CRO: C langage and programming Roots
Programmer proprement et pointeurs de fonction
Slides "advanced" pour le TP bigInt

tanguy.risset@insa-lyon.fr
Lab CITI, INSA de Lyon
Version du October 16, 2025

Tanguy Risset

October 16, 2025

Table of Contents

Plan

- Pointeurs de fonctions
- Les erreurs courantes en C (source `http://nicolasj.developpez.com/articles/erreurs/` par exemple)
- Règles pour coder proprement (source “Coder proprement” de Robert C. Martin)

Utilité des pointeurs de fonction

- Mécanismes dynamiques
  - plug-in
  - Modifier une fonctionnalité sans arrêter le programme
  - ajouter de nouvelles fonctionnalités
- Exemple: fonction de décodage de trame niveau 2: dépendant de l'interface connectée (ethernet, wifi, etc.)

Un premier exemple

Comprendre les déclarations

- Déclaration d'une variable: `int  *q[3]`
  - `[]` plus prioritaire que `*`, donc:
    `int  *q[3]` $\Leftrightarrow$ `int  (*(q[3]))`
  - l'expression `(*(q[3]))` est de type `int`
  - l'expression `q[3]` est de type *pointeur vers un* `int`
  - l'expression (i.e. la variable) `q` est de type *tableau de pointeur vers un* `int`
- Déclaration d'une fonction:
  `int  fonct1(int a)`
  - l'expression `fonct1(int a)` est de type `int`
  - l'expression (i.e. la variable) `fonct1` est de type fonction qui prend un `int` et renvoie un `int`
  - Les parenthèses après un symbole indique que le symbole est une fonction (de même que les crochets après un symbole indique que le symbole est un tableau).

Déclaration d'un pointeur de fonction

- Déclaration d'un pointeur de fonction:
  `int (*foncPtr)(int a)`
  - l'expression `(*foncPtr)(int a)` est de type `int`
  - l'expression `(*foncPtr)` est de type *fonction qui prend un int est renvoie un* `int`
  - l'expression (i.e. la variable) `foncPtr` est de type *pointeur vers une fonction qui prend un* `int` *et renvoie un* `int`
- lors de l'utilisation, (presque) tout se passe comme si la fonction était une Lvalue:
  - On peut affecter une adresse de fonction au pointeur de fonction:
    `foncPtr=&fonct1;`
  - Si on déréférence le pointeur de fonction, on obtient une fonction:
    l'exécution de `(*foncPtr)(10);` affiche:
    `Je suis la fonction fonct1(10)`

En fait, c'est un peu plus compliqué...

- En C, une fonction est presque toujours automatiquement castée en pointeur de fonction (et inversement):
  `foncPtr=&fonct1` $\Leftrightarrow$ `foncPtr=fonct1`
  `(*foncPtr)(10);` $\Leftrightarrow$ `(foncPtr)(10)`
- Tout comme pour les tableaux:
  `tab` $\Leftrightarrow$ `&tab`
- Pour les fonctions et les tableaux qui ne sont pas des L-values (on les appelle quelquefois des labels) le compilateur identifie `a` et `&a` (sauf dans certains cas comme le passage par référence).
- On peut donc écrire:

```
int (*foncPtr)(int a);

foncPtr=fonct1;
foncPtr(10);
```

On peut donc ecrire:

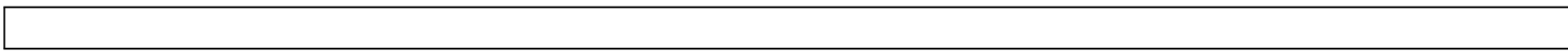
$\square$

Un autre exemple

```
//comparaison de deux entiers

int croissant(int i, int j)
{
    if (i<=j) return 0;
    else return 1;
}
int decroissant(int i, int j)
{
    if (i<=j) return 1;
    else return 0;
}
```

```
int main(void)
{
  int i,t[6]={1,5,2,3,6,4};

    trie(t, 6, croissant);
    for(i=0;i<6;i++)
      fprintf(stdout," %d ",t[i]);
    fprintf(stdout,"\n");

    trie(t, 6, decroissant);
    for(i=0;i<6;i++)
      fprintf(stdout," %d ",t[i]);
    fprintf(stdout,"\n");
    return 0;
}
```

... la fonction tri



Passage de fonction par référence

- Comme pour une variable normale, on peut modifier un pointeur de fonction en le passant en paramêtre *par référence*.
- Pour avoir une fonction qui modifie un pointeur de fonction (i.e. qui modifie la qui fonction appelée lorsque le pointeur est invoqué), il faut que son paramêtre soit un pointeur sur un pointeur de fonction.

Passage de fonction par référence

```
changeOrdre(int (**fcomp1)(int, int), int (*fcomp2)(int, int))
{
  *fcomp1=fcomp2;
}

int main(void)
{
  int i,t[6]={1,5,2,3,6,4};
  int (*fcomp)(int,int);

  fcomp=croissant;
  trie(t, 6, fcomp);
  [...] /* afficher le tableau t --> ordre croissant */

  changeOrdre(&fcomp,decroissant);
  trie(t, 6, fcomp);
  [...] /* afficher le tableau t --> ordre decroissant */
  return 0;
}
```

Table of Contents

Les erreurs courantes en C

Confusion entre == et =

- À ne pas faire:

  `if (size = 0) ....`

- Détection: l'option `-Wall` du compilateur avertit le programmeur (`warning`)

Confusion entre opérateurs logiques et binaires

- Le ET logique (`&&`) qui retourne 0 ou 1 (en s'arrêtant au premier argument s'il est faux)
- Le ET binaire (`&`) évalue ses deux opérandes et effectue le ET bit à bit entre ses deux opérandes.
- Le OU logique (`||`) qui retourne 0 ou 1 (en s'arrêtant au premier argument s'il est vrai)
- Le OU binaire (`|`) évalue ses deux opérandes et effectue le OU bit à bit entre ses deux opérandes.
- Impossible à détecter à la compilation.

Problèmes de macros

- On n'écrit pas
  `#define MAX 10;`
  - `A[MAX]` devient `A[10;]`
- On n'écrit pas
  `#define MAX=10`
  - Erreur détectée à la compilation, mais lors de l'utilisation de `MAX` (la ligne référencée n'est pas celle de la définition de `MAX`).
- On écrit
  `#define MAX 10`
- En cas de doute, on peut utiliser gcc -E pour vérifier l'expansion correcte des macros.

Fonctions retournant un caractère getc...

```
char c;
while ( (c = getchar ()) != EOF)
...
```

- La fonction `getchar` retourne un **entier**
- Les cast implicites effectués sont:
  `while ( (int)(c = (char)getchar ()) != EOF)`
- le caractère EOF (qui marque la fin d'un fichier : End Of File) est un caractère invalide généralement égal à -1 mais il peut parfaitement être égal à 128, dans ce cas on dépasse la capacité de stockage d'un char et l'on se retrouve avec un résultat égal à -128 : la condition du while sera toujours fausse, le programme boucle indéfiniment!
- Il faut écrire:

```
int cInt;
char c;
while ( (cInt = getchar ()) != EOF)
   {
     c=(char)cInt;
```

Erreurs avec `if` et `for`

- point-virgule mal placé

```
if (a < b) ;
    a = b;
```

```
for (i=0; i<N; i++);
    printf("%d", i);
```

- Le mauvais `else`

```
if (a < b)
    if (b < c) then b = c;
else
    b = a;
```

- Ce qu'il fallait faire:

```
if (a < b)
    { if (b < c) then b = c; }
else
    b = a;
```

Et les plus courantes...

- Mauvaise indentation
- Pas de makefile (ou pas de cible clean dans le makefile)
- exemple de fichier configuration vi:

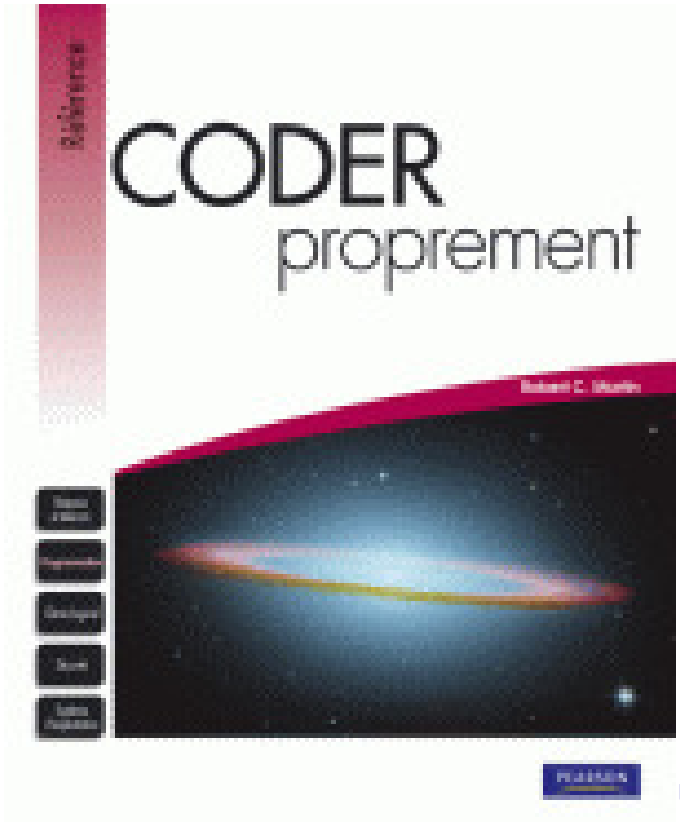
```
syntax on           " coloration syntaxique
set autoindent      " identation
set cindent    "     pour C
set nopaste
set ts=4    " tabulation a 4 caracteres
set sw=4
```

- Sous vi:
  - == pour indenter la ligne courante,
  - =G pour identer tout le fichier:

Table of Contents

Règles de programmation

Coder proprement, extraits de :



Avertissement

- Les méthodes de programmation évoluent très vite
- Cette partie du cours est tiré du livre "Coder Proprement (Robert C. Martin) et s'inspire des concept de l'*eXtreme programming* (programmation Agile)
- La programmation Agile s'appui sur le cycle suivant:

  While (1)

  Ecrire un test pour une fonction

  Ecrire une fonction qui valide le test

  Améliorer la fonction (refactoring)

- La durée du cycle peut se compter en minutes, au pire en heures
- Aujourd'hui les tests sont presque plus importants que les programmes: (TDD, *Test Driven Development*).
- Le programmeur est principalement concerné par les tests unitaires

Règles abordées

- Nom de variables
- Quelles caractéristiques pour les fonctions
- Des commentaires *utiles*
- Présentation générale et tests

Noms significatifs

- Choisir des noms de variables révélateurs des intentions

Ne faites pas:

```
int d; //Temps ecoulé en jours
```

Faites:

```
int tempsEcouleEnJours;
int joursDepuisCreation;
int joursDepuisModification;
```

ou mieux:

```
int timeElapsedInDays;
int daysSinceCreation;
int daysSinceModification;
```

Noms de variables

- Utiliser une règle systématique pour majuscule, minuscule, underscore...
- Les noms de variables *informent*, éviter la désinformation:
  - `hp` *n'est pas* une abréviation *commune* de hypoténuse.
- Éviter les lettre `l` et `O`:

```
int a = l;
if (O == l) a = O1 else l = O1;
```

- Évitez les séries: `a1, a2,.....` En général les variables ont un sens plus précis (source, destination, etc.)
- Choisir des noms prononçables
- Les variables d'une lettre sont limitées à des portées très courtes
- N'hésitez pas à avoir des noms longs: ils servent de commentaire.

Fonctions

- Deux règles importantes:
  - Faire le plus court possible
  - Faire encore plus court que ça
- 20 lignes est une bonne taille, on peut faire des programmes avec des fonctions de 4-5 lignes.
- Cela implique
  - Pas plus de deux niveaux d'imbrication de bloc (deux `if`; un `for`, un `if`, etc.), on appelle une fonction si il y en a plus.
  - Faire une seule chose dans une fonction
- Prenez le temps de choisir un bon nom de fonction
- Minimisez le nombre d'arguments (1 ou 2)
- Objectif: le principe de Ward: *Vous savez quand vous travaillez avec du code propre lorsque chaque fonction que vous lisez correspond parfaitement à ce que vous attendiez*

Commentaires

- Un commentaire bien placé est extrêmement utile
- Un commentaire redondant, mal placé, obsolète est extrêmement préjudiciable.
- Le commentaire est un *mal nécessaire*, idéalement il ne devrait pas y en avoir:
  - Chaque fois que le code transmet vos intentions, c'est un succès
  - Chaque fois que vous écrivez un commentaire, c'est un échec
- Ne soyez pas redondant:

```
[...]
//Initialisation de la pile des valeurs
InitPile(pileValeur)
[...]
```

Ne faites pas:

```
// verifier si l'employe peut
// beneficier de tous les avantages
if ((employee.flags &&
     benefitAvailable(employee)) &&
     employee.age > 65)
    ...
```

Faites:

```
 if (isEligibleForFullBenefits(employee))
    ...
```

Commentaire

- Ne mettez jamais de code en commentaire (c'est que vous n'avez pas le courage de le supprimer)
- N'écrivez pas un roman.
- Le lien entre le code et le commentaire doit être évident. Exemple du code source Apache:

```
/*
*   Débuter avec un tableau suffisament grand de
*   pixels (plus octest de filtrage) et 200
*   octets pour l'en-tête
*/
pngBytes = malloc(((Current.width+1)*Current.height*3)+200);
```

Mise en forme

- Plus votre code est élégant, plus votre travail paraît professionnel.
- Les fichiers courts sont plus facile a comprendre (200 lignes)
- Le code se lit comme un journal:
  - Chaque ligne représente une expression ou une clause
  - Chaque groupe de ligne représente une idée, ces idées sont séparées par des lignes vides
- Il faut des lignes courtes (80/100 caractères)
- INDENTATION!
  - Certain langages (Python) construisent la structure des blocs uniquement par l'information d'indentation.

Test

- Il existe maintenant des langages de test, des environnements permettant de faire des test de non-regression, des test automatiques (nightly build)
- Les tests propres suivent les cinq règles suivantes (F.I.R.S.T):
  - Fast: les tests doivent être rapides pour être exécutés très souvent
  - Independent: ils ne doivent pas dépendre les uns des autres.
  - Repeatable: ils doivent pouvoir être reproduits dans tous les environnement (connexion réseaux, etc.), pas d'excuser pour leur échec.
  - Self-Validating: il réussissent ou ils échouent, pas de validation complexe à la main d'un fichier de résultat.
  - Timely: ils doivent être écrits au moment opportun: juste avant la production de code.

Conclusion

- Certain en ont déjà fait l'expérience:
  - *on peut perdre un temps incommensurable à traquer un bug stupide*
- Un des ingrédient les plus important de la recette miracle est de coder proprement:
  - *Ne négligez pas l'investissement nécessaire à un code propre*
  - Ne dites pas "Je code çà en 5 minutes". Pensez au cycle de vie du logiciel.
- Le développeur doit programmer des programmes simples
- Il doit pouvoir évaluer le temps de développement
- Il doit fournir un ensemble de test le plus complet possible