Programmation avancée en C :

Bibliothèques, et comment écrire du bon code

Licence informatique 3e année

Université Gustave Eiffel

Section 1

Bibliothèque

Bibliothèque

- Bibliothèque = boîte noire capable de rendre des services
- 2 aspects :
  - le code (fichier binaire)
  - la liste des fonctions, types, variables (horrible !) et constantes utilisables (fichier `foo.h`)
- Fichier `foo.a` ou `foo.so`

Utiliser une bibliothèque

- Inclure les `.h` si nécessaire
- Indiquer au compilateur (linker) qu'il doit utiliser la bibliothèque `foo` (`foo.a` ou `foo.so`) :
  `gcc ......... -lfoo`
- Si le fichier n'est pas dans `/usr/lib`, il faut indiquer son chemin avec `-Lchemin`.

Bibliothèque statique

- Bibliothèque statique = fichier `.a`, contenant un ou plusieurs fichiers `.o`
- À la compilation, les portions de code nécessaires sont copiées dans l'exécutable.

  - **Avantage** : L'exécutable n'a plus besoin de la bibliothèque.

  - **Inconvénient** : Redondance de code entre les exécutables (les mises à jour sont sans effet)
- Peu utilisées, sauf pour la performance, intérêt surtout historique

Exemple : Faire des piles pour un `dc`

Un module pour manipuler une pile d'entier

Fichier **stack.h** :

```
1  /* This module allows the manipulation of a single integer stack */
2
3  /* Initialize the stack at the empty stack */
4  int stack_init(void);
5
6  /* Pop the top element of the stack and return its value */
7  int stack_pop(void);
8
9  /* Push the given element at the top of the stack */
10 void stack_push(int n);
11
12 /* Return the value of the top element of the stack */
13 int stack_top(void);
14
15 /* Display all element contained in the stack */
16 void stack_display(void);
```


Exemple : une bibliothèque de pile

- Créer le fichier objet `.o` (compilation seule sans lien)

  `$>gcc -c stack.c`
- Créer la bibliothèque `.a`

  `$>ar rs libstack.a stack.o`
- visualiser son contenu

```
nborie@perceval:~> nm --defined-only libstack.a

stack.o:
0000000000000000 b S
0000000000000106 T stack_display
0000000000000000 T stack_init
0000000000000010 T stack_pop
0000000000000063 T stack_push
00000000000000bf T stack_top
```


Test

```
1  #include "stack.h"
2  int main(int argc, char* argv[]){
3    stack_init();
4    stack_push(1); stack_push(2); stack_push(3);
5    stack_pop();
6    stack_push(4); stack_push(5);
7    stack_display();
8    return 0;
9  }
```

```
nborie@perceval:~> gcc test.c
/tmp/ccKPCclN.o: dans la fonction main:
test.c:(.text+0x10): référence indéfinie vers stack_init
test.c:(.text+0x1a): référence indéfinie vers stack_push
 ....
nborie@perceval:~> gcc test.c -L. -lstack
nborie@perceval:~> ./a.out
1 2 4 5
```

L'option `-L.` serait inutile si `libstack.a` était bien dans `/usr/lib`.

Bibliothèque partagée

- Fichier `.so` (shared object)
- À l'exécution, l'éditeur de liens dynamiques ira chercher le code dans le `.so`
- Économie : Si plusieurs programmes partagent le code, il n'est qu'une seule fois en mémoire.
- En cas de mise à jour de la bibliothèque, tous les exécutables en profitent automatiquement.

Exemple

- Créer le fichier `.o` avec l'option `fPIC` (Position Independant Code)

  ```
  $>gcc -fPIC -c stack.c
  ```

- Créer la bibliothèque avec l'option `-shared` :

  ```
  $>gcc -shared -o libstack.so stack.o
  ```

- Création de l'exécutable

  ```
  $>gcc -o test test.c -L. -lstack
  ```


Exemple

- Exécution qui ne marche pas :

```
nborie@perceval:~> ./test
./a.out: error while loading shared libraries: libstack.so:
cannot open shared object file: No such file or directory
```

- Explication avec `ldd` (affichage des dépendances) :

```
nborie@perceval:~> ldd test
linux-vdso.so.1 =>  (0x00007fffddce0000)
libstack.so => not found
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f413b887000)
/lib64/ld-linux-x86-64.so.2 (0x00007f413bc6f000)
```

Le linker dynamique ne peut trouver la bibliothèque `libstack.so` car celle-ci ne se trouve pas dans `/usr/lib`

Solution

- Si on n'a pas accès à `/usr/lib`, on doit compiler l'exécutable avec l'option `-Wl,-rpath,chemin_du_so` :

```
nborie@perceval:~> gcc -o test test.c -L. -lstack -Wl,-rpath,.
nborie@perceval:~> ./test
1 2 4 5
nborie@perceval:~> ldd test
linux-vdso.so.1 =>  (0x00007fff4effe000)
libstack.so => ./libstack.so (0x00007ffd74aa8000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffd746c2000)
/lib64/ld-linux-x86-64.so.2 (0x00007ffd74cac000)
```

Nommage

- **Linker name** : `libstack.so`
  - nom de fichier utilisé pour compiler un exécutable
- **Soname** : linker name + numéro de la version majeure : `libstack.so.1`
  - même numéro de version = update possible (les différentes versions d'une même majeure ne doivent pas casser la backward compatibility)
  - si on remplace `libstack.so.1` par `libstack.so.2`, certains programmes risquent de ne plus fonctionner.

Nommage

- **Real name** : soname + numéro de version mineure + numéro optionnel de release : `libstack.so.1.0.2` ou bien `libstack-1.0.2.so`
- C'est ici que se trouve vraiment le code !
- Les autres noms ne sont souvent que les liens symboliques.

```
nborie@perceval:~/ > ls -l /usr/lib/libMLV.so
lrwxrwxrwx 1 root 15 juil. 12  2012 /usr/lib/libMLV.so -> libMLV.so.0.0.0
nborie@perceval:~/ > ls -l /usr/lib/libMLV.so.0
lrwxrwxrwx 1 root 15 juil. 12  2012 /usr/lib/libMLV.so.0 -> libMLV.so.0.0.0
nborie@perceval:~/ > ls -l /usr/lib/libMLV.so.0.0.0
-rw-r--r-- 1 root 209296 juil. 12  2012 /usr/lib/libMLV.so.0.0.0
```


Bibliothèque dynamique

- DL = Dynamically Loaded Libraries
- Fichier `.so` chargé à l'exécution du programme
- Permet la mise en place de plugins
- Pratique pour écrire un JIT (Just In Time compiler) :
  - compiler le code
  - charger le fichier objet
  - exécuter le code

Bibliothèque dynamique

- Fonctions définies dans `<dlfcn.h>`
- Pour l'utiliser dans l'exécutable, compiler avec `-ldl`
- `void *dlopen(const char *filename, int flag);`
- `dlopen` charge la DL indiquée et retourne un pointeur la désignant (le `*handle`), ou `NULL` si erreur.
- `flag` :
  `RTLD_LAZY`=résolution quand nécessaire,
  `RTLD_NOW`=résolution de tous les noms de symboles utilisés dans la DL

Bibliothèque dynamique

- `int dlclose(void *handle);`
- Décharger la DL indiquée si elle n'est plus utilisée par la suite
- `void *dlsym(void *handle, const char *symbol);`
- Chercher le symbole (constantes, variables, fonctions, ...) indiqué et le retourne, ou retourne `NULL` si non trouvé
- `char *dlerror(void);`
- Retourner un pointeur sur une chaîne décrivant la dernière erreur qui s'est produite, ou `NULL` si la dernière erreur a déjà été gérée par un appel à `dlerror`

Exemple

- Localisation de `hello_world` :

hello_world_fr.c

```c
#include <stdio.h>

void hello_world() {
  printf("Bonjour monde!\n");
}
```

hello_world_en.c

```c
#include <stdio.h>

void hello_world() {
  printf("Hello world!\n");
}
```

- Préparation des DL :

```
nborie@perceval:~> gcc -fPIC -c hello_world_fr.c
nborie@perceval:~> gcc -fPIC -c hello_world_en.c
nborie@perceval:~> gcc -shared -o libhello_world_fr.so hello_world_fr.o
nborie@perceval:~> gcc -shared -o libhello_world_en.so hello_world_en.o
nborie@perceval:~> ls
hello_world_en.c  hello_world_en.o  hello_world_fr.c  hello_world_fr.o
libhello_world_en.so  libhello_world_fr.so
```


Exemple

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

int main(int argc, char* argv[]){
  void* dl;
  void (*hello)(void);
  dl = dlopen("./libhello_world_fr.so", RTLD_LAZY);
  hello = dlsym(dl,"hello_world");
  hello();
  dlclose(dl);
  dl = dlopen("./libhello_world_en.so", RTLD_LAZY);
  hello = dlsym(dl,"hello_world");
  hello();
  dlclose(dl);
  dl = dlopen("./libhello_world_de.so", RTLD_LAZY);
  fprintf(stderr, "%s\n", dlerror());
  return 0;
}
```

```
nborie@perceval:~> gcc test.c -ldl
nborie@perceval:~> ./a.out
Bonjour monde!
Hello world!
./libhello_world_de.so: cannot open shared object: No such file or directory
```


Bibliothèques et visibilité

- Même si un élément n'est pas déclaré dans un `.h`, il est accessible.
- Exemple : la fonction `hello_world` n'était pas déclarée.
- Pour rendre un élément non visible, il faut le déclarer avec `static`.
- Sert à interdire l'accès à l'implémentation

Quelques mots pour Windows

- Les bibliothèques statiques se comportent de la même manière que sous `Unix`.
- Les bibliothèques partagées ne s'appellent pas "shared object" mais DLL (pour Dynamic Link Library). Elles sont aussi compilées avec l'option `-shared` mais l'option `-fPIC` n'est plus utile.
- Les DLL n'ont pas de préfixe `lib` comme sous `Unix`, elles s'appellent ainsi `foo.dll`.
- À l'exécution, la DLL doit se trouver dans le même répertoire que le fichier à compiler ou bien dans un des chemins indiqués par la variable d'environnement `PATH`.

Quelques mots pour Windows

- L'inclusion est alors :

  `#include <windows.h>`

  (c'est à dire toute l'API windows...)
- L'ouverture de la DLL se fait avec `LoadLibrary`.
- Les symboles peuvent être recherchés avec `GetProcAddress`.
- Les fonctions récupérées doivent encore être bien castées à l'assignation.
- Mêmes règles d'accessibilité que pour les DLL précédemment

Section 2

Bon pratique pour bon code

C'est quoi du bon code ?

```
1  int nextcomb(int* ptrs, int cnt,
2               int startp, int endp){
3     if((cnt==0)||(cnt==endp-startp+1))
4        return -1;
5     if(((endp-startp+1)>>1)>=cnt){
6        int top=endp, i=0;
7        while((ptrs[i]==top)&&(i<cnt)){
8           i++;
9           top--;
10       }
11       if(i==cnt) return -1;
12       spm[ptrs[i]]--;
13       ptrs[i]++;
14       spm[ptrs[i]]++;
15       i--;
16       while(i>=0){
17          spm[ptrs[i]]--;
18          ptrs[i]=ptrs[i+1]+1;
19          spm[ptrs[i]]++;
20          i--;
21       }
```

```
1     }else{
2        int mycnt=endp-startp+1-cnt;
3        int top=endp, i=0;
4        while((ptrs[i]==top)&&(i<mycnt)){
5           i++;
6           top--;
7        }
8        if(i==mycnt) return -1;
9        spm[ptrs[i]]++;
10       ptrs[i]++;
11       spm[ptrs[i]]--;
12       i--;
13       while(i>=0){
14          spm[ptrs[i]]++;
15          ptrs[i]=ptrs[i+1]+1;
16          spm[ptrs[i]]--;
17          i--;
18       }
19    }
20    return 0;
21 }
```

Est-ce que c'est du bon code ?

Critère du bon code

La clé : **Lisibilité**

- Un code est écrit 1 fois, mais lu au moins 10 fois (souvent par nous-même)
- Bonne lisibilité facilite le débougage, qui prend le plus de temps.
- Lisibilité = code limpide + commentaires propres
- Code limpide = style + qualité + logique
- Commentaires propres = clarté + la bonne quantité
- Un effort significatif, mais qui vaut souvent le coût

Problème de ce code ?

```
int nextcomb(int* ptrs, int cnt,
             int startp, int endp){
   if((cnt==0)||(cnt==endp-startp+1))
      return -1;
   if(((endp-startp+1)>>1)>=cnt){
      int top=endp, i=0;
      while((ptrs[i]==top)&&(i<cnt)){
         i++;
         top--;
      }
      if(i==cnt) return -1;
      spm[ptrs[i]]--;
      ptrs[i]++;
      spm[ptrs[i]]++;
      i--;
      while(i>=0){
         spm[ptrs[i]]--;
         ptrs[i]=ptrs[i+1]+1;
         spm[ptrs[i]]++;
         i--;
      }
   }else{
      int mycnt=endp-startp+1-cnt;
      int top=endp, i=0;
      while((ptrs[i]==top)&&(i<mycnt)){
         i++;
         top--;
      }
      if(i==mycnt) return -1;
      spm[ptrs[i]]++;
      ptrs[i]++;
      spm[ptrs[i]]--;
      i--;
      while(i>=0){
         spm[ptrs[i]]++;
         ptrs[i]=ptrs[i+1]+1;
         spm[ptrs[i]]--;
         i--;
      }
   }
   return 0;
}
```

- Pas assez d'espaces, obscur (style)
- Répétition de code, variables globales (qualité)
- Mauvais usage de construction (logique)

Bon style du code

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

– Dishonorable mention, Obfuscated C Code Contest, 1984.

- Il faut que le code ne pique pas aux yeux...
- Espaces propre
  - Toujours autour de l'affectation : `sum += arr[i];`
  - Souvent autour des opérateurs binaires : `delta = b*b + 4*a*c;`
  - Jamais autour des opérateurs unaires (aussi `.` et `->`) : `pile->top++;`
- Éviter des lignes trop longues
- Briser et indenter la ligne aux opérateurs et aux virgules

```
screen = SDL_CreateWindow("Mandelbrot",
                          SDL_WINDOWPOS_UNDEFINED,
                          SDL_WINDOWPOS_UNDEFINED,
                          LEN, LEN, SDL_WINDOW_SHOWN);
```


Bon style du code (suite)

- Indentation obligatoire pour les blocs de code
- Si le bloc est court, on écrit sur une ligne.
- Accolades à la fin si le bloc est vide
- Pas trop de parenthèses

```
int first_zero(int* arr, int n){
    int i = 0;
    if(n < 0) return -1;
    while(i < n && arr[i] != 0) i++;
    return i;
}
```

- Préférer toujours +=, -=, *=, %= etc. si possible
- Donner un nom descriptif aux variables, mais pas trop long
- Conventions et acronymes usuelles :
  - `i`, `j`, `k` pour les indices de boucle `for`
  - `str`, `ptr`, `src`, `dest`, `size`, `cnt`, `new`, `old`, `tmp`, ...

Bonne qualité du code

- **Ne pas se répéter ! ! !**
- S'il faut changer quelque chose, il faut la changer partout.
- Si on en rate une, BOOOOOM ! ! !
- Pour les valeurs : utiliser les **constantes**
- Pour les codes : refactoring en une fonction, simplifier le code

```c
enum {EAST, WEST, NORTH, SOUTH};

void move_hole(Board b, int x, int y,
               int dir){
  switch(dir){
  case EAST:
    exchange(b, x, y, x + 1, y); return;
  case WEST:
    exchange(b, x, y, x - 1, y); return;
  case NORTH:
    exchange(b, x, y, x, y - 1); return;
  case SOUTH:
    exchange(b, x, y, x, y + 1); return;
  default: return;
  }
}
```

```c
enum {EAST, WEST, NORTH, SOUTH};

void move_hole(Board b, int x, int y,
               int dir){
  /* Offset for each direction */
  static const int offsetx[] =
      {1, -1, 0, 0};
  static const int offsety[] =
      {0, 0, -1, 1};
  exchange(b, x, y, x + offsetx[dir],
           y + offsety[dir]);
  return;
}
```


Bonne qualité du code (suite)

- Éviter les variables globales, passer plutôt par paramètre
- Initialiser les variables à déclaration tant que possible
- Éviter les nombres magiques, sauf évident ; sinon, donner des explications

```c
int is_square(uint64_t p){
    uint64_t c;
    if((int64_t)(0xC840C04048404040ULL << (p & 63)) >= 0)
        return 0;
    c = rint(sqrt(p));
    return p == c * c;
}
```

- C'est juste incompréhensible sans plus de commentaires...

Parfois c'est un choix

```c
int list_len(List s){
   List current = s;
   int size = 0;
   while(current != NULL)
      size++;
      current = current->next;
   }
   return size;
}
```

```c
int list_len(List s){
   int size = 0;
   for(; s != NULL; s = s->next) size++;
   return size;
}

int list_len(List s){
   int size = 0;
   List cur;
   for(cur = s; cur != NULL; cur = cur->next)
      size++;
   return size;
}
```

- Est-ce que l'un est mieux que l'autre ?
- Le premier : facile à comprendre, mais plus long
- Le deuxième : court, mais un peu cryptique ...
- Le troisième : bonne balance, mais un peu redondant ?

Bonne logique du code

- **Bien réfléchir avant d'écrire le code**
- Il faut choisir la bonne structure de données.
- Il faut choisir le bon algorithme.
- Il faut que le code fasse la bonne chose. (Mais oui ! !)
- Typage : struct, pointeur ou pointeur de pointeur ?
- Scope : où est visible cette variable, et sa durée de vie ?
- Mémoire : les données se trouve où ?
- Initialisation : quelle valeur ?

Exemple : Ensemble d'éléments

- **Qu'est-ce qu'on veut en faire ?**
- Des `int` dans un intervalle pas trop grand : tableau booléen, tableau à bit
- Le nombre borné, addition et itération : tableau d'éléments
- Le nombre inconnu, addition et itération : tableau dynamique
- Suppression à un endroit fixé : pile, file, ...
- Suppression à tout endroit, parcours dans un sens : liste simplement chaînée
- Suppression à tout endroit, parcours dans les deux sens : liste doublement chaînée
- Difficulté d'implantation très différente ! Il faut choisir la plus simple qui remplit le besoin.

Exemple : Traiter une liste chaînée

```c
typedef struct node {
  Data data;
  struct node * next;
} Node, *List;
```

- Comment faire une fonction qui ajoute un élément au début ?
- Le début change, donc il faut que ce soit passé par pointeur.
- La nouvelle cellule doit persister à la sortie, donc `malloc`.
- Et il faut que ça marche avec `NULL` (liste vide).

```c
int push_list(List* lst, Data data){
  Node* newnode;
  if(lst == NULL) return -1; /* Invalid pointer */
  newnode = malloc(sizeof(node));
  if(newnode == NULL) return -2; /* Unable to allocate memory */
  newnode->data = data;
  newnode->next = *lst; /* Works even for *lst empty (NULL) */
  *lst = newnode;
  return 0;
}
```


Exemple : implanter une file

- File (*queue* en anglais) : premier entré, premier sorti
- Idée naïve : liste chaînée, difficile à gérer
- Idée moins naïve : liste chaînée dans un tableau dynamique
- Trouver une case vide pour un nouvel élément, pas évident !

```c
struct cell {
  Data data;
  int next;
};
struct Queue {
  int head, tail, size;
  struct cell *arr;
};
```

- Idée maline : juste un tableau dynamique, mais cyclique
- Le cas de réallocation doit être géré soigneusement.

```c
struct Queue {
  int head, tail, size, count;
  Data *arr;
};
```


Un mot sur la performance

- Estimer le temps qu'un calcul va prendre
- Règle générale : environ 1 milliard opérations simple par seconde
- Ce qui est coûteux : fichier, affichage, graphique, ...
- Attention aux coûts implicites des fonctions !

```c
int rev2_list(List* lst){
  if(list_len(lst) < 2) return -1;
  Node* snd = (*lst)->next; /* Pointer to the second node */
  (*lst)->next = snd->next;
  snd->next = *lst;
  *lst = snd;
  return 0;
}
```

- **Très inutile** de parcourir toute la liste pour deux éléments...

Commentaires

- Il faut une bonne quantité, obligatoire à certains endroits
- Dans les `.h`
  - Expliquer le sens des types et des champs des structures
  - Expliquer le sens, l'utilité et la convention des variables
  - Expliquer les paramètres et les fonctionnalités des fonctions
- Dans les `.c`
  - Expliquer les choix d'implantation
  - Expliquer le fonctionnement et l'algorithme, surtout les "astuces"
- Noms de variable bien choisis ⇒ Commentaires épargnés
- Éviter des commentaires inutiles

Exemple de code bien commenté

`list.h`

```c
#ifndef __LIST_H
#define __LIST_H

/* Definition of Data */
#include <data.h>

/* Error codes */
#define INVALID_PARAM -1
#define FAILED_MALLOC -2

/* Node of simply chained list */
typedef struct node {
  Data data;
  struct node * next;
} Node;

/*
 * A list is a pointer
 * to its first node.
 */
typedef Node *List;

/* Returns an empty list */
List empty_list();

/* Returns the length of the list. */
int list_len(List lst);

/*
 * Add an element at the beginning
 * of the list.
 *
 * Param:
 * lst : pointer to the list
 * data : the new element
 *
 * Returns 0 for success,
 * INVALID_PARAM for invalid parameter,
 * FAILED_MALLOC for failed allocation.
 */
int push_list(List* lst, Data data);
#endif
```


Exemple de code bien commenté (suite)

`list.c`

```c
#include <list.h>

/* Returns an empty list */
List empty_list(){
  return NULL; /* Empty list points to nowhere */
}

/* Returns the length of a list */
int list_len(List lst){
   int size = 0;
   Node* cur; /* The current cell we count */
   for(cur = lst; cur != NULL; cur = cur->next) size++;
   return size;
}

/* Add an element at the beginning of a list*/
int push_list(List* lst, Data data){
  Node* newnode;
  if(lst == NULL) return INVALID_PARAM; /* Invalid pointer */
  newnode = malloc(sizeof(node));
  if(newnode == NULL) return FAILED_MALLOC; /* Unable to allocate memory */
  newnode->data = data;
  newnode->next = *lst; /* Works even for *lst empty (NULL) */
  *lst = newnode;
  return 0;
}
```