Cours de C - IR1
2007-2008

Les limites du C
ou
l'art d'apprendre à changer de paradigme quand c'est nécessaire

Sébastien Paumier

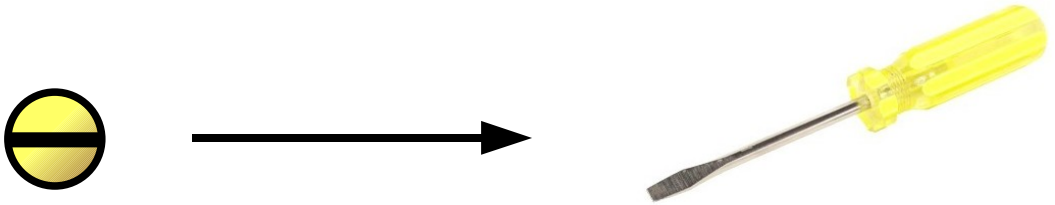
MCours.com

Paradigme

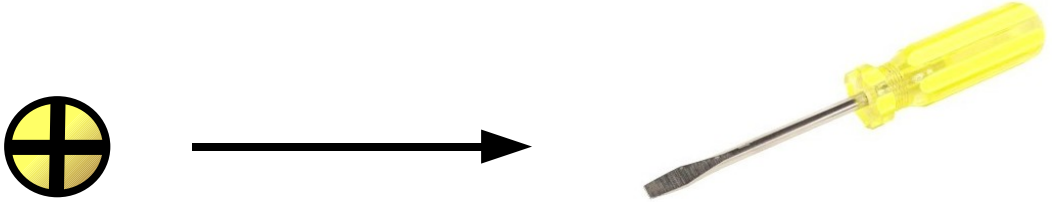
Un paradigme est une représentation du monde, une manière de voir les choses, un modèle cohérent de vision du monde qui repose sur une base définie (matrice disciplinaire, modèle théorique ou courant de pensée).

Wikipédia

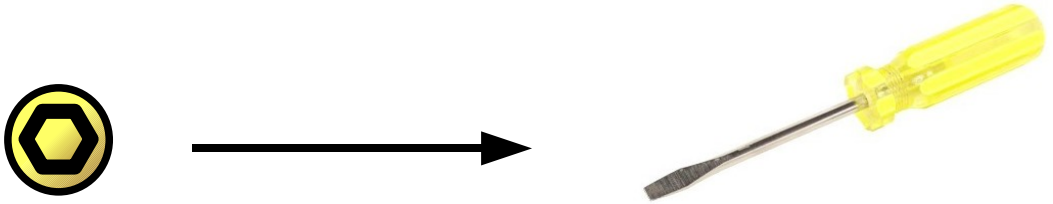
Prendre un outil adapté



OK



bof



vis et tournevis abîmés !

Les questions de chapelle...

- ne pas être aveuglé par ce qu'on préfère:
  - "*Faites du Perl* " (Sylvain L.)
  - "*Faites du Python*" (Jérôme P.)
  - "*fète du Java*" (Rémi F.)
  - "*Use .NET, or we'll kill you*" (Bill G.)
- quand les caractéristiques du langage ne conviennent plus, il faut savoir en changer

C=bas niveau

- accès possible à la machine physique, mais risque d'erreurs
- exemple: débordement des tableaux

```
float get1(float t[],int n,int size) {
   if (t==NULL) {
      fprintf(stderr,"NULL array in get\n");
      exit(1);
   }
   if (n<0 || n>=size) {
      fprintf(stderr,"Index %d out of bounds in get\n",n);
      exit(1);
   }
   return t[n];
}
```

C=bas niveau

- oui mais, si on ne veut pas quitter sauvagement ?

```
int get2(float t[],int n,int size,float *res) {
   if (t==NULL || n<0 || n>=size) {
      return 0;
   }
   (*res)=t[n];
   return 1;
}
```

C=bas niveau

- oui mais, je dois faire une fonction pour chaque type ?

```
int get3(void* t,int n,int size,int size_n,void *res) {
   if (t==NULL || n<0 || n>=size) {
      return 0;
   }
   char* foo=(char*)t;
   memcpy(res,foo+n*size_n,size_n);
   return 1;
}
```

C=bas niveau

- oui mais, si je veux pouvoir faire `t[i]=t[i]` sans risque d'erreur ?

Extrait de `man memcpy`:

The **memcpy**() function copies *len* bytes from memory area *src* to memory area *dst*. **If *src* and *dst* overlap, behavior is undefined**. Applications in which *src* and *dst* might overlap should use **memmove(3)** instead.

```
int get4(void* t,int n,int size,int size_n,void *res) {
    if (t==NULL || n<0 || n>=size) {
        return 0;
    }
    char* foo=(char*)t;
    memmove(res,foo+n*size_n,size_n);
    return 1;
}
```

C=bas niveau

- oui mais, ça pourrait déborder dans la zone **res**

```
int get5(void* t,int n,int size,int size_n,void *res,int size_res) {
   if (t==NULL || n<0 || n>=size || size_res<size_n) {
      return 0;
   }
   char* foo=(char*)t;
   memmove(res,foo+n*size_n,size_n);
   return 1;
}
```

MCours.com

C=bas niveau

- oui mais, si je me trompe de paramètres dans l'appel de `get5`, ça va planter quand même
- *oui, mais c'était déjà le cas avec* `get1`
- j'ai fait tout ça pour rien alors ?
- *ben oui*
- alors comment on fait en C ?
- *on ne fait pas*

C=gestion de la mémoire

- pas de garbage collector interne au langage
- on peut utiliser celui de Boehm, Demers & Weiser
- mais: risque si l'on se mélange les pinceaux avec les `malloc`/`free` normaux, car ce n'est qu'une couche supplémentaire

C=langage compilé

- un programme C existe en 2 versions:
  - les sources
  - le binaire exécutable
- si on n'a que le binaire, le programme mourra
- problème inexistant pour les langages interprétés (Perl, Python, ...)

C=typage statique

- déclaration des variables avec leurs types
- besoin de conversions explicites:

```
6   ⇒ "6" sprintf(tmp,"%d",n);
"6" ⇒  6  sscanf(tmp,"%d",&n);
```

- typage dynamique: variables de bash
  - la conversion est faite automatiquement quand c'est nécessaire

C=typage statique

- idée similaire: autoboxing/unboxing en Java:
- si on a `Integer foo(Integer i);` on peut écrire:

```
int i=foo(45);
```

- attention: toujours du typage statique, car effectué à la compilation, pas à l'exécution

C=typage faible

- on doit toujours préciser les types utilisés
- typage fort: le type est déterminé automatiquement en fonction de l'utilisation
- exemple: inférence de types en Caml

C=typage faible

- somme d'une liste d'entiers:

```
# let rec sum l = match l with
                  [] -> 0
                  | x::y -> x+sum(y);;
val sum : int list -> int = <fun>
```

- on n'a pas précisé qu'il s'agissait d'entiers
- Caml l'a déduit du +

C=typage faible

- somme d'une liste de réels:

```
# let rec sum l = match l with
                  [] -> 0.
                  | x::y -> x+.sum(y);;
val sum : float list -> float = <fun>
```

- taille d'une liste:

```
# let rec size l = match l with
                  [] -> 0
                  | x::y -> 1+size(y);;
val size : 'a list -> int = <fun>
```

n'importe quel type convient dans ce cas

C=opérateurs figés

- pas de surcharge, impossible d'écrire **a+b** pour des nombres complexes
- surcharge d'opérateur possible en C++:

```
class Complex {
   public:
   float re,im;

   Complex() {re=0; im=0;}
   Complex(float a,float b) {re=a; im=b;}

   friend Complex operator+(const Complex &a,const Complex &b) {
      Complex c;
      c.re=a.re+b.re;
      c.im=a.im+b.im;
      return c;
   }
};
```

C=opérateurs figés

- utilisation:

```
int main(int argc,char* argv[]) {
   Complex a=Complex(4,5);
   Complex b=Complex(0,7);
   Complex c=a+b;
   printf("c=%g+%g*i\n",c.re,c.im);
   return 0;
}
```

```
$>g++ surcharge.cpp
$>./a.out
c=4+12*i
```

C=pas d'exceptions

- gestion des erreurs manuelles
- propager une erreur est lourd:

```
int fprintf_utf8(FILE* f,unichar* s) {
   while (*s) {
      if (!fputc_utf8(*s,f)) return 0;
      s++;
   }
   return 1;
}

int save_HTML(FILE* f,unichar* s) {
   if (!fprintf_utf8(f,begin_tag)) return 0;
   if (!fprintf_utf8(f,s)) return 0;
   if (!fprintf_utf8(f,end_tag)) return 0;
   return 1;
}
```

C=pas d'exceptions

- solution: les exceptions (C++, Java, ...)

```
void fprintf_utf8(FILE* f,unichar* s) {
   while (*s) {
      if (!fputc_utf8(*s,f)) throw IOException;
      s++;
   }  /* Le throw pourrait même être dans */
}     /* la fonction fputc_utf8            */

int save_HTML(FILE* f,unichar* s) {
   try {
      fprintf_utf8(f,begin_tag);
      fprintf_utf8(f,s);
      fprintf_utf8(f,end_tag);
      return 1;
   } catch (MyException exception) {
      return 0;
   }
}
```

C=noms globaux

- pas d'espaces de nommage paramétrables

nommage.c:

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

int main(int argc,char* argv[]) {
    /* ... */
    return 0;
}
```

algo.h:

```c
#ifndef algo_H
#define algo_H

typedef struct noeud {
    int val;
    struct noeud *g,*d;
} ARBRE;

typedef struct file {
    struct file* premier;
    struct file* dernier;
} FILE;

void largeur(ARBRE* a);
#endif
```

```
In file included from nommage.c:2:
algo.h:12: conflicting types for `FILE'
c:/dev-cpp/include/stdio.h:159: previous
declaration of `FILE'
```

C=noms globaux

- autre exemple: `main`
- si une bibliothèque `foo` contient une fonction `main` de test oubliée...
- solution: avoir un contrôle de portée (`foo.toto()` par exemple)
  - en objet: encapsuler les choses
  - avoir des espaces de noms (packages Java)

C=pas d'objets

- je veux manipuler des automates avec une interface très propre, mais je veux cacher le type, pour empêcher l'accès aux champs
- comment faire en C ?
- *on ne peut pas*

- solution: POO avec des contraintes de visibilité

C=pas d'objets

- **`carre`**, **`rond`**, **`tache`** et **`trapeze`** sont des **`forme`** (aire, périmètre)
- **`carre`** et **`trapeze`** ont des propriétés de polygônes (nombre de sommets/arêtes)
- **`carre`** et **`rond`** ont des propriétés de formes régulières (symétrie)
- comment représenter tout ça en C ?
- *c'est dur, car il n'y a pas d'héritage*

C=pas de polymorphisme

- le calcul de l'aire dépend du type réel de forme que l'on a
- sans polymorphisme, on doit écrire 4 fonctions et savoir laquelle appeler pour une forme donnée:

  `aire_carre(carre c)`, etc
- avec, 4 fonctions de même nom et un seul appel générique:

  `aire(forme f)`

C=précision limitée

- taille des nombres limitée
- problèmes d'arrondi:

```c
#include <stdio.h>

int main(int argc,char* argv[]) {
    int i;
    float sum=0.f;
    for (i=0;i<999999997;i++) {
        sum=sum+0.001f;
    }
    printf("sum=%f\n",sum);
    return 0;
}
```

```
$>time ./a.out
sum=32768.000000
8.04user 0.00system 0:08.06elapsed 99%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (94major+14minor)pagefaults 0swaps
```

C=précision limitée

- solution: calcul formel
- juste, mais (très) lent!

```
mantisse:=proc(pas,n)
    local somme;
    local i;
begin
    somme:=0;
    for i from 1 to n do
        somme:=somme+pas;
    end_for;
    return(somme);
end_proc;
mantisse(0.001,999999997);
quit;
```

```
$>mupad mantisse.mu

   *----*     MuPAD 2.5.3 -- The Open Computer Algebra System
  /|   /|
 *----* |     Copyright (c)  1997 - 2003  by SciFace Software
 | *--|-*                   All rights reserved.
 |/   |/
 *----*       Licensed to:   MuPAD Combinat Developer

                  proc mantisse(pas, n) ... end

                            999999.997
```

C=langage non fonctionnel

- on veut représenter des formules logiques sous forme d'arbres
- on veut calculer la liste des atomes distincts présents dans les formules:

  A v (B→A) doit donner la liste "A", "B"

- comment faire en C ?

C=langage non fonctionnel

type d'une formule:

```
#define MAX 32

typedef enum {ATOME,VALEUR,NON,IMP,ET,OU} Type;

struct formule {
    Type type;
    union {
        char atome[MAX];
        char bool;
        struct formule* A;
        struct {
            struct formule* A;
            struct formule* B;
        };
    };
};
```

C=langage non fonctionnel

construction de la liste d'atomes:

```
struct liste* liste_atomes(struct formule* f) {
switch(f->type) {
   case ATOME: return new_liste(f->atome);
   case VALEUR: return NULL;
   case NON: return liste_atomes(f->A);
   default: return fusion(liste_atomes(f->A),liste_atomes(f->B));
}
}
```

C=langage non fonctionnel

fusion de listes d'atomes:

```
int appartient(struct liste* x,struct liste* l) {
while (l!=NULL) {
   if (!strcmp(x->s,l->s)) return 1;
   l=l->suivant;
}
return 0;
}

struct liste* fusion(struct liste* A,struct liste* B) {
while (A!=NULL) {
   struct liste* tmp=A->suivant;
   if (appartient(A,B)) {
      free_liste(A);
      A=tmp;
   } else {
      A->suivant=B;
      B=A; A=tmp;
   }
}
return B;
}
```

C=langage non fonctionnel

fonctions de base pour manipuler des listes de chaînes:

```
struct liste {
   char s[MAX];
   struct liste* suivant;
};

struct liste* new_liste(char s[]) {
struct liste* l=(struct liste*)malloc(sizeof(struct liste));
if (l==NULL) {
   fprintf(stderr,"Erreur memoire\n");
   exit(1);
}
strcpy(l->s,s);
return l;
}

void free_liste(struct liste* l) {
if (l!=NULL) free(l);
}
```

C=langage non fonctionnel

code de test pour la formule A ∨ (B→A):

```
int main(int argc,char* argv[]) {
   struct formule A,B,C,D,E;
   A.type=ATOME; strcpy(A.atome,"A");
   B.type=ATOME; strcpy(B.atome,"B");
   C.type=ATOME; strcpy(C.atome,"A");
   D.type=IMP;   D.A=&B; D.B=&C;
   E.type=OU;    E.A=&A; E.B=&D;
   struct liste* l=liste_atomes(&E);
   while (l!=NULL) {
      printf("%s ",l->s);
      l=l->suivant;
   }
   printf("\n");
   return 0;
}
```

→

```
$>./a.out
B A
```

au total: 91 lignes de code avec de l'allocation dynamique et des formules pas conviviales à tester

C=langage non fonctionnel

- si l'on prend un langage adapté (Caml):

définition du type formule:

```
$> ocaml
        Objective Caml version 3.09.1

# type formule =
    Atome of string
  | Valeur of bool
  | Non of formule
  | Imp of formule * formule
  | Et of formule * formule
  | Ou of formule * formule;;
type formule =
    Atome of string
  | Valeur of bool
  | Non of formule
  | Imp of formule * formule
  | Et of formule * formule
  | Ou of formule * formule
```

C=langage non fonctionnel

- calcul+test (et c'est fini):

```
# let rec liste_atomes f =
   let rec appartient l a = match l with
           [] -> false
           | x::y -> if a=x then true else (appartient y a)
   in
   let rec fusion l m = match l with
      [] -> m
      | x::y -> if (appartient m x) then (fusion y m) else x::(fusion y m)
   in
       match f with
     Atome(x) -> [x]
   | Valeur(_) -> []
   | Non(x) -> liste_atomes(x)
   | Imp(x,y)
   | Et(x,y)
   | Ou(x,y) -> (fusion (liste_atomes x) (liste_atomes y));;
val liste_atomes : formule -> string list = <fun>
# liste_atomes (Ou(Atome("A"),Imp(Atome("B"),Atome("A"))));;
- : string list = ["B"; "A"]
```

C=peu d'API

- exemple: pas de couche graphique standard comme swing ou awt en Java
- il faut utiliser des bibliothèques en plus
- si on veut de la portabilité, il faut en utiliser une bonne:
  - openGL, Allegro (http://alleg.sourceforge.net)
- même comme ça, il faut prendre des précautions:

http://alleg.sourceforge.net/docs/portability_guidelines.en.html

Épilogue

Apprendre à revenir sur une décision

Théorie de l'engagement

- attachement naturel à:
  - ce que nous avons décidé (ne pas vouloir avoir tort)
  - ce qui nous a coûté (protéger ses investissements)
- il est très difficile de revenir en arrière
- mais: l'obstination peut être catastrophique

Exemple

- problème initial: lire un fichier HTML en ne prenant que le texte sans les balises

```
int get_char(FILE* f) {
int c=fgetc(f);
if (c==EOF) return EOF;
if (c=='<') {
   while ((c=fgetc(f))!=EOF && c!='>');
   return fgetc(f);
}
return c;
}
```

Exemple

- bug 1: si on a < après > (deux balises qui se suivent)

```
int get_char2(FILE* f) {
int c=fgetc(f);
if (c==EOF) return EOF;
while (c=='<') {
   while ((c=fgetc(f))!=EOF && c!='>');
   c=fgetc(f);
}
return c;
}
```

Exemple

- bug 2: si une balise contient > entre guillemets

```
int get_char3(FILE* f) {
int c=fgetc(f);
if (c==EOF) return EOF;
while (c=='<') {
   while ((c=fgetc(f))!=EOF && c!='>') {
      if (c=='"') {
         do {
            c=fgetc(f);
         } while (c!=EOF && c!='"');
      }
   }
   c=fgetc(f);
}
return c;
}
```

Exemple

- bug 3: si le texte contient `<`, `>` ou `&`

```
int get_char4(FILE* f) {
/* ...tout ce qu'on faisait déjà dans get_char3... */
if (c=='&') {
   c=fgetc(f);
   switch (c) {
      case 'l': /* &lt; pour < */
         if ((c=fgetc(f))==EOF || c!='t') return EOF;
         if ((c=fgetc(f))==EOF || c!=';') return EOF;
         return '<';
      case 'g': /* &gt; pour < */
         if ((c=fgetc(f))==EOF || c!='t') return EOF;
         if ((c=fgetc(f))==EOF || c!=';') return EOF;
         return '>';
      case 'a': /* &amp; pour & */
         if ((c=fgetc(f))==EOF || c!='m') return EOF;
         if ((c=fgetc(f))==EOF || c!='p') return EOF;
         if ((c=fgetc(f))==EOF || c!=';') return EOF;
         return '&';
      default: return EOF;
   }
}
return c;
}
```

Exemple

- bug 4: et tout les autres codes (** **) ?

```
int get_char5(FILE* f) {
/* ...tout ce qu'on faisait déjà dans get_char3... */
if (c=='&') {
   char tmp[128];
   int i=0;
   while ((c=fgetc(f))!=EOF && c!=';') {
      tmp[i++]=c;
   }
   if (c==EOF) return EOF;
   tmp[i]='\0';
   if (!strcmp(tmp,"lt")) return '<';
   if (!strcmp(tmp,"gt")) return '>';
   if (!strcmp(tmp,"amp")) return '&';
   if (!strcmp(tmp,"nbsp")) return 0xA0;
   /* ...environ 250 lignes... */
   return EOF;
}
return c;
}
```

Exemple

- bug 5: et si l'encodage change ?
- bug 6: et si la page contient du javascript ?
- bug 7: et si on veut distinguer le vrai `EOF` des cas d'erreurs ?
- bug 8: et si on veut gérer les différents types d'erreurs
- ...
- bug 245: et si on veut garder certaines balises ?

Exemple

- si l'on veut pas se retrouver avec une usine à gaz ingérable, il faut accepter d'arrêter de patcher un truc inadapté pour passer à un modèle viable à plusieurs niveaux:
  - décodage des caractères (UTF8, UTF16, ...)
  - analyse de la structure du document HTML
  - filtrage des choses que l'on souhaite garder
  - décodage des symboles spéciaux (`&`)
  - etc.

Exemple

- chaque niveau pourra alors gérer ses propres erreurs en levant des exceptions:
  - `ÃÃ` `MalformedUTF8Character`
  - `</html` `UnclosedTag`
  - `&anp;` `InvalidHTMLTag`
  - etc.
- conclusion: la première version n'était pas du tout adaptée au problème; mieux vaut tout recommencer

Fin

MCours.com