Programmation avancée en C :

L'esprit, les limites et le paradigme du C

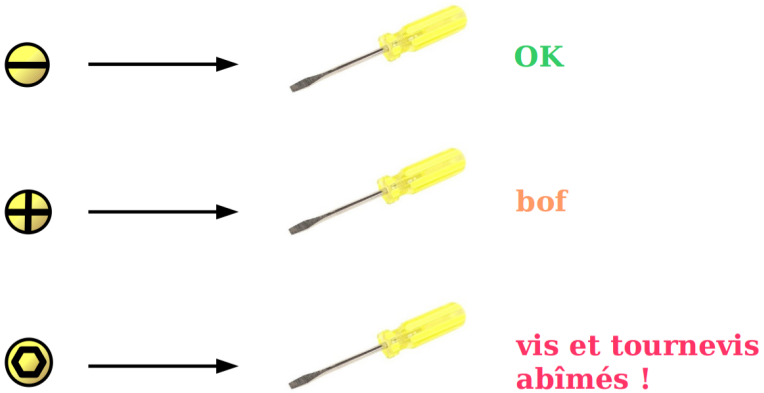
Licence informatique 3e année

Université Gustave Eiffel

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*)
- Un paradigme de programmation est un style fondamental de programmation informatique qui traite de la manière dont les solutions aux problèmes doivent être formulées dans un langage de programmation. (*Wikipédia (informatique)*)

Prendre un outil adapté



Problèmes métaphysiques...

- Ne pas être aveuglé par ce qu'on préfère :
  - "Faites du C" (W. Fang)
  - "Faites du Python" (N. Borie)
  - "Faites du Java" (R. Forax)
  - "Haskell sinon rien. Et oui, Haskell est Turing-complet..." (S. Giraudo)
  - "On devrait plutôt leur faire du C++ au L3." (D. Revuz)
- Quand les caractéristiques du langage ne conviennent plus au besoin, il faut savoir en changer.

L'esprit du langage C

- Faire confiance au programmeur (C'est donc un langage permissif et difficile en ce sens)
- Garder le langage réduit et simple (peu de mots clés et mécanismes)
- Tenter d'offrir au programmeur une seule manière de faire une opération
- Objectif d'efficacité même si le code n'est plus portable
- Peu de concepts difficiles (le passage de fonction est peut-être le concept le plus complexe)
- Aucun mécanisme pour empêcher le programmeur de faire ce dont il a besoin de faire (y compris les bêtises...)

Le langage est cohérent avec son esprit

- Seul les statiques et globales sont initialisées à zéro :
  Oui, ce n'est pas le cas de la pile ! Cela surchargerait le coût d'appel de fonction (nettoyage de la pile)
- Le compilateur ne dispose pas toujours les structures de la même manière :
  Oui, on souhaite aligner les données sur des mots machines pour faciliter leur chargement sur les registres
- On peut interpréter un `void*` avec un type dont la taille dépasse la place mémoire préparée derrière :
  Oui, imaginez encore le surcoût d'un mécanisme de vérification à chaque transtypage.

Les outils pour programmer

- Il sont peu nombreux dû à l'esprit du langage...
- Les flags de compilation (c'est un cadeau, pas une punition !)
- Le débogueur : difficile à utiliser
- Les syntax-checker : emacs indente automatiquement si et seulement si le code est syntaxiquement correct
- Un profileur

Les objectifs

- Le bas-niveau : disposer d'un langage qui permet d'exploiter complètement la machine
- Facile à apprendre (si, si, si !)
- L'efficacité des exécutables
- Vision directe de la mémoire
- Choix et contrôle de la représentation et modélisation bit à bit des données
- Adapté aux calculs intensifs et aux machines à ressources limitées

Un langage pédagogique

- La plupart des mécanismes courants de la programmation trouve leur illustration en C

  Ex : hachage, cache-remember, généricité, entrées/sorties, plugins, ...

- Liens profonds avec l'architecture
- Connaître le C permet de mieux appréhender les autres langages

  Ex : qu'est ce qu'un *garbage collector* pour une personne qui n'a jamais vu `malloc` et `free` ?

- La plupart des systèmes d'exploitations modernes sont écrits en C ; un grand nombre d'autres langages sont nés d'un compilateur/interpréteur écrit en C.

C = bas niveau

- Accès possible à la machine physique, mais risque d'erreur
- Exemple : débordement des tableaux

```
1  /* Safe access of an array element */
2  float get1(float t[], int n, int size){
3    if(t == NULL){
4      fprintf(stderr, "NULL array in get\n");
5      exit(1);
6    }
7    if(n < 0 || n >= size){
8      fprintf(stderr, "Index %d out of bounds in get\n");
9      exit(1);
10   }
11   return t[n];
12 }
```


C = bas niveau

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

```
1  /* Safe access of an array element
2     without hard exit */
3  int get2(float t[], int n, int size, float *res){
4    if(t == NULL || n < 0 || n >= size) return 1;
5    *res = t[n];
6    return 0;
7  }
```


C = bas niveau

- Oui, mais faut-il faire une fonction pour chaque type ?

```
1  /* Generic safe access of an array element
2     without hard exit */
3  int get3(void* t, int n, int size, int size_n, void* res){
4    if(t == NULL || n < 0 || n >= size) return 1;
5    char* foo = (char*)t;
6    memcpy(res, foo + n * size_n, size_n);
7    return 0;
8  }
```

C = bas niveau

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

Le comportement de `memcpy` est définit seulement si la zone source et la zone cible ne se chevauchent pas... Sinon, il faut utiliser memmove.

```
1  /* Generic safe access of an array element
2     without hard exit and supporting overwrite */
3  int get4(void* t, int n, int size, int size_n, void* res){
4    if (t == NULL || n < 0 || n >= size) return 1;
5    char* foo = (char*)t;
6    memmove(res, foo + n * size_n, size_n);
7    return 0;
8  }
```


C = bas niveau

- Oui, mais ça pourrait déborder dans la zone cible...

```
1  /* Generic very safe copy of an array element
2     without hard exit and supporting overwrite */
3  int get5(void* t,int n,int size,int size_n,
4           void* res,int res_size){
5    if (t == NULL || n < 0 || n >= size || res_size <= size_n){
6      return 1;
7    }
8    char* foo = (char*)t;
9    memmove(res, foo + n * size_n, size_n);
10   return 0;
11 }
```

⇒ Affreux, peu lisible, trop de paramètres, tant de tests pour une si petite chose (accès sécurisé dans un tableau...)

C = bas niveau

- Mais si on se trompe de paramètres dans un appel de `get5`, ça risque de produire une segfault ?
- Oui, mais c'était déjà le cas avec `get1`.
- On a fait tout ça pour rien alors ?
- Oui !
- Alors comment on fait en C ?
- On ne fait pas (et il faut savoir l'accepter...).

C = gestion manuelle de la mémoire

- Pas de garbage collector interne au langage
- Gérer manuellement permet de modéliser l'information de manière très fine (place minimale)
- Gérer manuellement demande une rigueur extrême pour que les choses fonctionnent
- Le contrôle est très fin : mémoire préparée et libérée à la demande

C = langage compilé

- Un programme C existe en deux versions :
  - les sources
  - le binaire exécutable
- Si on 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 : Python

  L'interpréteur définit le type à la volée...

  Une tache en moins à gérer pour le programmeur...

  Mais une grosse perte de performance...

C = typage statique

```
fwjmath@fwjmath-hover:~$ python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> def add_typefree(a, b):
...     c = a + b
...     print((c, type(c)))
...
>>> add_typefree(4, 2)
(6, <class 'int'>)
>>> add_typefree(35.6, 44.5)
(80.1, <class 'float'>)
>>> add_typefree([0, 1], [2, 3, 4])
([0, 1, 2, 3, 4], <class 'list'>)
>>> add_typefree('super', 'mutation')
('supermutation', <class 'str'>)
>>> exit()
```


C = typage faible

- On doit toujours préciser les types utilisés.
- Typage fort : le type déterminé automatiquement suivant l'utilisation
- En fait, ce sont les pointeurs qui affaiblissent le typage en C.
- Typage fort : OCaml, Haskell
- Typage statique faible ⇒ meilleure performance (la machine ne vérifie rien, le code est cru sur parole (jusqu'à la segfault ?))

C = typage faible

- OCaml contraint les types des variables en récupérant les contraintes contextuelles (ex : les opérateurs utilisant les variables).

```
fwjmath@fwjmath-hover:~$ ocaml
        OCaml version 4.08.1

# let rec sum = function
    | [] -> 0
    | hd :: tl -> hd + (sum tl) ;;
val sum : int list -> int = <fun>
# sum [1; 2; 3; 4; 5] ;;
- : int = 15
# exit ;;
- : int -> 'a = <fun>
# exit 0 ;;
fwjmath@fwjmath-hover:~$
```


C = opérateurs figés

- Pas de surcharge, impossible d'écrire `a+b` pour des nombres complexes
- Frein énorme à généricité !
- Surcharge d'opérateur possible en C++

```
1  class Complex{
2    public:
3    float re, im;
4
5    Complex() {re=0; im=0;}
6    Complex(float a, float b) {re = a; im = b;}
7
8    friend Complex operator+(const Complex &a,const Complex &b){
9      Complex c;
10     c.re = a.re + b.re;
11     c.im = a.im + b.im;
12     return c;
13   }
14 };
```


C = opérateurs figés

- En python, tout objet ayant une méthode `__add__` supporte l'addition.

```
#!/usr/bin/python3

class myObject():
    def __add__(self, other):
        print("L'addition n'est qu'une fonction...")

a = myObject()
b = myObject()

a + b
```

Script, qui une fois ayant les droits d'exécution, donne

```
fwjmath@fwjmath-hover:~$ ./test.py
L'addition n'est qu'une fonction...
fwjmath@fwjmath-hover:~$
```


C = pas d'exceptions

- Gestions manuelles des erreurs
- Propager une erreur est lourd et pénible.
- Solution : les exceptions (C++, Java, Python, ...)

```
fwjmath@fwjmath-hover:~/sagemath$ ./sage
┌────────────────────────────────────────────────────────────────────┐
| SageMath version 9.4, Release Date: 2021-08-22                     |
| Using Python 3.9.5. Type "help()" for help.                        |
└────────────────────────────────────────────────────────────────────┘
sage: S = Permutations(-1)
sage: S.random_element()
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-6da243bc27d3> in <module>
----> 1 S.random_element()

...

ValueError: Sample larger than population or is negative
sage:
```

C = noms globaux

- Pas d'espace de nommage paramétrable

```c
#include <stdio.h>

typedef struct file{
  struct file *avant;
  struct file *apres;
} FILE;

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

donne

```
nborie@perceval:~> gcc -o test test1.c -Wall -ansi
test1.c:6:2: erreur: conflicting types for 'FILE'
In file included from test1.c:1:0:
/usr/include/stdio.h:48:25: note: previous declaration
of 'FILE' was here
```


C = noms globaux

- Autre exemple : `main`
- Si une bibliothèque `foo` contient une fonction `main` de test oubliée...
- En C, toutes les fonctions non statiques sont globales (du point de vue du linker...) ; c'est la compilation séparée qui contraint l'écriture de headers.
  (`gcc *.c` : tout est global mais ça compile)
- Solution : avoir un contrôle de portée (`foo.toto()`)
- Un champ n'est visible que sur la structure correspondante...

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 cela en C ?
- On ne peut pas le faire simplement (et il faut savoir l'accepter).
- Solution : POO avec des contraintes de visibilités
- En C, on peut taper sur les champs d'un `FILE*`, et c'est très dangereux...

C = pas d'objets

- `carre`, `rond`, `tache` et `trapeze` sont des `forme` (aire, périmètre).
- `carre` et `trapeze` ont des propriétés de polygônes (nombres de sommets/arêtes).
- `carre` et `rond` ont des propriétés de formes régulières (symétries)
- Comment représenter tout cela 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 pour nos 4 formes et savoir laquelle il faut appeler :
  `aire_carre(carre c)`, etc.
- Avec 4 fonctions dédiées et une fonction de dispatch, on peut alors avoir :
  `aire(forme f)`

C = précision limitée

- Taille limitée des nombres
- Problèmes d'arrondis
- Rappelez-vous des premiers TP :

$$\sum_{i=1}^{N} \frac{1}{i} \text{ devient stationnaire pour N grand}$$

  Alors que l'on a

$$\lim_{N \to +\infty} \sum_{i=1} \frac{1}{i} = +\infty$$

- Solution : Il existe des bibliothèques pour les entiers à précision arbitraire (GMP, MPIR) et pour les flotants multi-précisions (MPFR).

C = précision limitée

- Solution : calcul formel
- Toujours juste, mais très lent (spécialité de vos combinatoristes préférés : Giraudo, Novelli, Thibon, Borie, ...)

```
nborie@perceval:~/ > sage
Sage Version 5.12, Release Date: 2013-10-07

sage: Partitions(4).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: Partitions(4).cardinality()
5
sage: Partitions(100000).cardinality()
27493510569775696512677516320986352688173429315980054758203125
98430214732811496417305505074166073662159015784477429624894049
30630702004617927644930335101160793424571901557189435097253124
66108452006369558934464248716828789832182345009262853831404597
02130713067451062441922731123899970228440860937093553162969785
15695698921961084801586005694210985 19
```


C = langage non fonctionnel

- On veut représenter des formules logiques sous forme d'arbre.
- On veut calculer la liste des atomes distincts présents dans les formules :
  A | (B → A) doit donner la liste "A", "B"
- Comment faire en C ?
- Au fait, on pourraient faire des plugins automatiques pour les opérations logiques...

C=langage non fonctionnel

Type d'une formule :

```
1  #define MAX 32
2
3  typedef enum {ATOME, VALEUR, NON, IMP, ET, OU} Type;
4
5  struct formule{
6    Type type; /* dispatch sur l'union qui suit */
7    union{
8      char atome[MAX];       /* symbole si atome */
9      char bool;             /* booleen si valeur */
10     struct formule* N;     /* operateur unaire */
11     struct {               /* operateur binaire */
12       struct formule* A;
13       struct formule* B;
14     };
15   };
16 };
```


C = langage non fonctionnel

Construction de la liste d'atomes :

```
1  struct liste* liste_atomes(struct formule* f){
2    switch(f->type){
3      case ATOME: return new_liste(f->atome);
4      case VALEUR: return NULL;
5      case NON: return liste_atomes(f->N);
6      default: return fusion(liste_atomes(f->A),
7                             liste_atomes(f->B));
8    }
9  }
```


C = langage non fonctionnel

Fusion de liste d'atomes :

```
1  int appartient(struct liste* x, struct liste* l){
2    while(l != NULL){
3      if(!strcmp(x->s, l->s)) return 1;
4      l = l->suivant;
5    }
6    return 0;
7  }
8
9  struct liste* fusion(struct liste* A, struct liste* B){
10   while(A != NULL){
11     struct liste* tmp = A->suivant;
12     if(appartient(A, B)){
13       free_liste(A); A = tmp;
14     }else{
15       A -> suivant = B; B = A; A = tmp;
16     }
17   }
18   return B;
19 }
```


C = langage non fonctionnel

fonctions de base pour les chaînes :

```
1  struct liste{
2    char s[MAX];
3    struct liste* suivant;
4  };
5
6  struct liste* new_liste(char s[]){
7    struct liste* l;
8    l = (struct liste*)malloc(sizeof(struct liste));
9    if(l == NULL){
10     fprintf(stderr, "Erreur memoire\n"); exit(1);
11   }
12   strcpy(l->s, s);
13   return l;
14 }
15
16 void free_liste(struct liste* l){
17   free(l);
18 }
```

C = langage non fonctionnel

Code de test pour la formule : A | (B → A)

```
1  int main(int argc, char* argv[]){
2    struct formule A,B,C,D,E;
3    A.type = ATOME; strcpy(A.atome, "A");
4    B.type = ATOME; strcpy(B.atome, "B");
5    C.type = ATOME; strcpy(C.atome, "A");
6    D.type = IMP; D.A = &B; D.B = &C;
7    E.type = OU; E.A = &A; E.B = &D;
8    struct liste* l = liste_atomes(&E);
9    while(l != NULL){
10     printf("%s ", l->s);
11     l = l->suivant;
12   }
13   printf("\n");
14   return 0;
15 }
```

Bilan : 94 lignes de codes, 11 aspirines pour voir :

```
nborie@perceval:~> gcc -o test test.c -Wall -ansi
nborie@perceval:~> ./test
B A
```


C = langage non fonctionnel

- Et si on prend un langage adapté : OCaml

```
nborie@perceval:~> ocaml
        OCaml version 4.01.0

# 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

- Cela s'écrit comme ça se prononce...

```
# let rec liste_atomes f =
    let rec appartient a = function
      | [] -> false
      | hd::tl -> if hd = a then true else appartient a tl
    in
    let rec fusion m = function
      | [] -> m
      | h::q -> if appartient h m then fusion m q else h::(fusion m q)
    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
- Il faut utiliser des bibliothèques en plus.
- Si on veut de la portabilité, il faut utiliser de bonnes bibliothèques...

  OpenGL, SDL, Allegro
- Et prendre de nombreuses précautions...

  http://wiki.allegro.cc/index.php?title=Hardware_Portability

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 toujours difficile de revenir en arrière.
- Mais : l'obstination peut être catastrophique
- La question "Est-ce qu'on patch ou est-ce que l'on refactorise le code ?" revient systématiquement dans tous les projets de longue durée.

Théorie de l'engagement

- Patcher un problème, c'est proposer un correctif (souvent sale) pour éviter un problème.
- Avec le temps, on se retrouve avec plus de patch que de code intelligent.
- Il est peut-être alors temps de refactoriser pour repartir sur des bases encore plus solides.
- Sur Linux, `X11` ne supporte pas les cartes graphiques multiples ; pour éviter de patcher encore le serveur X, il devrait être remplacé à terme par `wayland`.

Théorie de l'engagement

- Lorsqu'un langage n'est plus adapté au design imposé par les fonctionnalités voulues, il faut savoir en changer !
- L'ancienne version est alors considérée comme un prototype de la version à venir.
- L'ancienne version donne une expérience formidable sur :
  - ce qui marche et qu'il faut reproduire ;
  - ce qui est limité et qui nécessite une nouvelle modélisation.
- Repartir de zéro coûte très cher mais permet d'aller plus loin.

Au final

- On peut "a peu près" tout faire dans le langage C.
  Mais à quel prix ?
- On peut faire des choses génériques.
  Mais à quel prix ?
- On peut modéliser tout type d'information
  Mais ...
- Connaître paradigme et philosophie d'un langage permet de l'utiliser plus efficacement.
- Connaître l'algorithmique et la syntaxe du langage ne suffit pas pour être un bon programmeur.