Programmation avancée en C :

Système de type Unix et langage C

Licence informatique 3e année

Université Gustave Eiffel

Pourquoi encore du C ?

Que donnent ces codes ?

1. Manipulation des chaînes

```c
char* cutoff(char* str, int k){
    if (k > strlen(str)) return NULL;
    str[k] = 0;
    return str;
}
```

Que donne
char str[] = "University"; printf ("%s", cutoff(str, 4)); ?

2. Moyenne de deux niveaux de gris (entre 0 et 255) :

```c
unsigned char avg_grayscale(unsigned char gs1,
                            unsigned char gs2){
    return (gs1 + gs2) / 2;
}
```

Que donne printf ("%d", (**int**)average_grayscale(135, 145)); ?

Introduction de ce cours

Objectifs :

- approfondir sur le langage C
- connaître certains notions élémentaires des machines réelles
- analyser un problème et le résout avec un programme efficace et modulaire
- produire du code propre
- savoir développer efficacement et à plusieurs
- comprendre les points forts et faibles du C
- prendre du recul pour mieux voir les autres langages et paradigmes

Modalités :

- Projet en binôme
- Examen
- TP (note facultative pour certains TP)

Système Unix

- 1969 : Système d'exploitation Unix, écrit en assembleur (Ken Thompson et Dennis Ritchie)
- 1970 : Nouveau langage (C) pour réécrire Unix (Thompson et Ritchie)
- 1991 : Linux, réécriture libre du noyau Unix (Linus Torvalds, à l'époque étudiant finlandais)


Thompson


Ritchie


Torvalds

Linux

Généralités :

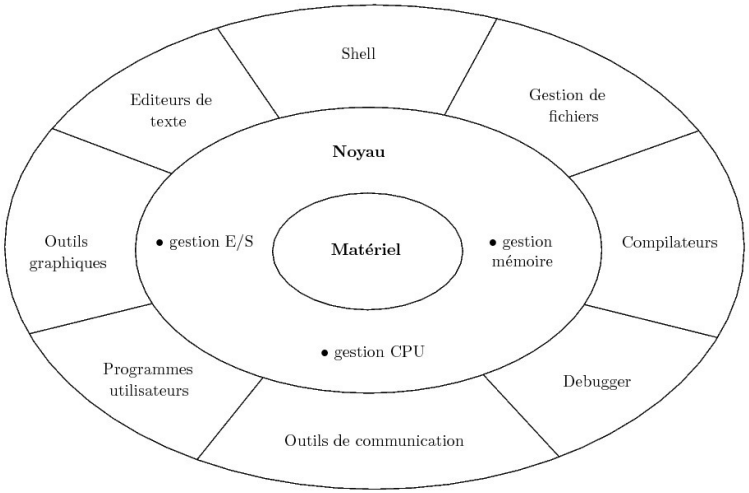
- système d'exploitation libre : open source et gratuit
- multi-utilisateurs
- multi-tâches

Rappel :

Un système d'exploitation est un ensemble de programmes faisant l'interface entre les ressources matérielles d'un ordinateur et les applications informatiques de l'utilisateur. Il fournit aux applications une abstraction pour accéder aux périphériques de façon uniforme.

!!! Nécessité d'un langage / système non machine dépendant !!!

Composants principaux d'un système Linux



Système de fichier `Unix`

Principales caractéristiques :

- racine unique : /
- les périphériques sont vus comme des sous-répertoires
- sensibilité à la case (majuscules ≠ minuscules, `A.txt` ≠ `a.txt`)
- droit sur tous les fichiers (répertoires ⊂ fichiers spéciaux) : lire - écrire - exécuter, pour les utilisateurs propriétaire - groupe - autres

Commandes courantes :

`ls, cd, mv, cp, rm, rmdir, pwd, find, grep, chmod, chown, ...`

Aide sous `Unix`

Tout est dans le manuel `man` d'`Unix` ! La politique de développement d'`Unix` impose que toutes les fonctionnalités soient documentées. Le manuel est un exemple sérieux de modélisation de spécifications.

Pour chaque fonction :

- NAME : nom de la (ou des) fonction(s)
- SYNOPSIS : prototype(s) et bibliothèque(s)
- DESCRIPTION : description comportementale
- RETURN VALUE : description de la valeur de retour
- NOTES : toutes informations méritantes d'être consignées
- BUGS : bogues connus et confirmés
- EXAMPLES : exemples d'utilisation
- SEE ALSO : fonctionnalités similaires
- COLOPHON : informations sur la dernière révision

Section 3 du manuel Unix

Dédiée à la programmation en langage C. Pour tout savoir sur une fonction `foo` du langage C (et des bibliothèque standards), taper `man 3 foo` dans un terminal.

exemple : `man 3 qsort`

```
QSORT(3)            Linux Programmer's Manual                   QSORT(3)

NAME
       qsort, qsort_r - sort an array

SYNOPSIS
       #include <stdlib.h>

       void qsort(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *));

       void qsort_r(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *, void *),
                  void *arg);
```


Licence 3

Pour une utilisation raisonnable et efficace d'un système de type `Unix` et votre niveau d'étude, vous devez connaître/savoir utiliser/être capable de retrouver les commandes suivantes :

cat	cd	chgrp	chmod
chsh	cp	cut	diff
echo	find	gcc	gdb
grep	head	kill	less
ls	man	mkdir	more
mv	ps	pwd	rm(dir)
sed	sort	screen	ssh
tail	tee	top	umask
uniq	valgrind	wc	yes
>	>>	2>	2>>
2&1>	<	<<	| (pipe)

Lire le manuel en cas de besoin !

Le langage C

Pourquoi et comment :

- C, c'est le langage apparu après le B
- 1972 : laboratoire Bell, D. Ritchie et K. Thompson
- 1989 : Norme ANSI du langage C
- langage impératif
- à la fois haut et bas niveau
- programmation système (noyau Linux)
- programmation haute-performance

Pourquoi apprendre le C ?

C est un langage de programmation :

- très répandu, portable
- facile à comprendre
  - peu de syntaxe
  - syntaxe efficace, concise
- disposant des principaux types de base généraux
- permettant de définir de nouveaux types et de structures
- permettant d'avoir un très bon contrôle de la machine
  - la compilation est aisée et intuitive
  - le programmeur a une **vision directe de la mémoire**
- gratuit
- amusant (mais si, mais si...)

Quelques défauts quand même

Sur le mécanisme du langage :

- Le typage est affaibli par l'utilisation de pointeurs.
- La syntaxe est parfois un peu ambiguë.
- Certains opérateurs n'ont pas la priorité qu'ils devraient avoir.
- Écrire du code réutilisable est difficile.
- Pas de gestion automatique d'erreurs (exceptions).

Aussi sur les qualités du langage :

- La syntaxe concise peut rendre les programmes très difficiles à lire.
- L'objectif de bon contrôle de la machine rajoute du travail pour le programmeur, qui ne peut recourir à des mécanismes automatiques (ramasse-miette, vérification d'indices, ...).

Les bogues dans les programmes sont courants et difficiles à trouver.

On peut écrire des programmes illisibles.

Il faut du **bon sens** et des **bons pratiques** !

Mais quel langage C ?

Les normalisations du langage C sont en perpétuelle évolution. Certains documents fixent les règles (C89, ISO/IEC 9899 :1989, ISO C9X, C99, etc).

. : Nous allons faire du C ANSI :.
(ainsi, on compilera toujours avec `-Wall -ansi`)

Le C ANSI

- standard
- plus portable et lisible que le C classique
- permet d'éviter quelques erreurs de programmation difficiles à retrouver
- est plus largement répandu que le C ISO
- est plus stable que les dernières normalisations ISO

Quand utiliser le C ?

- Pour écrire des programmes portables
- Pour écrire des programmes efficaces
- Gestion de la mémoire
- Maîtrise du code exécuté

Applications typiques :

- Systèmes d'exploitation
- Machine à ressources limitées
- Applications pour calculs intensifs
- Tout autre type d'applications

Caractéristiques du langage C

C est un langage :

- compilé
- à effets de bords
- impératif
- structuré
- typé moyennement
- déclaratif
- de niveau moyen

C est un langage impératif

Programmation par l'ordre, par le verbe :

```
manger(chat, souris);
```

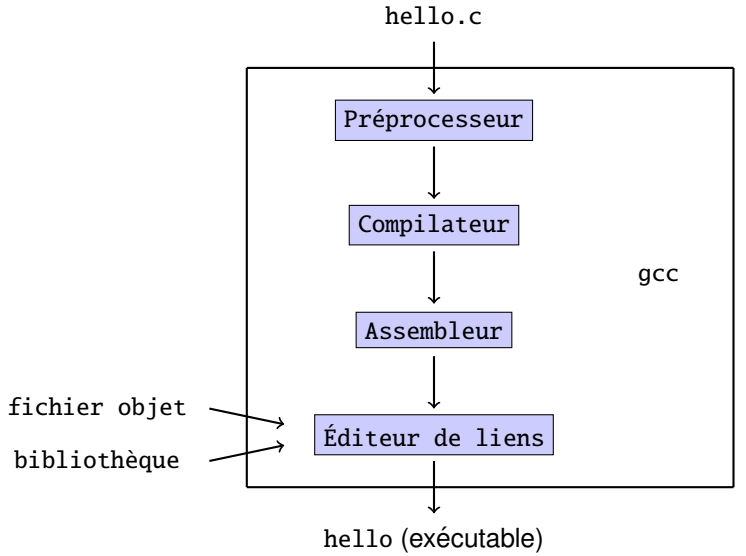
le code définit des types, utilise des variables sur ces types et contient des fonctions ayant pour arguments tous les intervenants du problème résolu par la fonction.

Programmation objet, par le sujet :

```
chat.manger(souris);
```

le code modélise des objets et chaque objet dispose de méthodes (fonctions rattachées à un objet) où l'objet rattaché joue un rôle central pour le problème résolu.

Qu'est ce qu'un programme C ?



Les fichiers sources

- fichiers textes contenant des instructions
- extensions spéciales `.c`
- syntaxe très précise à respecter
- **programmation modulaire** : un fichier par thème
  - bibliothèque mathématique
  - interface graphique
  - entrées/sorties
  - ...

Le premier programme

fichier `hello.c` :

```
1  /* Affichage du message Hello world! */
2
3  #include <stdio.h>
4
5  int main(int argc, char* argv[]){
6    printf("Hello world!\n");
7    return 0;
8  }
```

- Commentaires entre `/* ... */`. Pas de `//` pour `C ANSI` ! Pas d'imbrication non plus.
- Les commandes commençant par # : directives pour le préprocesseur.
  Ici l'inclusion de la bibliothèque contenant la fonction `printf`.

Le premier programme

fichier `hello.c` :

```
1 /* Affichage du message Hello world! */
2
3 #include <stdio.h>
4
5 int main(int argc, char* argv[]){
6   printf("Hello world!\n");
7   return 0;
8 }
```

- `main` : commande principale, point d'entrée d'un programme.
- Le type de retour apparaît en premier dans le prototype de la fonction. La fonction `main` retourne ici un entier (qui sert comme le code d'erreur).
- `argc` : nombre de paramètres, `argv` : tableau des paramètres.
  Attention : `argv[0]` est systématiquement le nom de l'exécutable.

Le premier programme

fichier `hello.c` :

```
1 /* Affichage du message Hello world! */
2
3 #include <stdio.h>
4
5 int main(int argc, char* argv[]){
6   printf("Hello world!\n");
7   return 0;
8 }
```

- Les accolades { et } délimitent des blocs de code qui peuvent s'imbriquer.
- La ligne 6 affiche à l'écran la chaîne de caractère souhaitée.
- La valeur de l'expression après le mot clé `return` est retournée par la fonction.
  `return;` signifie "retourner rien", pour un type de retour `void`.

Le premier programme

- **Compilation :**
  `gcc hello.c -c`
  `gcc hello.o -o Helloworld`

  en une seule ligne :
  `gcc hello.c -o Helloworld`
- **Exécution :**
  `./Helloworld`

L'option `-c` de `gcc` compile le code sans le faire passer au linker (faiseur de liens). Cela produit des fichiers objets non exécutables mais contenant du code exécutable.

C'est l'option récurrente utilisée dans les `Makefile` pour la **compilation séparée**, qui peut économiser la compilation.

Quelques options courantes de `gcc`

- `-ansi` : compilation de C à la norme ANSI
- `-Wall` : Mise en oeuvre de certains warnings très utiles (mais pas tous)
- `-g` : inclusion dans le code produit d'informations pour le débogueur
- `-pg` : inclusion dans le code produit d'informations pour le profiler
- `-c` : compilation produisant du code objet (sans édition de lien)
- `-Dmacro=valeur` : définition d'une macro pour le préprocesseur
- `-E` : préprocesseur uniquement
- `-O(0,1,2,3)` : optimisation de la taille du code et du temps d'exécution (lettre O + un chiffre, `-O0` par défaut)

Un autre exemple

```c
#include <stdio.h>
#define PI 3.14      /* une constante */

float a,b;     /* variables globales */

float sum(float x, float y){     /* une fonction */
  return x+y;
}

int main(int argc, char* argv[]){
  float c,d;
  a = PI;
  b = 1;
  c = sum(a,b);
  printf("%f + %f = %f\n", a, b, c);
  printf("d = %f", d); /* d n'est pas initialisee !!! */
  return 0;
}
```

Le C permet l'utilisation d'une variable **non initialisée**. Ce fait est à considérer comme une "faiblesse" (des flags de `gcc` peuvent détecter ces problèmes).

En deux fichiers

fichier **`f1.c`** :

```c
#include <stdio.h>
#define PI 3.14

float a,b;
/* Declaration d'une fonction definie dans un autre fichier */
extern float sum(float x, float y);

int main(int argc, char* argv[]){
  float c,d;
  a = PI;
  b = 1;
  c = sum(a,b);
  printf("%f + %f = %f\n", a, b, c);
  printf("d = %f", d);
  return 0;
}
```

fichier **`f2.c`** :

```c
float sum(float x, float y){
  return x+y;
}
```


En trois fichiers

fichier **`f1.c`** :

```c
#include <stdio.h>
#include "f2.h"
#define PI 3.14

float a,b;

int main(int argc, char* argv[]){
  float c,d;
  a = PI;
  b = 1;
  c = sum(a,b);
  printf("%f + %f = %f\n", a, b, c);
  printf("d = %f", d);
  return 0;
}
```

fichier **`f2.c`** :

```c
float sum(float x, float y){
  return x+y;
}
```

fichier **`f2.h`** :

```c
float sum(float x, float y);
```


Sources et entêtes

- **`.c`** : source code
  - Ici se trouve le code source complet de chaque fonction.
  - On inclut le fichier d'entête correspondant s'il existe.
  - **Jamais** inclure les fichier d'extension **`.c`** !
  - Créer un fichier d'entête s'il y a une fonction utilisée ailleurs.
- **`.h`** : entête
  - Déclaration des fonctions visibles de l'extérieur du **`.c`**
  - Définition de constantes
  - Description de la bibliothèque
  - Licence du code (ou votre nom ! Soyez fier de votre code ! (ou codez mieux...))

Fichiers d'entête

Les fichiers d'entête **NE DOIVENT PAS** être compilés avec `gcc`.

Pour éviter les problèmes de double définition, on les sécurise avec des macros.

```
1  #ifndef __F2_H_
2  #define __F2_H_
3
4  float sum(float x, float y);
5
6  #endif
```

Pour faciliter la lecture, utilisez toujours la même structure pour chaque projet.

Structuration d'une entête `foo.h`

Pour le fichier `foo.h`, un choix raisonnable est de faire apparaître toujours dans l'ordre suivant les déclarations qui suivent :

- Macro de Sécurité pour les inclusions multiples
- Inclusions des dépendances
- Définition des structures
- Prototypes des fonctions de foo.c
- Fin de sécurisation

Code d'une entête `foo.h`

Pour le fichier `foo.h`

```
1  #ifndef __FOO__
2  #define __FOO__
3
4  #include "autre_module.h"
5  #include "troisieme_module.h"
6
7  typedef struct bla{
8    int bar;
9    float blo;
10 }Bla;
11
12 int une_fonction(Bla a, int b);
13 void autre_fonction(Bla a, Bla b);
14
15 #endif
```


Code du fichier `foo.c`

Pour le fichier `foo.c`

```
1  #include "foo.h"
2  #include <bliotheque.h>
3
4  int une_fonction(Bla a, int b){
5  /*
6  *  VRAI CODE DE LA FONCTION
7  */
8  }
9
10 void autre_fonction(Bla a, Bla b){
11 /*
12 *  VRAI CODE DE LA FONCTION
13 */
14 }
```

Options de compilation

- -ansi : Norme qui ”garantit” la portabilité du code
- -Wall : Affiche les avertissements

La compilation (après possiblement grand ajout de code) peut provoquer l'affichage d'une quantité d'erreurs et de warning impressionnantes :

- Compiler RÉGULIÈREMENT ! (voire tester les fonctions à mesure de leur implantation)
- Commencer toujours par la PREMIÈRE erreur (ou warning) ! L'option `-Wfatal-errors` pourrait être utile.
- Méfiez-vous des traductions ! (une division par zéro dans les **entiers** provoque une ”floating point exception”)

Fichier Makefile

- Fichier qui contient les instructions nécessaires pour compiler un projet.
- N'importe qui peut alors compiler en tapant seulement la commande `make` (utilitaire d'`Unix`).
- Le fichier **DOIT** s'appeler `Makefile` ou `makefile`.
- `make -f foo` utilise le fichier `foo` comme un makefile.

LA SYNTAXE DES MAKEFILES EST PÉNIBLE ET TRÈS IMPORTANTE !
(comme tout ce qui tourne autour du `bash` d'`Unix`)

```
cible: dependance1 dependance2 ...
(tabulation)gcc -c fichier1.c
(tabulation)rm fichier2.txt
(tabulation)make cible4
...
```

Il faut bien sûr remplacer `(tabulation)` par une tabulation (pas avec des espaces !). On saute une ligne en chaque cible.

Fichier Makefile

Structure d'un Makefile classique :
A dépend de B et C, B dépend de C et main dépend de A.

```
1  exe: main.o A.o B.o C.o
2          gcc -o exe main.o A.o B.o C.o -Wall -ansi
3
4  A.o: A.c A.h B.h C.h
5          gcc -c A.c -Wall -ansi
6
7  B.o: B.c B.h C.h
8          gcc -c B.c -Wall -ansi
9
10 C.o: C.c C.h
11         gcc -c C.c -Wall -ansi
12
13 main.o : main.c A.h
14         gcc -c main.c -Wall -ansi
15
16 clean :
17         rm *.o
18         rm exe
```


Fichier Makefile avec variables

Les variables simplifient l'écriture du Makefile et le rendent plus maintenable.

```
1  CC=gcc
2  CFLAGS=-Wall -ansi
3  OBJ=main.o A.o B.o C.o
4  exe: $(OBJ)
5          $(CC) -o exe $(OBJ) $(CFLAGS)
6
7  A.o: A.c A.h B.h C.h
8          $(CC) -c A.c $(CFLAGS)
9
10 B.o: B.c B.h C.h
11         $(CC) -c B.c $(CFLAGS)
12
13 C.o: C.c C.h
14         $(CC) -c C.c $(CFLAGS)
15
16 main.o: main.c A.h
17         $(CC) -c main.c $(CFLAGS)
```

Un projet : Réaliser un répertoire

```
1  CC=gcc
2  CFLAGS=-Wall -ansi
3  OBJ=main.o date.o fiche.o repertoire.o in_out.o tri.o groupe.o
4
5  repertoire : $(OBJ)
6        $(CC) -o repertoire $(OBJ) $(CFLAGS)
7
8  main.o : main.c in_out.h repertoire.h fiche.h
9        $(CC) -c main.c $(CFLAGS)
10
11 date.o : date.c date.h
12       $(CC) -c date.c $(CFLAGS)
13
14 fiche.o : fiche.c fiche.h date.h groupe.h
15       $(CC) -c fiche.c $(CFLAGS)
16
17 repertoire.o : repertoire.c repertoire.h fiche.h tri.h
18       $(CC) -c repertoire.c $(CFLAGS)
19
20 in_out.o : in_out.c in_out.h repertoire.h
21       $(CC) -c in_out.c $(CFLAGS)
22
23 tri.o : tri.c tri.h
24       $(CC) -c tri.c $(CFLAGS)
25
26 groupe.o : groupe.c groupe.h
27       $(CC) -c groupe.c $(CFLAGS)
```


Code propre

Votre code est écrit une fois, mais lu ≥ 10 fois (y compris par vous-même) !

- Penser aux lecteurs !
- Indentation claire et constante
- Accolades consistants
- Commentaires pour tout choix d'implémentation
- Nom de variables, structures et fonctions avec du sens et formalisés de manière unifiée dans tout le code (`nom_de_fonction`, `NomDeFonction`, `Nom_De_Fonction`, ...)
- Code lisible par tous (pourquoi pas en anglais avec des mots standards)
- Éviter le mode gourou ou obscurantiste (du moins si vous voulez une note...)
- fixer et tenir des standards de développement :
  - organisation des headers
  - structure de documentation
  - entête de documentation avec référence de l'auteur
  - disposition du code, sauts de ligne réguliers, ...