Programmation avancée en C :

Functions, entrée-sortie de base

Licence informatique 3e année

Université Gustave Eiffel

Section 1

Fonctions

Les fonctions

- Une fonction a un prototype composé de :
  - un nom (identificateur)
  - des paramètres nommés et typés
  - un type de retour
- Exemple de fonction :

```c
int average(int t[], int n){
  float sum=0;
  int i;
  for (i = 0; i < n; i++) sum += t[i];
  return sum / n;
}
```

Le type `void` :

- En valeur de retour → pas de valeur de retour
- En paramètre → pas de paramètre (facultatif)

Définition VS déclaration

- Une fonction doit être déclaré avant d'être appelée.
- Définition = code de la fonction
- Déclaration = juste son prototype avec `;`
- Prototype dans le header `.h` pour les fonctions à utiliser ailleurs
- Prototype dans le code `.c` pour les fonctions privées

```c
void get_oeuf(void);

void get_poule(void){
  /* ... */
  get_oeuf();
  /* ... */
}

void get_oeuf(void){
  /* ... */
  get_poule();
  /* ... */
}
```

return

- Pour quitter la fonction
- Renvoie une valeur si retour ≠ `void`
- Obligatoire, sauf les fonctions avec le type de retour `void`
- Il faut couvrir tous les chemins d'exécution par des `return` !

```
1 int print_sum2(int a, int b){
2   if(a > 0){
3     printf("%d + %d = %d\n", a, b, a+b);
4     return a+b;
5   }
6 }
```

donne

```
warning: control reaches end of non-void function [-Wreturn-type]
```


Valeurs de retour

- On peut ignorer une valeur de retour, *e.g.*, `printf`, `scanf`.
- On ne peut pas utiliser une fonction `void` dans une expression :

```
1 void print_sum(int a, int b){
2   printf("%d + %d = %d\n", a, b, a + b);
3 }
4
5 int main(int argc, char* argv[]){
6   float f = print_sum(5, 6);
7   printf("essai : %f\n", f);
8   return 0;
9 }
```

donne

```
error: void value not ignored as it ought to be
```


Les paramètres

- Tous les paramètres sont passé **par valeur** !
- Avant chaque appel, les paramètres sont tous recopiés.
- Conséquence 1 : On peut utiliser les paramères comme des variables.
- Conséquence 2 : Les changements effectués aux paramères n'ont pas d'effet à la sortie de la function.
- Conséquence 3 : Coût considérable si les paramètres sont gros (les grosse structures).

Passage d'adresse

- Comment faire pour avoir un effet de bord ?
  ⇒ Donner l'adresse (pointeur) de `foo` pour pouvoir modifier le contenu de la zone mémoire correspondante
- C'est fait avec l'opérateur d'adressage `&`.
- Pour indiquer qu'une fonction reçoit une adresse, on met un type de pointer (avec `*`) :
  `void copychar(char src, char* dest);`
- `dest` : l'adresse d'un `char`
- `*dest` : le zone de mémoire pointé par `dest`, taille indiquée par le type (`char` ici), et vu comme une variable

Passage d'adresse

```
1  void add_3(int *a){
2    *a = *a + 3;
3  }
4
5  int main(int argc, char* argv[]){
6    int foo = 14;
7    add_3(&foo);
8    printf("foo = %d\n", foo);
9    return 0;
10 }
```

donne

```
$> ./a.out
foo = 17
```

- Ça marche !

Passage d'adresse

- Utile

  - pour modifier un paramètre

  - pour retourner plusieurs valeurs

  - pour retourner une ou plusieurs valeurs + un code d'erreur

  - pour éviter de copier des grosses structures en paramètre

Modifier une variable

- Exemple classique : les compteurs
- Attention : `*n++` n'est pas `(*n)++`, mais `*(n++)`

```
1  /* Reads from the given file the first character */
2  /* that is not a newline. Returns it or -1 if the */
3  /* end of file is reached. If there are newlines, */
4  /* '*n' is updated. */
5  int read_char(FILE* f, int *n){
6    int c;
7    while ((c=fgetc(f)) != EOF){
8      if (c=='\n') (*n)++;
9      else return c;
10   }
11   return -1;
12 }
```


Retourner plusieurs valeurs

- On utilise des passages par adresse sauf potentiellement pour une seule donnée.

```
1  /* compute the irreducible fraction x/y such that */
2  /* x/y = a1/b1 + a2/b2 */
3  void add_fractions(int a1, int b1, int a2, int b2,
4                                     int *x, int *y){
5    int p;
6    *x = a1*b2 + a2*b1;
7    *y = b1*b2;
8    p = gcd(*x, *y);
9    *x /= p;
10   *y /= p;
11 }
```

- Pourquoi favoriser le numérateur ou le dénominateur ?
  ⇒ Ce serait une faute de goût de ne pas mettre les deux variables au même niveau.

Avec un code d'erreur

- On choisit plutôt de retourner le code d'erreur et de récupérer les résultats par passage par adresse.
- L'écriture suivante s'inscrit complètement dans l'esprit du langage :

```
...
if (fonction_code_erreur(&arg1, &arg2, &arg3)){
  fprintf(stderr, "Il aurait fallu des choses, "
                  "des choses et des choses....\n");
  return -1;
}
var = arg1 + arg2 +arg3;
...
```

⇒ C'est un `if` avec effet de bord qui poursuit l'erreur ou laisse continuer l'algorithmique.

Quelques détails sur les tableaux

- Les tableaux sont des pointeurs.
- Donc ils sont passés par adresse automatiquement.

```
/* equivalent : void foo(int* t) */
void foo(int t[]){
  t[1] = 42;
}

int main(int argc, char* argv[]){
  int t[3];
  t[0] = t[1] = t[2] = 0;
  foo(t);
  printf("%d\n",t[1]);
  return 0;
}
```

donne bien 42 si vous en doutiez...

Quelques détails sur les tableaux

- Pas de différence formelle entre un tableau de `truc` et un `truc` passé par adresse.
- Pour le compilateur, les 3 prototypes suivants sont les mêmes !
  `void foo(char s[]);`
  `void foo(char* s);`
  `void foo(char *s);`
- Pour s'y retrouver, on pourrait noter par convention :
  `float s[]` : `s` est un tableau de `float`
  `float *s` : `*s` est un `float` passé par adresse
- Ou bien de traiter tous comme des pointeurs.
- Donc la fonction ne connais pas la taille du tableau !

Fonction à nombre de paramètres variable

- Comme `int printf(const char*, ...);`
- Type et macros dans `stdarg.h`
- Voir `man va_arg` pour les détails
- Au moins un paramètre
- Les types sont à gérer par le programmeur (souvent à partir de l'argument obligatoire).
- Attention ! Les "fonctions" dans `stdarg.h` pourrait être des macros, donc faire gaffe au parenthésage !
- Assez compliqué, à utiliser si strictement nécessaire.

Structures dans les fontions

- En paramètre : passage par valeur !
  - Occuper beaucoup de place sur la pile (tableaux)
  - Prendre du temps à recopier
- Idem en tant que valeur de retourne.
- Conclusion : passer les grosses structures par adresse (qu'on les modifie ou pas)
- Les petites structures (quelques `int`), ça va...
- Si on retourne une (adresse d'une) structure, alors il faut penser à l'allocation dynamique.

Exemple de structures dans les fonctions

```
1  struct array{
2    int t[100000];
3    int size;
4  };
5
6  void f(struct array* a){
7    int i;
8    for (i=0 ; i<(*a).size ; i++)
9      printf("%d\n", (*a).t[i]);
10 }
```

- Attention au parenthèsage : `(*a).t[i]` ≠ `*a.t[i]`
- **Notation simplifiée** : `foo->bar` ⇔ `(*foo).bar`
- Donc on préfère d'écrire `a->t[i]`.

Le cas de `main`

- Fonction particulière :
  - `return` quitte le programme
  - renvoie le code de retour du programme (par convention, on a 0 = OK, $\neq 0$ = erreur)
- Si `main` est appelé explicitement par une fonction, son `return` fonctionne alors normalement.
- Un programme se termine "normalement" lorsque la pile d'appel de fonctions est vide. Il faut ainsi fermé le `return` de l'appel originel de `main` (celui qui est fait par la ligne de commande lors de l'appel de l'exécutable).

Écrire une fonction

- Réfléchir à l'utilité de la fonction
- 1 fonction = 1 seule tâche
- Ne pas mélanger calcul et affichage

```
1  int minimum(int a, int b){
2    int min = (a < b) ? a : b;
3    printf("min␣=␣%d\n", min);
4    return min;
5  }
```

On fait de l'affichage dans des fonctions de type de retour `void` ou pour du débogage. Mais ici, c'est une faute de goût.

Les paramètres

- Ne pas mettre trop de paramètres !
- Encapsuler les paramètres trop nombreux
- Si la structure est trop grande, passer son pointeur.

```
1  typedef struct{
2    int nb_largeur_pixel;
3    int nb_hauteur_pixel;
4    int background_color;
5    int set_option;
6    int ...
7  }ParamGraph;
8
9  void create_image(ParamGraph pg);
10 void change_image(ParamGraph pg, int action);
11 ...
```


Définir le prototype

- De quoi la fonction a-t-elle besoin ?
- Retourne-t-elle quelque chose ?
- A-t-elle un effet de bord ?
- Y a-t-il des cas d'erreurs ?
- Si oui, 3 solutions :

  - mettre un commentaire (bof, solution du feignant...)

  - renvoyer un code d'erreur (qu'on peut ré-exploiter après)

  - afficher un message et quitter le programme (cas les plus critiques)

Le commentaire

- Solution du programmeur pressé : utile quand on maîtrise tous les appels de la fonction, et on assure tous les conditions nécessaires à chaque appel :

```
1  /* Copies the array 'src' to the 'dest' one.
2     'dest' is supposed to be large enough. */
3  void copy(int src[], int dest[]);
```

ou encore

```
1  /* returns 1 if 'w' is an English word,
2     returns 0 otherwise.
3     'w' is not supposed to be NULL. */
4  int is_english_word(char* w);
```


Le code d'erreur

- Pas de problème si la fonction n'a pas vocation à retourner quelque chose.

```
1  int init(int t[], int size){
2    int i;
3    if (size <= 0) return 1;
4    for (i = 0; i < size; i++) t[i] = 0;
5    return 0;
6  }
```

- Sinon, passer le résultat par adresse, ou bien la confusion...

```
1  int main(int argc, char* argv[]){
2    printf("%s --> %d\n", argv[1], atoi(argv[1]));
3    return 0;
4  }
```

donne

```
nborie@perceval:> ./a.out 0
0 --> 0
nborie@perceval:> ./a.out foo
foo --> 0
```

Le code d'erreur

- Dans certains cas, si le code d'erreur ne peut pas être un résultat valid, alors on peut le retourner directement.

```
1 int str_length(char* s){
2   int i;
3   if (s == NULL) return -1;
4   for (i = 0; s[i] != '\0'; i++) {}
5   return i;
6 }
```

⇒ Ok, -1 indique que l'argument n'était pas une vraie chaîne C.

- On retourne dès qu'il y a une erreur, pour éviter les `else` inutiles.

L'interruption du programme

- À n'utiliser que dans des cas **TRÈS TRÈS GRAVES** !
  - rien de mémoire disponible
  - erreurs d'entrées/sorties dangereuses pour la suite du programme
  - mauvais paramètres donnés au programme et risques pour la suite
- Message d'erreur sur `stderr` + `exit(`$\neq 0$`)`

Dans la plupart des cas, on préférera ne pas quitter brutalement le programme mais retourner dans le `main` qui peut, par exemple, arrêter l'algorithmique et sauvegarder les résultats partiels calculés.

Esthétique des fonctions

- Fait : on lit 10 fois un code écrit 1 fois.
- Soigner la présentation :
  - Commentaires
  - Noms explicites des fonctions et variables
  - Indentation
- Regrouper les fonctions de même thème dans les `.c`

Fonction récursive

- Fonction s'appelant elle-même.
- Attention à la condition d'arrêt
- Fait gonfler la pile
- Éviter s'il on peut faire facilement autrement
  `Python`, par default : 1000 appels récursifs ⇒ `max depth of recursion`
- Utilisation judicieuse : voir cours d'algo !

Fonction récursive

- Ne tester qu'une seule fois les cas d'erreurs

```
1  int sum(Tree* t, int* s){
2    if (t == NULL) return ERROR;
3    *s = *s + t->value;
4    if (t->left != NULL) sum(t->left, s);
5    if (t->right != NULL) sum(t->right, s);
6    return OK;
7  }
```

est mieux écrit comme

```
1  int sum_aux(Tree* t){
2    if (t == NULL) return 0;
3    return t->value + sum_aux(t->left) + sum_aux(t->right);
4  }
5
6  int sum(Tree* t, int *s){
7    if (t==NULL) return ERROR;
8    *s = sum_aux(t);
9    return OK;
10 }
```


Section 2

Utilités et entrée-sortie de base

Quelques fonctions sur les chaînes

- Fonctions définies dans `string.h`

```
char* strcpy(char* dest, const char* src);
char* strcat(char* dest, const char* src);
size_t strlen(const char* s);
char* strdup(const char* s);
int strcmp(const char* a, const char* b);
```

- `strcpy` copie, `strcat` concatène, `strlen` mesure, `strdup` duplique, `strcmp` compare
- Attention, aucun test sur la validité des chaînes.
  - Les chaînes ne doivent pas être `NULL`.
  - `dest` doit avoir assez d'espace mémoire réservé.
  - Le `\0` à la fin doit bien être placé (segfault sinon).

Entrée-sortie formatté avec les chaînes

- `int sprintf(char* str, const char* format, ...)`
  - Comme `printf`, mais en écrivant à `str`.
  - Attention aux débordements ! Il faut assez d'espace réservé pour str.
- `int sscanf(char* str, const char* format, ...)`
  - Comme `scanf`, mais en lisant `str`.
  - Il faut que `str` termine bien avec `\0` (sinon segfault).

Fonctions mathématiques

- Définies dans `math.h` :
  - `double cos(double x);`
  - `double sin(double x);`
  - `double tan(double x);`
  - `double sqrt(double x);`
  - `double exp(double x);`
  - `double log(double x);`
  - `double log10(double x);`
  - `double pow(double x, double y);`
- Compiler avec l'option `-lm`

Nombres aléatoires

- Fonctions définies dans `stdlib.h`
- `int rand(void);`
  - retourne un entier entre `0` et `RAND_MAX`
- `void srand(unsigned int seed);`
  - initialise le générateur avec une graine
- Pour une graine qui change :
  - `time_t time(time_t*);` dans `time.h`
- Initialiser avec `srand(time(NULL));`

Qu'est-ce qu'un fichier ?

- Espace de stockage (permanent) des données
- Presque tout est fichier sous `Unix`
- Opérations permises :
  - ouvrir, lire, écrire, fermer
- Il n'est pas prévu de mécanisme pour enlever un morceau au milieu d'un fichier.
- Tous les fichiers ne sont pas en *random access* (ex : `/dev/mouse`)

Les primitives

- Primitives systèmes :

```
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

- Peu pratique car ne manipule que des octets, à éviter

```
1 #include <unistd.h>
2
3 /* Ouais on ecrit a l'ecran sans <stdio.h>, meme pas peur ! */
4 int main(int argc, char* argv[]){
5   char* s="Salut!\n";
6   write(1, (const void*)s, 7);
7   return 0;
8 }
```

E/S civilisées

- `FILE` = structure décrivant un fichier (ouf !)
- Varie selon les implémentations
  ⇒ ne jamais utiliser les champs directement
- On exploite cette structure avec les fonctions de `stdio.h`

```
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
FILE *stream);
```

- Normalement on ne manipule que les `FILE*` !

Noms de fichiers

- Un nom de fichier peut être relatif ou absolu.
- Extension facultative, possibilité d'avoir plusieurs `.` `projet_c_L3.tar.gz`
- attention aux fichiers cachés sur Linux (*e.g.*, `.bashrc`)
- Le séparateur varie selon les systèmes (portabilité...).
  - `/` sous Linux
  - `\` sous Windows
  - `:` sous les vieux Mac

Ouvrir un fichier

- Options de `fopen` :
  - `"r"` : lecture seule
  - `"w"` : écriture seule (fichier créé si nécessaire, écrasé si existant)
  - `"a"` : ajout en fin (fichier créé si nécessaire)
  - `"b"` : mode binaire
- Option composées :
  - En mode binaire : `"rb"`, `"wb"`, `"ab"`
  - En mode lecture et écriture : `"r+"`, `"w+"`, `"a+"`
- Combinaisons possibles : `"rb+"`, `"a+b"`, *etc.*
- Par défaut, opération en mode texte (et pas binaire)
  - différences pour les retours à la ligne suivant les systèmes
- **Toujours** tester la valeur de retour !

Ouvrir un fichier

- Un nom du fichier relatif est par rapport au répertoire courant.
- En création, les droits dépendent du `umask` (souvent `rw-rw-r--` par défaut)

```
1 int main(int argc, char* argv[]){
2   FILE* foo = fopen("foo", "w");
3   if (foo == NULL) exit(1);
4   fprintf(foo, "Hello you\n");
5   fclose(foo);
6   return 0;
7 }
```

donne

```
nborie@perceval:~> ./test
nborie@perceval:~> ls -al foo
-rw-rw-r-- 1 nborie nborie 10 oct.  29 14:04 foo
```

Fermer un fichier

- `fclose(f);`
- `f` doit être l'adresse d'un `FILE` valide (donc non `NULL`)
- `f` ne doit pas avoir déjà été fermé
- Tout fichier ouvert doit être fermé
- Petit astuce : toujours coupler `fopen(f)` et `fclose(f)`

E/S formatées

- `fprintf` et `fscanf` fonctionnent comme `printf` et `scanf`
- Elles sont bufferisées.
- Évacuer les buffers : `fflush`
- En fait, `printf(...);` et `scanf(...);` sont équivalents à `fprintf(stdout,...);` et `fscanf(stdin...);`
- `stdin`, `stdout` et `stderr` sont des `FILE*` spéciaux.

fgetc/fputc

- `int fgetc(FILE* stream);`
- `int fputc(int c, FILE* stream);`
- Servent à lire ou écrire un seul caractère
- Retourne `EOF` en cas d'erreur :
  - pas de caractère de fin de fichier
  - `EOF` n'est pas un caractère

fgets

- `char* fgets(char *s, int size, FILE *stream);`
- `fgets` lit des caractères jusqu'à :
  - un `\n` (qui est copié dans le résultat `s`)
  - la fin du fichier
  - avoir lu `size-1` caractères (à cause de `'\0'`)
- Retourne `NULL` en cas erreur, `s` sinon
- Évite les problèmes de débordement

fputs

- `int fputs(const char *s, FILE *stream);`
- `fputs` écrit la chaîne `s` dans le fichier `f`
- retourne `EOF` en cas d'erreur, `0` sinon

Fin de fichier en lecture

Attention à toujours vérifier les descriptions de valeur de retour (en cas de fin de fichier) dans le manuel `man` !

- `fscanf` : retourne `EOF`
- `fgetc` : retourne `EOF`
- `fgets` : retourne `NULL`
- `fread` : retourne `0`

```
1 void copy(FILE* src, FILE* dest){
2   char buffer[4096];
3   size_t n;
4   while ((n=fread(buffer, sizeof(char), 4096, src)) > 0){
5     fwrite(buffer, sizeof(char), n, dest);
6   }
7 }
```

- On peut aussi tester avec `int feof(FILE *stream);` (déconseillé).

Créer un répertoire

- `int mkdir(const char *pathname, mode_t mode);`
- `mode` spécifie les droits du repertoire
- ces droits seront combinés avec le `umask`
- fonction non portable !
- nécessite les bibliothèques `sys/types.h` et `sys/stat.h`