Septembre 2026 – Version étudiante

C2



loic.yon@isima.fr
https://perso.isima.fr/loic

CLERMONT AUVERGNE INP UCA Isima

⊗ 0 ⚠ 0

Être un ingénieur ?



- Présence en cours
- Présence **active** en TP
  - Présence contrôlée
  - Préparer les TPs
  - **SAVOIR FAIRE**
- 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)

Objectifs du cours

- Connaître les bases du langage et les mécanismes mis en œuvre sans aide de l'IA
- Transposer en C des algorithmes et des structures des données
- Connaître la chaine de compilation
- Manipuler des outils d'aide au développement
- Découvrir à la programmation graphique

Plan



- Outillage & méthodes 5
- Compilation séparée / Makefile 16
- Sous le capot 51
- Pointeurs
  - de variable 68
  - de fonction 206
- Macros 120
- Structures 151
- Enum/Unions 191
- Arguments var 199
- Flux 226
- SDL 2 239

Pré-requis :
- Initiation algo
- Initiation C/UNIX



OUTILLAGE & MÉTHODES





Plan



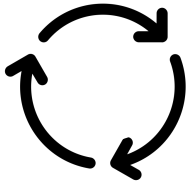
- Logiciel ?
- Méthode(s) de développement
- Tests unitaires
- Outils
  - Compilateur
  - Débogueur
  - Profileur
  - Gestionnaire de versions

Un logiciel

- Utilisateurs ?
  - Besoins
  - Spécifications
- Développeurs
  - Conception : analyse // Algorithmique // SDD
  - Développement
  - Tests
  - Maintenance / évolutions

Cycle de développement

- Méthodes
  - Cycle en V
  - Agile
- Spécifications
- Analyse
- Développement
- Tests
  - Unitaires – intégration - fonctionnels
- Déploiement
  - Environnement ?





Compilateur



- ISIMA
  - `gcc 14.2` [PC] [VM]

Norme C17

- À la maison :
  - `gcc` sous une distribution LINUX
  - `gcc` sous Cygwin / MinGW
  - `gcc` sous WSL2 (W11)
  - `gcc` sous mac (brew / macport)
- Visual Studio ?
  - C'est du C++
- Bon éditeur ou EDI (VS code)

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
```

Support C3

Profileur

- Analyse de l'exécution
  - Variables initialisées ?
  - Indices des tableaux
  - Respect des zones mémoire
- 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



Git (2005)



https://gitlab.isima.fr

Tests unitaires

- Tester des petits bouts des programmes
  - Avant leur intégration
- À la main ou **automatique**
  - Répétabilité ?
  - Régression ?
- Plus ou moins exhaustifs
  - Couverture des tests
- Conditions ± réelles
- Développement orienté Tests (TDD)

Coder n!

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

   return r;
}
```

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

```
TEST(factorielle)
{
   CHECK(    1 == f(0) );
   CHECK(    2 == f(2) );
   CHECK(    6 == f(3) );
   CHECK( 5040 == f(7) );
}
```

Tests exhaustifs ?

Toujours contrôler les tests

Et encore ?

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

COMPILATION SÉPARÉE & MAKEFILE





CLERMONT AUVERGNE INP ISIMA

Plan



- Structure d'un programme
  - Déclaration / implémentation / mots-clés
- Étapes de compilation
- Compilation séparée
  - Module
  - Variable globale
  - Gardiens
- Make/ Makefile
  - Règle
  - Variable


```c
#include <stdlib.h>

int main() {

   int x = 0;

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

   printf("Somme %d+%d = %d",
     x, y, somme(x, y));

   return 0;
}

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


Inconnu au bataillon ?

On ne peut utiliser que des choses qui sont CONNUES !

- Déclarer des variables
  = donner le type

```c
int y;
```

- Déclarer des fonctions
  = donner le prototype ou la signature

```c
int somme(int, int);
```

- L'identifiant ne doit être **ni réservé** ni avoir été **déjà utilisé**
  - Règles de nommage

Mots-clés en C17

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

```
_Alignas
_Alignof
_Atomic
_Bool
_Complex
_Generic
_Imaginary
_Noreturn
_Static_assert
_Thread_local
```

```c
#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;
}
```


Structure d'un programme

- Inclusion des **bibliothèques standards**
- Définition des **constantes symboliques**
- Déclaration des **variables globales**
  - + initialisation
  - + description
- Définition des **fonctions**
  - Fonction `main()`

La structure évoluera en fonction des concepts vus en cours

Ordre dans la liste des fonctions ?


```c
#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;
}
```



```c
#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;
}
```

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

int somme(int, int);
// int somme(int e, int f);

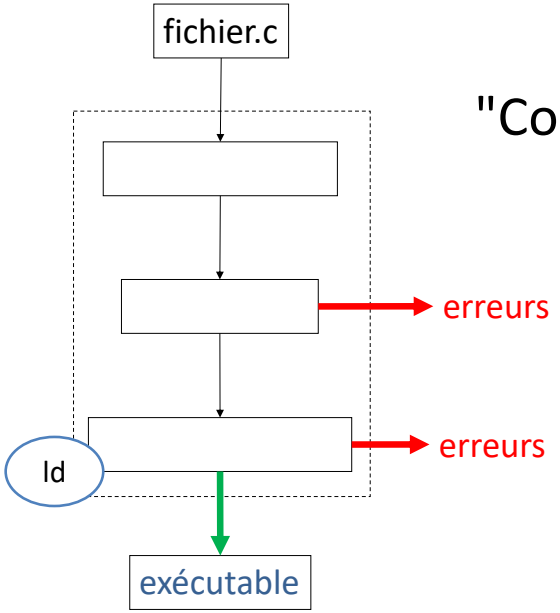
int main() {

   int x = 3, y = 5;

   printf("%d", somme(x, y));

   return EXIT_SUCCESS;
}

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

"Compilation"


```c
#include <stdio.h>

#define T 10

int tab[T];

int main() {
  int i;
  for(i=0; i<T; ++i)
     tab[i] = 0;
  return 0;
}
```

123o

préprocesseur

```c
/** fichier stdio.h **/

int tab[10];

int main() {
  int i;
  for(i=0; i<10; ++i)
     tab[i] = 0;
  return 0;
}
```

12826o

Code asm
Code machine

```
89e5 c745 fc00 0000 00eb 1848 8d05
6e000000 8b55 fc48 63d2 c704 9000 0000
008345fc 0183 7dfc 097e e2b8 0000 0000
5dc31400 0000 0000 0000 017a 5200 0178
1001100c 0708 9001 0000 3400 0000 1c00
0000aeff ffff ffff ffff 3200 0000 0000
00000004 0100 0000 0e10 8602
000d0604 2d00 0000 0c07 0800
00000001 5f00 0500 035f 6d68 5f5f 7865
```

Editions des liens

```
cffa edfe 0700 0001 0300 0000 0100
00000300 0000 5001 0000 0020 0000 0000
00001900 0000 e800 0000 0000
00000000 0000 0000 0000 0000
00008800 0000 0000 0000 7001 0000 0000
00008800 0000 0000 0000 0700 0000 0700
```

4280o

```
cffa edfe 0700 0001 0300 0000 0100
00000300 0000 5001 0000 0020 0000 0000
00001900 0000 e800 0000 0000 0000 0000
00000000 0000 0000 0000 0000 0000 0000
00008800 0000 0000 0000 7001 0000 0000
0000f801 0000 0100 0000 0004 0080 0000
```

600o

#include (1)

Insère le contenu du fichier (copier/coller)

- à partir du chemin du répertoire courant

```c
#include "fichier1"
```

- à partir d'une liste de chemins fixés

```c
#include <fichier2>
```

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

`#include` (2)



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

#include "a_moi.h"
#include "teZZt.h"
```

```
#include <sdl2/sdl.h>
```


Compilation séparée

- Fichier de code unique ?

Mettre le code dans des fichiers séparés




```
#include <stdio.h>

int main() {
  printf("%d\n", f());
  return 0;
}
```

main.c

```
int f() {

  return 3;
}
```

f.c

Plusieurs fichiers ?

```
#include <stdio.h>

int main() {
  printf("%f\n", f());
  return 0;
}
```

main.c

```
float f() {

  return 3.0;
}
```

f.c


```
#include <stdio.h>

float f();

int main() {
  printf("%f\n", f());
  return 0;
}
```

main.c

```
float f() {

  return 3.0;
}
```

f.c

Module

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

int main() {
  printf("%f\n", f());
  return 0;
}
```

main.c

Un module

```c
float f();
```

f.h

```c
#include "f.h"

float f() {

  return 3.0;
}
```

f.c

- Séparer en "modules"

Variable globale ?

```c
double a = 7.0;

float f() {
  a = 5.0;
  return 3.0;
}
```

f.c

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

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

  return 0;
}
```

main.c

Variable globale ? (2)

```c
double a = 2.0;
float f();
```

f.h

```c
#include "f.h"

float f() {
  a = 5.0;
  return 3.0;
}
```

f.c

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

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

  return 0;
}
```

main.c

Variable globale !

```c
#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

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

f.h

```c
#include "f.h"

double a = 2.0;

float f() {
  a = 5.0;
  return 3.0;
}
```

f.c

Gardiens

```
#ifndef LOIC_F_H
#define LOIC_F_H

extern double a;

float f();

#endif
```



f.h

~~#pragma once~~



- Bonne pratique
- Éviter les inclusions multiples

Module - Synthèse

Fichier d'entête	Fichier d'implémentation
• Extension .h • **Déclaration** des variables globales • **Déclaration** des fonctions – Prototype / signature • Types • Constantes	• Extension .c • **Définition** des variables globales • **Définition** des fonctions – Code du .h – Fonctions privées



Comment compiler plusieurs sources ?

```
$ gcc main.c f.c -o main
```

- Tous les fichiers sont compilés en un exécutable
  - Même si un fichier n'a pas été modifié

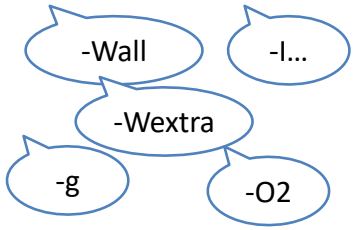
Compilation séparée

- … des différents fichiers de code
- Compilation des sources en fichiers objet

```
$ gcc -c main.c
$ gcc -c f.c
```



```
main.o
f.o
```

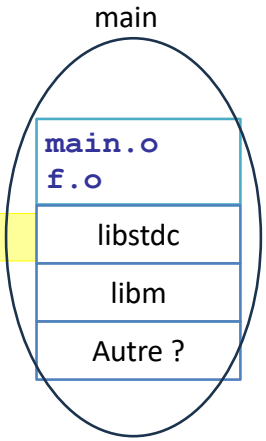


Édition des liens

- Dernière étape
- Bibliothèques standards
  et moins standards

```
$ gcc main.o f.o -o main
```





- Édition statique ou dynamique

Comment compiler ?

1. Écrire les différentes lignes de commandes correspondantes

```
$ gcc -c main.c
$ gcc -c f.c
$ gcc main.o f.o -o main
```

2. Écrire un petit programme *shell* qui lance les différentes étapes de compilation

```
$ ./script
```

3. Utiliser un programme spécial comme `make`

```
$ make
```


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

(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

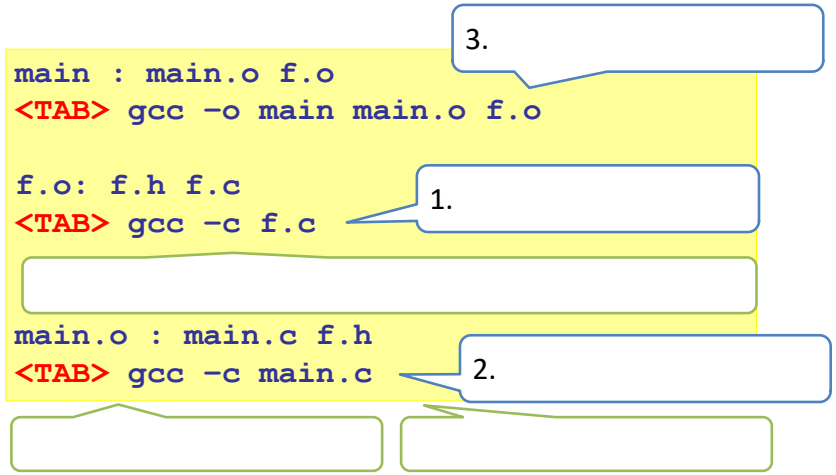
```
$ make
$ make clean
$ make install
```

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)


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



Variable personnalisée

Variable = nom donné à un "texte"

Personnalisée | Automatique

- Définition

```
NOM = texte \
<TAB> suite du texte
```

Liste de fichiers

Compilateur

Options

- Utilisation `$(NOM)`

Convention de nommage


```
# choix du compilateur ou version
CC = gcc-12
# options compilation
CFLAGS  = -Wall -Wextra -g -Werror=vla
# options edition des liens
LDFLAGS = -lm
# liste des fichiers objets
OBJ = main.o \
<TAB> f.o
# règle de production de l executable
main : $(OBJ)
<TAB> $(CC) $(OBJ) -o main $(LDFLAGS)
# règle de production pour chaque objet
main.o : main.c f.h
<TAB> $(CC) -c main.c $(CFLAGS)
f.o : f.h f.c
<TAB> $(CC) -c f.c $(CFLAGS)
```

Règle *clean*

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

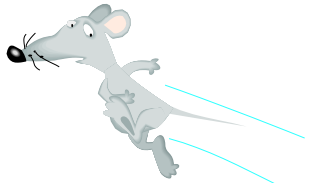
```
clean :
<TAB> rm -f *.o
```

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
  - Variables automatiques $@ $< $^
  - Génération automatique des dépendances (ZZ2, option –MM ou -MMD)
  - Gestion de configuration
  - Installation d'un programme

MÉMOIRE – SOUS LE CAPOT





CLERMONT AUVERGNE INP Isima

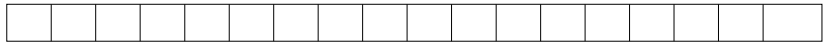
Plan



- Mémoire
- Les types entiers
- Exécution d'un programme
- Variable locale / globale
- Tableaux à 1 et 2 dimensions

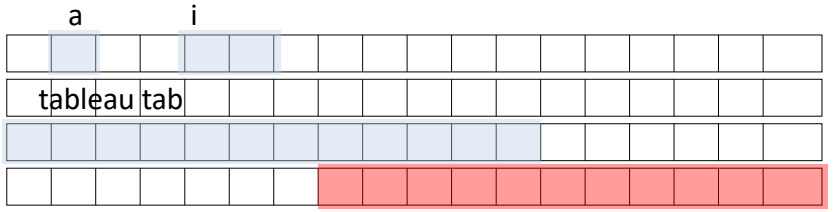
Mémoire de l'ordinateur (1)

- Abstraction : tableau à une dimension
  - Grande dimension (4 Go -> 64 Go)



- Unité de base : l'octet
- Gestion plus ou moins automatique
  - Fonctions par zones
- Partagée entre utilisateurs
  - Sécurité ?
- Un objet prend plus ou moins de mémoire
  - fonction `sizeof`
  - alignement possible

Mémoire de l'ordinateur (2)




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

Zone protégée :
- code programme courant
- zone d'un autre utilisateur
- ...

Types standards entier C17

```
char
short int
long
long int

signed (par défaut)
unsigned
```

```
sizeof(type) ?
```

```
stdint.h
```

```
int8_t
int16_t
int32_t
int64_t
```

```
uint8_t
uint16_t
uint32_t
uint64_t
```

```
int_least32_t
int_fast16_t
```

```
stdbool.h
```

```
_Bool ou bool
```


Comment utiliser ces entiers ?

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

int main() {
  int8_t i8 =3;
  int32_t i32 = -17;
  printf("concat" "enation\n");
  printf("Entier 8 bits: %" PRId8  "\n", i8);
  printf("Entier32 bits: %" PRId32 "\n", i32);

  return 0;
}
```

Système actuel pas "classique"
Amélioration en C17

Concaténation à la compilation

"%d"

"%d" ou "%ld"

Exécution d'un programme ?

- Chargement en mémoire du programme
- Création de zones mémoire "indépendantes"

1. Zone de code (RO)
2. Zone des données (RW)
   - Variables globales et statiques
3. Zone de tas
   - Mémoire à la demande
4. Pile d'exécution
   - Fonctions "actives" (non terminées)
   - Variables locales, paramètres, adresse de retour


```
int gj = 7;
int gi;

int fonction1(int p1)
{
  int li = 1;
  gj = 4;
}

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

int main()
{
  int li = 3;

  gi = 5;
  return 0;
}
```



Variables globales ?
Variables locales ?

Variable globale

- Portée (*scope*) : le programme
- Créée et initialisée au lancement du programme
- Détruite à la fin du programme

- À utiliser avec parcimonie
  - Collision de noms
  - Frein à la réutilisabilité

Variable locale

- Portée limitée à la fonction

1. Automatique
   - Par défaut
   - Mot-clé `auto` optionnel (⚠ C17)
   - Créée à chaque appel de fonction
   - Détruite à chaque fin de fonction
   - Pas de valeur par défaut (norme)
2. Statique
   - Initialisée au premier passage dans la fonction
     - À 0 en cas d'omission
   - Détruite à la fin du programme

```c
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());
  printf("%d", fonction1());
  printf("%d", compteur);
  return 0;
}
```

Autres variables

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

```c
extern int g;
```

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

```c
register int r;
```




```c
#include <stdio.h>

int a = 0;

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

int main()
{
  int a = 3;
  f(2.000);
  printf("%d\n", a);

  return 1;
}
```




```c
#include <stdio.h>


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

int main()
{
  int a = 3;

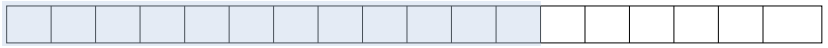
  f(a);
  printf("%d", a);

  return 0;
}
```

Trace ?

Tableau à 1 dimension

tableau t1



```
int    t1[12];
double t2[12];
```

Taille fixée à la compilation

Zone contiguë d'éléments de même type




```
#include <stdio.h>


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

int main()
{
  int t[3];

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

  return 0;
}
```

Pas nécessaire de préciser la taille MÊME si c'est une bonne pratique



Passage d'un tableau en paramètre

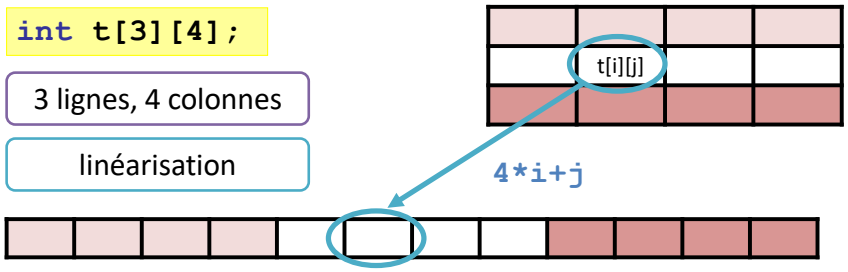
Trace ?

Tableau à deux dimensions

```
int t[3][4];
```

3 lignes, 4 colonnes

linéarisation




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


POINTEURS





CLERMONT AUVERGNE INP ISIMA

Plan



- Variable, adresse
- Pointeur
- Pointeurs & tableaux
- Pointeurs & chaînes de caractères
- Allocation dynamique de mémoire

Points à résoudre



```
char    tab[100];
```

```
void f(int a)
{
  a /= 3;
}
```



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

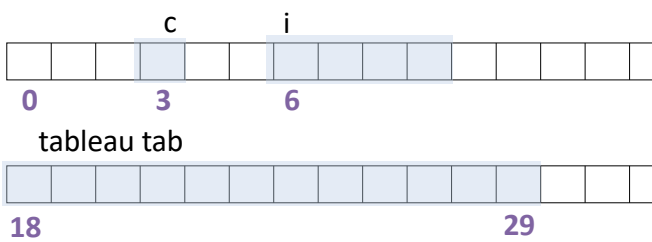
  c = a;
  a = b;
  b = c;
}
```



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

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


Variable : nom <-> adresse





```
char c;
int  i;
char tab[12];
```

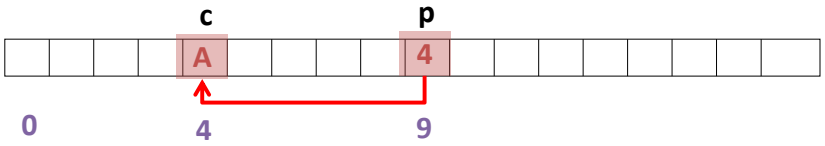
```
sizeof(c) ?
```



```
&c
&i
&tab[0]
```

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

Pointeur ?




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

Variable classique
Variable contenant une adresse

```
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.


```
int main()
{
  int a, b, *p;
  a = 4;
  p = &a;

  printf("contenu %d",        a);
  printf("adresse de a %p",  &a);
  printf("contenu de p %p",   p);
  printf("contenu pointé %d",*p);

  b = *p;

  (*p)++;
  printf("valeur de a est %d", a);

  return 0;
```


Déréférencer un pointeur ?

- Adresse valide ou non ?
- Valeur particulière

```
int    *  p1 = 0;
char   *  p2 = NULL;
```

```
double * p3 = nullptr // C23 - gcc-15
```

```
printf("%p",  p2);
printf("%c", *p2);
```


Affectation de pointeurs

```
int  a = 3;
int  *pi, *pj;

pi  = &a;
pj  = pi;
*pj = 7;

printf("%d", a);
```

```
float *pf;

pf = pi;
```

Changement de type ?

- *Cast*
- Opérateur de transtypage
- Opérateur de coercition

```
int     i;
float   f;

i = (int) f;
```

```
int     * pi;
float   * pf;

pi = (int *) pf;
```


Pointeur typé ou "universel"

- Pointeur classique typé

```
int     *pi;
```

- Pointeur "universel"
  - Type non fixé ou tout type de pointeur

```
void  *pv;
```

```
int     i;
void * pv = &i;
int  * pi = (int *) pv;
```

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


Paramètre de fonction

```
#include<stdio.h>

void fv(int a) {
  a = 5;
}

void fp(int *p) {
  *p = 7;
}
```

```
int main(int, char **)
{
  int a = 3;
  fv(a);
  printf("%d", a);

  fp(&a);
  printf("%d", a);

  return 0;
}
```



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

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

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

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

Pointeurs & tableaux (1)

- Un tableau est un pointeur **constant**

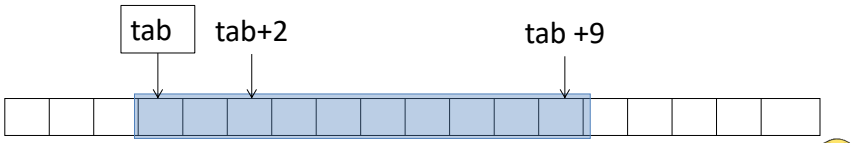
```
char tab[10];
```

tab ?

- Adresse du premier élément du tableau

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

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



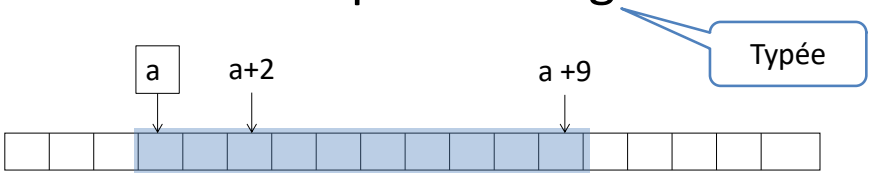
Arithmétique




```
POINTEUR' = POINTEUR + DEPLACEMENT
Adresse     adresse.   offset
```

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

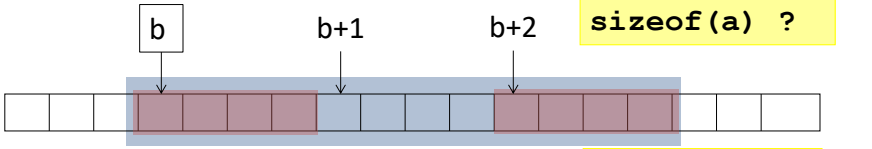
Arithmétique "intelligente"




```
char a[10];
```

1

```
sizeof(char)
sizeof(a) ?
```




```
int b[3];
```

Exemple : 4

```
sizeof(int)
sizeof(b) ?
```


Pointeurs & tableaux (2)

```
int tab[3];
int * p = &tab[0];
```

tab		p
&tab[0]		p
tab[0]		*p
tab[i]	*(tab+i)	*(p+i)
tab[i]	*(p+i)	p[i]
tab[i+1]		*(p+i+1)
tab[i+1]		++p; *p;



Pointeurs & chaînes

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

Réservation mémoire d'un tableau de 8 caractères

Initialisation de la chaine

```
s1[3] = 'I';
```

```
char * s2  = "chaine2";
```

Réservation mémoire d'un tableau de 8 caractères

Réservation pointeur

```
s2[3] = 'I';
```


Compter les lettres d'un mot …

```
char mot[20];

strcpy(mot, "Pointeur");
printf("%d", strlen(mot));
printf("%d", compter2(mot));
```

```
int compter1(char chaine[]);
```

```
int compter2(char * chaine);
```

Toutes les fonctions de <string.h> utilisent char * pour désigner des chaînes de caractères


```
int compter1(char chaine[])
{
  int i = 0;

  while (chaine[i] != '\0')
     ++i;

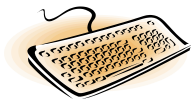
  return i;
}
```

strlen()

```
int compter2(char * chaine)
{
  int i = 0;

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

  return i;
}
```

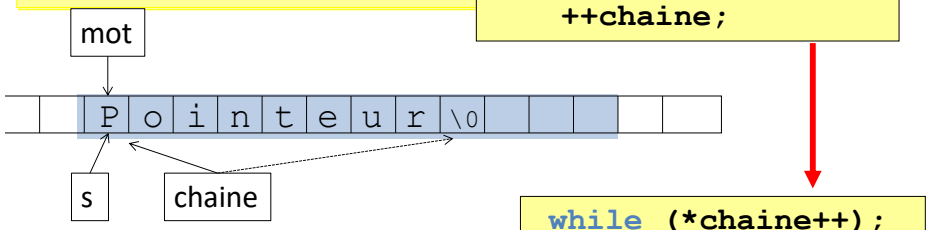



```
int compter2(char * chaine)
{
  char * s = chaine;

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

  return chaine - s;
}
```

```
while (*chaine)
   ++chaine;
```




```
while (*chaine++);
```

```
return chaine - s - 1;
```



Copie de chaînes de caractères

```
char source[20] = "origine";
char dest1[20], dest2[30];

strcpy(dest1, source);
copie2(dest2, source);
printf("%s", strcmp(dest1, dest2);
```

```
int copie1(char d[],  char s[]);
```

```
int copie2(char * d, char * s);
```



```
void copie1(char d[], char s[])
```

```
void copie1(char d[], char s[])
{
  int i = 0;

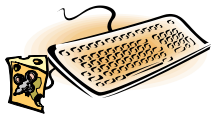
  while (s[i] != '\0')
  {
    d[i] = s[i];
    ++i;
  }
  d[i]=s[i];
}
```

strcpy()

```
while ((d[i] = s[i])!= '\0')
  ++i;
```

```
while ((d[i] = s[i]))
  ++i;
```

```
while ((d[i] = s[i++]));
```



Effets de bord ?

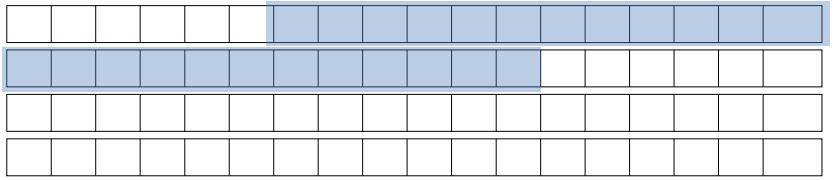

```
void copie2(char * d, char * s)
{
  while (*s != '\0')
  {
     *d = *s;
     ++d;
     ++s;
  }
  *d=*s;
}
```

```
while ((*d = *s) != '\0')
{
  ++d;
  ++s;
}
```

```
while ((*d++ = *s++)!='\0') ;
```

```
while ((*d++ = *s++)) ;
```


Allocation dynamique de mémoire ?



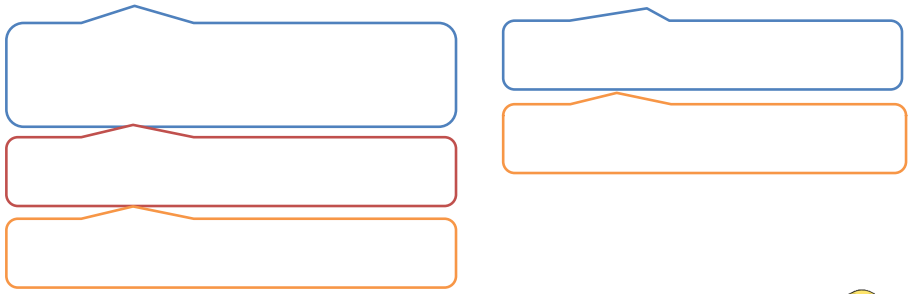
- Allouer de la mémoire = réserver une zone mémoire contiguë
  - Pas toujours possible
  - Appel système
- Dynamique : en cours d'exécution de programme (et non au moment de la compilation)

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

malloc()

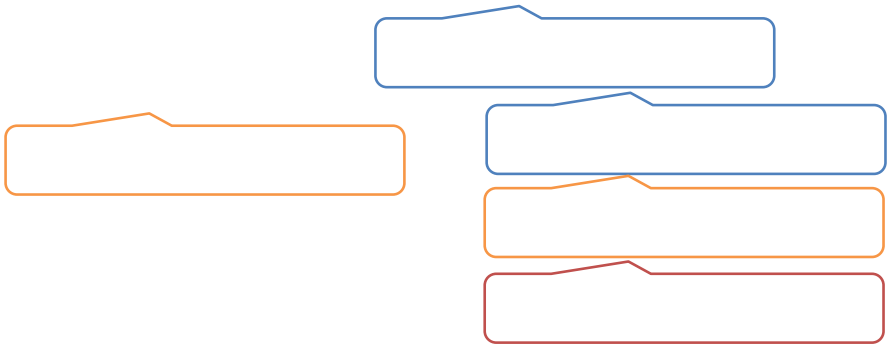
- Allocation d'un bloc contigu de mémoire

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



Libérer la mémoire

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



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


Libérer la mémoire (2)

- Oublier de rendre la mémoire =
  - Fuite mémoire
  - Vers un redémarrage de la machine



- Outils de détection : valgrind, sanitizer, …


```
int   taille;
int * tableau = NULL;

printf("Taille du tableau ? ");
scanf("%d", &taille); // taille > 0

tableau = (int *) malloc (taille*sizeof(int));

if (tableau == NULL)
{
   printf("allocation mémoire impossible");
} else {
  /* l'allocation mémoire a eu lieu */
  /* …                              */

  free(tableau);
}
```

calloc()

- Allocation et initialisation d'un bloc contigu de mémoire

```
void * calloc (size_t nb, size_t t);
```

Toujours plus rapide que `malloc()` + `memset()` sur systèmes actuels

```
tableau = (int *) calloc (taille, sizeof(int));
```

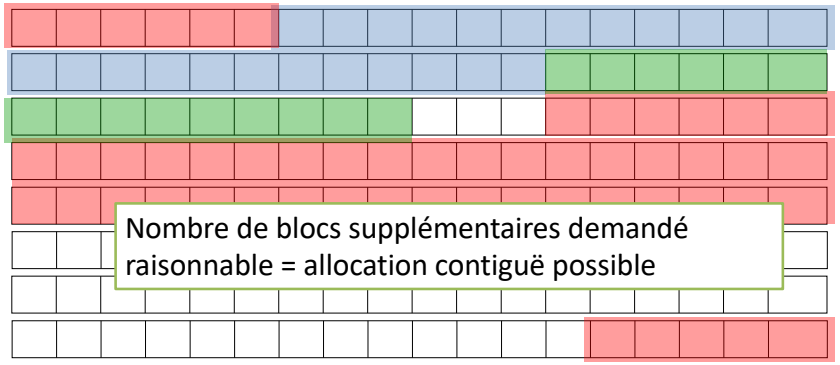

realloc()

- Redimensionner la taille d'un tableau en mémoire

```
void * realloc (void * o, size_t t);
```

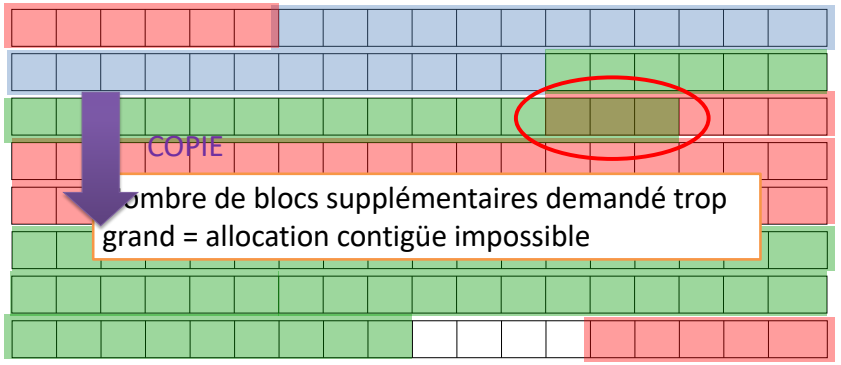
Demander plus de mémoire ?

Cas 1



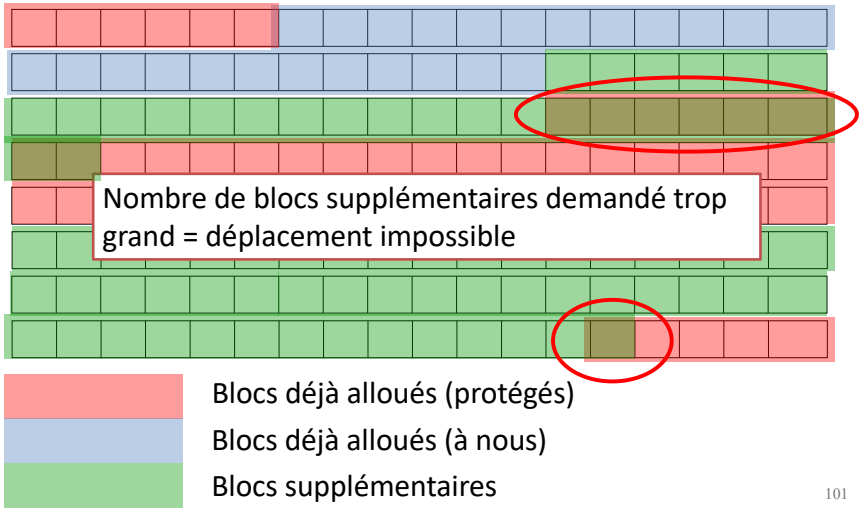
Blocs déjà alloués (protégés)
Blocs déjà alloués (à nous)
Blocs supplémentaires

Cas 2



Blocs déjà alloués (protégés)
Blocs déjà alloués (à nous)
Blocs supplémentaires

Cas 3



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
```


Variable Length Array (VLA)

PILE

TAS

Toujours implémenté ?

Taille maximale ?

Sécurité

```
void f(int n)
{
   int   i[n]; // VLA sur la pile
}
```


Erreurs typiques

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

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

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



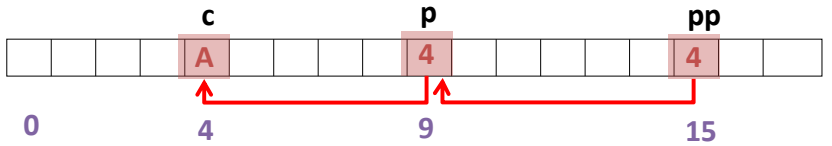
Tableau de pointeurs

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

```
int * tableau [2];
int   i = 10, j = 20;
```

```
tableau[0] = &i;
tableau[1] = &j;
printf("%d", (*tableau[0]));
```


****




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

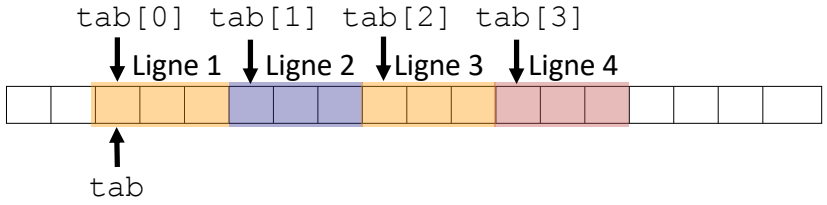
```
pp = &p;
```

```
type * pp;
```

```
char * * pp;
```


Tableau statique à 2 dimensions & pointeur

```
int tab [4][3];
```




```
tab[i][j]
```

```
*(tab[i] + j)
```

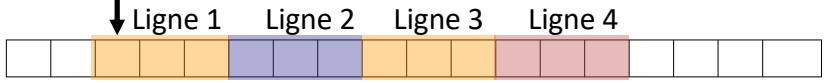
```
*(*(tab +i) + j)
```


Tableau dynamique à 2 dimensions (1)

1. Implémentation avec un tableau à une dimension
   - m lignes et n colonnes

```
int * tab1;
```

```
tab1 = malloc(n*m*sizeof(int));
```




```
tab[i][j]
```

?

```
tab1[i*n+j]
```

Tableau dynamique à 2 dimensions (2)

2. Implémentation avec une dimension maximale et une dimension libre

```
int * tab2[M];
```

```
for(int m=0; m<M; ++m)
   tab2[m] = malloc(n*sizeof(int));
```

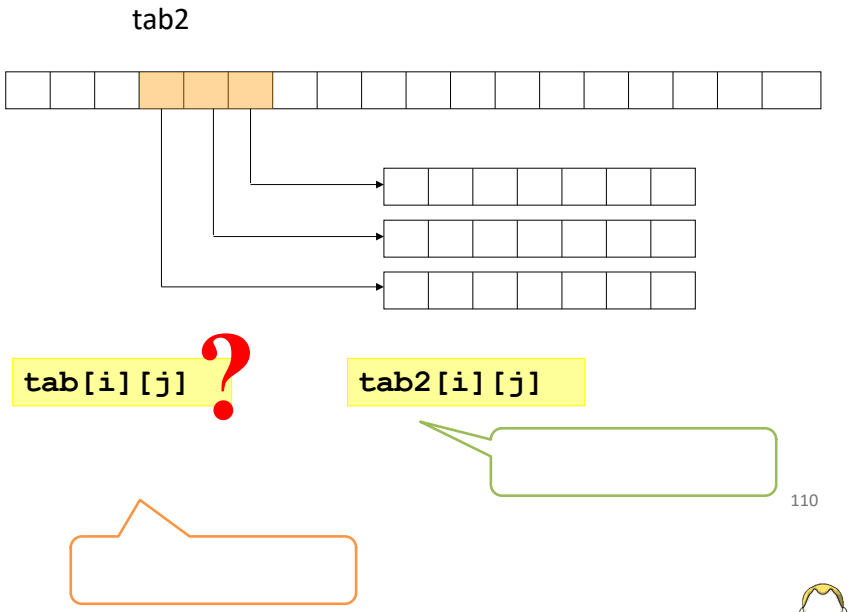


Tableau dynamique à 2 dimensions (3)

- Choisir la représentation par lignes ou par colonnes

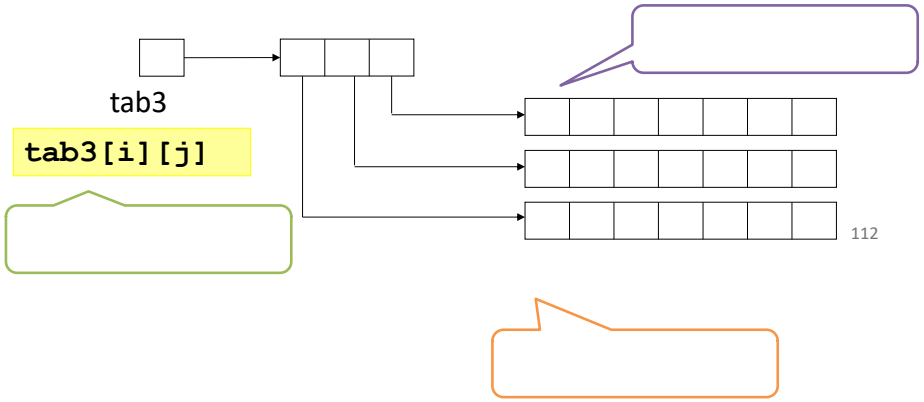
```
int ** tab3;
```

- Allouer un premier tableau de pointeurs

```
tab3 = malloc(m*sizeof(int *));
```

- Allouer chaque ligne/colonne

```
for(int k=0; k<m; ++k)
   tab3[k] = malloc(n*sizeof(int));
```



Équivalence tableau / pointeur

`int tab[]` ⟺ `int * p`

`int mat[M][N]` ⟺ `int ** v`

- Zone mémoire contiguë
- Lignes de même taille
- Statique

- Zones mémoires PAS contiguës
- Lignes de taille différente
- Éventuellement non allouées
- Dynamique


```c
int k = 2;

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

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

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

   return 0;
}
```

```c
int k = 2;

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

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

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

   return 0;
}
```



typedef

Faire un alias de type

```c
typedef int entier;
```

```c
int i;
entier j;
```

```c
typedef int * pint;
```



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

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

Échanger deux pointeurs

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

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

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

const (1)

Variable non modifiable

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

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


const (2)

- Pointeur sur objet **constant**

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

- Pointeur **constant**

```
char * const sz3 = …;
```

- Pointeur **constant** sur objet **constant**

```
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;
```

```
char * strcpy(char *d, const char * s);
```


MACROS







CLERMONT AUVERGNE INP Isima

Macros



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

Préprocesseur

1ère étape de la "compilation"

```
#include <stdio.h>

#define T 10

int tab[T];

int main() {
  int i;
  for(i=0; i<T; ++i)
     tab[i] = 0;
  return 0;
}
```

préprocesseur

```
/** fichier stdio.h **/

int tab[10];

int main() {
  int i;
  for(i=0; i<10; ++i)
     tab[i] = 0;
  return 0;
}
```


Macro de type constant

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

```
#define NOM texte_remplacement
```



```
float tab1[N];

#define N 100

float tab2[N];

#undef  N
#define N 200

float tab3[N];

#undef N

float tab4[N];
```

Code source

```
float tab2[100];
```

```
float tab3[200];
```

Après préprocesseur

Constante & compilation conditionnelle

```
#define VARIABLE
```

PAS DE VALEUR

Constante définie

Ou pas …

- Utile pour la compilation conditionnelle
  - `#ifdef`, `#ifndef`, `#endif`,
  - `#else`, `#elif`

Gardiens

Mode DEBUG / RELEASE

Test OS

Test compilateur

Constantes symboliques prédéfinies

```
__cplusplus
__DATE__
__TIME__
__FILE__
__LINE__
```

Contextuelles

https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html

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));
```


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

Calcul de x^2 !

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

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

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


Usage des macros

S'affranchir des types

Éviter la duplication de code

Effets de bord ?

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

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


Parenthésite aiguë (1)

```
#define CARRE2(A)  (A)*(A)
#define CARRE3(A)  (A*A)
#define CARRE4(A)  ((A)*(A))
```

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

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

printf("%d", 6+MAX1(i,j));
```



```c
#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);
```


Avec une fonction ?

```c
int max_int(int a, int b)
{
  return a > b ? a : b;
}

int i = 2;
int j = 3;

printf("%d"%, max_int(i++, j++));
```


Écrire une "fonction" d'allocation

```c
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));
```

Alléger la syntaxe

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

int   * pi, * pj;
float * pf;

pi = MALLOC(int, 10);

pf = MALLOC(float, 100);

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


Une macro complexe ?

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

```
void fonction(int p1, double p2)
{
   fonction1(p1);
   fonction2(p2);
}
```

```
MACRO(a, b);
```

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

```
fonction(a, b);
```

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


Ne compile pas

```
if (a!=b)
  MACRO(a, b);
else fonction3(a);
```

```
if (a!=b)
  fonction(a,b);
else fonction3(a);
```

✅

```
if (a!=b) {
  fonction(a,b);
} else {
  fonction3(a);
}
```

✅

```
if (a!=b) {
  fonction(a,b);
}; else {
  fonction3(a);
}
```


Solution pour macro complexe

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

```
if (a!=b) do {
   fonction1(a);
   fonction2(b);
} while(0); else fonction3(a);
```

```
if (a!=b) MACRO(a, b); else fonction3(a);
```

Opérateur

- Entoure l'élément de guillemets: `#A ⇨ "A"`

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

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

"A" "B" ⇨ "AB"

```
int a = 1;

DUMP1(a);
DUMP2(a);
```

a == 1


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

#define DUMPD(X) printf("%s == %lf", #X, X)
```

```
int a = 1;
double d = 7.1;

DUMPI(a);
DUMPD(d);
```

Macros "typées"

```
#define DUMP(X,T) printf("%s == " T, #X, X)

DUMP(a, "%d");
DUMP(d, "%lf");
```

Connaitre le type de a


```
_Generic( (expression),
    type_1: instruction1,
    type_2: instruction2,
    ...
    default: instruction_def
)
```

```
#define DUMP(X)                                  \
   printf ("[TRACE] " #X " == ");                \
   _Generic((X),                                 \
      int     : printf("%d (int)", X),           \
      float   : printf("%f (float)", X),         \
      void*   : printf("%p (void*)", X) ,        \
      default : printf("inconnu" ))
```

```
int    a = 1;
int *  p = &a;
```

```
DUMP(a);
DUMP((void*)p);
```

a == 1 (int)

p == 12344 (void*)

Mode DEBUG

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

3 manières d'activer

- `#define DEBUG`
- `$ DEBUG=1 make`
- `$ make -DDEBUG`

Opérateur

- Concatène deux éléments

```
A ## B ⇨ AB
```

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

```
COLLER(A, 1)
```

A1

Utilisé pour coder "en objet" en C

https://gcc.gnu.org/onlinedocs/gcc-15.2.0/cpp.pdf (p18)

auto (C23)

Déduction (inférence) de type

```
#define SWAP(A,B) do { \
  auto c=A; A=B; B=c;  \
} while (0)
```

```
int i = 3;
int j = 4;
int * pi = &i;
int * pj = &j;
```

```
SWAP(i, j);
SWAP(pi, pj);
```


Macro …

- Macro variadique
- Nombre d'arguments variables

```
#define ESSAI1(...) printf(__VA_ARG__)
```

```
#define ESSAI2(A, ...) \
     printf(A, __VA_ARG__)
```

```
ESSAI1("%s", "ZZ" );
ESSAI2("%s %d", "ZZ", 1);
```


Note sur la vitesse d'exécution

- Appel de macro ⇨ 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

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

Peut-on éviter l'appel de fonction ?

- Choix du compilateur – heuristique

```
gcc -O2 source.c
```

`-O` `-O1` `-O2`

- Déclaration manuelle
  - Code dans le fichier d'entête

```
inline double max_d(double a, double b)
{ return (a<b)? b:a; }
```


Synthèse



GUIDE DE STYLE : nom en majuscules

STRUCTURE
(ET LISTE CHAÎNÉE)







CLERMONT AUVERGNE INP UCA Isima



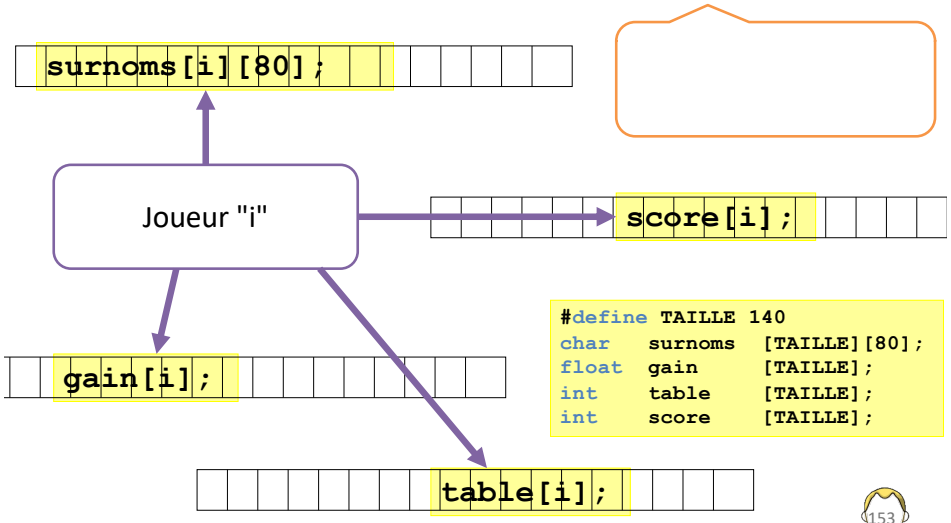
Un concours au BdE

- Gérer un concours de SKYJO
  - surnom : 80 caractères
  - table en cours : un entier
  - score de la partie : un entier
  - gain (unité : EuroZZ) : un réel



Capacité maximale de la salle : 140 joueurs

Solution à 



Une structure ?

Réunir les différentes informations pour un joueur

```
struct joueur {
  char  surnom[80];
  int   score;
  int   table;
  float gain;
};
```


Manipulations de base

```
struct joueur zz;
```

```
zz.score = 0;
zz.table = 1;
zz.gain  = .0;
scanf("%s", zz.nom);
```

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

```
80 * sizeof(char) + 2 * sizeof(int) + sizeof(float)
```


Paramètre de fonction

```
void afficher(struct joueur j)
{
   printf("[JOUEUR] : %s\n", j.surnom);
   printf("gain   : %f\n", j.gain);
   printf("score : %d\n", j.score);
   printf("table : %d\n", j.table);
}
```

```
struct joueur zz = { "ZZ", 100, 1, 10.0} ;

afficher(zz);
```

Retour de fonction

```
struct joueur initialiser1(char nom[80])
{
   struct joueur j;
   strcpy(j.surnom, nom);
   j.gain  = .0;
   j.score =  0;
   j.table = -1;
   return j;
}
```

```
struct joueur zz;

zz = initialiser("SouriZZ");
```


On ne joue pas tout seul…

```
#define MAX 140
struct joueur joueurs[MAX];
```

```
for(int i = 0; i < MAX; ++i)
{
  char nom[80];
  sprintf(nom, "ZZ%d", i+1);

  joueurs[i] = initialiser1(nom);
}
```

ZZ1 ZZ2 ooo ZZ139 ZZ140

Pointeur de structure

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

*psj ?

(*psj).score ⟺ psj->score

```
psj = (struct joueur *)
      malloc(140*sizeof(struct joueur));
```


Passage par adresse

```
void initialiser2(struct joueur * pzz,
                  char nom[80])
{
   strcpy(pzz->surnom, nom);

   pzz->gain    = .0;
   (*pzz).score =  0;
   (*pzz).table = -1;
}
```

```
struct joueur j;

initialiser2(&j, "SouriZZ");
```

typedef (1)

```
struct joueur {
  char  surnom[80];
  int   score;
  int   table;
  float gain;
};
typedef struct joueur joueur_t;
typedef struct joueur joueur;
```

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

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


typedef (2)

```
typedef struct joueur {
  char  surnom[80];
  int   score;
  int   table;
  float gain;
} joueur_s;
```

```
struct joueur  zz1;
joueur_s       zz2;
```

On peut vous interdire d'utiliser le typedef pour éviter les erreurs



typedef (3)

```
typedef struct {
  char  surnom[80];
  int   score;
  int   table;
  float gain;
} joueur_s;
```

```
struct joueur {
  char  surnom[80];
  int   score;
  int   table;
  float gain;
} quoi;
```

?

```
struct joueur zz1;
```

```
joueur_s      zz2;
```



Compilation séparée

Structure

Type de données

Doit être connu pour être utilisé

Fichier d'entête

Avec les prototypes de fonctions qui utilisent cette structure

Structure autoréférentielle (1)

```
typedef struct {
  char              surnom[80];
  int               score;
  int               table;
  float             gain;

  struct joueur *   suivant;

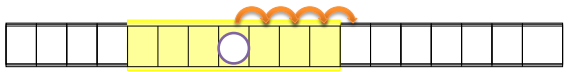
} joueur_s;
```


Structure autoréférentielle (2)

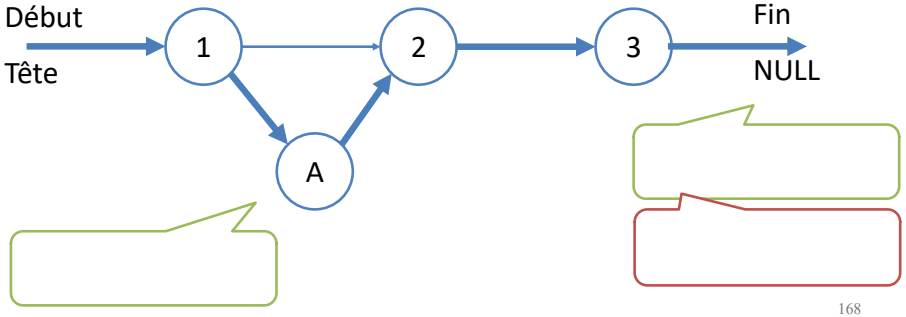
```
typedef struct joueur  joueur_s;

struct joueur {
  char              surnom[80];
  int               score;
  int               table;
  float             gain;
  joueur_s      *   suivant;
};
```


Tableau



Liste chaînée



C'est compliqué une liste chaînée ?

- Qu'est-ce qu'un élément de liste ?
  - La tête ?
  - La fin ?

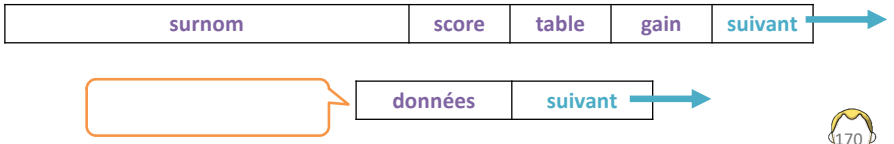
- Quelques opérations élémentaires
  - Sans passer par une fonction
  - En passant par une fonction
- Insertion d'un élément en tête
- Ajout d'un élément en fin
- Suppression de la liste

Une cellule de liste chaînée

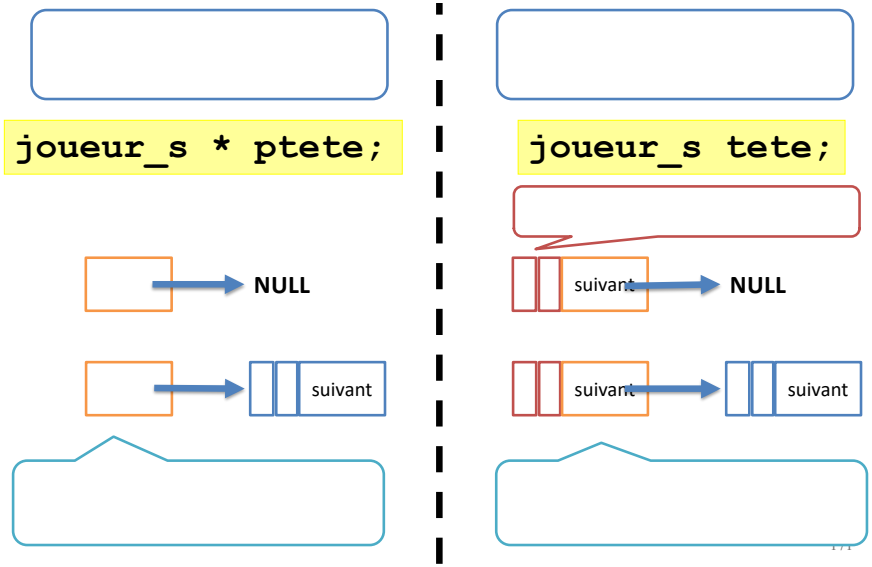
```
typedef struct joueur {
  char              surnom[80];
  int               score;
  int               table;
  float             gain;

  struct joueur * suivant;

} joueur_s;
```



Tête de liste



Liste vide

```
tf.suivant = NULL;
```



Liste à 1 élément

```
tf.suivant = malloc(…);
tf.suivant->suivant = NULL;
tf.suivant->score   = 0;
```



Liste à 2 éléments

```
tf.suivant->suivant = malloc(…);
tf.suivant->suivant->suivant = NULL;
tf.suivant->suivant->score   = 0;
```

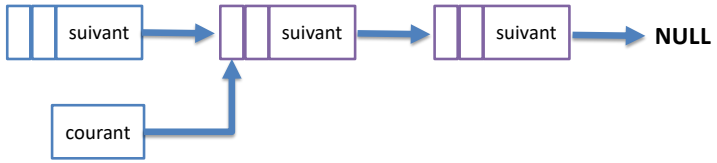


Affichage de la liste (1)




```
if (tf.suivant == NULL)
{
   printf("liste vide");
} else {
   printf("%s\n", tf.suivant->surnom);
   if (tf.suivant->suivant !=NULL) {
      printf("%s\n", tf.suivant->
                        suivant->surnom);
   }
}
```

Affichage de la liste (2)




```
joueur_s * courant = tf.suivant;

while (courant != NULL)
{
   printf("%s\n", courant->surnom);
   courant = courant->suivant;
}
```

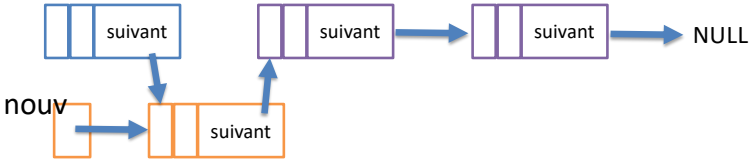

Affichage de la liste (3)

```
void afficher(joueur_s tete)
{
   joueur_s * courant = tete.suivant;

   while (courant != NULL)
   {
      printf("%s : %d\n", courant->surnom,
                          courant->score );
      courant = courant->suivant;
   }
}
```

```
afficher(tf);
```


Insertion en tête (1)




```
joueur_s * nouv = malloc(…);

if (nouv) {
   strcpy(souv->surnom, "nouveau");
   nouv->suiv = tf.suivant;
   tf.suiv = nouv;
}
```

Insertion en tête (2)

```
void insertion(joueur_s tete)
{
    joueur_s * nouv = malloc(…);

    if (nouv) {
        strcpy(nouv->surnom, "nouveau");
        nouv->suiv = tf.suivant;
        tf.suiv = nouv;
   }
}
```

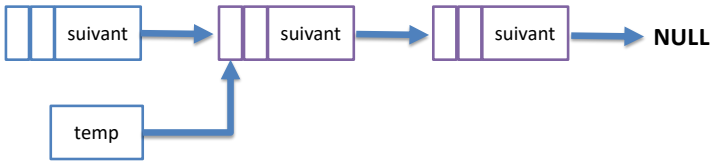
```
insertion(tf);
afficher(tf);
```


Insertion en tête (3)

```
void insertion(joueur_s * ptete)
{
    joueur_s * nouv = malloc(…);

    if (nouv) {
        strcpy(nouv->surnom, "nouveau");
        nouv->suiv   = ptete->suivant;
        ptete->suiv = nouv;
   }
}
```

```
insertion(&tf);
afficher(tf);
```

Suppression de la liste

```
joueur_s *temp;

while (tf.suivant != NULL)
{
  temp = tf.suivant;
  tf.suiv = tf.suivant->suivant;
  /* libérer les champs de temp */
  free(temp);
}
tf.suivant = NULL;
```


Et la suite ?

- Ce que l'on fera en TP
  - La version avec pointeur (tête "réelle")
  - Ajout en fin avec un autre pointeur
  - Insertion dans une liste triée
- Ce que l'on ne fera pas en cours de C
  - Insertion ou suppression quelconque
  - Liste double chaînée
  - D'autres structures de données

STRUCTURES PARTICULIÈRES

ENUM
UNION
CHAMP DE BITS





```
int jour = MARDI;
int jour = 4;
int jour = 10;
int jour = JANVIER;
```

```
#define JANVIER   1
#define FEVRIER   2
#define MARS      3
#define AVRIL     4
#define MAI       5
#define JUIN      6
#define JUILLET   7
#define AOUT      8
#define SEPTEMBRE 9
#define OCTOBRE   10
#define NOVEMBRE  11
#define DECEMBRE  12
```


Enumération ?

```
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);
```


Enumération !

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

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

Point de vue du compilateur

```
enum SEMAINE
{
  lundi,      → 0
  mardi,
  mercredi,
  jeudi,
  vendredi,
  samedi,
  dimanche    → 6
};
```

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


```
enum SEMAINE
{
  lundi = 1,  → 1
  mardi,
  mercredi,
  jeudi,
  vendredi,
  samedi,
  dimanche    → 7
};
```

Choisir des valeurs

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

Compatibilité ?

```
enum PAYS pays;

pays = france;

pays = mercredi;
```


Collision de noms

```
enum Mois
{
  JANVIER,
  FEVRIER,
  MARS
};
```

```
enum Barre
{
  TWIX,
  BOUNTY,
  MARS,
};
```

error: redeclaration of enumerator '**MARS**'
note: previous definition of '**MARS**' with type '**enum Mois**'

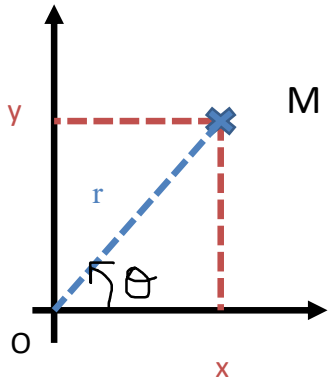
```
enum Mois mois = MARS;
enum Barre jaimepas = MARS;
```

`enum` *vs* macros

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

On peut choisir le type en C23

Représentation d'un point ?



- Un point
- Plusieurs représentations
  - Cartésienne
  - Polaire
- Utilisation des deux formes dans un même programme ?
  - Conversions ?
  - Changement de type ?

Point[1]

```
typedef struct
{
  double r;
  double t;
} polaire_s;
```

```
typedef struct
{
  double x;
  double y;
} cartesien_s;
```

```
cartesien_s p1
= { 0, .1};

polaire_s p2;
p2.r = 0;
P2.t = PI / 2;
```

Facile de convertir mais un point a une forme figée

```
cartesien_s convertirEnCartesien(polaire_s p) {
   cartesien_s retour;
   retour.x = p1.r * cos(p1.t);
   retour.y = p1.r * sin(p1.t);
   return retour;
}
```

Point[2]

```
typedef struct
{
  double a;
  double b;
} point_s;
```

```
// polaire
point_s p1 = { 1, 0};

// cartesien
point_s p2;
p2.a = p1.a * cos(p1.b);
p2.b = p1.a * sin(p1.b);

printf("%lf", p2.a);
```

Le point peut avoir deux formes
Pas de sécurité

```
point_s convertirEnCartesien(point_s p) {
   point_s retour;
   retour.a = p1.a * cos(p1.b);
   retour.b = p1.b * sin(p1.b);
   return retour;
}
```

Point^3

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

```
typedef struct
{
   double a;
   double b;
   enum FORMES f;
} point_s;
```

```
// polaire
point_s p1;
= { 1, 0, POL};

// cartesien
point_s p2;
p2.a = p1.a * cos(p1.b);
p2.b = p1.a * sin(p1.b);
p2.f = CAR;

printf("%lf", p2.a);
```

```
point_s convertirEnCartesien(point_s p) {
   point_s retour;
   // vérifier p.f
   // Flag si d'erreur ?
   return retour;
}
```

Le point peut avoir deux formes
Sécurité relative

Union

```
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

Point^4 (1)

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

```
typedef struct
{
   double r;
   double t;
} polaire_s;
```

```
typedef union
{
   cartesien_s car;
   polaire_s   pol;
} point_u;
```

```
typedef struct
{
   double x;
   double y;
} cartesien_s;
```

```
typedef struct
{
   point_u  val;
   enum FORMES forme;
} point_s;
```


Point^4 (2)

```
double distance_origine( point_s a)
{
   double res;

   switch (a.forme) {
      case CAR :
         res = sqrt(a.val.car.x*a.val.car.x
                  + a.val.car.y*a.val.car.y);
            break;
      case POL :
         res = a.val.pol.r;
            break;
      default : fprintf(stderr, "Erreur\n");
   };
   return res;
}
```

```c
typedef union SDL_Event
{
  Uint32 type;
  SDL_CommonEvent    common;
  SDL_DisplayEvent   display;
  SDL_WindowEvent    window;
  SDL_KeyboardEvent key;
  SDL_UserEvent      user;

  // plein d'autres
} SDL_Event;
```

```c
typedef struct SDL_UserEvent
{
    Uint32 type;
    Uint32 timestamp;
    Uint32 windowID;
    Sint32 code;
    void *data1;
    void *data2;
} SDL_UserEvent;
```

```c
SDL_Event user_event;
SDL_zero(user_event);
user_event.type = SDL_USEREVENT;
user_event.user.code = 2;
user_event.user.data1 = NULL;
user_event.user.data2 = NULL;
SDL_PushEvent(&user_event);
```

https://wiki.libsdl.org/SDL2/SDL_UserEvent

Champs de bits

```c
typedef struct couleur1
{
  unsigned int R;
  unsigned int G;
  unsigned int B;
  unsigned int A;
} couleur1_s;
```

```c
typedef struct couleur2
{
  unsigned int R:8;
  unsigned int G:8;
  unsigned int B:8;
  unsigned int A:8;
} couleur2_s;
```


FONCTIONS À NOMBRE VARIABLE D'ARGUMENTS





CLERMONT AUVERGNE INP UCA Isima



État des lieux

Fonctions avec nombre fixé d'arguments

```c
int max(int a, int b);
```

Liste des types

`[s|f] printf` `[s|f] scanf`

Comment peut-on les coder ?

Nombre d'arguments variable

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

Liste non vide de paramètres fixes

Sauf C23

Variable

Aucun moyen de connaître le nombre d'arguments

Fonction standard vérification compilo

Autant d'arguments que de %

```
int fscanf(FILE *,char *, ...);
fscanf(fichier, "%d %d", &nb1, &nb2);
```


Ce que l'on peut faire..

stdarg.h

`va_list` — Type liste de paramètre

`va_start(liste, nom)` — Initialiser la liste de paramètres

Nom du dernier paramètre fixe

`va_copy(dest, src)` — copier la liste de paramètres

`va_end(liste)` — Libération du pointeur de liste

Conseillé par la norme

`va_arg(liste, type)`

Récupération du paramètre courant de la liste

Conversion obligatoire

Combien ?

int

char

double

float

pointeurs

Conversions …


```
double moyenne(int nombre, ...) {
  double  result = 0;
  int     nb     = 0;
  va_list liste;

  va_start(liste, nombre);
  while (nombre>0) {
    result += nombre;
    nb     += 1;
    nombre = va_arg(liste, int);
  }
  va_end(liste);
  if (nb>0) result /= nb;
  return result;
}
```

Optionnel en C23

Ne Pas Oublier

```
/* exemple d'appel */
printf("%lf", moyenne(-1));
printf("%lf", moyenne(2, 2, 5, -1));
```

```c
int vsscanf(const char * s,
    const char * fmt, va_list args);
```

```c
int sscanf(const char * s,
   const char * fmt , ...)
{
  int ret;
  va_list ap;

  va_start(ap, fmt);
  ret = vsscanf(s, fmt, ap);
  va_end(ap);

  return ret;
}
```

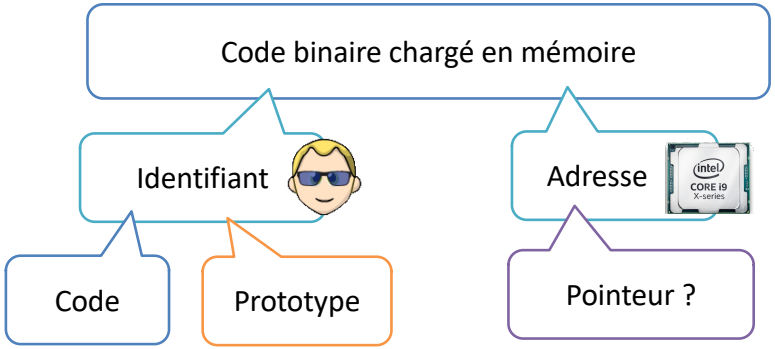

POINTEURS DE FONCTION





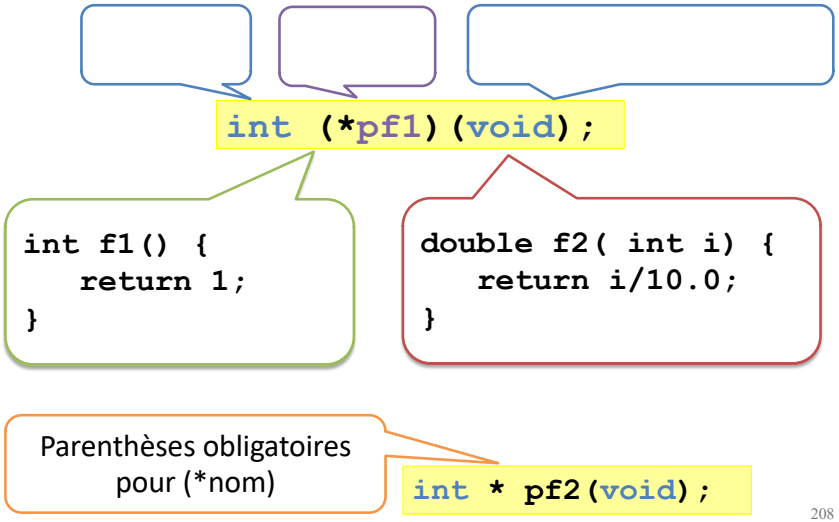
CLERMONT AUVERGNE INP ISIMA

Une fonction ?




```
retour identifiant(paramètres typés)
```


Pointeur de fonction (1)



Pointeur de fonction (2)

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

```
double distance2(double a, double b) {
  return a*a+b*b;
}
```

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

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

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



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

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

compréhension

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

```
pf2 =
```

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

```
pf2 =
```

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

portabilité

Exemple



```
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;
}
```



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

```
int choix, val;
```

```
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 … */
```

```
int   t1[4];
```

```
t1[choix]= val;
```

```
int * t2[4]
  = {&a, &b, &c, &d };
```

```
(*t2)[choix] =val;
```

Mise en œuvre

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

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

OU

```
int (*tabptrfn[4]) () = { f1, f2, f3, f4 };
```

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

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


Une fonction en paramètre

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

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

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


Trier un tableau ?

```
void tri(double t[]
  for (int i = 1;
    double cle = t[i]
    int j = i - 1

    while (j >= 0 && t[j] > cle) {
      t[j + 1] = t[j];
      --j;
    }
    t[j +
  }
}
```


Travail préalable

```
while (j >= 0 && t[j] > cle) {
      t[j + 1] = t[j];
      --j;
}
```

"Même" code

Paramétrable !

```
while (j >= 0 && inferieur(cle, t[j])) {
      t[j + 1] = t[j];
      --j;
}
```

Efficace ?

```
int inferieur(double a, double b) {
  return a < b;
}
```

Paramétrer le tri ?

Écrire une macro

TP

```
while (j >= 0 && C(cle, t[j])) {
      t[j + 1] = t[j];
      --j;
}
```

```
TRI(t, n, inferieur);
```

Utiliser une fonction en paramètre

```
tri(t, n, inferieur);
```

```
tri(t, n, superieur);
```


Tri paramétré par une fonction

```
int inferieur(double a, double b) {
  return a < b;
}
```

```
void trier(double t[], int n ,
  int (*compare) (double, double) )
{
  // …
  while (j >= 0 && compare(cle, t[j]))
  // …
}
```


Peut-on faire abstraction du type ?

```
double a;
```

```
int b;
```

```
struct Personne {
  char n[50];
  char p[50];
} c;
```

Point comun ?

variable

adresse

type *

void *

La macro reste possible

```
int inferieur(void * p1, void * p2) {
  int * i1 = (int *)p1;
  int * i2 = (int *)p2;
  return *i1 < *i2;
}
```


Tri complètement paramétré

```
void tri(void * t, int n , int s
    int (*comp)(void *, void *))
{
}
```

```
double cle = t[i];
```

```
double * cle = &t[i];
```

```
double * cle = t + i;
```

```
void * cle
    = t + s *i;
```

Taille d'un élément en octet(s)

```
t[j + 1] = t[j];
```

```
memcpy(t+s*(j+1), t+s*j, s);
```

Tri et recherche dans stdlib

```
void qsort (
  void *base, size_t nel, size_t width,
  int (*compar)(const void *, const void *)
);
```

Renvoie -1, 0 ou 1

Recherche dans un tableau trié

```
void *
bsearch(const void *key,
 const void *base, size_t nel, size_t width,
 int (*compar) (const void *, const void *));
```


Sortie "propre" de programme

```
FILE * log = NULL;
double * g = NULL;

void procedure() {
  if (erreur)
     exit(-1);
);

int main() {
  atexit(sortie);
  procedure(…);
  return 0;
);
```

Autre sortie

Sortie usuelle

```
void sortie() {
  free(g);
  if (log)
    fclose(log);
}
```


Réagir à un signal

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

```
void avertissement()
{
   printf("CTRL+C ne marche pas \n");
}

int main(){

  signal(SIGINT, avertissement);

  for(;;) {
    printf("Zzzz");
    sleep(1);
  }

 return 0;
}
```


typedef

- Forme habituelle

```
typedef nom_deja_defini nouvel_alias;
```

```
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;
```

Retour de fonction

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

```
double (*) (double) f(char);
```

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

```
PTR f(char);
```

```
double (*f (char))(double);
```


FLUX

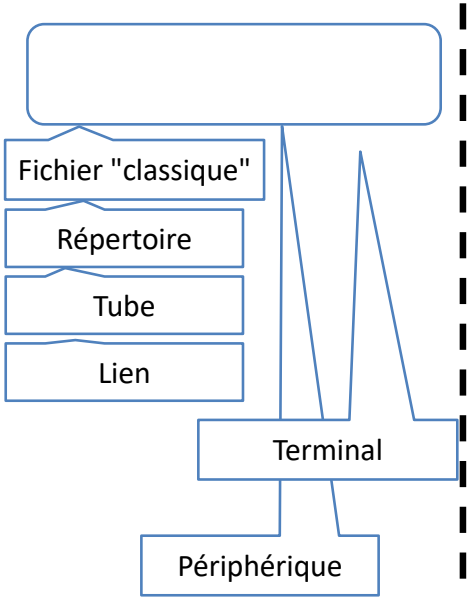
1. Compléments sur les fichiers / flux
2. Flux binaires



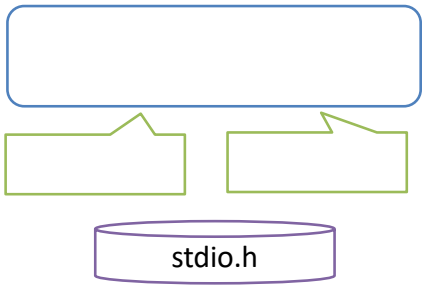


CLERMONT AUVERGNE INP ISIMA

Fichiers



flux



Il existe des fonctions de bas niveau

Structure FILE

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

Emplacement disque

Mode d'ouverture

Type

Nom

Mémoire-tampon

BUFSIZ

Accès asynchrone

Issu du stdio.h de MinGW

Ouverture

r+ w+ a+

NULL ? Identification read write append

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

Préparer la manipulation du fichier

binary

Fichiers déjà ouverts FOPEN_MAX

stdout stdin stderr

Fermeture de fichier

Flux valide

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

0 si succès

Forcer l'écriture des données pas encore écrites

Abandonner les informations pas encore prises en compte (lues)

Rendre les ressources associées à la manipulation



Mémoire-tampon

3 manières de gérer les lectures / écritures

Non bufferisé En ligne Complètement bufferisé

Gérer l'écriture du buffer ?



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

Buffer de sortie

Tous les buffers si NULL

Informations supplémentaires

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

Fin de fichier atteinte si non nul

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

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

erreur rencontrée si non nul

errno Variable globale

Comment lire ou écrire ?

- Accès séquentiel

Curseur "virtuel"

	Lecture	Écriture
Texte	`fscanf` `fgets`	`fprintf`
Binaire	`fread`	`fwrite`

- Accès aléatoire

`ftell`	`fseek`

Lecture / écriture binaire



Où placer ce qui est lu

Taille en mémoire d'un élément

```
size_t fread(void *ptr, size_t taille,
             size_t nbobj, FILE * flux);
```

Nb d'élément à lire

Ce qui a été lu / écrit

Ce qui est à écrire

```
size_t fwrite(const void *ptr, size_t taille,
              size_t nbobj, FILE * flux);
```


Exemple

- Gestion d'un répertoire personnel
  - Nom du contact
  - Numéro de téléphone

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

- Tableau en mémoire
  - À écrire/lire sur le disque

```
entree_s repertoire[MAX];
```



```
void enregistrer(entree_s repertoire[],
                 int taille)
{
  FILE * flux = fopen("sauvegarde", "wb");
  int    res = 0;

  if (flux)
  {
    res = fwrite(repertoire, sizeof(entree_s),
                 taille, flux);
    if (res != taille)
      fprintf(stderr, "probleme\n");
    fclose(flux);
  }
}
```

```c
int lire(entree_s repertoire[])
{
  FILE * flux = fopen("sauvegarde", "rb");
  int    taille = 0;

  if (fichier)
  {
    while(!feof(flux))
      fread(&repertoire[taille++],
            sizeof(entree_s), 1, flux);
    fclose(flux);
  }
  return taille;
}
```


Accès aléatoire

```c
long ftell(FILE * flux);
```

Position du curseur dans le flux

Changer la position du curseur

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

0 si OK

déplacement (+ ou -)

origine

SEEK_SET
SEEK_END
SEEK_CUR

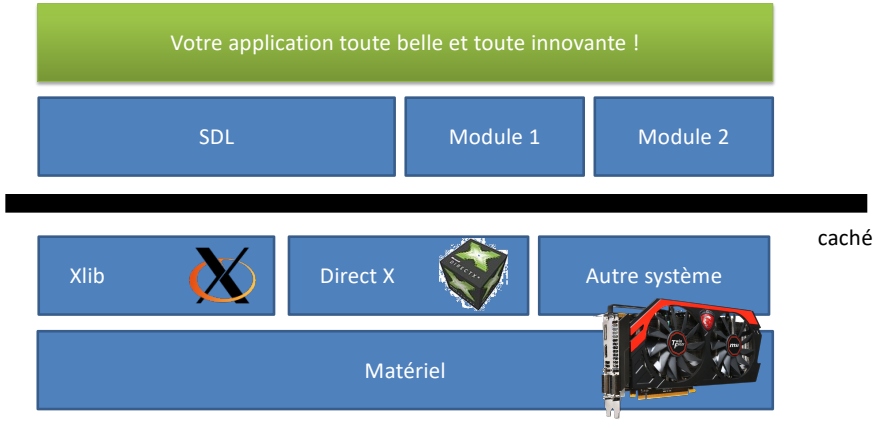
PROGRAMMATION D'UNE INTERFACE GRAPHIQUE



Simple DirectMedia Layer

- Bibliothèque de développement multiplateforme
  - UNIX, LINUX, WINDOWS, MAC, Mobiles
- Écrite en C, utilisable avec plein d'autres langages
- Née en 1998, version 2 depuis 2013, version 3 publiée en 2025
- Bas niveau aux périphériques et à la couche graphique
  - Vulkan (OpenGL) et Direct3D
  - Greffons standards et moins standards
    - SDL2_TTF, SDL2_mixer, SDL2_image, SDL2_net, SDL2_gfx
- Orientée événements
- Multifenêtres

Bibliothèque multiplateforme



Bibliothèque bas niveau

- Formes basiques
  - Points, lignes, rectangles et textures en standard !
  - Écrire à l'écran avec le greffon standard SDL2_TTF
  - Lire des formats images autre que BMP avec SDL2_Image
  - Afficher de ellipses dans le greffon non standard SDL2_GFX
- Pas de composants hormis les fenêtres ou les boites de dialogues
- D'autres greffons standards : son, réseau

Squelette d'une application SDL

- Initialisations graphiques
  - Variables globales ?
  - Structures
- Boucle principale = gestion des événements asynchrones
  - Attente d'un événement (bloquant ou non)
  - Gestion de l'événement
- Rendu des ressources graphiques

File d'attente des événements

- Permet une gestion asynchrone des événements
- Lecture bloquante ou non d'un événement
- Eventuellement avec une limite
- Insertion d'événement système / système graphique / utilisateur

- Filtrage possible avant insertion
- Désactivation d'événements

Types d'événements (1)

- Structure SDL_Event avec un champ *type*
- Système (notamment sur les mobiles)
- Clavier
- Souris
- Contrôleur / joystick / toucher / mouvement
- Moteur de rendu
- Audio
- Utilisateur / Personnel
- Fenêtre →

Types d'événements (2) - Fenêtre

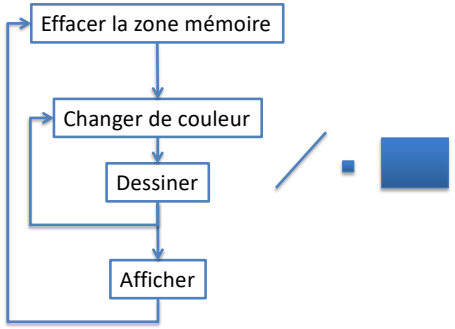
- Affichage / exposition

> Les applications sont responsables de leur affichage !!!

- Focus clavier
- Entrée ou sortie de la souris
- Fermeture
- Réduction / restauration / maximisation
- Redimensionnement

Contexte de moteur de rendu (*renderer*)

- Dessin 2D dans une zone mémoire-tampon (*buffer*)
- Affichage différé
- Exploitation des capacités des cartes graphiques



Texture

- Zone dans laquelle on peut dessiner
  - Chargement d'images
  - Police de caractères (police TTF)
  - Import de surface (SDL 1.2)
  - Potentiellement gérée/accélérée par le matériel
- Cible temporaire d'un *renderer*
- Copie d'une texture vers un *renderer*

Codes fournis

- Initialisation de la bibliothèque SDL2
- Exemple d'affichage d'un rectangle, d'un texte et d'un image
- Gestion de quelques événements

SDL_Event

```
typedef union SDL_Event
{
  Uint32 type;
  SDL_CommonEvent common;
  SDL_WindowEvent window;
  SDL_KeyboardEvent key;
  SDL_TextInputEvent text;
  SDL_MouseMotionEvent motion;
  SDL_MouseButtonEvent button;
  SDL_MouseWheelEvent wheel;
  SDL_JoyAxisEvent jaxis;
  SDL_QuitEvent quit;
  SDL_UserEvent user;
  SDL_TouchFingerEvent tfinger;
  // …
} SDL_Event;
```

```
SDL_WaitEvent()
SDL_WaitEventTimeout()
SDL_PollEvent()
```


Opérations binaires en C

OU
|

0	0	0
0	1	1
1	0	1
1	1	1

Mettre à 1 les bits
d'un masque en fonction de certains critères

ET
&

0	0	0
0	1	0
1	0	0
1	1	1

OU EXCLUSIF
^

0	0	0
0	1	1
1	0	1
1	1	0

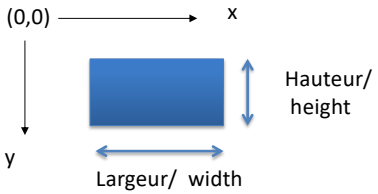
Afficher deux fois un pixel l'efface

Exemples

- 5 && 7
- 5 || 7
- !5
- 5 & 7
- 5 | 7
- 5 << 2
- 5 >> 2
- ~ 5

2^n	64	32	16	8	4	2	1
5	0	0	0	0	1	0	1
7	0	0	0	0	1	1	1

Dessiner





Couleurs : (ROUGE, VERT, BLEU, ALPHA)

Unix

- Notion de DISPLAY
  - Ecran(s)
  - Entrées : clavier, souris, joystick, dalle tactile …
  - 172.0.0.1:1
- Inversion géographique client / serveur
  - serveur physique / programme
  - client graphique / serveur graphique