



Systèmes d'exploitation

Fabien Calcado

Email: fabien.calcado@isty.uvsq.fr

EFREI L3 – LSI – 2019/2020

1

Plan général du cours

- Cours n°2 - Introduction
 - Rappels de programmation C

EFREI L3 – LSI – 2019/2020

2



Rappels de programmation

- Conception d'un programme informatique
 - Algorithme
 - Séquence bien définie d'opérations (calcul, manipulation de données, etc.) permettant d'accomplir une tâche en un nombre fini d'opérations
 - » En principe il reste indépendant de toute implémentation
 - Programme
 - Réalisation dans un langage (de programmation) particulier d'un algorithme destiné à être exécuté de manière automatique

Données en entrée ↔ Traitements ↔ Données en sortie

↓
Données intermédiaires

EFREI L3 – LSI – 2019/2020

3

Rappels de programmation

- Cycle de développement d'un programme
 - Analyse
 - Définition du problème
 - Conception (algorithme)
 - Définition précise des données, des traitements et de leur séquencement
 - » Un algorithme doit être indépendant du langage de programmation cible
 - Implémentation (codage ou programmation)
 - Traduction et réalisation des données et des algorithmes dans le langage de programmation cible
 - » Langage cible pour nous → Langage C
 - Test
 - Vérification du bon fonctionnement du programme

EFREI L3 – LSI – 2019/2020

4

Rappels de programmation

Structure d'un programme informatique

- Un programme est une suite d'instructions
- L'exécution est dite « **séquentielle** »
 - Le programme est exécuté pas à pas, les instructions sont donc exécutées les unes après les autres
 - Le programme a un début (point d'entrée) et une fin
 - Des branchements (ou saut) sont possibles
- Dépend du langage utilisé
- Séquentielle vers parallèle : outils de parallélisation en C
 - OpenMP : mémoire partagée
 - » plusieurs coeurs sur une même machine
 - MPI : mémoire distribuée
 - » plusieurs machines connectées par un réseau

EFREI L3 – LSI – 2019/2020

5

Rappels de programmation

Différents langages de programmation

- L'ordinateur ne comprends que le **langage machine** {0,1}
 - 011101000100011 → Besoin de langages simplifiés qui seront ensuite traduits en langage machine (ou binaire)
- Langage de **bas niveau** (spécifique à chaque machine)
 - Langage machine, assembleur
- Langage de **haut niveau** (indépendant du matériel)
 - Interprété
 - » Traduction en langage machine pendant l'exécution (Basic, PERL...)
 - Compilé
 - » Traduction avant l'exécution (C / C++...)
- Langage interprété vs langage compilé
 - Programme compilé moins exigeant en terme de ressources
 - Programme compilé plus rapide que le même programme interprété
 - Intérêt du langage interprété pour la facilité de mise en œuvre et la portabilité des programmes

EFREI L3 – LSI – 2019/2020

6

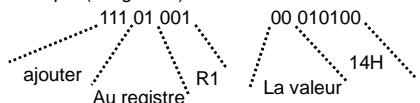
Rappels de programmation

Langage machine

Codage par champs des instructions

Code opération	Mode d'adressage	opérande
----------------	------------------	----------

- Opération: charger un registre, additionner, sauter à, ...
- Mode d'adressage : immédiat, mémoire, indirect, ...
- Opérandes : registre interne, emplacement mémoire, ...
- Exemple (imaginaire) d'une instruction sur 2 octets



EFREI L3 – LSI – 2019/2020

7

Rappels de programmation

Langage assembleur

Traduit un langage symbolique en langage machine

- Connaît les codes des opérations, les modes d'adressage, la longueur de chaque instruction
- Peut calculer les adresses de chaque donnée et de chaque instruction

Syntaxe typique

- Une opérande
 - » [étiquette] <codeop> <mode> <opérande>
- Deux opérandes
 - » [étiquette] <codeop><mode dest><destination>, <mode source> <source>
- L'étiquette (label) correspond à l'adresse courante

EFREI L3 – LSI – 2019/2020

8

Rappels de programmation

Langage assembleur

- Quelques exemples d'opérations effectuées par la machine de Von Neumann
 - ADD R1, #34h $R1 \leftarrow R1 + 34h$
 - MUL R1, @1234h $R1 \leftarrow R1 * \text{contenu_m\u00e9moire}[1234h]$
 - MOV R1, @1234h $R1 \leftarrow \text{contenu_m\u00e9moire}[1234h]$
 - MOV R2, R1 $R2 \leftarrow R1$
 - JMP @1430h $PC \leftarrow 1430h$
 - JSUP label, R1, #23h si $R1 > 23h$ $PC \leftarrow \text{adresse label}$
sinon $PC \leftarrow \text{adresse inst. suivante}$
 - CALL @1234h Appel sous programme: sauvegarde de
PC et du registre d'état, puis $PC \leftarrow 1234h$
 - RET Sortie du sous programme : restauration
du registre d'état et de PC



Rappels de programmation

- **Passage du langage haut niveau au langage machine**
 - Qu'est que le **code source** ?
 - Le code de votre programme écrit dans un langage de haut niveau
 - » C'est donc ce code source que vous écrivez et qui sera traduit en langage binaire
 - Comment le traduire ?
 - A l'aide d'un **programme** nommé **compilateur**
 - L'opération de traduction se nomme la **compilation**
 - » Il existe un compilateur différent pour chaque langage
 - Qu'obtient-on après la compilation ?
 - Le compilateur crée un programme binaire nommé l'**exécutable** (d'où une extension .exe sous windows)
 - » L'exécutable n'est pas généré s'il y a une erreur dans le code source

Rappels de programmation C

- Nous utiliserons le Langage C dans ce module
- Rappels de programmation C
 - Historique du langage C
 - Structure d'un programme C, type booléen et représentation des caractères
 - Opération de transtypage (cast)
 - Overflow et erreur de segmentation
 - Directives du préprocesseur & Compilation séparée

EFREI L3 – LSI – 2019/2020

11



Rappels de programmation C

Historique du langage C

- Créer en 1972 par Denis Ritchie
- Objectif : écrire un système d'exploitation (UNIX)
- Première définition rigoureuse du langage en 1978 par Kernighan et Ritchie
 - Ouvrage : *The C Programming Language*
- Différentes normalisations
 - C90 (aussi appeler « C ANSI » ou « C norme ANSI »)
 - C99 (extensions publiée en 1999)
 - C11 (extensions publiée en 2011)

EFREI L3 – LSI – 2019/2020

Rappels de programmation C

● Historique du langage C

– Différentes normalisations

- Prudence concernant les extensions C99 et C11 ! Elles sont loin d'être implémentées entièrement dans tous les compilateurs...
- Quelques exemples d'apports de la norme C99
 - Emplacement des déclarations ne doivent plus figurer au début d'un bloc, i.e. avant les 1ères instructions exécutables
 - » Il devient possible de déclarer une variable utilisée comme compteur de boucle au sein même de la boucle
 - Autorise les commentaires de fin de ligne (« // »)
 - Préconise l'utilisation de la norme IEE754 (ou ISO/IEC 60559) mais sans l'imposer
 - » Respect de la norme → « __STDC_IEE_559__ » est définie

EFREI L3 – LSI – 2019/2020

13

Rappels de programmation C

● Structure d'un programme C

– C'est une collection de fonctions et de variables

- L'une des fonctions doit s'appeler **main** (fonction principale)
 - » C'est le **point d'entrée** du programme
 - » Permet de repérer le début du programme et la fin de celui-ci (dans le cas où l'exécution n'a pas été terminée prématurément)
 - » Ne peut pas être appelée par le programme lui-même
- L'exécution du programme correspond à la création des **variables « globales »** puis à l'exécution de la fonction **main**
 - » Il est possible d'appeler (exécuter) d'autres fonctions, qui peuvent elles même faire appel à d'autres fonctions, à partir de la fonction **main**
 - » Certaines variables seront créées pendant l'exécution du programme (**variables « locales »**)

EFREI L3 – LSI – 2019/2020

14

Rappels de programmation C

● Type booléen en C

– N'existe pas

- Le type utilisé est un **type entier** (en général int parce que les expressions à valeur booléenne produisent une valeur de ce type)
 - » Valeur 0 correspond à FAUX
 - » Valeur différent de 0 correspond à VRAI
- Le fait qu'un booléen soit représenté par un type entier ne doit pas lui faire perdre son caractère booléen dans la logique d'écriture du programme
- Rappel des opérateurs logiques
 - et « && », ou « || », non « ! »
 - » Le « & » et le « | » sont des opérateurs binaires

EFREI L3 – LSI – 2019/2020

15

Rappels de programmation C

● Représentation des caractères

– Les caractères sont stockés comme des nombres

- additions, comparaisons, incrémentations, etc

– Pour que ces calculs aient un sens, il est nécessaire d'avoir en tête certains points relatifs aux codes ASCII :

- Les lettres majuscules de 'A' à 'Z' se suivent dans l'ordre de l'alphabet
 - » le code ASCII du caractère 'A' est 65, celui de 'B' est 66, ..., 'Z' 90
- Les lettres minuscules de 'a' à 'z' se suivent également dans l'ordre de l'alphabet
 - » le code ASCII du caractère 'a' est 97, ..., celui de 'z' est 122
- Les chiffres de '0' à '9' se suivent dans l'ordre de leur valeur
 - » Le code ASCII du caractère '0' est 48
- Les caractères accentués ne sont pas placés dans l'ordre alphabétique (utilisent des valeurs parmi les 128 restantes)

EFREI L3 – LSI – 2019/2020

16

Rappels de programmation C

● Transtypage (cast)

– Évaluation de l'expression $v1 \text{ op } v2$

- op est un opérateur, $v1$ et $v2$ sont d'un type arithmétique
- **V1 et V2 seront transtypés vers un type commun**
 - Les entiers de taille inférieur à *int* sont amenés au type *int*
 - Détermination du type commun
 - » S'ils sont rationnels, le type commun est le plus précis des deux types rationnels
 - » Si l'un est rationnel et l'autre entier, le type commun est le type rationnel
 - » S'ils sont entiers, le type commun est le plus « grand » des deux types entiers

Double op float → *double*

Float op int → *float*

EFREI L3 – LSI – 2019/2020

17

Rappels de programmation C

● Opération de transtypage (cast)

– On peut changer le type d'une **expression** par l'opérateur de cast « () »

(nouveau_type) variable

(nouveau_type) (expression)

- Après évaluation de *expression* (*ou de variable*), le résultat est convertit en le type *nouveau_type*
 - » On ne change donc pas le type de la variable mais le type de la valeur de la variable
 - » Rappel : le langage C est fortement typé !
 - » *! /! Cette opération de transtypage n'a pas toujours un sens /!*

EFREI L3 – LSI – 2019/2020

18

Rappels de programmation C

● Exemple de transtypage (cast)

```
float f;  
int i1 = 1;  
int i2 = 2;  
  
f = i1 / i2;           /* f = 0.0 */  
f = (float)(i1 / i2); /* f = 0.0 */  
f = (int)( (float)i1 / (float)i2 ) /* f = 0.0 */  
f = (float)i1 / (float)i2; /* f = 0.5 */  
f = (float)i1 / i2;   /* f = 0.5 */
```

EFREI L3 – LSI – 2019/2020

19

Rappels de programmation C

● Flux d'entrée / sortie

– Permet (notamment) au programme d'échanger des informations avec l'utilisateur

- L'interaction se fait à travers le clavier et l'écran (terminal)
 - » i.e., respectivement l'entrée standard (stdin) et la sortie standard (stdout)

– Toutes les opérations d'entrée / sortie se font à l'aide de fonctions codées dans la bibliothèque « stdio.h »

- Abréviation de standard input output
- Bibliothèque à inclure au début du programme « #include <stdio.h> »
- **On ne lit et écrit que des caractères**
 - Conversion du contenu d'une variable en caractère(s) lors de l'affichage ou de la saisie
 - » Ces règles de conversion dépendent du type de la variable

EFREI L3 – LSI – 2019/2020

20

Rappels de programmation C

- **Flux d'entrée / sortie**
- Règles de conversion
 - Spécifiées par l'utilisateur à travers des formats

%[taille][.prec]<lettre>

 - » [taille] → nombre minimal de caractères à afficher (facultatif)
 - » [prec] → nombre de caractères à afficher après la virgule (facultatif)

Type	Lettre	Type	Lettre
int	%d	string (char*)	%s
long	%ld	pointeur (void*)	%p
float/double	%f / %lf	short	%hd
char	%c	entier hexadécimal	%x

EFREI L3 – LSI – 2019/2020 21

Rappels de programmation C

- **Flux d'entrée / sortie**
- Ecriture à l'écran
 - Prototype de la fonction « printf »


```
int printf( " texte1 <format1> ... textN <formatN> ", variable1,..., variableN);
```

 - La valeur de retour est
 - » Si positive : le nombre de variables écrites avec succès
 - » Si négative : un code erreur
 - Exemple d'utilisation de la fonction *printf*

```
int a = 3 ;
printf( " La valeur de a vaut %d \n ", a );
```

Sur la console La valeur de a vaut 3

EFREI L3 – LSI – 2019/2020 22

Rappels de programmation C

- **Flux d'entrée / sortie**
- Ecriture à l'écran
 - Exemple d'utilisation de la fonction *printf*

```
float val = 103.85689;
printf("valeur arrondie a 2 chiffres après la virgule : %.2f \n",val);
printf(" valeur arrondie a 2 chiffres après la virgule : %8.2f \n",val);
val = 3.854;
printf(" valeur arrondie a 2 chiffres après la virgule : %8.2f \n",val);
```

Sur la console valeur arrondie a 2 chiffres après la virgule : 103.86
valeur arrondie a 2 chiffres après la virgule : 103.86
valeur arrondie a 2 chiffres après la virgule : 3.85

EFREI L3 – LSI – 2019/2020 23

Rappels de programmation C

- **Flux d'entrée / sortie**
- Ecriture à l'écran
 - La fonction *printf* n'envoie pas la chaîne interprétée directement à l'écran mais dans un buffer (de sortie)
 - » Pour forcer l'affichage à l'écran il faut mettre un retour à la ligne
 - Exemple


```
printf( " Blabla1" );
printf( " Blabla2" );
printf( " Blabla3" );
printf( " Blabla4\n" );
printf( " Blabla5\n" );
```

C'est ce *printf* qui provoque l'affichage de tous les précédents à l'écran

Sur la console Blabla1Blabla2Blabla3Blabla4
Blabla5

EFREI L3 – LSI – 2019/2020 24

Rappels de programmation C

- Flux d'entrée / sortie
 - Problème lors d'une saisie de caractère

```
#include <stdio.h>
int main(void)
{
    int var=0;
    char carac = 'o';
    do
    {
        printf("Tapez une valeur <100 : ");
        scanf("%d",&var);
    }while(var<100);

    do
    {
        printf("Voulez continuer y/o ?");
        scanf("%c",&carac);
    }while(carac=='y');

    return 0;
}
```

Tapez une valeur <100 : 10
Tapez une valeur <100 : 120
Voulez continuer y/o ?
D'où vient le problème ?

EFREI L3 – LSI – 2019/2020 25

Rappels de programmation C

- Flux d'entrée / sortie
 - Problème lors d'une saisie de caractère

```
#include <stdio.h>
int main(void)
{
    int var=0;
    char carac = 'o';
    do
    {
        printf("Tapez une valeur <100 : ");
        scanf("%d",&var);
    }while(var<100);

    do
    {
        printf("Voulez continuer y/o ?");
        scanf("%c",&carac);
        printf("*%c*",carac);
    }while(carac=='y');

    return 0;
}
```

Tapez une valeur <100 : 10
Tapez une valeur <100 : 120
Voulez continuer y/o ??*
**user@lubuntu:~/Bureau/SE_ASYRIA/cours\$

EFREI L3 – LSI – 2019/2020 26

Rappels de programmation C

- Flux d'entrée / sortie
 - Problème lors d'une saisie de caractère

```
#include <stdio.h>
int main(void)
{
    int var=0;
    char carac = 'o';
    do
    {
        printf("Tapez une valeur <100 : ");
        scanf("%d",&var);
    }while(var<100);

    do
    {
        printf("Voulez continuer y/o ?");
        scanf(" %c",&carac);
    }while(carac=='y');

    return 0;
}
```

Tapez une valeur <100 : 10
Tapez une valeur <100 : 120
Voulez continuer y/o ?y
Voulez continuer y/o ?y
Voulez continuer y/o ?o
user@lubuntu:~/Bureau/SE_ASYRIA/cours\$

Insertion d'un espace = plus de soucis !
Sinon il est possible d'utilisation la fonction fflush(.)

EFREI L3 – LSI – 2019/2020 27

Rappels de programmation C

- Boucle infinie et affichage sans retour à la ligne

– Si « test » est vrai ?

```
#include <stdio.h>
int main(void)
{
    int test= ??? ;
    printf("Hello world !\n");
    user@lubuntu:~/Bureau$
```

– Si « test » est faux ?

```
user@lubuntu:~/Bureau$ ./test
Hello world !
Avant la boucle
Apres la boucle
user@lubuntu:~/Bureau$
```

EFREI L3 – LSI – 2019/2020 28

Rappels de programmation C

■ Erreur de segmentation (segmentation fault)

- Tentative d'accès par l'application à une zone mémoire qui ne lui est pas allouée

- sûreté de fonctionnement garantie par le système d'exploitation
- Se produit uniquement durant l'exécution
- Exemple :

```
int tab[5] = {1, 2, 3, 4, 5} ;  
int i = 1;  
...  
printf("%d \n", tab[i-1]);  
printf("%d \n", tab[i+4]);  
printf("%d \n", tab[i-2]);
```

EFREI L3 – LSI – 2019/2020

29

Rappels de programmation C

■ Quelques règles d'écriture des programmes C

- Ne jamais placer plusieurs instructions sur une même ligne
- Utiliser des identificateurs significatifs
- Faire ressortir la structure syntaxique du programme grâce à l'indentation
 - Une accolade fermante est seule sur une ligne et est alignée avec l'accolade ouvrante du bloc correspondant → *Style Allman*
- Insérer des lignes blanches pour séparer des ensembles d'instructions
- Il est nécessaire de commenter le code (en évitant les commentaires triviaux)
 - On doit pouvoir comprendre le fonctionnement de votre programme à partir des commentaires

EFREI L3 – LSI – 2019/2020

30

Rappels de programmation C

■ Quelques règles d'écriture des programmes C

- Exemple de styles d'indentation

Style K&R

```
void a_function(void)  
{  
    if (x == y) {  
        something1();  
        something2();  
    } else {  
        somethingelse1();  
        somethingelse2();  
    } finalthing();  
}
```

Style Allman

```
void a_function(void)  
{  
    if (x == y)  
    {  
        something1();  
        something2();  
    }  
    else  
    {  
        somethingelse1();  
        somethingelse2();  
    }  
    finalthing();  
}
```

Style GNU

Rappels de programmation C

■ Préprocesseur

- Prétraitement du programme source avant la compilation proprement dite du fichier

- Visualisation du résultat au moyen de l'option « - E » sur la sortie standard
- Réécriture du programme pilotée par des instructions spéciales appelées *directives*
- Offre plusieurs fonctionnalités dont :
 - » Inclusion de fichier
 - » Définition de macroconstante / macrofonction
 - » Message d'erreur et avertissement
 - » Compilation conditionnelle
- C'est durant cette phase que les commentaires sont retirés du source du programme

EFREI L3 – LSI – 2019/2020

32

Rappels de programmation C



- **Syntaxe et traitement des directives**
 - Une **directive** est une ligne commençant par un « # » et suivie d'un mot-clé
 - Le caractère « \ » permet d'écrire la chaîne sur plusieurs lignes (à mettre à la fin de chaque ligne)
 - » Ex: include, define
 - » Directive nulle (ignorée) : # « ligne vide »
 - Traduction des directives précède la phase de compilation
 - Génère des portions de code C traitées ensuite par le compilateur

EFREI L3 – LSI – 2019/2020 33

Rappels de programmation C



- **Inclusion de fichier**
 - Permet d'importer dans un fichier source le contenu d'un autre fichier
 - Piloté par la directive « **include** »
 - Spécification du fichier à inclure correct → le fichier correspondant est inséré à la place de la directive
 - » Répertoire d'en-têtes standard " /usr/include/fichier "

```
#include <fichier>
#include " fichier "
```

EFREI L3 – LSI – 2019/2020 34

Rappels de programmation C



- **Inclusion de fichier**
 - Possible d'inclure n'importe quel fichier source
 - Limiter cette fonctionnalité à des fichiers contenant seulement des directives et des déclarations (types, prototypes de fonction...)
 - Ne pas hésiter à découper un programme en de nombreux fichiers
 - Reflète l'organisation logique du programme
 - Faciliter la définition de modules indépendants et réutilisables
 - Permettre de factoriser une portion de code commune à plusieurs fichiers (attention aux inclusions multiples !)
 - Un fichier « *nom.h* » est inclus dans le fichier « *nom.c* » et dans tout fichier utilisant une des fonctions du fichier « *nom.c* »

EFREI L3 – LSI – 2019/2020 35

Rappels de programmation C



- **Macroconstantes**
 - Une macroconstante est un symbole défini avec la directive « **define** »
 - La valeur est une suite de caractères
 - Composé de lettre et de chiffre (doit commencer par une lettre)
 - L'identificateur et la chaîne de substitution doivent être séparés par au moins un séparateur (espace ou tabulation)
 - Le préprocesseur recherche dans le source les occurrences de macroconstante et les remplace par leur valeur
 - Possible d'annuler une définition au moyen de la directive « **undef** »

EFREI L3 – LSI – 2019/2020 36

Rappels de programmation C

- Macroconstantes
 - Exemple :

```
#define TAILLE_TAB 100

int tab[TAILLE_TAB];           préprocesseur → int tab[100];

#define TAILLE_TAB
```

EFREI L3 – LSI – 2019/2020 37

Rappels de programmation C

- Macroconstantes
 - Exemple :

```
#define entier int

entier a, b;           préprocesseur → int a,b;
```

EFREI L3 – LSI – 2019/2020 38

Rappels de programmation C

- Macroconstantes
 - Le préprocesseur ignore la syntaxe du langage C lorsqu'il réalise les substitutions
 - Quelques erreurs classiques:

```
#define max 100          préprocesseur → int max; → int 100;
#define TMAX 100;         int tab[TMAX]; → int tab[100];
#define TMAX = 100         int tab[TMAX]; → int tab[=100];
```

EFREI L3 – LSI – 2019/2020 39

Rappels de programmation C

- Macrofonctions
 - Mécanisme de substitution de la directive « define » est paramétrable
 - Une définition paramétrée est appelée une **macrofonction**
 - Le corps de la macrofonction est une suite quelconque de caractères
 - Substitution → toute occurrence d'un paramètre dans le corps est remplacé par le paramètre correspondant
 - Les paramètres sont entre parenthèse et une virgule permet de séparer plusieurs paramètres
 - » Pas de séparateur entre le symbole et la parenthèse ouvrante !

EFREI L3 – LSI – 2019/2020 40

Rappels de programmation C

Macrofonctions

- La valeur absolue et le carré

```
#define ABS(x) x > 0 ? x : -x
```

```
#define CARRE(x) x*x
```

- La somme et la différence et multiplication

```
#define SOM(x,y) x+y
```

```
#define DIF(x,y) x-y
```

```
#define MULT(x,y) x*y
```

Rappels de programmation C

Macrofonctions

- Il est possible de faire des imbrications dans le code

```
MULT( DIF(a,b) , SOM(a,b) );
```

préprocesseur



```
MULT( a - b , a + b );
```



```
a - b * a + b ;
```

Cela ne correspond pas à ce qu'on voulait !

Rappels de programmation C

Macrofonctions

- Attention aux erreurs !

- Toute occurrence d'un paramètre dans le corps de la définition doit être placée entre parenthèse

préprocesseur

```
MULT(a-b,a+b) → a-b*a+b;
```

```
#define MULT(x,y) (x)*(y)
```

préprocesseur

```
MULT(a-b,a+b) → (a-b)*(a+b);
```

Rappels de programmation C

Macrofonctions

- Les macros précédentes bien écrites

```
#define ABS(x) (x) > 0 ? \  
           (x) : -(x)
```

```
#define CARRE(x) (x)*(x)
```

```
#define SOM(x,y) (x) + (y)
```

```
#define DIF(x,y) (x) - (y)
```

```
#define MULT(x,y) (x)*(y)
```

Rappels de programmation C

- Macrofonctions**
 - Attention à ne pas mettre d'espace dans le nom de la macro !

```
#define MULT (x,y) (x)*(y)
          ↑
          espace
```

préprocesseur

MULT(a-b, a+b); → (x,y) (x)*(y)(a-b, a+b);

EFREI L3 – LSI – 2019/2020 45

Rappels de programmation C

- Message d'erreur et avertissement**
 - Emission d'un message d'erreur à la compilation
 - Directive « **error** »
 - Affiche un message en arrêtant la compilation
 - Utiliser lorsqu'une condition fait que le programme ne peut pas s'exécuter sur la plate-forme

```
#error " message d'erreur "
```

- Emission d'un avertissement à la compilation (GCC)
 - Directive « **warning** »
 - Affiche un message sans arrêter la compilation

```
#warning " message d'avertissement "
```

EFREI L3 – LSI – 2019/2020 46

Rappels de programmation C

- Compilation conditionnelle**
 - Directives classées en 2 catégories
 - Existence (ou inexistence) de symboles
 - Ifdef , ifndef
 - Valeur d'une expression
 - If, else, elif, endif
 - En fonction du résultat, le préprocesseur conserve ou écarte certaine(s) ligne(s) de code

EFREI L3 – LSI – 2019/2020 47

Rappels de programmation C

- Compilation conditionnelle**
 - Valeur d'une expression

<code>#define BLOC 1</code>	<code>#define BLOC 1</code>
<code>#if BLOC == 1</code>	<code>#if BLOC == 1</code>
<code>... //bloc d'instructions 1</code>	<code>... //bloc d'instructions 1</code>
<code>#else</code>	<code>#elif BLOC == 2</code>
<code>... //bloc d'instructions</code>	<code>... //bloc d'instructions 2</code>
<code>#endif</code>	<code>#else</code>
	<code>... //bloc d'instructions</code>
	<code>#endif</code>

EFREI L3 – LSI – 2019/2020 48

Rappels de programmation C

Compilation conditionnelle

Existence d'une macroconstante

- Testée au moyen de la directive « `#ifdef` »

```
#define MACRO
```

...

`#ifdef MACRO` —> Fin de ligne

... //si MACRO existe

`#endif` —> Fin de ligne

- Équivalence avec « `defined` »

```
#ifdef MACRO → #if defined(MACRO)
```

EFREI L3 – LSI – 2019/2020

49

Rappels de programmation C

Compilation conditionnelle

La non-existence d'une macroconstante

- Testée au moyen de la directive « `#ifndef` »
» syntaxe identique à « `#ifdef` »

```
#ifndef MACRO
```

... //si MACRO n'existe pas

`#endif`

EFREI L3 – LSI – 2019/2020

50

Rappels de programmation C

Compilation conditionnelle

Une application importante est de prévenir des inclusions multiples d'une même en-tête

- Associer une macroconstante à chaque nom d'en-tête
- Teste la définition au début du fichier d'en-tête
» Règle : fichier « `nom.h` » → macroconstante « `NOM_H` »

```
#ifndef NOM_H
#define NOM_H
    ... /*contenu de nom.h*/
#endif /*NOM_H*/
```

EFREI L3 – LSI – 2019/2020

51

Rappels de programmation C

Compilation conditionnelle

Utile pour introduire dans le source des instructions de mise au point

- Modification mineur du source pour contrôler la présence ou non d'instructions

- Exemple : construction destinées à tracer son exécution lors de la mise au point du programme
» La recopie dans le code de l'instruction est conditionné par l'existence de la macroconstante

```
#ifdef DEBUG
    printf(" Erreur l.%d du fichier %s \n ", __LINE__ , __FILE__);
#endif /*DEBUG*/
```

EFREI L3 – LSI – 2019/2020

52

Rappels de programmation C

- **Compilation conditionnelle**
 - Utile pour adapter un programme à différents environnements
 - Possible de définir des macros pour l'environnement

```
#define LINUX

#endif LINUX
... //bloc d'instructions pour Linux
#endif

#ifndef WINDOWS
... //bloc d'instructions pour Windows
#endif
```

EFREI L3 – LSI – 2019/2020 53

Rappels de programmation C

- **Compilation conditionnelle**
 - Utile pour adapter un programme à différents environnements
 - Déclaration automatique de certaines macros
 - » dépendant du compilateur !

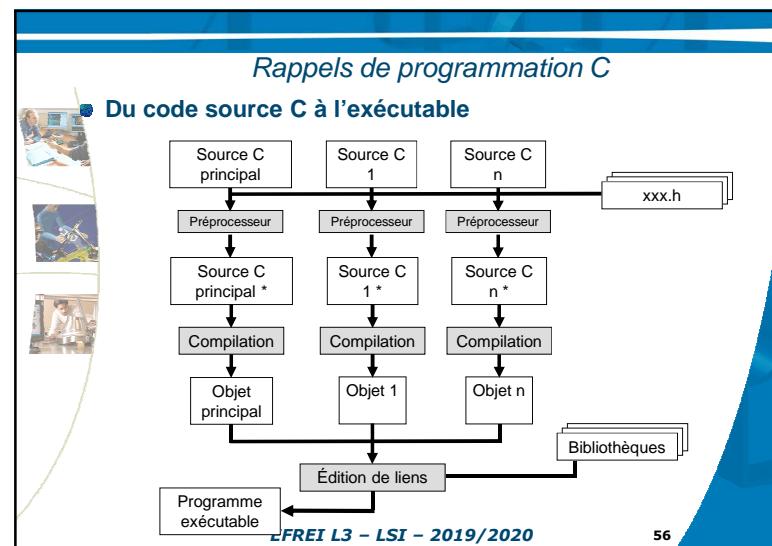
```
#if defined(linux) || defined(__linux__)
... //bloc d'instructions pour Linux
#elif defined(_WIN32) || defined(__WIN32__)
... //bloc d'instructions pour Windows
#else
... #error "Environnement incompatible..."
#endif
```

EFREI L3 – LSI – 2019/2020 54

Rappels de programmation C

- **Compilation séparée**
 - Découper un programme en plusieurs fichiers
 - Le compilateur traite chaque fichier source séparément
 - » Par conséquent on ne doit pas inclure de fichier source dans un autre sous peine de pouvoir avoir des erreurs
 - La solution consiste à ajouter un **fichier en-tête** (.h)
 - Portion de code commune entre les fichiers sources
 - Le compilateur relie ainsi l'utilisation de la fonction dans un fichier source à sa définition dans un autre fichier source
 - Seul manière pour avoir un contrôle syntaxique des appels de fonctions

EFREI L3 – LSI – 2019/2020 55



Rappels de programmation C

Compilation séparée

- Découper un programme en plusieurs fichiers

<i>main.c</i>	<i>afficher.h</i>	<i>afficher.c</i>
<pre>#include <stdio.h> #include <stdio.h> #include "afficher.h" int main() { afficher(); return 0; }</pre>	<pre>#ifndef AFFICHER_H #define AFFICHER_H #include <stdio.h> void afficher(void) { printf("fct afficher\n"); return; } #endif /* AFFICHER_H */</pre>	<pre>#include <stdio.h> #include "afficher.h" void afficher(void) { printf("fct afficher\n"); return; }</pre>
<pre>gcc -Wall main.c afficher.c -o main</pre>		

EFREI L3 – LSI – 2019/2020 57

Rappels de programmation C

Compiler et exécuter son programme

- Ouvrez une console (ou terminal) et placez-vous dans le répertoire de votre projet contenant votre fichier source

```
[utilisateur@ordi ~]$ gcc source.c
```

- Vous allez obtenir dans votre répertoire un exécutable portant le nom « a.out » (nom par défaut)
- Exécution du programme (lancement de l'exécutable)

```
[utilisateur@ordi ~]$ ./a.out
```

- Il est possible de définir un nom pour l'exécutable généré via l'option « -o »

```
[utilisateur@ordi ~]$ gcc -Wall source1.c source2.c -o mon_programme
```

```
[utilisateur@ordi ~]$ ./mon_programme
```

EFREI L3 – LSI – 2019/2020 58

Rappels de programmation C

Compiler et exécuter son programme

- Il est possible de faire la compilation en 2 étapes

- Génération des « .o » (compilation)
- Création de l'exécutable en liant tous les « .o » (édition de lien)

```
[utilisateur@ordi ~]$ gcc -c -Wall source1.c source2.c
```

```
[utilisateur@ordi ~]$ gcc source1.o source2.o -o mon_programme
```

```
[utilisateur@ordi ~]$ ./mon_programme
```

EFREI L3 – LSI – 2019/2020 59

Rappels de programmation C

Compilation séparée

- Ne pas hésiter à découper un programme en plusieurs fichiers

- Doit refléter l'organisation logique du programme
- Permet de faciliter la définition de modules indépendants et réutilisables
- On ne recompile que les fichiers sources qui ont été modifiés
 - On réutilise les « .o » déjà générés pour les autres

➔ favorise une **bonne structuration** de l'ensemble du projet et **accélère** le processus de compilation

EFREI L3 – LSI – 2019/2020 60

Rappels de programmation C

Le pointeur

- Une variable est représentée symboliquement par un identificateur (son nom)
 - Une variable dont le type de la valeur est une adresse de variable est appelée **pointeur**
 - Déclaration et initialisation d'un pointeur



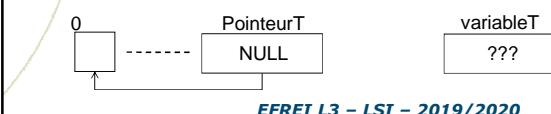
EFREI L3 – LSI – 2019/2020

61

Rappels de programmation C

Le pointeur : exemple d'utilisation

`T * PointeurT = NULL;` → création d'un pointeur sur un objet de type T
`T variableT;` → création d'une variable de type T



EFREI L3 – LSI – 2019/2020

62

Rappels de programmation C

Le pointeur : exemple d'utilisation

- `T * PointeurT = NULL;` → création d'un pointeur sur un objet de type T
`T variableT;` → création d'une variable de type T
`PointeurT = &variableT;` → affectation au pointeur de l'adresse de la variable de type T (adressage **directe**)



EFREI L3 – LSI – 2019/2020

63

Rappels de programmation C

Le pointeur : exemple d'utilisation

`T * PointeurT = NULL;` → création d'un pointeur sur un objet de type T
`T variableT;` → création d'une variable de type T
`PointeurT = &variableT;` → affectation au pointeur de l'adresse de la variable de type T (adressage **directe**)
`variableT = 0;` → modification de la valeur de variableT (adressage **directe**)



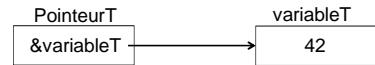
EFREI L3 – LSI – 2019/2020

64

Rappels de programmation C

Le pointeur : exemple d'utilisation

T * PointeurT = NULL; → création d'un pointeur sur un objet de type T
T variableT ; → création d'une variable de type T
PointeurT = &variableT; → affectation au pointeur de l'adresse de la variable de type T (adressage **directe**)
variableT = 0; → modification de la valeur de variableT (adressage **directe**)
*pointeurT = 42; → modification de la valeur de variableT (adressage **indirecte**)



EFREI L3 – LSI – 2019/2020

65

Rappels de programmation C

Pointeur

- Comment modifier ou accéder à la valeur d'une variable pointée ?
 - On utilise l'opérateur « * »
- int x = 2, resultat ;
- int * ptrx = &x ; → * signifie ici que c'est un pointeur sur un int !
- resultat = *ptrx + *ptrx ; → * signifie ici qu'on accède à la valeur pointée par x soit 2 (resultat = 4)
- *ptrx = *ptrx + 1 ; → *ptrx = 3, x = 3

EFREI L3 – LSI – 2019/2020

66

Rappels de programmation C

Pointeur sur void

- Type spéciale représentant une adresse de variable de type quelconque (pointeur « universel »)
 - Attention il est impossible d'accéder à la variable pointé et il nécessaire de préciser le type de la variable pointé par transtypage du pointeur

```
T * PointeurT;  
void * PointeurVoid ;  
...  
PointeurT = (T *) PointeurVoid;
```

EFREI L3 – LSI – 2019/2020

67

Rappels de programmation C

Allocation dynamique

- Permet la création d'objets pendant l'exécution à la demande explicite du programme
 - La taille n'a pas besoin d'être connue à la compilation
 - « void* malloc (dimension en octet) » : renvoie l'adresse du début du bloc mémoire alloué ou NULL si la mémoire ne peut pas être allouée
- Les objets sont accessibles par leur adresse
- Les objets doivent être détruits explicitement par le programme
 - Appel à la fonction « free(adresse) »
 - Adresse renvoyée par le malloc
- Eviter les fuites mémoires
 - Toujours autant d'appel à free() que d'appel à malloc()

EFREI L3 – LSI – 2019/2020

68



Pointeur de structure

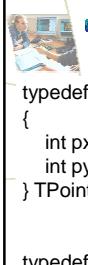
- Déclaration d'un pointeur de structure

```
typedef struct
{
    int x;
    int y;
} point_a;
point * ptr_a = &point_a;
```

- Accéder au champ d'une structure via un pointeur sur la structure
 - Deux solutions : `(*ptr_a).x` ou `ptr_a->x`

```
printf( " Les coordonnées du point sont <%d,%d> ", ptr_a->x, ptr_a->y );
```

EFREI L3 – LSI – 2019/2020 69



Rappels de programmation C

Utilisation de structures imbriquées

```
void TranslationPoint( TPoint * point , TPoint vecteur)
{
    point -> px += vecteur.px ;
    point -> py += vecteur.py ;
}

void TranslationCercle (TCercle * cercle , TPoint vecteur)
{
    TranslationPoint( &(cercle -> centre) , vecteur ) ;
}
...
TCercle cercle ;
TPoint vecteur ;
... //init
```

```
TranslationCercle( &cercle , vecteur )
```

EFREI L3 – LSI – 2019/2020 70



Rappels de programmation C

Ecrire une fonction de création d'une matrice

- Quel est le problème sur la fonction proposée ?
 - Proposer 2 solutions différentes

```
t_matrice* alloc_matrice(int lig, int col)
{
    t_matrice mat;
    int i = 0;

    mat.nb_lig = lig;
    mat.nb_col = col;

    mat.coeffs = (double**)malloc(mat.nb_lig*sizeof(double*));
    for(i=0; i<mat.nb_lig ; i++)
    {
        mat.coeffs[i] = (double*)malloc(mat.nb_col*sizeof(double));
    }

    return &mat;
}
```

EFREI L3 – LSI – 2019/2020 71



Rappels de programmation C

Passer des arguments à un programme au lancement

- Arguments fournis au programme lors de son lancement
 - Chaîne(s) de caractères séparée(s) par des espaces
- Transmission à la fonction main

```
int main (int nbarg, char * argv[ ])
```

- Le 1^{er} argument est représenté le nombre total de paramètre
 - » le nom (ou chemin) du programme compte pour un paramètre
- Le 2^{ème} argument est l'adresse d'un tableau de pointeur, chaque pointeur désigne la chaîne d'un paramètre

EFREI L3 – LSI – 2019/2020 72

Rappels de programmation C

● Arguments reçus par la fonction main

```
#include <stdio.h>

int main(int nbarg, char* argv[])
{
    int i = 0;
    printf("Commande de lancement : %s\n", argv[0]);
    if(nbarg>1)
    {
        for(i=1;i<nbarg;i++)
            printf("arg numero %d : %s\n",i,argv[i]);
    }
    else
    {
        printf("aucun argument\n");
    }
    return 0;
}
```

EFREI L3 – LSI – 2019/2020

73

Rappels de programmation C

● Les entrées / sorties (E/S)

- Regroupe les opérations de lecture / écriture et les opérations de contrôle associées
 - Transfert de données effectué entre une zone mémoire du programme et une unité périphérique (terminal , disque...)
 - Réalisé par l'OS à l'aide des fonctions d'E/S de bas niveau « read » et « write »
 - » On ne les utilisera pas directement
- Nous utiliserons les E/S de haut niveau de la bibliothèque standard (stdio.h)
 - » Interface avec les fonctions d'E/S de bas niveau

EFREI L3 – LSI – 2019/2020

74

Rappels de programmation C

● Les entrées / sorties (E/S)

- Pour unifier les mécanismes d'E/S (de haut niveau)
 - les E/S s'effectuent au moyen d'un seul type d'unité logique appelé **flot (stream)**
 - Les unités physiques sont vues à travers une abstraction unique appelée **fichier (FILE)**
- Utilise un tampon (buffer)
 - Mémoire où sont stockés les caractères transitant par le flot
 - Un buffer par flot

EFREI L3 – LSI – 2019/2020

75

Rappels de programmation C

● Les entrées / sorties (E/S)

- L'initialisation d'un flot est le résultat de l'ouverture d'un fichier
 - « Lecture seule », « écriture seule » ou « lecture / écriture »
 - » Conditionne les opérations possibles sur le flot
- Flot
 - Buffer
 - Paire d'indicateur : **fin de fichier** et **erreur**
 - **Position courante** : position de l'octet à partir duquel sera effectuée la prochaine opération de lecture ou d'écriture
 - » Chaque opération de lecture ou d'écriture met à jour la position courante du flot
 - » Exemple: si un fichier contient le mot « hello », après la lecture du mot dans le fichier le curseur se trouvera en position 5.

EFREI L3 – LSI – 2019/2020

76

Rappels de programmation C

● Les entrées / sorties (E/S)

- Certains flots sont prédefinis au démarrage d'un programme
 - stdin (en lecture), stdout (en écriture)
 - » Utilisable implicitement au moyen de scanf et printf
- Les fonctions de manipulation de flots se trouvent dans la bibliothèque « stdio.h »
 - Ces fonctions sont dites « d'entrée/sortie »
 - Un flot est codé par une structure dont l'identificateur est de type **FILE** (défini dans stdio.h)
 - » Regroupe l'ensemble des informations nécessaire à la mise en œuvre d'une entrée / sortie

EFREI L3 – LSI – 2019/2020

77

Rappels de programmation C

● Ouverture d'un flot

FILE * fopen (const char *path, const char *mode)

- La fonction renvoie un pointeur de type FILE
 - Création dynamique
 - Si un problème survient lors de l'ouverture, renvoie NULL
- Deux paramètres (chaînes de caractères) :
 - **path** : Chemin d'accès vers le fichier (absolue ou relatif)
 - » Nom du fichier à associer au nouveau flot
 - » Attention au chemin absolue, l'écriture est dépendante de l'OS
 - **mode** : Différente possibilité d'ouverture (**mode d'ouverture**)
 - » Type d'accès que l'on veut effectuer

EFREI L3 – LSI – 2019/2020

78

Rappels de programmation C

● Ouverture d'un flot

- Le mode d'ouverture du flot conditionne les opérations d'entrée / sortie qu'il sera possible d'effectuer
 - « r » → Mode lecture seule
 - » Le fichier doit exister et le curseur est placé au début du fichier
 - « r+ » → Mode lecture ET écriture
 - » Le fichier doit exister et le curseur est placé au début du fichier
 - « w » → Mode écriture
 - » S'il n'existe pas le fichier est créé
 - » s'il existe, son contenu est écrasé et le curseur est placé au début du fichier
 - « w+ » → Mode écriture ET lecture
 - » S'il n'existe pas le fichier est créé
 - » s'il existe, son contenu est écrasé et le curseur est placé au début du fichier

EFREI L3 – LSI – 2019/2020

79

Rappels de programmation C

● Ouverture d'un flot

– Les modes d'ouverture (suite) :

- « a » → Mode ajout en écriture
 - » S'il n'existe pas le fichier est créé et le curseur est placé à la fin du fichier
- « a+ » → Mode ajout en écriture ET lecture
 - » S'il n'existe pas le fichier est créé et le curseur est au début du fichier, attention l'écriture se fera TOUJOURS à la fin du fichier !
- Ajout du caractère « b » dans le mode d'ouverture
 - Indique au système d'exploitation que le fichier à exploiter est un fichier binaire
 - » Pas de sens sous un système UNIX (ou compatible POSIX), tous les fichiers sont gérés de façon homogène
 - rb, rb+, wb, wb+, ab, ab+

EFREI L3 – LSI – 2019/2020

80

Rappels de programmation C

● Fermeture d'un flot

- Un flot ouvert doit être fermé (vide le tampon et ferme la communication)
 - Même raison que pour les allocations dynamiques

```
int fclose (FILE *stream)
```

- La fonction renvoie 0 si elle réussit et le code de l'erreur dans le cas contraire
 - Voir page « man » pour les codes erreurs si besoin
- L'argument stream contient l'adresse du fichier
 - variable de type « FILE » dont l'adresse a été retournée par « fopen »

Rappels de programmation C

● Redéfinir un flot

- « freopen() » permet de redéfinir un flot déjà initialisé, en changeant le fichier qui lui est associé (redirection)
- Exemple de redirection de stdout

```
#include <stdio.h>
#define NOM_FICHIER_LOG "execution.log"
int main(void)
{
    freopen ( NOM_FICHIER_LOG, "w", stdout );
    printf(" ** DEBUT execution ** \n");
    printf("..... ..... \n");
    printf(" ** FIN execution ** \n");
    return 0;
}
```

Rappels de programmation C

● Lecture / Écriture dans un fichier formaté (texte)

- Ecriture sur stdout
 - « printf() »
- Lecture sur stdin
 - « scanf () »
- Ecriture sur un flot
 - « fprintf() » permet d'écrire dans un fichier texte

```
int fprintf ( FILE* stream, char* format, exp1, exp2...)
```
- Lecture sur un flot
 - « fscanf() » permet de lire dans un fichier texte
 - » Attention l'espace est considéré comme un séparateur !

```
int fscanf ( FILE* stream, char* format, adr_arg1,...)
```

Rappels de programmation C

● Lecture / Écriture dans un fichier

- Ecriture d'un caractère « c » sur un flot
 - int fputc(int c, FILE * stream)
- Ecriture d'une chaîne de caractère « s » sur un flot
 - int fputs(const char * s, FILE * stream)
- Lecture d'un caractère sur un flot
 - Fin du fichier si EOF est retourné
 - int fgetc(FILE * stream)
- Lecture d'une ligne sur un flot
 - « size » est le nombre de caractère à lire
 - On copie dans « s » ce qui a été lu dans « stream »
 - Renvoie NULL si la lecture rate
 - char* fgets(char * s, int size, FILE * stream)

Rappels de programmation C

Lecture / Écriture dans un fichier binaires

– « `fwrite()` » qui permet d'écrire dans un fichier binaire

```
size_t fwrite ( const void* source, size_t taille, size_t nbr, FILE * flobt)
```

- Ecrit sur *flobt*, *nbr* objets de taille *taille* placés en tableau à l'adresse *source*. Renvoie le nombre d'objets écrits.
- Il est possible d'écrire dans un fichier texte à l'aide de `fwrite` mais c'est moins facile et clair que `fprintf`

– « `fread()` » qui permet de lire dans un fichier binaire

```
size_t fread ( void* destination, size_t taille, size_t nbr, FILE * flobt)
```

- Lit sur *flobt*, *nbr* objets de taille *taille* et les copies à l'adresse *destination*. Renvoie le nombre d'objets effectivement lus.

Rappels de programmation C

Fonctions de contrôle

– Transfert et vidage du tampon associé au flot
int `fflush(FILE * stream)`

– Contrôle de terminaison

- Teste si l'indicateur de fin de fichier EOF est atteint
int `feof(FILE * stream)`

– Contrôle de positionnement

- Il est possible de gérer le positionnement des flots binaires
 - » Accéder à n'importe quel endroit du fichier en modifiant la position courante du flot

Rappels de programmation C

Contrôle de positionnement

– « `fseek()` » change la position courante dans du *flot*.

- *offset* : valeur du déplacement de position par rapport à « *whence* » (en octets)
- *whence* (origine) : SEEK_SET , SEEK_CUR , SEEK_END

```
int fseek ( FILE * stream, long offset , int whence )
```

– « `ftell()` » retourne la position courante en octets ou -1 en cas d'échec

```
long ftell ( FILE * steam )
```

– « `rewind()` » change la position courante du flot pour le début du fichier

```
void rewind ( FILE * stream )
```

Rappels de programmation C

Contrôle de positionnement

– Fonction utilisable avec des flots texte

- « `fgetpos` » permet de récupérer la position courante du flot

```
int fgetpos ( FILE * steam, fpos_t * pos )
```

- « `fsetpos` » permet de modifier le paramètre position du flot

```
int fsetpos ( FILE * stream, const fpos_t * pos )
```