



Être un ingénieur ?

- Présence en cours
- Présence **active** en TP
 - Présence contrôlée
 - Préparer les TPs
- Pas de document aux examens
 - Entretiens
 - Culture générale
- Savoir se documenter
- Rendre des TPs à l'image du travail
 - Guide de style & Génie Logiciel & Commentaires
 - Tests
 - Honnêteté (faute professionnelle)



Plan



- | | | | |
|----------------------------------|---------|---------------------|-----|
| • Outillage & méthodes | 4 | • Fichiers binaires | 167 |
| • Compilation séparée / Makefile | 15 | • Ligne de commande | 160 |
| • Révisions | 50 | • Système | 216 |
| • Pointeurs | | • Arguments var | 198 |
| ▪ de variable | 58, 116 | • SDL 2 | 225 |
| ▪ de fonction | 204 | | |
| • Structures | 95, 185 | | |
| • Macros | 132 | | |

Pré-requis :

- Initiation algo
- Initiation C/UNIX



Un logiciel

- Utilisateurs ?
 - Besoins
 - Spécifications
- Développeurs
 - Analyse // Algorithmique // SDD
 - Développement
 - Tests
 - Maintenance / Evolutions

5



Compilateur



- ISIMA
 - gcc 12.2 [ada] [PC] [VM]
- À la maison :
 - gcc sous une distribution LINUX
 - gcc sous Cygwin / MinGW
 - gcc sous WSL (W10 ou 11)
 - gcc sous mac (brew / macport)
- Visual Studio ?
- Bon éditeur ou EDI (VS code)

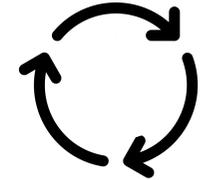
7

- Méthodes
 - Cycle en V
 - Agile

- Spécifications

- Analyse
- Développement
- Tests
 - Unitaires – intégration - fonctionnel
- Déploiement
 - Environnement ?

Cycle de développement



6

Débogueur

- Recherche de bogues à l'exécution
 - Analyse d'un programme en temps réel
 - Point d'arrêt
 - Exécution pas à pas
 - Inspection des variables `printf`
- GDB // DDD

\$ gcc -g

C3



8

Profileur

- Analyse de l'exécution
- Occupation mémoire
 - Eviter les fuites mémoires
- Temps d'exécution
 - Optimisation
- Gprof // valgrind



```
$ gcc -g
```



Gestionnaire de versions

- Développement à plusieurs
 - Diffusion du code
 - Gestion des conflits
- Historique
 - Régression ?
 - Retour en arrière ?
 - Fonctionnalités à venir



Tests unitaires

- Tester des petits bouts des programmes
 - Avant leur intégration
- A la main ou **automatique**
 - Répétabilité ?
 - Régression ?
- Plus ou moins exhaustifs
- Conditions \pm réelles
- Développement orienté Test (TDD)

```
int f(int x)
{
    int r = 1;
    if (x==3)
        r=6;
    else if (x==7)
        r = 5040;

    return r;
}
```

```
TEST(factorielle)
{
    CHECK( 1 == f(0) );
    CHECK( 6 == f(3) );
    CHECK( 5040 == f(7) );
}
```

0! = 1
3! = 6
7! = 5040
13! = 6227020800



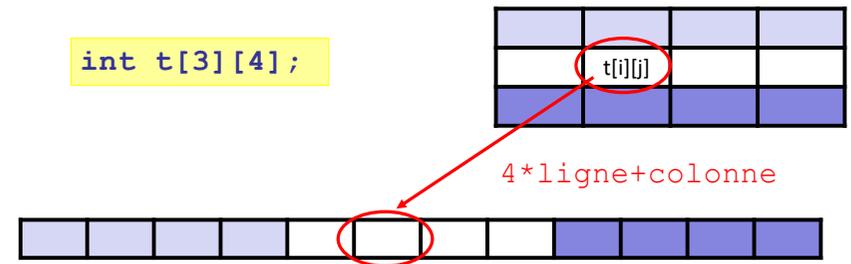
Et encore ?

- Documentation
 - Doxygen
- Analyse statique de qualité de code ?
 - Cppcheck
 - Clang-tidy
 - SonarQube
 - Les compilateurs ☺ (clang & gcc)
- Analyse dynamique avec les « sanitizers »

13



TP C Tableau à deux dimensions



```
void f1(int t[][4]);  
void f2(int t[3][4]);  
void f3(int t[][]);
```

14

Structure d'un programme

- Inclusions des bibliothèques standards et personnelles
- Définition des constantes symboliques et macros
- Déclaration des types personnels
- Déclaration, initialisation, description des variables globales
- **Définition** des fonctions
 - Fonction main()

Tout ce qui est utilisé doit être déclaré sans ambiguïté.
Attention aux mots réservés !

16

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

Mots réservés en C

17

Déclarer ou définir des fonctions ?

- Principe : on ne peut utiliser que des choses connues = déclarées au préalable
- Déclarer, c'est donner le prototype / la signature

NOM + types des paramètres + type de retour

```
float min(float, float);
```

- Définition = prototype et le corps de la fonction

```
float min(float a, float b) {
    return ... ;
}
```

19

```

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

#define C 100

int globale = 300;

int somme(int a, int b)
{
    return a+b;
}

int main() {

    int x = 3, y = 5;

    printf("%d %d", somme(x, y), globale);
    return EXIT_SUCCESS;
}

```

18

```

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

```

```

int main() {

    int x = 3, y = 5;

    printf("%d", somme(x, y));
    return EXIT_SUCCESS;
}

int somme(int a, int b) {
    return a+b;
}

```

20

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

int somme(int a, int b) {
    return a+b;
}

int main() {

    int x = 3, y = 5;

    printf("%d", somme(x, y));
    return EXIT_SUCCESS;
}
```

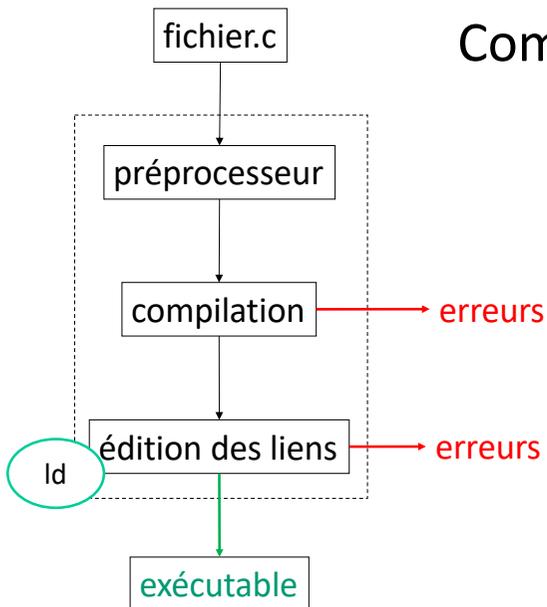


```
#include <stdio.h>
#include <stdlib.h>

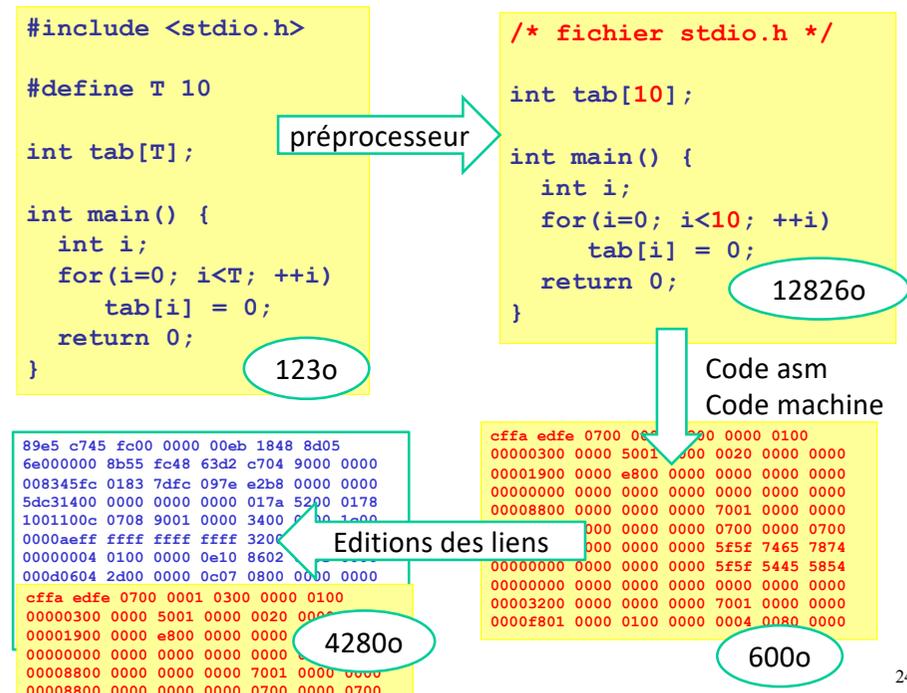
int somme(int a, int b);
//int somme(int, int);

int main() {
    int x = 3, y = 5;
    printf("%d", somme(x, y));
    return EXIT_SUCCESS;
}

int somme(int a, int b) {
    return a+b;
}
```



Compilation (rappel)



#include (1)

- Insère le contenu du fichier (copier/coller)

```
#include "fichier1"
```

à partir du chemin du répertoire courant

```
stdlib.h  
a_moi.h
```

```
#include <fichier2>
```

à partir d'une liste de chemins fixés

/usr/include (système, par défaut)
Tout chemin donné avec -I (i majuscule)



Compilation séparée ?

- Fichiers trop gros / denses ?
 - Pas facile à appréhender / de se déplacer
 - Hétérogène
- Mettre le code dans des fichiers séparés
- Facile ?
- Méthode ?
- Exemple : main() qui appelle f()



#include (2)

- Dans l'ordre lexicographique
- Bibliothèques systèmes d'abord
- Bibliothèques tierces ensuite
- Juste ce qu'il faut
- Caractère séparateur : /



```
#include <stdio.h>  
  
int main() {  
    printf("%d\n", f());  
    return 0;  
}
```

main.c

```
int f() {  
    return 3;  
}
```

f.c

Plusieurs fichiers ?

gcc main.c f.c

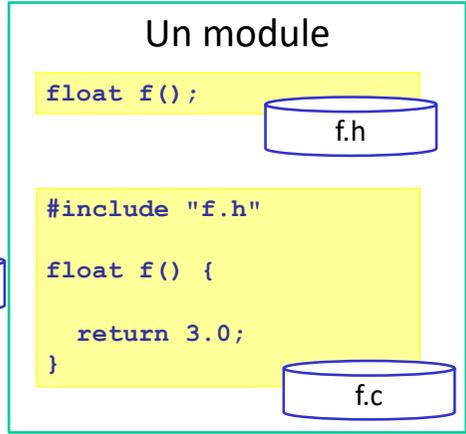
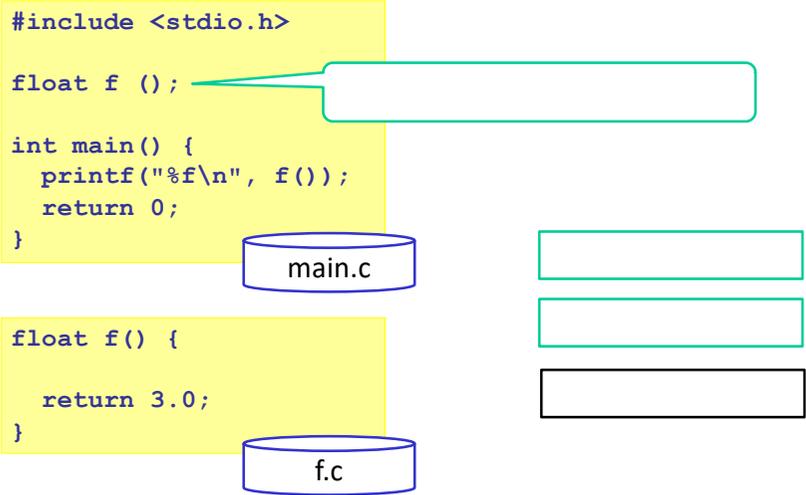
```
#include <stdio.h>  
  
int main() {  
    printf("%f\n", f());  
    return 0;  
}
```

main.c

```
float f() {  
    return 3.0;  
}
```

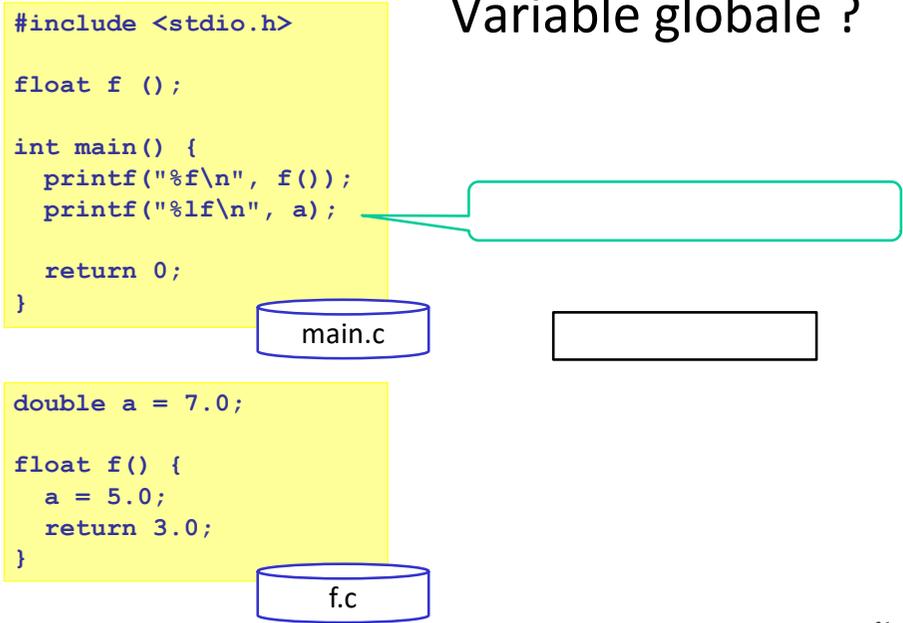
f.c

Module

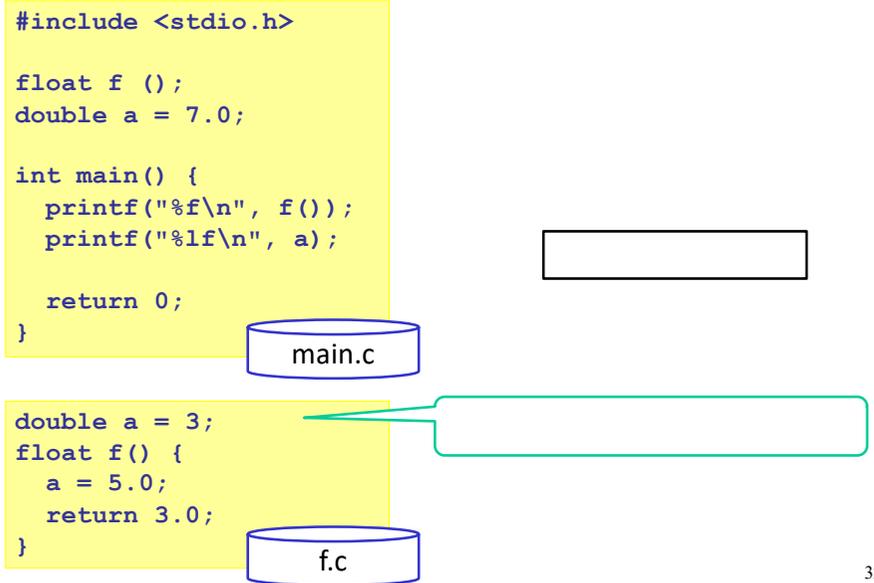


- Séparer en "modules"
 - Regroupement thématique
 - Plus facile à lire
 - Plus facile à tester
 - Réutilisabilité

Variable globale ?



Variable globale ? (2)



Variable globale !

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

int main() {
    printf("%f\n", f());
    printf("%lf\n", a );

    return 0;
}
```

main.c

- Déclaration externe .h
- Définition dans un .c

Un module

```
float f();
extern double a;
```

f.h

```
#include "f.h"

double a = 5;

float f() {

    return 3.0;
}
```

f.c

33

Gardiens

```
#ifndef __LOIC_DONNEES_H__
#define __LOIC_DONNEES_H__

#define TAILLE_MAX 300

extern int variable_globale;

void initialiser(float [], int);
int lireFichier(char[], float[]);
int calculer(float[], int);

#endif
```

~~#pragma once~~



donnees.h

- Bonne pratique
- Inclusion multiple



34

Module - Synthèse

Fichier d'entête

- Extension .h
- **Déclaration** des variables globales
- **Déclaration** des fonctions
 - Prototype / signature
- Types
- Constantes

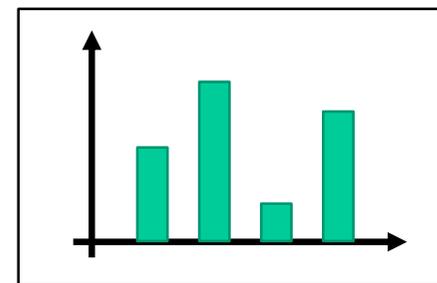


Fichier d'implémentation

- Extension .c
- **Définition** des variables globales
- **Définition** des fonctions
 - Code du .h
 - Fonctions privées

35

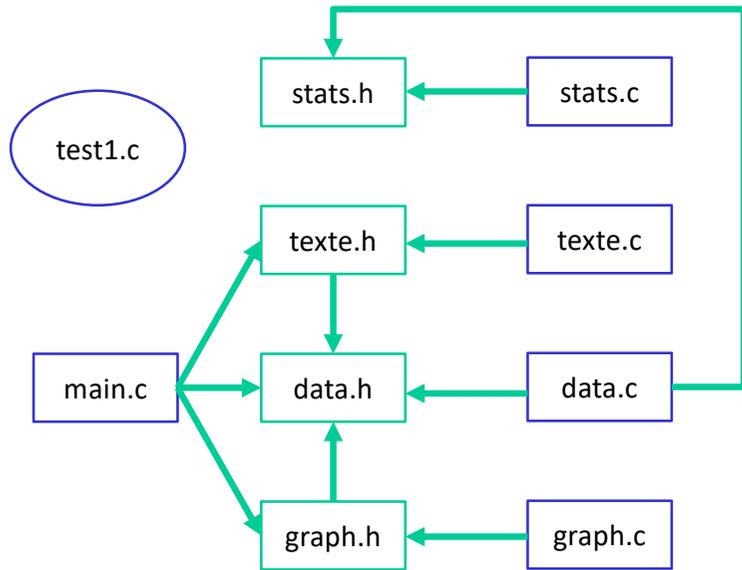
Projet



Un seul fichier = non gérable

- Données
 - Tableaux
 - Listes
 - Lecture de fichiers
- Affichage
 - Graphique
 - Texte
- Opérations
 - Statistiques

36



Compilation globale

```
gcc main.c data.c stats.c graph.c texte.c
-o prog
```

- Pas d'exploitation de la séparation
- Temps de compilation

37

38

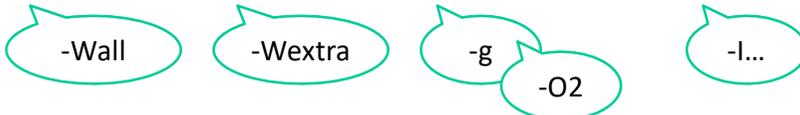
Compilation séparée

- ... des différents fichiers de code
- Compilation des sources en .objets

```
gcc -c main.c
gcc -c data.c
gcc -c stats.c
gcc -c graph.c
gcc -c texte.c
```



```
main.o
data.o
stats.o
graph.o
texte.o
```



39

Édition des liens

- Dernière étape
- Bibliothèques standards et moins standards

```
gcc main.o data.o stats.o
graph.o texte.o -o prog
```

```
main.o
data.o
stats.o
graph.o
texte.o
```

libstdc

libm

Autre ?



40

Comment compiler ?

1. Écrire les différentes lignes de commandes correspondantes (fastidieux)
2. Ecrire un petit programme *shell* qui lance les différentes compilations (mieux ?)
3. Utiliser un programme spécial comme `make` (obligatoire)



41

Commande GNU `make`

- Commande/utilitaire système
- Détection "automatique" des fichiers ayant besoin d'être recompilés
- Edition des liens si nécessaire
- Informations dans un fichier texte
 - `Makefile` ou `makefile`
- Autres utilisations :
 - Nettoyage des fichiers temporaires (objet, ...)
 - Installation dans un répertoire particulier

42

(M | m)*akefile*

- Fichier texte au format très strict
- Constitué de règles (commandes)
 - Fichiers à compiler
 - Fichier exécutable dont il faut faire l'édition de liens
 - Commandes nommées
- Exécuté par

```
$make  
$make règle
```

43

Règle

```
# commentaire  
CIBLE : DEPENDANCES  
<TAB> COMMANDES
```

- Exécution d'une règle
 - Cible qui n'existe pas
 - Une dépendance est plus récente que la cible
 - La première règle est exécutée par `make`
 - Spécifique : `make CIBLE (nom)`

44

```
prog :
<TAB> gcc -o prog main.o
```

```
prog : main.o
<TAB> gcc -o prog main.o
```

```
prog : main.o texte.o ...
<TAB> gcc -o prog main.o texte.o ...
```

```
main.o:
<TAB> gcc -c main.c
```

```
main.o: main.c data.h texte.h graph.h
<TAB> gcc -c main.c
```



```
# compilateur
CC = gcc-12
# options
CFLAGS = -Wall -Wextra -g
LDFLAGS = -lm
# liste des fichiers objets
OBJ = main.o data.o texte.o graph.o \
<TAB> stats.o
# règle de production finale tp :
prog : $(OBJ)
<TAB> $(CC) $(OBJ) -o prog $(LDFLAGS)
# règle de production pour chaque fichier
data.o : data.h data.c stats.h
<TAB> $(CC) -c data.c $(CFLAGS)
stats.o : stats.h stats.c
<TAB> $(CC) -c stats.c $(CFLAGS)
texte.o : data.h texte.h texte.c
<TAB> $(CC) -c texte.c $(CFLAGS)
```

Incomplet

Variable

- Définition

```
NOM = liste de fichiers \
<TAB> suite de la liste
```

- Utilisation

```
$(NOM)
```



Règle *clean*

- Nom usuel pour une règle qui efface les fichiers de travail
 - Exécutable, fichiers objets, fichiers temporaires
 - Pas de dépendance, règle nommée

```
clean :
<TAB> rm -f *.o
```

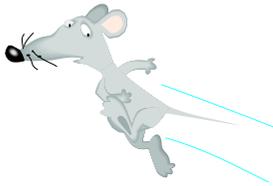
Version bourrine

```
clean :
<TAB> rm -f $(OBJ)
```

Version plus propre, sécuritaire

```
$ make clean
```





Et maintenant ???

- Sauf petit projet, utiliser la compilation séparée
- Gain de temps à utiliser le makefile
 - Copie d'un Makefile qui marche
 - Attention au format du fichier (espaces, tabulations)
- Nombreuses possibilités supplémentaires
 - Génération automatique des dépendances (ZZZ, option -MM ou -MMD)
 - Gestion de configuration
 - Installation d'un programme



RÉVISIONS / VARIABLES

```
#include <stdio.h>

int a;

void f1(double a)
{
    printf("%f\n", a);
}

int main()
{
    a = 3;
    f1(2.000);
    printf("%d\n", a);
    return 1;
}
```



```
#include <stdio.h>

void f1(int a)
{
    a=2;
}

int main()
{
    int a = 3;
    f1(a);
    printf("%d", a);
    return 1;
}
```



```

#include <stdio.h>

void f1(int t[])
{
    t[0] = 2;
}

int main()
{
    int t[3];

    t[0] = 1;
    f1(t);
    printf("%d", t[0]);

    return 0;
}

```



53

```

int gj = 0;
int gi;

int fonction1 (int p1)
{
    int li = 1;

    gj = 4;
    ...
}

int fonction2 (int p1)
{
    int li = 2;
    int gi = 3;
    ...
}

```



Variables globales ?
Variables locales ?

```

int main ()
{
    int li = 3;

    gj = 5;
    return 0;
}

```

54

Variable locale

- Automatique
 - Par défaut
 - Mot-clé `auto` optionnel
 - Créée à chaque appel de fonction
 - Détruite à chaque fin de fonction
 - Pas de valeur par défaut (norme)
- Statique
 - Initialisée au premier passage dans la fonction
 - À 0 en cas d'omission
 - Détruite à la fin du programme

55

```

int fonction1 ()
{
    static int compteur = 0;

    auto int i =1; /* auto est optionnel */
    printf("%d %d", ++compteur, ++i); ...

    return compteur;
}

int main ()
{
    printf("%d", fonction1(0));
    printf("%d", fonction1(0));
    printf("%d", compteur);
    return 0;
}

```

56

Autres variables

- Variables externes explicites
 - Déclaration dans l'entête en général

```
extern int g;
```

- Variable de type registre
 - Pas d'adresse mémoire
 - Accès hyper rapide
 - Vieille optimisation (-O2)

```
register int r;
```



Problèmes rencontrés

- Tableau dynamique ?
- Passage des paramètres par valeur...
 - Pas moyen de changer facilement la valeur d'une variable passée en paramètres



POINTEURS



```
void echange(int a, int b)
{
    int c;

    c = a;
    a = b;
    b = c;
}
```



```
int main ()
{
    int i=1, j=2;

    printf("%d %d", i, j);
    echange(i,j);
    printf("%d %d", i,j);
    return 0;
}
```

Mémoire de l'ordinateur (1)

- Tableau à une dimension
 - Grande dimension (256Mo -> 8Go)
- Unité de base : l'octet
- Gestion plus ou moins automatique
- Partage entre utilisateurs
 - Zones protégées
- Un objet prend plus ou moins de mémoire
 - fonction `sizeof`



Types standards entier

```
char
short int
long
long int

signed (par défaut)
unsigned
```

```
stdint.h
```

```
int8_t      uint8_t
int16_t     uint16_t
int32_t     uint32_t
int64_t     uint64_t
```

```
sizeof(type) ?
```

```
int_least32_t
int_fast16_t
```

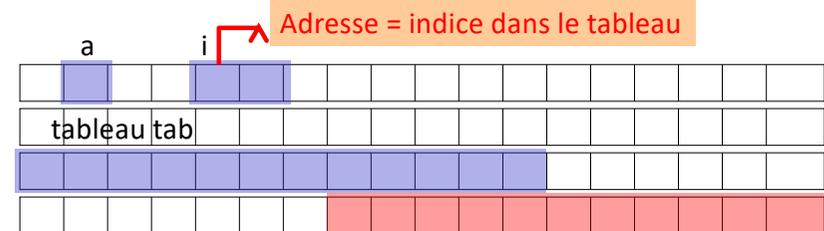


Mémoire de l'ordinateur (2)

- Que trouve-t-on en mémoire ?
 - Code du programme
 - Données du programme (variables)
- Rangement suivant des standards
 - Consultation possible de la mémoire
 - Attention au *segmentation fault*
- Chaque élément a une **adresse**



Mémoire de l'ordinateur (3)



```
char a;
int i;
char tab[12];
sizeof(a)
```

Zone protégée :

- code programme courant
- zone d'un autre utilisateur
- ...



Pointeur ? (1)

- Variable "classique"

```
int i, j;
```

- Opérateur &

- Adresse d'un objet en mémoire (l'index ...)

```
&i et &j
```

- Pointeur

- Variable contenant une adresse
- typé

```
int * p;  
<type> * p2;
```

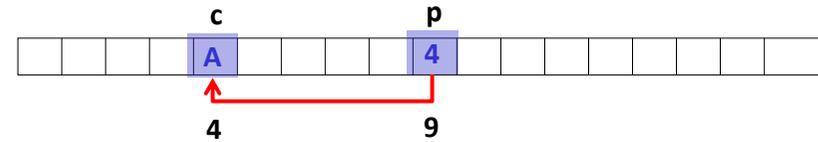


```
int main()  
{  
    int a, b, *p;  
    a = 4;  
    p = &a;  
  
    /* %x : affichage en hexa */  
    printf("contenu de p %p", p);  
    printf("adresse de a %p", &a);  
    printf("contenu pointé %d", *p);  
    printf("contenu %d", a);  
  
    b = *p;  
    (*p)++;  
    printf("valeur de a est %d", a);  
  
    return 0;  
}
```

"Warnings"
(conversion)
%d : décimal
%u : non signé
%x : hexa

Correct :
%p : pointeur

Pointeur ? (2)



```
char c = 'A';  
char * p = &c;
```

Variable (caractère)
Adresse en mémoire

```
printf("%c", c);  
printf("%c", *p);
```

Contenu pointé

* : Opérateur d'indirection / opérateur de **déréférencement** *

Abus de langage : le « contenu du pointeur » ne désigne pas la variable c, p ne contient que l'adresse de c. Le contenu de l'adresse pointée désigne bien c.



Affectation de pointeurs

- Types compatibles

```
int * i, *j;  
float * f;  
  
i = j;  
i = f;
```

- Valeur spéciale NULL

- Le pointeur ne pointe sur rien
- A utiliser avec précaution



Rappel : changer de type

- Cast
- Opérateur de transtypage
- Opérateur de coercition
- Transformer une valeur réelle en une valeur entière avec perte éventuelle d'informations

```
int i;
double d;

i = (int) d;
```

```
int * pi;
void * pv;

pi = (int *) pv;
```



```
void echangekimarche(int *a, int *b)
{
    int c;

    c = *a;
    *a = *b;
    *b = c;
}

int main ()
{
    int i=1, j=2;

    printf("%d %d", i, j);
    echangekimarche(&i,&j);
    printf("%d %d", i,j);
    return 0;
}
```

71

```
float *f = NULL;
int *g = (int *) 0;
char *c = 0;

printf("%f", *f);
```

```
int i;
void * v = &i;
int * j = (int *) v;
```

```
/* définition, fonction du compil */
#define NULL 0
#define NULL (void *) 0
```

70

Pointeurs & tableaux (1)

- Un tableau est un pointeur **constant**

```
char tab[10];
```

- Adresse du premier élément du tableau

```
char *p = &tab[0];
tab ?
```

```
printf ("%p %p %p ", tab, &tab[0], p);
```



72

Arithmétique

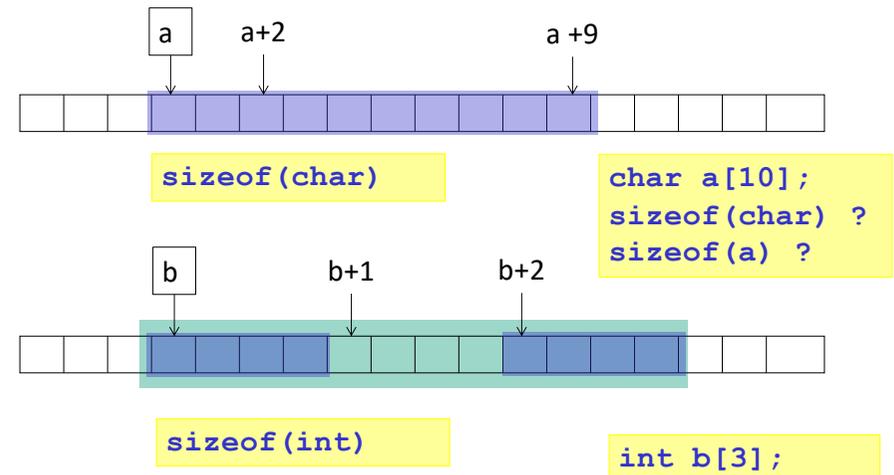


POINTEUR' = POINTEUR + DEPLACEMENT
 Adresse adresse

- Arithmétique intelligente
 - Typée
- Nombre limité d'opérations
 - Affectation (=)
 - Addition, soustraction (+, -, +=, -=)
 - Comparaison (==, !=, <, >)



Arithmétique "intelligente"



Pointeurs & tableaux (2)

- Deux notations : élément *i*

```
tab[i]
*(tab+i)
*(p+i) avec p=&tab[0]
```

- Élément *i*+1 ?

```
p = &tab[i];
++p;
=> *p
```

++tab



Pointeurs & chaînes

```
char s1[] = "chaine1";
char * s2 = "chaine2";
```

- 2^{ème} cas : pointeur + chaîne en mémoire
- *string.h* : Fichier d'entêtes des fonctions sur les chaînes de caractères



strlen()

```
int compteur3(char * chaine)
{
    char * s = chaine;

    while (*chaine != '\0')
        ++chaine;

    return chaine - s;
}
```

77

78

strcpy()



Mémoire "typée"

```
sizeof(char)

sizeof(int)
sizeof(long int)

sizeof(float)
sizeof(double)

sizeof(int *)
sizeof(double *)
sizeof(void *)
```

81

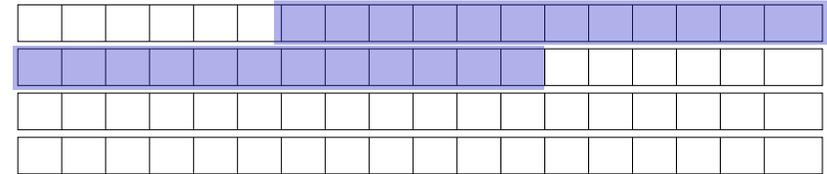
malloc()

- Allocation d'un bloc contigu de mémoire
 - Erreur, si le bloc est trop grand
- Demande d'un nombre d'octets à calculer
Nombre d'éléments x Taille d'un élément
- Retour : pointeur sur le début du tableau
 - NULL si l'allocation est impossible

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

83

Allocation dynamique de mémoire ?



- Allouer de la mémoire = réserver une zone mémoire contiguë
 - Pas toujours possible
- Dynamique : en cours d'exécution de programme (et non au moment de la compilation)

82

Libérer la mémoire (1)

- Obligatoire de rendre la mémoire demandée au système
- Ce n'est PAS AUTOMATIQUE
- Paramètre : un pointeur VALIDE
 - Donner la taille d'allocation n'est pas utile
 - Tester que le pointeur est non NULL même si ...

```
void free(void * pointeur);
```

84

Libérer la mémoire (2)

- Oublier de rendre la mémoire =
 - Fuite mémoire
 - Vers un redémarrage de la machine
- Conseil : mettre le pointeur à NULL si la libération n'est pas faite à la fin du programme

```
if (p) free(p);  
p = 0;
```

- Outils de détection : debugger, **valgrind** ...



calloc()

- Allocation mémoire + initialisation à 0
 - Connaître la taille d'un élément pour le mettre à 0
 - `calloc()` prend plus de temps que `malloc()`
 - **Mais toujours plus rapide** que `malloc()+memset()` sur systèmes actuels
- Paramètres différents de `malloc()`
 - Nombre d'éléments / Taille d'un élément
- Retour : pointeur sur le début du tableau
 - NULL si l'allocation est impossible

```
void * calloc (size_t nb_elem, size_t taille);
```



```
int  taille;  
int * tableau;  
  
printf("Taille du tableau ? ");  
scanf("%d", &taille);  
  
tableau = (int *) malloc (taille*sizeof(int));  
  
if (tableau == NULL)  
{  
    printf("allocation mémoire impossible");  
} else {  
    /* l'allocation mémoire a eu lieu */  
    /* ...                               */  
  
    free(tableau);  
}
```

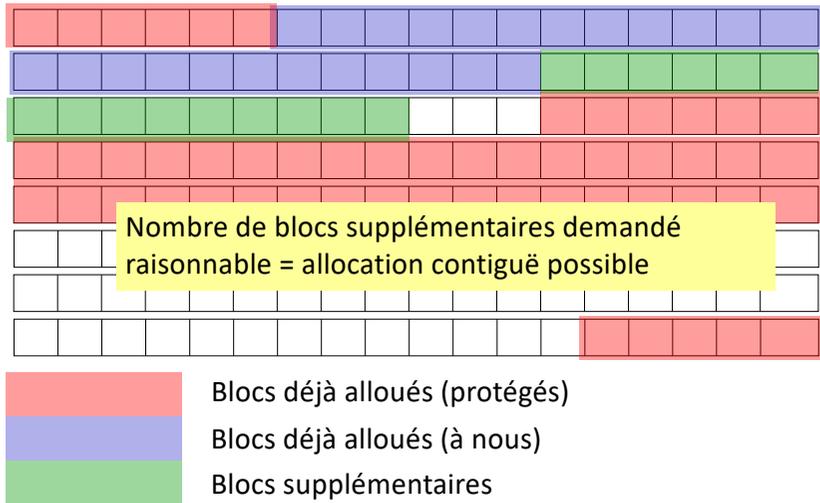


realloc()

- Redimensionner la taille d'un tableau en mémoire
- Problème : demander plus de mémoire
 1. Ajout au bloc précédent
 2. Allocation d'un nouveau bloc et déplacement du bloc initial
 3. Plus assez de mémoire disponible
 - Fragmentation de la mémoire

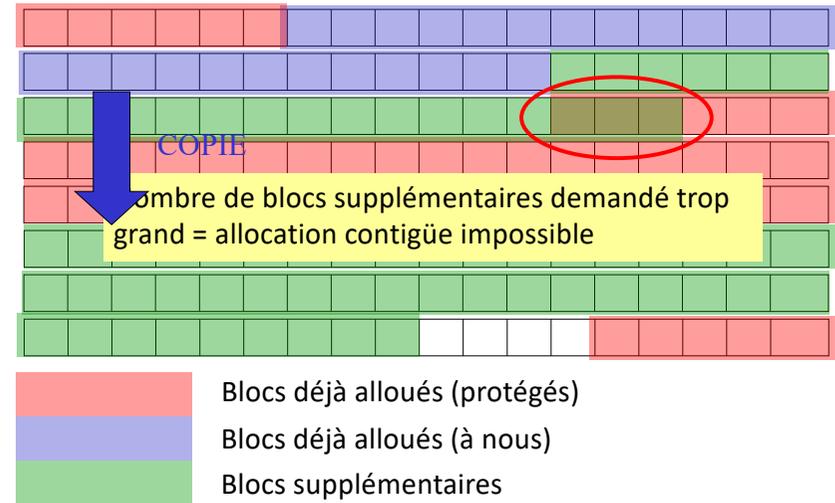


Cas 1



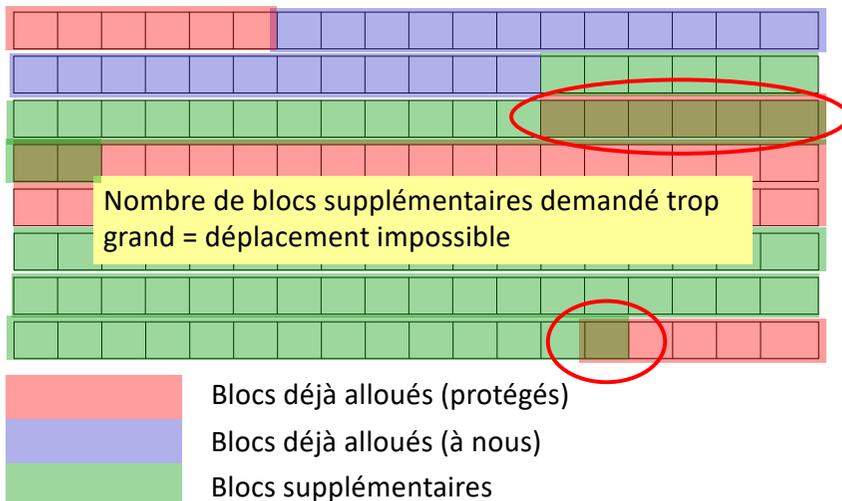
89

Cas 2



90

Cas 3



91

Allocation dynamique / Résumé

```
void * malloc (size_t taille);
void * calloc (size_t nb_elem, size_t taille);
void * realloc(void * pointeur,
               size_t nouvelle_taille);
```

```
void free(void * pointeur);
```

```
void * /* type pointeur universel */
      /* transtypage(cast) obligatoire */
size_t /* type entier positif */
```

```
NULL
```

92

Tableau de pointeurs

- Pointeur = un type de variable
- Possibilité d'en faire des tableaux

```
int * tableau [80];  
int i = 10;
```

```
tableau[0] = &i;  
printf("%d", (*tableau[0]));
```

93

Erreurs typiques

```
char * s;  
s = "chaine2";
```

```
char * s;  
scanf("%s", s);
```

```
char s1[10] = "chaine1";  
char s2[10];  
s2 = s1;
```

94

STRUCTURE



96

Problème ...

- On ne peut retourner **qu'un seul élément** par fonction

```
return variable;
```

- En algorithmique

```
retourne (élément1, élément2);
```



Exemple hypothétique

- Gérer un concours de poker au BDE
 - surnom : 80 caractères
 - score (unité : EuroZZ) : entier

Jouer en ligne est dangereux pour la santé ... et le porte monnaie



Réalisation

- Solution à deux balles :
 - 2 tableaux
 - Nom
 - Score



```
#define TAILLE 30
int score [TAILLE];
char surnoms [TAILLE][80];
```

- Solution propre
 - Un tableau d'un type complexe



Type complexe ?

- Réunir des informations dans un tout
 - Créer un **nouveau type** de données
 - Personnalisé avec 2 champs :

```
struct joueur {
    char nom[80];
    int score;
};
```

- Ne pas oublier les gardiens dans fichier entête
 - Redéfinition si inclusion multiple



Manipulations de base

- Déclaration de variable

```
struct joueur zz;
```

- Accéder à un champ d'une variable

```
zz.score = 0;
scanf("%s", zz.nom);
```

- Taille en mémoire du type complexe

```
sizeof(struct joueur);
```



Tableau de structures

```
struct joueur joueurs[50];
int i;

for(i = 0; i < 50; ++i)
{
    joueurs[i].score = 0;
    strcpy(joueurs[i].nom, "");
}
```

101

typedef

- Renommer un type existant
 - L'ancien nom est toujours disponible
1. Simplifier le nom des structures
 2. Eviter les erreurs avec les pointeurs

```
typedef ancien_nom nouveau_nom;
```

103

Pointeur de structure

- Déclaration d'un pointeur

```
struct joueur * psj = &zz;
```

- Accéder à un champ (pointeur non NULL)

```
(*psj).score      ↔      psj->score
```

- Allocation dynamique

```
psj = (struct joueur *)
      malloc(50*sizeof(struct joueur));
```

102

typedef & structures (1)

```
struct joueur{
    char nom[80];
    int score;
};
typedef struct joueur joueur_t;
```

```
struct joueur zz1;
joueur_t zz2;
```

```
struct joueur * pzz1;
joueur_t * pzz2;
```

typedef & structures (2)

- Condenser les deux écritures...

```
typedef struct joueur {  
    char nom[80];  
    int score;  
} joueur_t;
```

```
struct joueur zz1;  
joueur_t      zz2;
```



Structure autoréférentielle (1)

- Structure qui contient un pointeur sur un élément de même type

```
typedef struct joueur {  
    char          nom[80];  
    int           score;  
    struct joueur * suivant;  
} joueur_t;
```



typedef & structures (3)

- Simplifier ?

```
typedef struct {  
    char nom[80];  
    int score;  
} joueur_t;
```

```
struct {  
    char nom[80];  
    int score;  
} quoi;
```

- Structure "anonyme"

```
struct joueur zz1;
```

```
joueur_t      zz2;
```



Structure autoréférentielle (2)

- Inversion des déclarations possible

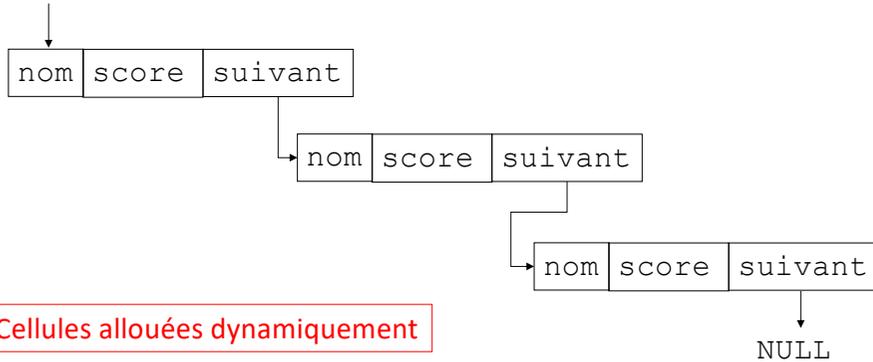
```
typedef struct joueur joueur_t;  
struct joueur {  
    char          nom[80];  
    int           score;  
    joueur_t *    suivant;  
}
```

- Pas de structure anonyme



Liste chaînée

Tête de la liste chaînée ?



Tête de la liste chaînée

1. Un pointeur simple

- NULL liste vide
- Non NULL pointe sur le premier élément

```
joueur_t * ptete;
```

2. Une tête fictive

- Une structure dont on ne se sert pas du contenu
- Seul le champ SUIVANT est utile

```
joueur_t tetefictive;
```

TD : liste chaînée

- Liste vide ?
 - Liste à un élément ?
 - Liste quelconque ?
- 
- Insertion d'un élément en tête
 - Insertion d'un élément en fin
 - Pointeur de fin
 - Insertion / Suppression quelconque
 - Opération hors fonction ou appel de fonction

Liste vide



Liste à 1 élément



Liste à 2 éléments

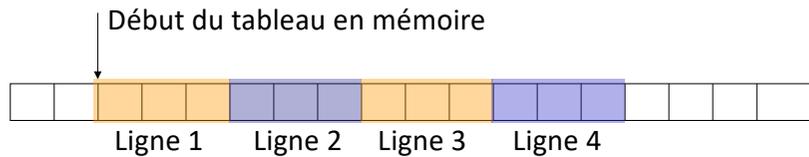


Tableau multidimensionnel statique

- Tableau de 4 lignes et 3 colonnes

```
int tableau [4][3];
```

- Stockage contigu, ligne par ligne



```
int tableau [][3]
```



Tableau à 2 dimensions dynamique ?

1. Implémentation avec un tableau à une dimension

- Taille de la ligne : n $i * n + j$
- Trouver l'élément de la ligne i et colonne j :

```
int * tab1;
```

2. Implémentation avec une dimension maximale et une dimension libre

```
int * tab2[MAX];
```



Accéder à un élément

- Tableau à une dimension

```
ligne[i]  
*(ligne + i)
```

- Tableau multidimensionnel

```
tableau[1][2]  
*(tableau[1]+2)  
*(*(tableau+1)+2)
```

Attention à la lisibilité !!!

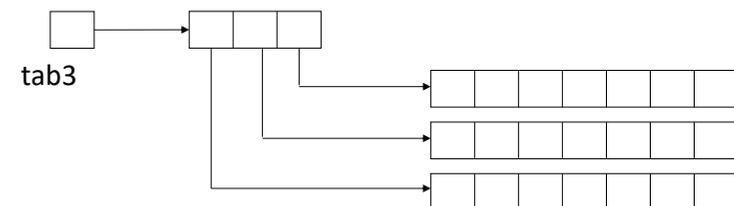


Tableau à 2 dimensions dynamique

- Choisir la représentation par lignes ou par colonnes

```
int ** tab3;
```

- Allouer un premier tableau de pointeurs
 - Un élément pointe vers une ligne ou une colonne
- Allouer chaque ligne/colonne



Equivalence ?

```
int tab[]
```



```
int * p
```

```
int mat[][N]
```



```
int ** v
```

- Zone mémoire contiguë
- Lignes de même taille
- Statique
- Zones mémoires PAS contiguës
- Lignes de taille différente
- Eventuellement non allouées
- Dynamique



Tableau dynamique / synthèse

```
int * tab1;
```

- ☺ dynamique
- ☹ calcul

```
int * tab2[MAX];
```

- ☺ moins de pointeurs
- ☹ pas complètement dynamique

```
int ** tab3;
```

- ☺ dynamique
- ☺ vrai tableau
- ☹ complexe

122

```
int k = 2;

void modifier(int * p)
{
    p = &k;
}

int main()
{
    int i = 3;
    int * p = &i;

    modifier(p);
    printf("%d", *p);

    return 0;
}
```

```
int k = 2;

void modifier(int ** p)
{
    *p = &k;
}

int main()
{
    int i = 3;
    int * p = &i;

    modifier(&p);
    printf("%d", *p);

    return 0;
}
```

123

typedef

- Utiliser typedef pour éviter un maximum d'erreurs avec les pointeurs
- "Créer" un type pointeur
 - Manipuler un pointeur comme une variable pour oublier les pointeurs de pointeurs

```
typedef int * pint;
```



124

Modifier un pointeur ? (1)

```
void echange2pointeurs (int **a, int **b)
{
    int * c;

    c = *a;
    *a = *b;
    *b = c;
}
```

Echanger deux pointeurs

```
int * t1 = ...;
int * t2 = ...;
echanger(&t1, &t2);
```

```
void echange2pointeurs(pint* a, pint *b)
{
    pint c;

    c = *a;
    *a = *b;
    *b = c;
}
```

125

```
void liberer(joueur_t * tete) {
    /* ... */
    if (tete) free(tete);
    tete = 0;
}
```

Liste chaînée avec
vrai pointeur en tête
Réutiliser le pointeur

```
int main() {
    joueur_t * tete = NULL;
    ajouter_element(&tete, element);
    /* utiliser tete */
    liberer(tete);
    ajouter_element(&tete, element);
    /* utiliser tete */
    liberer(tete);
}
```

126

Modifier un pointeur ? (2)

```
void liberer(joueur_t ** tete) {
    /* ... */
    if (*tete) free(*tete);
    *tete = 0;
}
```

Liste chaînée avec
vrai pointeur en tête
Réutiliser le pointeur

```
int main() {
    joueur_t * tete = NULL;
    ajouter_element(&tete, element);
    /* utiliser tete */
    liberer(&tete);
    ajouter_element(&tete, element);
    /* utiliser tete */
    liberer(&tete);
}
```

127

```
typedef joueur_t * pjoueur_t;

void liberer(pjoueur_t * tete) {
    /* ... */
    if (*tete) free(*tete);
    *tete = 0;
}
```

Cacher le double pointeur ..

```
int main() {
    pjoueur_t tete = NULL;
    ajouter_element(&tete, element);
    /* utiliser tete */
    liberer(&tete);
    ajouter_element(&tete, element);
    /* utiliser tete */
    liberer(&tete);
}
```

128

const (1)

- S'engager à ne pas modifier une variable
 - Valeur fixée à l'initialisation

```
/* notations équivalentes */  
const float pi = 3.14;  
float const pi = 3.14;
```

- Ne peut pas servir pour initialiser un tableau

```
const int I = 100;  
float tab[I];
```

(sauf depuis la norme C99)



const (2)

- Pointeur

```
const char * sz1;  
char const * sz2;
```

- Pointeur

```
char * const sz3 = ...;
```

- Pointeur

```
char const * const sz4 = ...;
```



const (3)

```
char a = 'a';  
char b = 'b';
```

```
const char * p1 = &a;  
*p1 = 'c';  
p1 = &b;
```

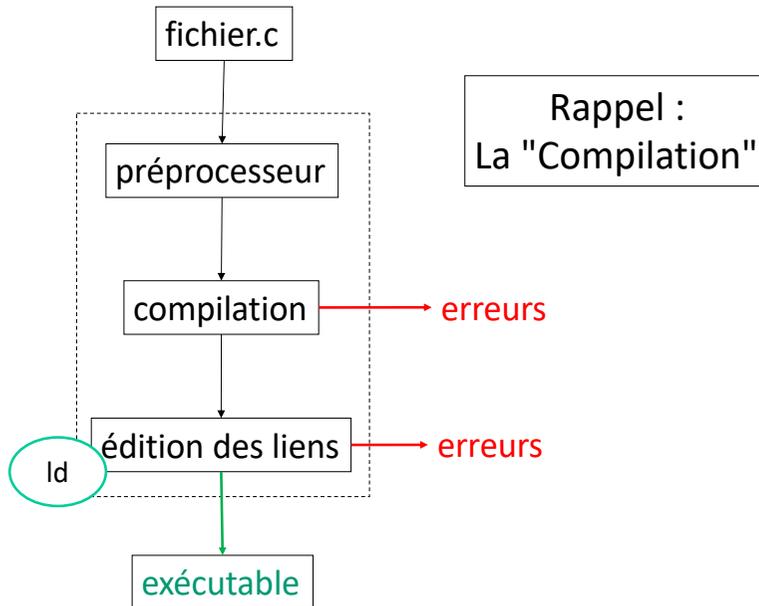
```
char * const p2 = &a;  
*p2 = 'd';  
p2 = &b;
```



MACROS

1. Type constant
2. Type fonction





133

Macros

- Synonymes
 - Macro-définition
 - Macro-commande
 - Macro-instruction
- 2 types de macros
 - Type constant
 - Type fonction

134

Macro de type constant (1)

- Synonymes
 - Macro définition
 - Constante symbolique

```

#define N      100
#define PI     3.14
#define EPSILON 1E-6
#define NOM    "loic"
  
```

PAS DE NOTION DE TYPE

```

#define nom texte_replacement
  
```

135

<code>float tab1[N];</code>	
<code>#define N 100</code>	
<code>float tab2[N];</code>	
<code>#undef N</code>	
<code>#define N 200</code>	
<code>float tab3[N];</code>	
<code>#undef N</code>	
<code>float tab4[N];</code>	

- Validité
 - De la déclaration à la fin du fichier
 - Sauf #define
 - Sauf #undef
- Evaluation par le préprocesseur
 - Inconnu du compilateur

136

Macro de type constant (3)

- Constante sans valeur
- Utile pour la compilation conditionnelle
 - #ifdef, #ifndef, #endif,
 - #else, #elif

```
#define VARIABLE    PAS DE VALEUR
```

- Gardiens

137

Macro de type fonction (1)

- Macros avec paramètre[s]

```
#define NOM(paramètres) texte
```

- Syntaxe
 - Pas de point-virgule (sauf...)
 - Pas d'espace entre le nom et la parenthèse
 - Nom en majuscules

139

Calcul de x^2 ?

```
printf("%f", carre(12.5));
```

```
printf("%d", carre(5));
```

```
double carre_double(double)
```

```
int    carre_int(int)
```

```
printf("%f", carre_double(12.5));
```

```
printf("%d", carre_int(5));
```

138

Calcul de x^2 !

```
#define CARRE1(A) A*A
```

```
int i = 2;  
printf("%d", CARRE1(i));
```

```
printf("%f", CARRE1(5.1));
```

Usage des macros

- Simplifier un code => générique
 - Code plus simple à écrire ou à lire
 - Parfois plus dur à déboguer
- Eviter la duplication de code
 - Chercher / Remplacer par le préprocesseur
 - Une définition pour le développeur
- S'affranchir des types

141

Effets de bord ?

```
#define CARRE1(A) A*A
```

```
int i=2;  
printf("%d", CARRE1(i+1));  
/* résultat ? */
```

143

Macro de type fonction (2)

- Substitution de texte
- Ce ne sont PAS des fonctions !
 - Pas d'appel de fonction => plus rapide
- Attention
 - Aux messages d'erreur du compilateur
 - Aux effets de bord
 - Parenthésage systématique



142

Parenthésite aiguë

```
#define CARRE2(A) (A*A)
```

```
#define CARRE3(A) ((A)*(A))
```

Ne règle pas tout !

144

Exemple 2a

```
#define MAX1(A,B) A>B?A:B
```

```
int i = 2;  
int j = 3;  
printf("%d", MAX1(i,j));
```

145

Exemple 2b

```
#define MAX1(A,B) A>B?A:B
```

```
int i = 2;  
int j = 3;  
printf("%d", 6+MAX1(i,j));
```

146

Exemple 2c

```
#define MAX2(A,B) ((A)>(B))?(A):(B)
```

```
int i = 2;  
int j = 3;  
printf("%d\t", 6 + MAX2(i,j));  
printf("%d\n", MAX2(i++, j++));  
printf("i = %d j = %d\n", i, j);
```

147

Avec une fonction ?

- Si l'on veut calculer le maximum avec une fonction, il faut autant de fonctions qu'il y a de type
- Identification de paramètres
 - Paramètres évalués une fois

```
int max_int(int a, int b)  
{  
    return a > b ? a : b;  
}  
max(i++, j++) renvoie le bon résultat
```

148

Allègement de syntaxe

```
#define MALLOC(TYPE,NOMBRE) \  
((TYPE *)malloc(sizeof(TYPE)*NOMBRE))
```

```
int * pi, * pj;  
float * pf;
```

☺ pas de pointeur de pointeur
☹ attention aux messages du compilateur

```
pi = MALLOC(int, 10);
```

```
pf = MALLOC(float, 100);
```

```
pj = MALLOC(float, 100);
```

149

Écrire une "fonction" d'allocation

```
void allouer1(void ** pointeur,  
              int nb, int taille) {  
    *pointeur = malloc(nb*taille);  
}
```

```
void * allouer2(int nb, int taille) {  
    return malloc(nb*taille);  
}
```

```
int *pi;  
  
allouer1(&pi, 10, sizeof(int));  
pi = (int *) allouer2(10, sizeof(int));
```

150

Allouer avec erreur

```
void allouer(void ** pointeur,  
            int nb, int taille) {  
    if ((*pointeur = malloc(nb*taille))  
        == NULL) erreur();  
}
```

```
int *pi;  
allouer(&pi, 10, sizeof(int));
```

```
#define ALLOUER(PTR,NB,TYPE) \  
if ((PTR = (TYPE *)malloc(NB*sizeof(TYPE))) \  
    == NULL) erreur()
```

```
int *pi;  
ALLOUER(pi, 10, int);
```

151

Ecrire une macro complexe

```
#define MACRO(ARG1, ARG2) { \  
    fonction1(ARG1); \  
    fonction2(ARG2); \  
}
```

```
if (a!=b) fonction(b) ; else fonction(a);
```

```
if (a!=b) MACRO(a, b) ; else fonction(a);
```

```
#define MACRO(ARG1, ARG2) do { \  
    fonction1(ARG1); \  
    fonction2(ARG2); \  
} while (0)
```

Ne compile pas !

152

Aller plus loin... opérateur

- Entoure l'élément de guillemets
- Note : "A" "B" ⇒ "AB"

```
#define DUMP1(X) printf("%s == %d", #X, X)
```

```
#define DUMP2(X) printf(#X " == %d", X)
```

```
int a = 1;  
DUMP1(a); /* affiche a==1 */  
DUMP2(a); /* affiche a==1 */
```



Aller plus loin... opérateur ## (1)

- Concatène deux éléments

```
#define COLLER(A,B) A##B
```

```
COLLER(A, 1) ⇒ A1
```

```
COLLER(COLLER(A,1), 2)  
donne COLLER(A,1)2  
et non pas A12
```

empêche un développement supplémentaire



Mode DEBUG

```
#ifdef DEBUG  
    #define DUMP1(X) printf("%s == %d", #X, X)  
#else  
    #define DUMP1(X)  
#endif
```

3 manières d'activer

```
#define DEBUG
```

```
$ DEBUG=1 make
```

```
$ make -DDEBUG
```

- TP sur les outils



Aller plus loin... opérateur ## (2)

- Concatène deux éléments

```
#define COLLER(A,B) A##B  
#define XCOLLER(A,B) COLLER(A,B)
```

```
XCOLLER(XCOLLER(A,1), 2) ⇒ A12
```



Macro ...

- C99 (pas ansi ☺)
- Macro variadique
- Nombre d'arguments variables

```
#define ESSAI(A, ...)
```

```
#define ESSAI(A, ...) \  
    printf(A, __VA_ARGS__)
```

157

Synthèse

- Avantages
 - Simplification d'écriture de code
 - Rapidité par rapport aux fonctions
- Désavantages
 - Augmente la taille du code
 - Effets de bords
 - Lenteur à cause de l'évaluation systématique
- Guide de style
 - Nom en MAJUSCULES

159

Note sur la vitesse d'exécution

- Appel de macro \Rightarrow Pas d'appel de fonction
 - Pas de saut mémoire
 - Pas de paramètres empilés
- Évaluation systématique des arguments
 - Dans MAX, un des arguments est évalué deux fois
- Perte de temps ???

Gain de temps

MAX(f(x), g(x))

158

LIGNE DE COMMANDE



Ligne de commande ?

- Fournir des informations au programme qui s'exécute à partir du système

```
$programme param1 -l -r  
$programme "une chaine avec espaces"
```

- Solution : le point d'entrée du programme
 - La fonction `main()`
 - Normes ANSI et ISO



Jusqu'à maintenant...

```
void main();
```

```
void main(void);
```

```
int main();
```

Pas d'autres types de retour possibles



Fonction "main"

```
int main(int argc, char ** argv);
```

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

- Seuls prototypes minimaux reconnus par les normes ANSI et ISO
- Tout programme renvoie un état
- Paramètres séparés par au moins un espace



argc / argv

- *arg count*
 - Nombre d'éléments de la ligne de commande
- *arg vector*
 - Liste des arguments (chaînes de caractères)
 - Au moins un argument, le nom du programme
- Norme

```
argv[0]      -> nom du programme  
argv[argc-1] -> dernier paramètre  
argv[argc]  -> NULL
```



Transformer les arguments

- Tous les paramètres sont des chaînes de caractères...
- Conversion éventuelle dans un type différent
- Exemple : transformer en entier

```
int atoi(char *);  
sscanf(chaine, format, variables);
```



Récupérer la valeur de retour

```
$ echo $?
```



```
> print ERRORLEVEL
```



```
EXIT_FAILURE  
EXIT_SUCCESS
```



FICHIERS (BINAIRE)

1. Compléments sur les fichiers
2. Fichiers binaires



Fichiers

- Texte
 - Compréhensible par l'humain
 - Editeur de texte
 - Par défaut
- Binaire
 - Informations codées (illisibles)
 - Périphériques UNIX
 - Préciser le mode b
- Traitement similaire
- Flot / Flux

**Semaines
bloquées**



Un fichier...

- Informations suivantes
 - Nom et type du fichier
 - Mode d'ouverture (Lecture/Ecriture)
 - Emplacement sur le disque
 - Emplacement de la mémoire tampon en RAM
- Fichiers standards prédéfinis
 - `stdin`
 - `stdout`
 - `stderr`



Structure FILE

- Bibliothèque `stdio.h`
 - Stocker des informations sur le fichier à manipuler
- Mémoire-tampon
 - De taille `BUFSIZ` (pas de E)
 - Machine asynchrone



Structure FILE

```
typedef struct _iobuf {
    char*   _ptr;
    int     _cnt;
    char*   _base;
    int     _flag;
    int     _file;
    int     _charbuf;
    int     _bufsiz;
    char*   _tmpfname;
} FILE;
```

FOPEN_MAX

- Nombre limité de fichiers ouverts (ressources systèmes)
- Au moins 8
- Fichiers déjà ouverts
 - `stdin`
 - `stdout`
 - `stderr`



Modes d'utilisation

- LECTURE (r)
 - *read*
- ECRITURE (w)
 - *write*
- AJOUT (a)
 - *append*
- Panachage : r+, w+, a+



Utilisation des fichiers binaires

- Ouverture/création/fermeture
 - *fopen* / *fclose*
- Lecture / Ecriture
 - Séquentielle *fread* / *fwrite*
 - Aléatoire *fseek* / *ftell*
- Gestion des fichiers
 - *feof* / *ferror* / *fflush*



fopen / *fclose*

```
FILE * fopen(const char * nom, char * mode);
```

- "Ouvrir" un fichier dont le nom est donné
 - Chemin valide
 - Mode "rb", par exemple

```
int fclose(FILE * fic);
```

- "Fermer" le fichier
 - Nécessaire
 - Force l'écriture des données non encore écrites
 - Vide les mémoires tampons



fread

1. Lire des objets d'un flux donné
2. Les placer en mémoire

```
size_t fread(void *ptr, size_t taille,  
             size_t nobj, FILE * fic);
```

- Quoi : lire *nobj* objets de taille *taille*
- De : *fic*
- Où : zone mémoire, tableau *ptr*
- Retourne le nombre d'éléments **effectivement** lus



fwrite

1. Lire les données en mémoire
2. Ecrire des objets dans un flux donné

```
size_t fwrite(const void *ptr, size_t taille,
              size_t nboj, FILE * fic);
```

- Quoi : écrire nboj objets de taille taille
- De : zone mémoire ptr
- Où : fichier fic
- Retourne le nombre d'éléments **effectivement** écrits



Gestion des flux (2)

```
int fflush(FILE * fic);
```

- Vide la mémoire tampon associée à fic
- Force l'écriture/lecture
- Vide tous les tampons si fic est nul
- **NON défini pour les flux d'entrée**



Gestion des flux (1)

```
int feof(FILE * fic);
```

- Retourne une valeur non nulle si l'indicateur de fin de fichier est positionné

```
int ferror(FILE * fic);
```

- Retourne une valeur non nulle si l'indicateur d'erreur de fichier est positionné

```
#define EOF (-1)
```



Accès aléatoire (1)

- Accès au fichier à un endroit particulier
 - Gestion d'un curseur

```
long ftell(FILE * fic);
```

- Retourne la position du curseur fichier



Accès aléatoire (2)

```
int fseek(FILE * fic, long dep, int o);
```

- Déplacer le pointeur/curseur fichier
 - dep :déplacement (+ ou-)
 - o : origine dans les valeurs suivantes

SEEK_SET : du début du fichier (dep>0)

SEEK_END : de la fin de fichier (dep<0)

SEEK_CUR : de la position courante

- Retourne 0 si le déplacement est effectué



Exemple

- Gestion d'un répertoire personnel
 - Nom du contact
 - Numéro de téléphone
- Tableau en mémoire
 - À écrire/lire sur le disque

```
typedef struct  
{  
    char nom[30];  
    char num[14];  
} entree_t;
```



```
void enregistrer(entree_t repertoire[],  
                int taille)  
{  
    FILE * fichier = fopen("sauvegarde", "wb");  
    int res = 0;  
  
    if (fichier)  
    {  
        res = fwrite(repertoire, sizeof(entree_t),  
                    taille, fichier);  
        if (res != taille)  
            fprintf(stderr, "probleme\n");  
        fclose(fichier);  
    }  
}
```

```
int lire(entree_t repertoire[])  
{  
    FILE * fichier = fopen("sauvegarde", "rb");  
    int taille = 0;  
  
    if (fichier)  
    {  
        while(!feof(fichier))  
            fread(&repertoire[taille++],  
                sizeof(entree_t), 1, fichier);  
        fclose(fichier);  
    }  
    return taille;  
}
```

STRUCTURES PARTICULIÈRES

1. enum
2. union
3. champs de bits



Enumération (1)

```
const int lundi    = 0;
const int mardi   = 1;
const int mercredi = 2;
const int jeudi   = 3;
const int vendredi = 4;
const int samedi  = 5;
const int dimanche = 6;

#define LUNDI    0
#define MARDI   1
#define MERCREDI 2
#define JEUDI   3
#define VENDREDI 4
#define SAMEDI  5
#define DIMANCHE 6
```

```
printf("On est %d", mardi);
```

```
printf("On est encore %d", MARDI);
```

186

Enumération (2)

```
enum SEMAINE
{
    lundi, mardi,
    mercredi,
    jeudi,
    vendredi, samedi,
    dimanche
};
```

```
enum SEMAINE jour;
for (jour = lundi; jour <= vendredi; ++jour)
    printf("On travaille...");
```

188

Point de vue du compilateur

```
enum SEMAINE
{
    lundi,
    mardi,
    mercredi,
    jeudi,
    vendredi,
    samedi,
    dimanche
};
```

Diagram illustrating the compiler's view of the enum SEMAINE. The elements are mapped to integer values: lundi (0), mardi (1), mercredi (2), jeudi (3), vendredi (4), samedi (5), and dimanche (6). A red dashed arrow indicates the sequence of values from 0 to 6.

- Affecte une valeur entière à chaque élément
- Sans précision
 - Commence à zéro
 - Incrmente la valeur pour chaque nouvel élément

189

```
enum SEMAINE
```

```
{
  lundi = 1,
  mardi,
  mercredi,
  jeudi,
  vendredi,
  samedi,
  dimanche
};
```

1

7

```
enum PAYS
```

```
{
  belgique = 32,
  france,
  angleterre = 44,
  etats_unis = 1,
  allemagne = 49,
  royaume_uni = 44
};
```

190

"Perturber"
le compilateur

Compatibilité ?

```
enum PAYS pays;
```

```
pays = france;
```

```
pays = mercredi;
```

191

enum vs macros

- Constante symbolique
 - Substituée par le préprocesseur
 - N'existe pas pour le compilateur
- enum
 - Traduction en "entiers"
 - Vérification de type
 - Vraies constantes

Union (1)

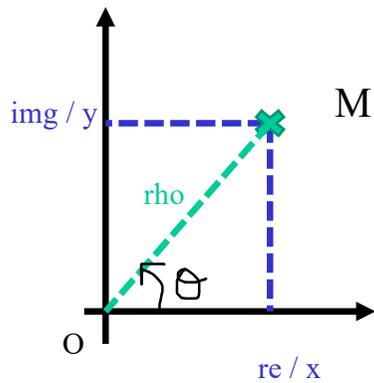
```
struct s
{
  int i;
  double d;
} s1;
```

```
union u
{
  int i;
  double d;
} u1;
```

- Ranger des variables différentes dans le même espace mémoire
- Stockage d'un champ à la fois

sizeof ?

Union (2)



```
enum FORMES {  
    CAR, POL  
};
```

```
typedef struct  
{  
    double rho;  
    double theta;  
} polaire_t;
```

```
typedef struct  
{  
    double reel;  
    double imag;  
} cartésien_t;
```

```
typedef union  
{  
    cartésien_t car;  
    polaire_t pol;  
} complexe_u;
```

```
typedef struct  
{  
    complexe_u val;  
    enum FORMES forme;  
} complexe_t;
```

194

Union (3)

```
double modulo_compl( complexe_t a)  
{  
    double res;  
  
    switch (a.forme) {  
        case CAR :  
            res = sqrt(a.val.car.reel*a.val.car.reel  
                + a.val.car.imag*a.val.car.imag);  
            break;  
        case POL :  
            res = a.val.pol.rho;  
            break;  
        default : fprintf(stderr, "Erreur\n");  
    };  
    return res;  
}
```

Champs de bits

```
typedef struct couleur1  
{  
    unsigned int R;  
    unsigned int G;  
    unsigned int B;  
    unsigned int A;  
} couleur1_t;
```

- types obligatoires :
 - int, unsigned int

```
typedef struct couleur2  
{  
    unsigned int R:8;  
    unsigned int G:8;  
    unsigned int B:8;  
    unsigned int A:8;  
} couleur2_t;
```

FONCTION À NOMBRE D'ARGUMENTS VARIABLE



Etat des lieux

- Prototype de fonctions à nombre d'arguments fixe
 - Liste des types
- Fonctions qui ne respectent pas ce schéma !
 - `printf scanf`
 - Essayer de définir des fonctions avec le même nom mais avec une liste d'arguments différents
 - ▶ IMPOSSIBLE



Prototype

```
<type_retour> nom(liste_fixe, ... );
```

- Liste NON VIDE de paramètres fixes
- Prototype de `fscanf` ?

```
int fscanf(FILE *,char *, ... );
```

```
fscanf(fichier, "%d %d", &nb1, &nb2);
```



Utilisation (1)

```
#include<stdarg.h>
```

Macros dans fichier include

```
va_list
```

Type liste de paramètre

```
va_start(liste, nom)
```

Initialiser la liste de paramètres

nom est le nom du dernier paramètre fixe de la fonction

```
va_end(liste)
```

Libération du pointeur de liste

Seulement conseillé par la norme donc ... OBLIGATOIRE

Utilisation (2)

`va_arg(liste, type)`

- Récupération paramètre par paramètre
 - Un à chaque appel
- Types possibles
 - int, double,
 - pointeurs comme `char *`
- Types convertis
 - char, float



```
int ouvrable(char * mode, int nombre, ...) {
    int    result = 1;
    char * fileName;
    va_list liste;

    va_start(liste, nombre);
    while (nombre--) {
        fileName = va_arg(liste, char *);
        if (!fopen(fileName, mode)) {
            result = 0;
            break;
        }
    }
    va_end(liste);
    return result;
}
/* exemple d'appel */
ouvrable("r", 2, "pg1.c", "pg2.c")
```

3

POINTEURS DE FONCTION



Une fonction ?

- Nom + type de retour + liste de paramètres
- Ce n'est pas une variable !
- Nom ► adresse en mémoire
- ⇨ Pointeurs de fonction



Exemples

```
int (*pf1)(void);
```

```
double (*pf2)(double, double);
```

```
void (*pf3)();
```

On ne connaît pas les paramètres de pf3.
(ancienne formulation du C)

Parenthèses obligatoires
pour (*nom)

```
void f3(nom1, nom2)
int nom1;
double nom2;
{ ... }
```

206

```
double (*pf2)(double, double);
```

```
double max(double a, double b) {
    return (a > b)? a : b;
}
```

```
pf2 = &max;    Privilégier la compréhension
```

```
pf2 = max;    Privilégier la portabilité
```

```
max(2.0, 5.0);
```

```
pf2(2.0, 5.0);
```

```
(*pf2)(2.0, 5.0);
```

207

Tableau de pointeurs (1)

```
int f1();
int f2();
int f3();
int f4();
```

```
switch(choix)
{
    case 0 : f1(); break;
    case 1 : f2(); break;
    case 2 : f3(); break;
    case 3 : f4(); break;
}
```



208

Analogie

```
int choix, val;
int a,b,c,d;
```

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

```
switch(choix)
{
    case 0 : a = val; break;
    case 1 : b = val; break;
    case 2 : c = val; break;
    case 3 : d = val; break;
}
/* ou if ... else if else ... */
```

```
t[choix]
= val;
```

209

Tableau de pointeurs (2)

```
int (*tabptrfn[4]) ();
```

```
tabptrfn[0] = f1;  
tabptrfn[1] = f2;  
tabptrfn[2] = f3;  
tabptrfn[3] = f4;
```

OU

```
int (*tabptrfn[]) () = { f1, f2, f3, f4 };
```

```
tabptrfn[choix] ();           "SWITCH"
```

```
(*tabptrfn[choix]) ();  
(**(tabptrfn+choix)) ();
```

210

Pointeur de fonction en paramètre

```
appliquer(10.0, 2.0, max );
```

```
double appliquer(double a, double b,  
                 double (*pf) (double, double) )  
{  
    pf(a,b);  
}
```

```
double appliquer(double, double,  
                 double (*) (double, double) );
```

Prototype = pas de nom pour les paramètres

211

Pointeur de fonction en paramètre

- atexit()
- signal()
- qsort()

212



exit()

- Définie dans `#include <stdlib.h>`
- Sortie propre du programme
 - Tampons vidés
 - Fichiers fermés
- Libérer la mémoire allouée sur le tas



```
ptr = malloc(...);  
if (!ptr) exit(2);
```

213

typedef

- Forme habituelle

```
typedef ancien_nom nouveau_nom;
```

```
typedef double(*) (double, double) NOM;
```

- Forme spéciale

```
typedef double (*NOM) (double, double);
```

```
double (*p1) (double, double) = min;  
double (*p2) (double, double) = max;  
NOM p1 = min, p2 = max;
```

(214)

Retour de fonction

Fonction f1 qui prend un char en paramètre et retourne un pointeur de fonction ?

```
double (*) (double) f1(char);
```

```
typedef double (*PTR) (double);
```

```
PTR f1(char);
```

```
double (*f1 (char)) (double);
```

(215)

C & UNIX

1. Processus
2. Interface
3. Entrées/sorties de bas niveau



Processus

- Identifiant unique *processus identifier*
 - Type spécial `pid_t`

```
pid_t getpid();
```

```
$ echo $$ (shell)
```

- Processus père (sauf processus 0)

```
pid_t getppid();
```

(217)

Copie de processus (1)

- Création d'un processus fils, **copie conforme** du processus père
 - Même zone de données et de code
 - Même environnement
 - Même priorité
 - Même description de fichiers
- Exécution **concurrente** des deux processus



Terminaison de processus

- Terminaison "normale"
 - Fin de la fonction `main()`
 - Fonction `void exit(int);`
- Valeur transmise au shell
 - Retour du `main()`
 - Paramètre d'`exit()` } 0 : fin normale

```
$ echo $? (shell [k]sh)
> print ERRORLEVEL (dos/win)
```



Copie de processus (2)

```
pid_t fork();
```

```
#include <unistd.h>
```

Valeur de retour :

-1 : erreur lors de la création
limite du nombre de processus

0 : processus fils

>0 : processus père, pid du processus fils

Après un `fork()`, tout se passe comme si chacun venait de faire un `fork()`



Exemple 1

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

void main() {
    pid_t pid;

    switch(pid=fork()) {
        case (pid_t) -1 :
            printf("Erreur");
            exit(2);
        case (pid_t) 0 :
            /* sleep(100); */
            printf("FILS: %d (PERE: %d)",
                getpid(), getppid());
            exit(0);
        default :
            printf("PERE:%d (PÈRE: %d)",
                getpid(), getppid());
    }
}
```

Exemple 2

```
int i;  
for (i = 0; i < 3; i++) fork();
```

- Combien de processus sont créés ?

222



Bibliographie

- La programmation sous UNIX, Jean-Marie RIFFLET, Ediscience, 2003
- The C Programming Language, KERNIGHAN, RITCHIE, Prentice-Hall, 1989
- Guide de développement sécurisé en langage C, ANSSI, 2020

224

Interface C / Système

- Faire un appel système `int system(char *);`

```
system("ls -l");  
system("date");
```

- Passer des paramètres (ligne de commande)

```
int main(int, char **);
```

- Flux standards

```
stdin  : entrée standard      <  
stdout : sortie standard     >  
stderr : sortie d'erreur standard 2>
```

223