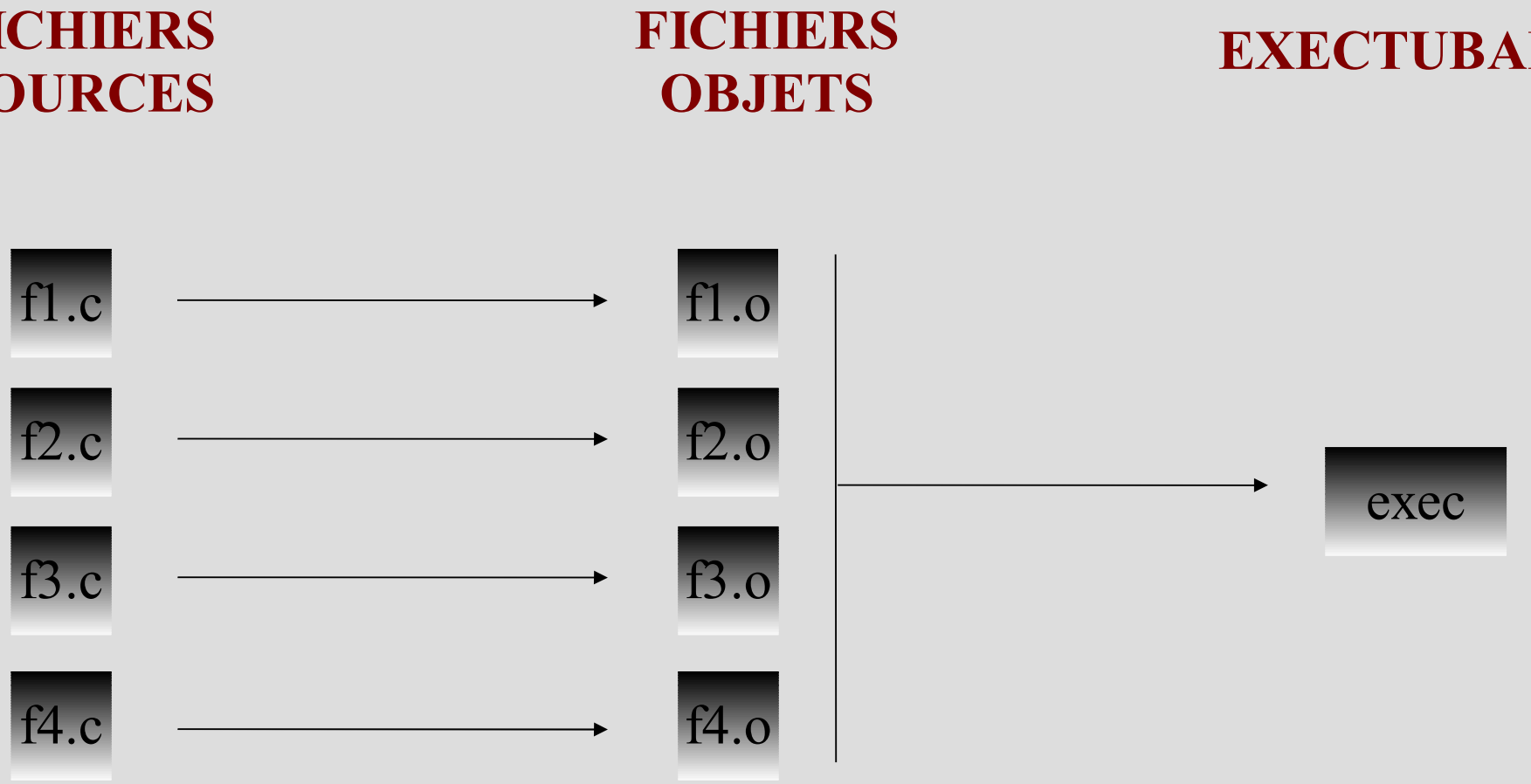
Gestion des différents fichiers

- Plusieurs fichiers sources .c
- Regroupement des déclarations dans une bibliothèque de fonctions (library), fichier d'en-tête .h (header)
- #include “entete.h”
- Compilation : *gcc -o exec f1.c f2.c ...*
- Mais inutile de tout recompiler

⇒ Compilation séparée

Rappels

- Deux étapes : compilation et édition de liens
- **Compilation** : transforme un programme C en langage machine (sans se soucier des appels de procédures)
- **Edition de liens** : Vérification des appels de procédures et laison des appels de procédures à leur déclaration
- Option -c : le compilateur n'effectue pas l'édition de liens


FICHIERS SOURCES
FICHIERS OBJETS
EXECTUBALE
f1.c
f2.c
f3.c
f4.c
f1.o
f2.o
f3.o
f4.o
exec
Compilation
Édition de liens

Pré-processeur

- Toutes les directives au pré-processeur sont traitées avant le début de la compilation
- Directives : #include , #define, ...
- #define : remplacement des occurrences
- Attention au parenthésage !
- #define MAX 15
- #define MAX2 (2 * MAX)
- #define MAX(a,b) (a>b?a:b)
- #define PERMUT(a,b,type) {type t_;t_=a;a=b;b=t_;}

Makefile

- Fichier de dépendances
- Utilitaire make
- Description des fichiers sources nécessaires à l'obtention d'un exécutable
- Recompilation de seulement les fichiers modifiés
- Lignes de commentaires : #.....
- Variables : par convention que des majuscules et _

Structure du Makefile

- Déclaration : NOM_VARIABLE = valeur
- Règles de compilation : 3 parties
  - Nom du fichier à créer (cible/target) suivi de **:**
  - Liste des fichiers dont le fichier cible dépend (si l'un est modifié alors le fichier cible est regénéré)
  - Caractère de **tabulation** suivi de la commande pour générer le fichier cible

Exemple de Makefile

```
#voici un fichier makefile exemple

CFLAGS = -ansi

exec : f1.o f2.o f3.o

    gcc $ (CFLAGS) -o exec f1.o f2.o f3.o

    f1.o : f1.c f.h

    gcc $ (CFLAGS) -c f1.c

    f2.o : f2.c f.h

    gcc $ (CFLAGS) -c f1.c

    f3.o : f3.c f.h

    gcc $ (CFLAGS) -c f3.c
```

EXERCICE

- Ecrivez un programme qui demande à l'utilisateur le rayon d'un cercle puis calcule et affiche la circonférence et la surface associées. Votre programme soit être composé de plusieurs fichiers :
  - princ.c contenant la fonction main
  - pi.h contenant la définition de la constante π
  - surf.c contenant la fonction surface
  - circ.c contenant la fonction circonference
- Ecrire le makefile

Correction

pi.h

```
#define PI 3.14
extern float surface(float);
extern float circonference(float);
```

surf.c

```
#include "pi.h"
float surface(float r){
     return(PI * r * r);
}
```

circ.c

```
#include "pi.h"
float circonference(float r){
     return(PI * 2 * r);
}
```

princ.c

```
#include "pi.h"
int main(){
    float ray ;
        printf("rayon?\n");
    scanf("%f",&ray);
    printf("%f %f\n",
                surface(ray),
          circonference(ray));
    return 0;
}
```

Compilation

```
gcc -c surf.c
gcc -c circ.c
gcc -c princ.c
gcc -o princ princ.o surf.o circ.o
```

makefile

```
CC=gcc
princ : princ.o surf.o circ.o
<TAB>$(CC) -o princ princ.o surf.o circ.o
.o : surf.o circ.o pi.h
<TAB> $(CC) -c surf.o surf.c
<TAB> $(CC) -c circ.o circ.c
```

dépendances

commandes

IX. En vrac ...

- rappels sur scanf
- atoi
- sprintf
- scanf
- strcpy, strcmp
- allocation dynamique de mémoire : malloc
- debuggage

scanf : Saisie de plusieurs valeurs

- *scanf("%d %d", &x, &y);*
- Que se passe-t-il si on saisit **1 2** ?
- Et **1 (entrée) 2** ?
- Et **1, 2** ?
- Quels résultats de saisie avec :
  - *scanf("%d%d", &x, &y);*
  - *scanf("%d, %d", &x,&y);*
  - *scanf("%d , %d", &x,&y);*
  - *scanf("%d\n%d", &x,&y);*

Résultats de saisie

- scanf("%d %d", &x, &y);
- 1 2 Entrée : 1 afffecté à x, et 2 affecté à y.
- 1, 2 Entrée : 1 affecté à x mais rien affecté à y
- scanf("%d, %d", &x,&y);
- 1 2 Entrée : 1 affecté à x mais 2 non affecté à y
- 1,2 Entrée : OK
- 1, 2 OK aussi (scanf ignore les espaces, tabulations et retours charriot après les délimiteurs)
- scanf("%d\n%d", &x,&y);
- Pas de délimiteur après le dernier format !

Exemple

avec *scanf("%d, %d", &x,&y);*

```
#include <stdio.h>

int main() {

  int a,b;

  scanf("%d, %d",&a,&b);
  printf("%d %d\n",a,b);

  return 0;
}
```

```
gcc -o t t.c
116 laurent@lirmm.lirmm.fr:laurent % t1
1 4
117 laurent@lirmm.lirmm.fr:laurent % t
1 2
1 4
118 laurent@lirmm.lirmm.fr:laurent % t
1,2
1 2
119 laurent@lirmm.lirmm.fr:laurent % t
1,    2
1 2
120 laurent@lirmm.lirmm.fr:laurent % t
1 ,2
1 4
```

atoi

- Que renvoient
  - *atoi(“23”);*
  - *atoi(“29abc”);*
  - *atoi(“a”);*
  - *atoi(“2.3”);*

```
#include <stdio.h>
int main() {
  printf("%d %d %d %d\n",atoi("23"),atoi("29abc"),atoi("a"),atoi("2.3"));
  return 0;
}
```

23 29 0 2

- itoa (non ANSI C !!!) ⇒ sprintf

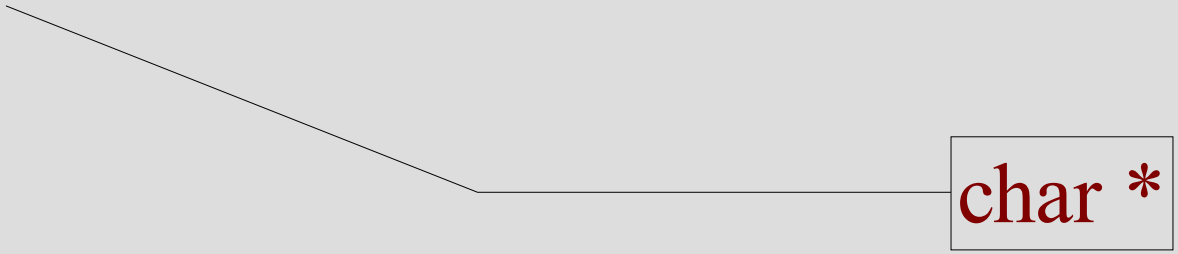
Exemple

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

int main() {
  char str1[] = "124z3yu87";
  char str2[] = "-3.4";
  char *str3 = "e24";
  printf("str1: %d\n", atoi(str1));
  printf("str2: %d\n", atoi(str2));
  printf("str3: %d\n", atoi(str3));
  return 0;
}
```

```
str1: 124
str2: -3
str3: 0
```

sprintf

- 3 arguments
- Convertit le nombre_source en chaîne de caractères dans dest
- Retourne le nombre de caractères dans la chaîne dest
- *sprintf(dest, format, nombre_source);*



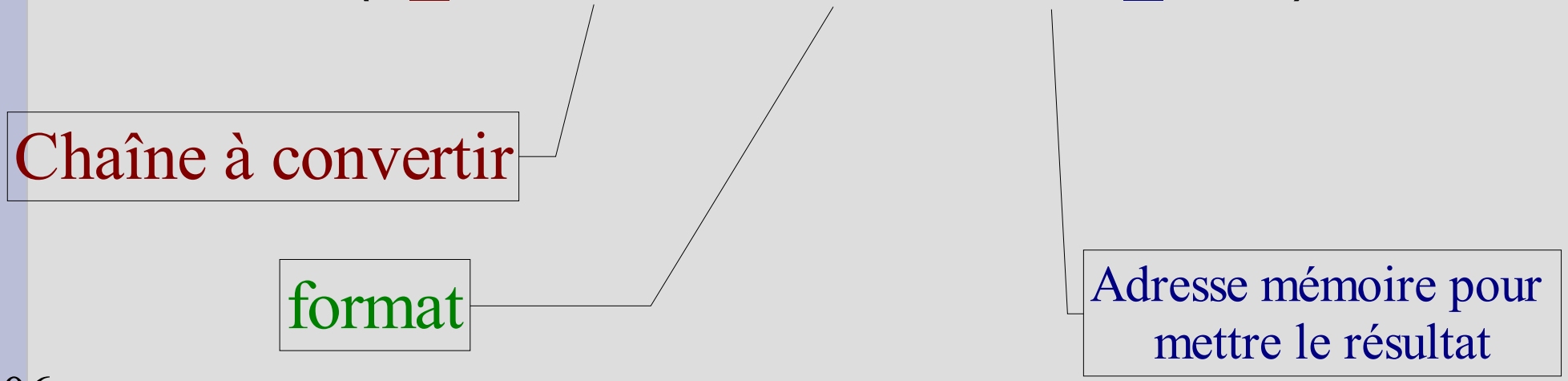
Exemple

```
#include <stdio.h>

int main() {
  char str[10]; /* PREVOYEZ ASSEZ GRAND ! */
  int i;

  i = sprintf(str, "%o", 15);
  printf("15 in octal is %s\n",   str);
  printf("sprintf returns: %d\n\n", i);

  i = sprintf(str, "%d", 15);
  printf("15 in decimal is %s\n", str);
  printf("sprintf returns: %d\n\n", i);

  i = sprintf(str, "%x", 15);
  printf("15 in hex is %s\n",     str);
  printf("sprintf returns: %d\n\n", i);

  i = sprintf(str, "%f", 15.05);
  printf("15.05 as a string is %s\n", str);
  printf("sprintf returns: %d\n\n", i);

  return 0;
}
```

```
15 in octal is 17
sprintf returns: 2

15 in decimal is 15
sprintf returns: 2

15 in hex is f
sprintf returns: 1

15.05 as a string is
15.050000
sprintf returns: 9
```

sscanf

- Conversion d'une chaîne de caractères en différents formats
- 3 arguments
- Retourne le nombre d'objets convertis
- *sscanf(a_convertir, "%d", &nbre_dest);*



Exemple

```
#include <stdio.h>

int main() {
  char* ints = "20, 40, 60";  char* floats = "10.4, 24.66";
  char* hex = "FF, F";

  int i;  int n;   float f;   int h;   char* s;

  i = sscanf(ints, "%d", &n);
  printf("n: %d\n", n);   printf("sscanff returns: %d\n\n", i);

  i = sscanf(floats, "%f", &f);
  printf("f: %f\n", f);   printf("sscanff returns: %d\n\n", i);

  i = sscanf(hex, "%x", &h);
  printf("h: %d\n", h);   printf("sscanff returns: %d\n\n", i);


  s = (char *)malloc(50);
  i = sscanf(ints, "%s", s);
  printf("s: %s\n", s);   printf("sscanff returns: %d\n\n", i);

  return 0;
}
```

```
n: 20
sscanff returns: 1

f: 10.400000
sscanff returns: 1

h: 255
sscanff returns: 1

s: 20,
sscanff returns: 1
```

Exemple

```
#include <stdio.h>

int main() {

  int i,a,b; char op;

  i = sscanf("2+3","%d%c%d",&a,&op,&b);
  printf("operation : %d %c %d\n", a, op, b);
  printf("%d objets convertis\n",i);

  return 0;
}
```

operation : 2 + 3
3 objets convertis

Exemple

```
#include <stdio.h>

int main() {

  int i,a,b; char op;

  i = sscanf("2 + 3","%d%c%d",&a,&op,&b);
  printf("operation : %d %c %d\n", a, op, b);
  printf("%d objets convertis\n",i);

  return 0;
}
```

operation : 2 -4261436
2 objets convertis

strcpy, strcmp

- Rappels sur les chaînes de caractères :
  - **char s[8]={'b','o','n','j','o','u','r','\0'};**
  - **char s[ ]={'b','o','n','j','o','u','r','\0'};**
  - **char s[13]="bonjour";**
  - **char s[ ]="bonjour";**
- ***strcpy(<adresse_tableau_cible>,<adresse_tableau_source>);***
- **Recopie la chaîne source dans cible**
- **char s[8]; strcpy(s,"bonjour");**
- **#include <string.h>**

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

int main(void){
  char *s1,*s2="coucou";
  strcpy(s1,s2);
  printf("%s\n",s1);
  return 0;
}
```

strcmp

- Ecrivez un programme permettant la saisie de 2 chaînes de caractères et l'affichage de
  - ... > ...
  - ... < ...
  - ... == ...

selon les cas

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
    char s1[20]; char s2[20];
    int comp ;
    printf("chaine 1 ?\n");
    scanf ("%s",&s1);
    printf("chaine 2 ?\n");
    scanf("%s",&s2);
    comp = strcmp(s1,s2);
    if (comp < 0)
          printf("%s < %s",s1,s2);
    else
      if (comp > 0)
          printf("%s > %s",s1,s2);
      else printf("%s == %s",s1,s2);
    printf("\n");
    return 0;
}
```

malloc et free

- Allocation dynamique de mémoire **pendant l'exécution**
- stdlib.h
- Complétez le programme pour permettre de réserver un espace mémoire suffisant pour la saisie des notes des étudiants en langage C.

```c
#include <stdio.h>
#include <stdlib.h> /* pour malloc et free */

int main() {
  int n;
  int *ptr;
  int i;

  printf("combien d\'etudiants ? ");
  scanf("%d", &n);

  ptr = ...

  ...
}
```

```c
#include <stdio.h>
#include <stdlib.h> /* pour malloc et free */

int main() {
  int n;
  int *ptr;
  int i;

  printf("combien d etudiants ? ");
  scanf("%d", &n);

  ptr = malloc(n*sizeof(int)); /* allocation mémoire */

  if(ptr!=NULL) {
    for(i=0 ; i<n ; i++) {
       printf("note de l\'etudiant %d ?\n",i+1);
       scanf("%d",(ptr+i));
    }

   printf("\n");
    free(ptr); /* liberation de la memoire allouee */
    return 0;
  }
  else {
    printf("\nEchec allocation - pas assez de memoire.\n");
    return 1;
  }
}
```

Debuggage

- Gdb. Option -g à la compilation
- *gcc -o prog prog.c -Wall -ansi -g* *gdb prog*
- *break main* *break 12* points d'arrêt
- *run*
- *next* : on passe à la suite sans entrer dans le code des fonctions
- *step* : on passe à la suite en entrant dans le code des fonctions
- *display* i : affichage de la valeur de la variable i

Exemple

```
82 @lynx:alaurent % gcc -o t t.c -Wall -ansi -g
83 @lynx:alaurent % gdb t
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.9"...
```

```
(gdb)  help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the
program
user-defined -- User-defined commands
(gdb) break main
Breakpoint 1 at 0x10614: file t.c, line 7.
```

```
(gdb) run
Starting program: /public/alaurent/t

Breakpoint 1, main () at t.c:7
7         i = sprintf(str, "%o", 15);
(gdb) next
8         printf("15 in octal is %s\n",   str);
(gdb) next
15 in octal is 17
9         printf("sprintf returns: %d\n\n", i);
(gdb) continue
Continuing.
sprintf returns: 2

15 in decimal is 15
sprintf returns: 2

15 in hex is f
sprintf returns: 1

15.05 as a string is 15.050000
sprintf returns: 9

Program exited normally.
```