

COURS 1 - INTRODUCTION

G. Bianchi, G. Blin, A. Bugeau, S. Gueorguieva, R. Uricaru

2015-2016

Programmation 1 - uf-info.ue.prog1@diff.u-bordeaux.fr

université
de **BORDEAUX**

- ▷ Apprendre le langage C
- ▷ Analyser un problème pour écrire un programme qui le résout, de façon élégante et modulaire
- ▷ Coder proprement
- ▷ Savoir respecter un cahier des charges lors de projets

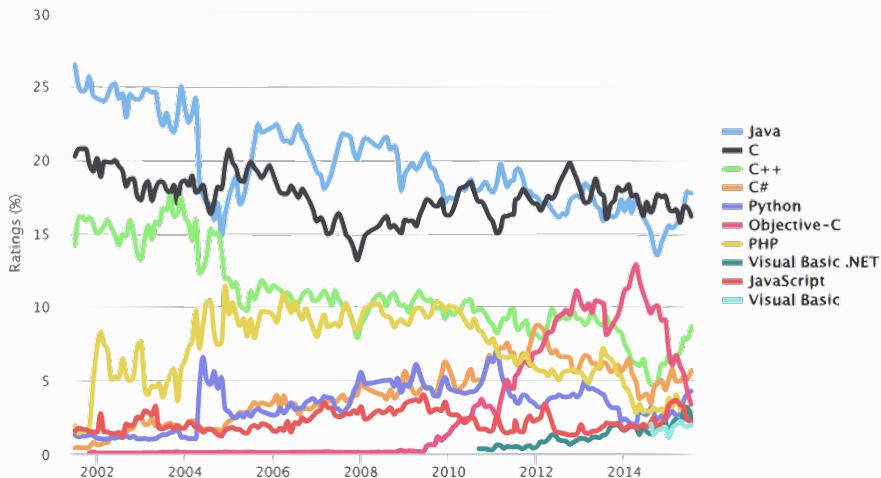
DÉROULEMENT DU COURS DE PROGRAMMATION 1

Semaine	Cours	TD et TP
37	3 séances, lun, mer et jeu	
38	2 séances, lun et jeu	1 TD/TP
39	2 séances, lun et jeu	1 TP/TP et 1 TD/TP
40	1 séance, lun	1 TP/TP et 1 TD/TP
41	1 séance, lun	1 TP/TP et 1 TD/TP
42	1 séance, lun	1 TP/TP et 1 TD/TP
43	2 séances, lun et jeu	1 TD/TP
44	Vacances	
45	1 séance, lun (Devoir Surveillé)	1 TD/TP
46	2 séances, lun et jeu	1 TP/TP
47	2 séances, lun et jeu	1 TP/TP et 1 TD/TP
48	1 séance, lun	1 TD/TP
49	1 séance, lun	1 TP/TP et 1 TD/TP
50	1 séance, lun	1 TP/TP et 1 TD/TP
Total	20 créneaux	19 séances

POURQUOI LE C ?

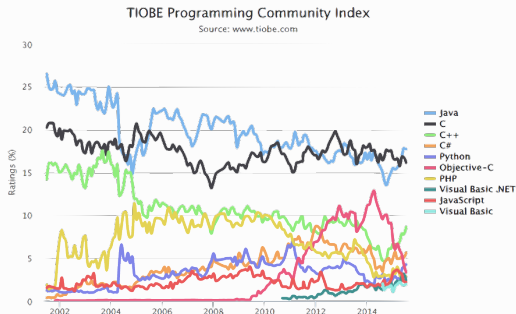
TIOBE Programming Community Index

Source: www.tiobe.com



▷ Indicateur de popularité des langages de programmation

POURQUOI LE C ?



- ▷ Toujours autant utilisé
- ▷ Privilégié pour sa proximité avec le système
- ▷ Portable (avec des efforts du programmeur)

- ▷ Langage compilé
 - ▷ Typage statique (chaque variable, constante et expression a un type défini à la compilation)
 - ▷ Sûreté du typage (détection d'erreur à la compilation)
 - ▷ Optimisation (e.g., éliminer les informations de typage dans le fichier objet produit - gain de mémoire)
 - ▷ A la fois haut et bas niveaux (1 instruction = nombre +/- prévisible d'instructions machines)
 - ▷ Gestion de la mémoire explicite (gain en performance)
 - ▷ Pas de surcoût à l'exécution
- Il ne fait rien d'autre que ce que vous lui dites!

HISTORIQUE

- ▷ C, parce qu'après le langage B (évolution avec e.g., typage)
- ▷ Inventé en 1972 (en même temps qu'UNIX), dans les Laboratoires Bell, par Dennis Ritchie & Kenneth Thompson (à l'origine du langage B)
- ▷ Normalisé en 1989 (norme ANSI* C) puis suivie de C99 (ISO†) et C11 (2011)
 - ▷ Définition de "règles à suivre" à des fins de portabilité

*American National Standards Institute

†International Organization for Standardization

- ▷ Langage impératif

 - ▷ Un état (mémoire) + des instructions élémentaires exécutables par le processeur pour modifier cet état

 - ▷ 4 types d'instructions (séquence, affectation, conditionnelle et la boucle) + branchements (déplacements dans la mémoire)

- ▷ B. Kernighan et D. Ritchie, The C Programming Language
- ▷ I. Horton, Beginning C, Apress, 2006
- ▷ J.-M. Léry, Le langage C, Pearson Education, 2005
- ▷ A. Braquelaire, Méthodologie de la programmation en C, Dunod, 2005
- ▷ C. Delannoy, Programmer en langage C - Cours et exercices corrigés, Eyrolles, 2006

PREMIER PROGRAMME EN C

C'EST QUOI UN PROGRAMME ?

- ▷ Succession d'instructions assez simples exécutables par l'ordinateur souvent regroupées en fonctions *
- ▷ Opérations arithmétiques, de chargement et stockage de valeurs, d'envoi à des périphériques, ...
- ▷ Une succession de 0 et de 1
- ▷ Un langage de programmation = { instructions dans un langage lisible par un humain }
- ▷ Traduits en langage machine (en binaire) par un compilateur, un assembleur ou un interpréteur

*Factorisation du code = portion de code effectuant des instructions sur de possibles entrées et pouvant renvoyer un résultat

HELLO, WORLD! DE 1978

Proposé en exemple en 1978 dans The C Programming Language de B. Kernighan et D. Ritchie.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

▷ Programme affichant `hello, world` suivi d'un retour à la ligne

HELLO, WORLD! DE 1978

Proposé en exemple en 1978 dans The C Programming Language de B. Kernighan et D. Ritchie.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

▷ `#include <stdio.h>` inclut les déclarations des fonctions d'entrées-sorties de la bibliothèque standard du C[†] (dont `printf`)

[†]Des fonctions utiles à votre disposition

HELLO, WORLD! DE 1978

Proposé en exemple en 1978 dans The C Programming Language de B. Kernighan et D. Ritchie.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

▷ `main` est le nom de la fonction principale, aussi appelée point d'entrée du programme

HELLO, WORLD! DE 1978

Proposé en exemple en 1978 dans The C Programming Language de B. Kernighan et D. Ritchie.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

▷ Les parenthèses après `main()` indiquent que `main` est une fonction.

HELLO, WORLD! DE 1978

Proposé en exemple en 1978 dans The C Programming Language de B. Kernighan et D. Ritchie.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

▷ Les accolades { et } entourent les instructions constituant le corps de la fonction `main()`

HELLO, WORLD! DE 1978

Proposé en exemple en 1978 dans The C Programming Language de B. Kernighan et D. Ritchie.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

▷ `printf` est une fonction d'écriture sur la sortie standard (la console par défaut)

HELLO, WORLD! DE 1978

Proposé en exemple en 1978 dans The C Programming Language de B. Kernighan et D. Ritchie.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

▷ " " délimite une chaîne de caractères terminée ici par une séquence d'échappement représentant le saut de ligne

HELLO, WORLD! DE 1978

Proposé en exemple en 1978 dans The C Programming Language de B. Kernighan et D. Ritchie.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

▷ Un point-virgule termine l'instruction

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    printf("hello, world\n");
    return EXIT_SUCCESS;
}
```

- ▷ `int` est le type renvoyé par la fonction `main`
- ▷ Implicite jusqu'en C89, il était couramment omis mais ce n'est plus le cas en C99

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    printf("hello, world\n");
    return EXIT_SUCCESS;
}
```

▷ Le mot clé **void** entre parenthèses signifie que la fonction n'admet aucun paramètre

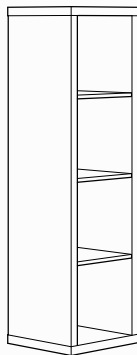
```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    printf("hello, world\n");
    return EXIT_SUCCESS;
}
```

▷ L'instruction `return EXIT_SUCCESS;` indique que la fonction `main` retourne la valeur de la constante `EXIT_SUCCESS` (à savoir `0`; qui est bien du type entier) définie dans `stdlib.h`

DES SOURCES À L'EXÉCUTABLE

▷ Un fichier source est un texte exprimé dans un langage de programmation décrivant des instructions permettant d'obtenir un objet fonctionnel précis

KALLAX

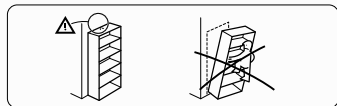
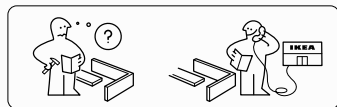
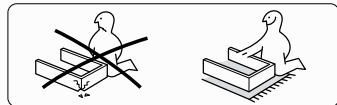
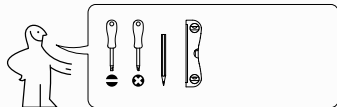


LES FICHIERS SOURCES

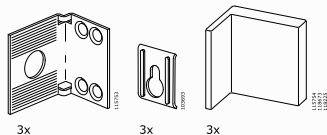
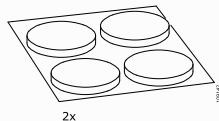
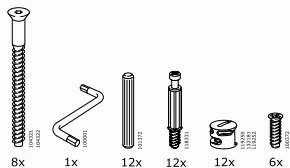
▷ Pouvant suivre des conventions données

▷ Ce sont bien des règles de bons sens

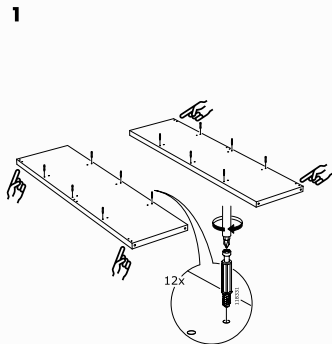
▷ Que l'on est pas obligé de suivre mais ⚠



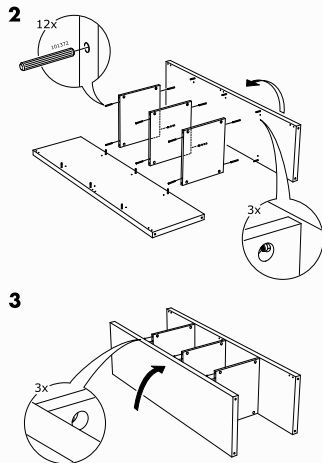
▷ Utilisant des outils pré-existants et disponibles (librairies)



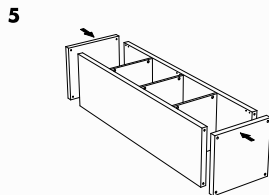
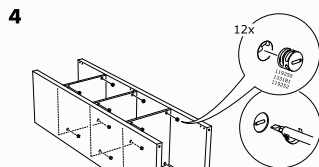
▷ Correspondant à une suite d'instructions (souvent répétitives)



▷ Correspondant à une suite d'instructions (souvent répétitives)

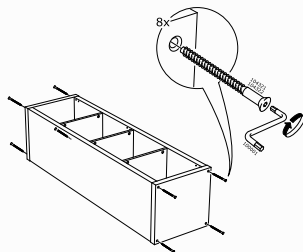


▷ Correspondant à une suite d'instructions (souvent répétitives)



▷ Correspondant à une suite d'instructions (souvent répétitives)

6



- ▷ Fichiers textes contenant les instructions
- ▷ Extension spéciale `.c`
- ▷ Syntaxe très précise à respecter
- ▷ Séparateurs = espace + tabulation + retour à la ligne
- ▷ Le C est sensible à la casse
- ▷ Les séparateurs peuvent être librement utilisés pour la mise en page, car ils sont équivalents à un seul espace dans la plupart des cas

▷ Un binaire est un fichier qui contient des instructions machines correspondant à une entité que l'on ne peut plus modifier mais que l'on peut utiliser dans un exécutable



LES FICHIERS EXÉCUTABLES

▷ Un exécutable est un binaire ayant un point de début d'exécution et pouvant correspondre à plusieurs fichiers binaires.



- ▷ La génération d'un exécutable à partir des fichiers sources se fait en plusieurs étapes, qui sont souvent automatisées à l'aide d'outils (e.g., make)
- ▷ Le compilateur est un programme qui lit des sources et produit un binaire possiblement exécutable

- ▷ La compilation comporte 4 étapes :
 - ▷ 1 - pré-compilation (.c → .c)
 - ▷ 2 - compilation (.c → .s)
 - ▷ 3 - assemblage [‡] (.s → .o)
 - ▷ 4 - édition de liens (.o → exe)

[‡]Parfois regroupée avec la précédente par établissement d'un flux de données interne

1 - LA PRÉ-COMPILATION

- ▷ La pré-compilation prend un ou plusieurs fichiers sources en entrée et produit un unique fichier source exempt de directives de pré-compilation (définies par des lignes commençant par #)
- ▷ Les fichiers sources sont analysés par le préprocesseur qui effectue des transformations purement textuelles
 - ▷ Les macros `#define` sont interprétées et remplacées par leurs évaluations → chercher-remplacer
 - ▷ Tous les `#include` sont remplacés par leurs contenus → copier-coller

LA PRÉ-COMPILATION - EXEMPLE 1 - #DEFINE

```
#define MAX 10
int main(){
    int i=0, j=0;
    for(i=0;i<MAX;++i){
        j+=i;
    }
    return j;
}

int main(){
    int i=0, j=0;
    for(i=0;i<10;++i){
        j+=i;
    }
    return j;
}
```

- ▷ Directive de substitution : `#define NAME VALUE`
- ▷ Par convention, les noms de macro sont en majuscules

LA PRÉ-COMPILATION - EXEMPLE 2 - #INCLUDE

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

#define MESSAGE "hello, world!\n"

int main(void)
{
    printf(MESSAGE);
    return EXIT_SUCCESS;
}

typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef short __int16_t;
...
void perror(const char *);
int printf(const char * rest...
int putc(int, FILE *);
...
int main(){
    printf("hello, world!\n");
    return 0;
}
```

▷ Directive d'inclusion : #include NOM_FICHIER

▷ Les fichiers .h sont disponibles dans /usr/include

2 - COMPILATION

▷ Traduction du fichier généré par le préprocesseur en assembleur

▷ Instructions du microprocesseur dans un langage lisible (si, c'est vrai)

```
int main(void){
    int i=0, j=0;
    for(i=0;i<10;++i){
        j+=i;
    }
    return j;
}
```

```
0: push    %ebp
1: mov     %esp,%ebp
3: sub     $0xc,%esp
6: movl   $0x0,-0x4(%ebp)
d: movl   $0x0,-0x8(%ebp)
14: movl   $0x0,-0xc(%ebp)
1b: movl   $0x0,-0x8(%ebp)
22: cmpl   $0xa,-0x8(%ebp)
29: jge    4a <_main+0x4a>
2f: mov    -0x8(%ebp),%eax
32: mov    -0xc(%ebp),%ecx
35: add    %eax,%ecx
37: mov    %ecx,-0xc(%ebp)
3a: mov    -0x8(%ebp),%eax
3d: add   $0x1,%eax
42: mov    %eax,-0x8(%ebp)
45: jmp    22 <_main+0x22>
4a: mov    -0xc(%ebp),%eax
4d: add   $0xc,%esp
50: pop    %ebp
51: ret
```


▷ Même code généré avec de l'optimisation.

0x2d = 45 = somme des nombres de 0 à 9

```
int main(void){
    int i=0, j=0;
    for(i=0;i<10;++i){
        j+=i;
    }
    return j;
}
```

```
0: push    %ebp
1: mov     %esp,%ebp
3: mov     $0x2d,%eax
8: pop     %ebp
9: ret
```

3 - ASSEMBLAGE - FICHER OBJET

▷ Transformation du code assembleur en un fichier binaire (i.e., directement compréhensible par le processeur)

```
int main(void){
    int i=0, j=0;
    for(i=0;i<10;++i){
        j+=i;
    }
    return j;
}
```

```
<v1.o>
55 89 e5 83 ec 0c c7 45 fc
00 00 00 00 c7 45 f8 00 00
00 00 c7 45 f4 00 00 00 00
c7 45 f8 00 00 00 00 81 7d
f8 0a 00 00 00 0f 8d 1b 00
00 00 8b 45 f8 8b 4d f4 01
c1 89 4d f4 8b 45 f8 05 01
00 00 00 89 45 f8 e9 d8 ff
ff ff 8b 45 f4 83 c4 0c 5d
c3
```

```
<v2.o>
55 89 e5 b8 2d 00 00 00 5d
c3
```

▷ Généralement, la compilation et l'assemblage se font dans la foulée

- ▷ Un programme est souvent séparé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des bibliothèques de fonctions standard déjà écrites
- ▷ Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets en utilisant les symboles fournis (e.g., `printf`) et manquants de chaque fichier objet
- ▷ Lorsqu'un symbole est manquant dans un objet mais fournit dans un autre, alors les deux sont liés et le symbole est résolu
- ▷ L'édition de liens produit alors un fichier dit exécutable

▷ GNU Compiler Collection - GCC

▷ Compilation et assemblage

```
$>gcc HelloWorld.c -c
```

▷ Edition des liens

```
$>gcc HelloWorld.o -o HelloWorld
```

▷ Execution

```
$>./HelloWorld  
hello, world!
```

- ▷ `-std=c99` : définition de la norme à suivre
- ▷ `-Wall` : affiche tous les avertissement

```
int main(void){
    int i, j;
    for(i=0;i<10;++i){
        j+=i;
    }
    return j;
}
```

```
toto.c:4:5: warning: variable 'j' is
uninitialized when used here
```

```
    j+=i;
    ^
```

```
toto.c:2:11: note: initialize the
variable 'j' to silence this warning
```

```
    int i, j;
        ^
```

```
    = 0
```

```
1 warning generated.
```

LES MESSAGES D'ERREURS

- ▷ Lors de la compilation, le compilateur indique, lorsqu'il ne peut compiler, des messages d'erreur
- ▷ Il indique parfois des avertissements (`warnings`) qui ne l'empêchent pas de compiler mais qui selon lui peuvent être source d'erreur

```
err.c:12:5: error: ...
```

indique que lors de la compilation du fichier `err.c`, une erreur a été trouvée à la ligne 12, au caractère 5


▷ `err.c:21:12 error: parse error before ';'`
Erreur de syntaxe repérée avant le caractère `' ; '` à la ligne 21 du fichier `err.c`

▷ `err.c:13:4 error: use of undeclared identifier 'j'`
La variable `'j'` n'a pas été déclarée

EXEMPLES D'ERREURS DE COMPILATION

```
▷ err.c:16:6 error: redeclaration of 'i'  
err.c:15:5 error: 'i' previously declared here
```

La variable 'i' a été déclarée deux fois, lignes 15 et 16

▷  Comme pour les trains, une erreur peut en cacher une autre ...

▷ Toujours les traiter dans l'ordre !

▷ `err.c:17:12 warning: unused variable 'dy'`
La variable `'dy'` est déclarée à la ligne 17 mais pas utilisée

▷ `err.c:25:16 warning: control reaches end of non-void function`
Une fonction qui doit retourner une valeur n'en retourne pas à la ligne 25

▷ `err.c:21:20` warning: statement with no effect

Une expression à la ligne 21 n'a pas d'effet

▷ `err.c:25:12` warning: implicit declaration of function `'calc'`

La fonction `'calc'` n'a pas été déclarée avant son utilisation à la ligne 25

▷ Après l'assemblage, la liaison entre eux des différents fichiers objets peut échouer

▷ Suite à l'absence de définition d'un symbole (fonction ou variable)

```
bar.c:2:12 undefined reference to 'foo'
```

La fonction `foo` n'a pas de définition.

▷ Ou à la présence de symbole en plusieurs versions

```
multiple.c:12:2 multiple definition of 'pow'  
firsttime.c:1:1 first defined here
```

La fonction `pow` est définie plus d'une fois

PROPRETÉ D'UN CODE

```
int main(void)
{
    int i,
        j=0;

    int u;
    for(i =0 ;i < 10 ;i ++){ j=j+i;
    u=u+ j;
    }
    return u;
}
```

```
int main(void){
    int i,j=0,u;
    for(i=0;i<10;i++){
        j=j+i;
        u=u+j;
    }
    return u;
}
```

```
int    main(    void)

{
    int i,
      j=0;

int u;
for(i =0 ;i < 10 ;i ++) {    j=j+i;
u=u+    j;
    }
return    u;
    }
```

```
int main(void){
    int i,j=0,u;
    for(i=0;i<10;i++){
        j=j+i;
        u=u+j;
    }
    return u;
}
```

▷ L'indentation permet la relecture aisée du code et la détection d'erreur de syntaxe (e.g., absence d'une accolade fermante)

RÉUTILISABILITÉ

```
/* Alloue et retourne un tableau de 'taille' entiers, ou NULL en
   cas d'erreur. Chaque case d'indice i du tableau est initialisee
   avec f(i), ou f est une fonction passee en argument de f_alloc.*/
int* f_alloc(int taille, int (*f)(int)) {
    int* t=(int*)malloc(taille*sizeof(int));
    if (t==NULL) return NULL;
    int i;
    for (i=0; i<taille; i++){ t[i]=f(i);}
    return t;
}
```

```
/* Affiche un tableau passe en parametre*/
void affiche_tableau(int tableau[], int taille){
    int i; /* declaration d'une variable i*/
    for(i=0;i<taille;i++){ /* on parcourt le tableau*/
        printf("%d", tableau[i]); /* en affichant chaque case*/
    }
}
```


RÉUTILISABILITÉ

```
/* Alloue et retourne un tableau de 'taille' entiers, ou NULL en
   cas d'erreur. Chaque case d'indice i du tableau est initialisee
   avec f(i), ou f est une fonction passee en argument de f_alloc.*/
int* f_alloc(int taille, int (*f)(int)) {
    int* t=(int*)malloc(taille*sizeof(int));
    if (t==NULL) return NULL;
    int i;
    for (i=0; i<taille; i++){ t[i]=f(i);}
    return t;
}
```

```
/* Affiche un tableau passe en parametre*/
void affiche_tableau(int tableau[], int taille){
    int i; /* declaration d'une variable i*/
    for(i=0;i<taille;i++){ /* on parcourt le tableau*/
        printf("%d", tableau[i]); /* en affichant chaque case*/
    }
}
```

▷ Il faut commenter son code mais judicieusement

```
a f(a b, c d){  
  if (b==NULL) return NULL;  
  if (b->e==d) return b;  
  return f(b->a,d);  
}
```

```
ColorList find_color(ColorList l, Color c){  
  if (l==NULL) return NULL;  
  if (l->color==c) return l;  
  return find_color(l->next,c);  
}
```

```
a f(a b, c d){  
  if (b==NULL) return NULL;  
  if (b->e==d) return b;  
  return f(b->a,d);  
}
```

```
ColorList find_color(ColorList l, Color c){  
  if (l==NULL) return NULL;  
  if (l->color==c) return l;  
  return find_color(l->next,c);  
}
```

▷ Il faut choisir des identificateurs adaptés

```
ColorList find_color(ColorList l, Color c){
    if (l==NULL) return NULL;
    if (l->color==c) return l;
    return find_color(l->next,c);
}
```

```
listacores achacor(listacores l, cor c){
    if (l==NULL) return NULL;
    if (l->cor==c) return l;
    return achacor(l->proxima,c);
}
```

```
ColorList find_color(ColorList l, Color c){
    if (l==NULL) return NULL;
    if (l->color==c) return l;
    return find_color(l->next,c);
}
```

```
listacores achacor(listacores l, cor c){
    if (l==NULL) return NULL;
    if (l->cor==c) return l;
    return achacor(l->proxima,c);
}
```

▷ Il faut écrire du code lisible par tous

```
int* pstrand(int* p){  
    while (!*p++);  
    return (*p++&=0xE8)?p:0;  
}
```

```
i,j;main(n){for(;i<1E4;j||printf("%d ", n,++i))for(j=n++;n%j--;);)}
```

```
int* pstrand(int* p){  
    while (!*p++);  
    return (*p++&=0xE8)?p:0;  
}
```

```
i,j;main(n){for(;i<1E4;j||printf("%d ", n,++i))for(j=n++;n%j--;);)}
```

▷ Il faut éviter le style gourou

COHÉRENCE

```
/* This function looks for the given
 * color in the given color list.
 */
ColorList findColor(ColorList l, Color c) {
    if (l==NULL) {
        return NULL;
    }
    if (l->color==c) {
        return l;
    }
    return findColor(l->next,c);
}
```

```
/* This function looks for the given color */
/* in the given color list. */
ColorList find_Color(ColorList l, Color c)
{
    if (l == NULL) return NULL ;
    if (l->color == c) return l ;
    return find_Color(l->next , c) ;
}
```



```
/* This function looks for the given
 * color in the given color list.
 */
ColorList findColor(ColorList l, Color c) {
    if (l==NULL) {
        return NULL;
    }
    if (l->color==c) {
        return l;
    }
    return findColor(l->next,c);
}
```

```
/* This function looks for the given color */
/* in the given color list. */
ColorList find_Color(ColorList l, Color c)
{
    if (l == NULL) return NULL ;
    if (l->color == c) return l ;
    return find_Color(l->next , c) ;
}
```

▷ Il faut une cohérence de présentation

DOGGY BAG

- ▷ Le Langage C est un langage compilé du paradgigme impératif
- ▷ Syntaxe précise à respecter compréhensible
- ▷ Compilé en assembleur (encore compréhensible)
- ▷ Assemblé en binaire
- ▷ Liés pour former un exécutable
- ▷ Des normes à suivre pour permettre l'inter-opérabilité
→ nécessité de compiler sur chaque type de machines et de systèmes d'exploitation

EXERCICE À LA MAISON

QUESTIONS?