

CRO: C langage and programming Roots

tanguy.risset@insa-lyon.fr

Lab CITI, INSA de Lyon

Version du April 1, 2026

Tanguy Risset

April 1, 2026

1 Introduction

2 Rapides rappel d'architectures et de compilation

3 Quelques caractéristiques du langage C

4 Quelques questions importantes en rapport avec la mémoire

5 Quelques notions pour commencer

6 Un exemple complet pour résumer

7 Quelques notions plus avancées

Table of Contents

- 1 Introduction
- 2 Rapides rappel d'architectures et de compilation
- 3 Quelques caractéristiques du langage C
- 4 Quelques questions importantes en rapport avec la mémoire
- 5 Quelques notions pour commencer
- 6 Un exemple complet pour résumer
- 7 Quelques notions plus avancées

CRO: présentation du cours

- Cours de 32h répartis en:
 - 1 CM (2h)
 - 7 TD, en fait cours-TP machine (14h)
 - 4 TPs (16 h)
 - Évaluation: examen papier classique
- Principes de “cours-TP”, chaque TD est associé à des slides que vous devez regarder avant de faire le TD.
- Compétences et connaissances du cours CRO:
 - Syntaxe du langage C
 - Notion d'outils de compilation (compilateur, Makefile)
 - Pointeurs, manipulation explicite de la mémoire
 - Algorithmique élémentaire bas niveau (pointeurs et structures)
 - Introduction à la programmation de micro-contrôleurs embarqués
- Moodle (ouvert): Programmation C

Comment *rentabiliser* son CRO

- Notions renforcées dans CRO:
 - Fondements et principes fondamentaux de l'informatique.
 - Connaître le fonctionnement des ordinateurs.
 - Concevoir et réaliser un programme.
 - Comprendre l'exécution (universelles) des programmes.
- ⇒ S'adapter rapidement aux nouvelles technologies informatiques (en les re-situant par rapport à celles que vous connaissez)
- ⇒ Vous maîtrisez la machine...
- Expérience acquise (dans la difficulté quelquefois)
 - Apprendre la rigueur dans la production
 - Comprendre l'utilité de la phase de *conception*
 - "Ce que l'on conçoit bien s'énonce clairement" cette maxime s'applique tout à fait à la programmation, en C en particulier
- On n'est pas tous égaux devant la programmation en C...

Installer les outils utiles pour CRO

- Allez lire "README-TOOL-CRO.org" sur le Moodle du cours CRO
 - Les outils utilisés dans CRO:
 - Le compilateur gcc
 - L'outil make
 - un éditeur de texte pour la programmation (emacs, vi, atom, sublimeText...), pas gedit!
 - On peut trouver ces outils sur Windows et Mac (Ubuntu subsystem ou WSL sur Windows)
- ⇒ Sinon c'est la clé bootable TC.
- On travaille en ligne de commande = Un éditeur de texte et une fenêtre de commande (pas de logiciel de développement intégré comme eclipse).
 - Éditeur possible: emacs, submimeType, visualStudioCode

Un mot sur le plagiat

- Le plagiat selon Wikipedia: *Le plagiat est une faute morale, civile, commerciale et/ou pénale consistant à copier un auteur ou créateur sans le dire, ou à fortement s'inspirer d'un modèle que l'on omet délibérément ou par négligence de désigner. Il est souvent assimilé à un vol immatériel.*
- En programmation, on utilise très souvent des programmes écrits par d'autre personne ⇒ mais on les cite!!!
- Ici, vous êtes en formation initiale:
 - On vous demande de produire du code et des rapports,
 - La production est la meilleure manière pour vous d'apprendre
 - Il est normal de vous classer sur vos capacité à produire plutôt qu'à reproduire
 - Je n'ai pas de problème avec l'utilisation de chatGPT, c'est un formidable outil pour augmenter sa productivité de programmeur, mais apprenez à écrire des programme (utilisez le pour débbugger).
 - Soyez honnête avec vous-même..

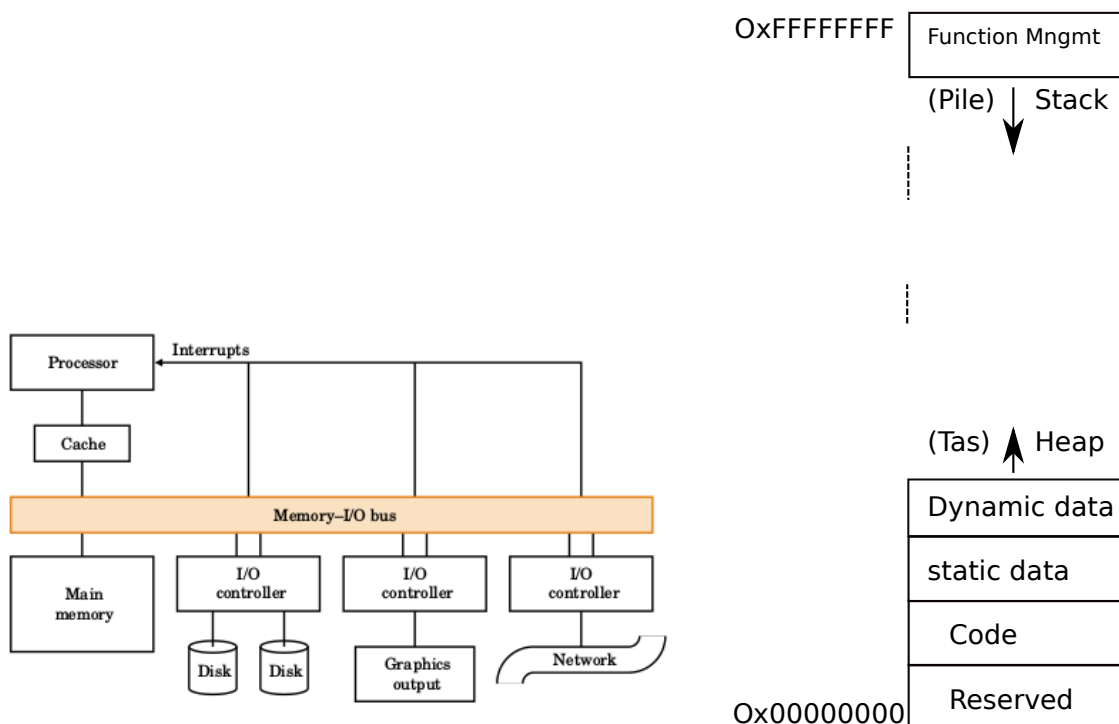
Table of Contents

- 1 Introduction
- 2 Rapides rappel d'architectures et de compilation
- 3 Quelques caractéristiques du langage C
- 4 Quelques questions importantes en rapport avec la mémoire
- 5 Quelques notions pour commencer
- 6 Un exemple complet pour résumer
- 7 Quelques notions plus avancées

Rappels sur l'architecture d'un ordinateur

- Un ordinateur de bureau est composé (au moins):
 - D'un processeur
 - D'une mémoire (dite *vive*: rapide et non rémanente)
 - D'un espace de stockage (disque dur: lent, rémanent)
 - De périphériques d'entrée/sortie (écran, claviers, etc.)
- Principe du processeur programmable:
 - Le processeur lit un programme en mémoire (programme *exécutable*, dépendant du type de processeur).
 - En fonction de ce programme
 - Il lit ou écrit des données en mémoire à une certaine *adresse mémoire* (nombre entier sur 32 bits)
 - Il effectue des calculs entre ces données

Rappels d'architecture

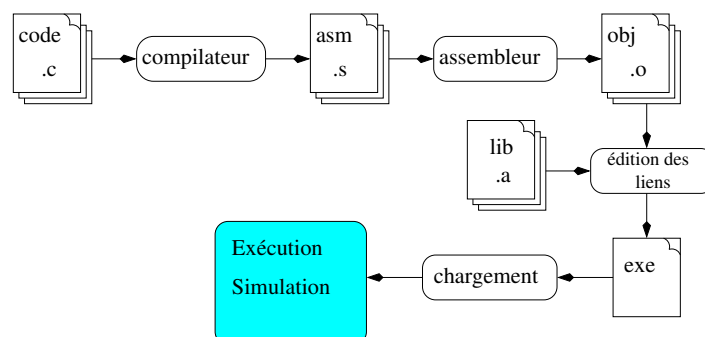


Architecture vue du programmeur

- Les systèmes modernes permettent
 - D'exécuter plusieurs programmes indépendants en parallèle (processus)
 - D'accéder à un espace mémoire plus grand que la mémoire physique disponible (mémoire virtuelle)
- Pour le programmeur: tout cela est transparent
 - Un seul programme s'exécute avec une mémoire très grande disponible
- La mémoire vue du processeur contient:
 - Le code à exécuter
 - Les données statiques (taille connue à la compilation)
 - Les données dynamiques (taille connues à l'exécution: le tas)
 - L'espace nécessaire à l'exécution du programme: la pile
- Le programmeur lui ne voit que les données (statiques et dynamiques)

Processus de compilation

- le processus complet va traduire un programme C en code exécutable (le chargement et l'exécution auront lieu plus tard).



- On nomme souvent *compilation* l'ensemble compilateur+assembleur
- Le compilateur gcc inclut aussi un assembleur et un éditeur de lien (accessibles par des options)

Votre processus de compilation

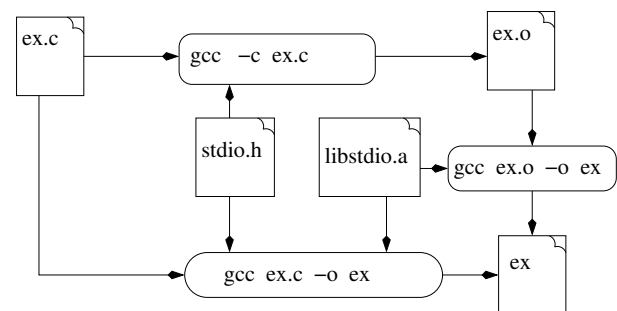
- Le programmeur:
 - Écrit le programme C: ici le dans le fichier `ex.c`
 - Compile vers un programme objet `ex.o`
 - Fait l'édition de lien pour générer l'exécutable `ex`

contenu de `ex.c`

```
#include <stdio.h>

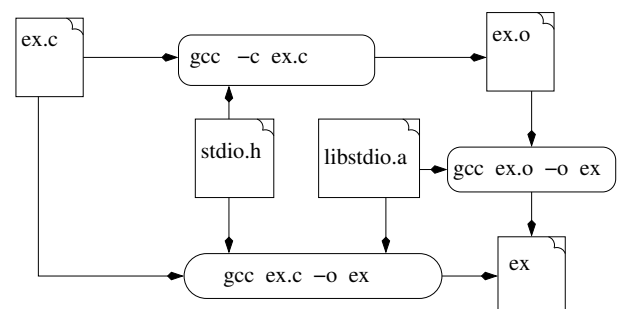
int main()
{
    printf("hello World\n");

    return 0;
}
```



Dénomination à retenir

- Le programme C que vous écrivez est le *programme source* (`ex.c`)
- Le programme source est compilé en un *programme objet* par le compilateur (`ex.o`)
- Le programme source inclut un fichier d'en-tête (`stdio.h`) pour vérifier les noms des fonctions externes utilisées (exemple: `printf`)
- le *programme exécutable* est construit par l'éditeur de liens grâce aux programmes objets et aux bibliothèques.
- Les *bibliothèques* (*library* en anglais) sont des codes assembleurs (i.e. déjà compilés) de fonctions fréquemment utilisées. Ici la bibliothèque utilisée est `libstdio.a` qui contient le code de la fonction `printf`



Options utiles de gcc

- -c : pas d'édition de liens (produit un fichier objet .o)
- -o file: renomme le fichier de sortie file (au lieu de a.out)
- -g : insère les informations nécessaires à l'utilisation d'un débogueur (gdb).
- -Wall : fait le maximum de vérifications statiques possibles
- -Ipath : recherche les fichiers d'en-tête dans le répertoire path avant de les rechercher dans les répertoires standards (/usr/include, /usr/local/include).
- -Lpath : recherche les bibliothèques dans le répertoire path avant de les rechercher dans les répertoires standards (/usr/lib, /usr/local/lib).
- -Dflag=val : équivalent à écrire la directive `#define flag val` dans le code

Table of Contents

- 1 Introduction
- 2 Rapides rappel d'architectures et de compilation
- 3 Quelques caractéristiques du langage C
- 4 Quelques questions importantes en rapport avec la mémoire
- 5 Quelques notions pour commencer
- 6 Un exemple complet pour résumer
- 7 Quelques notions plus avancées

Caractéristiques du langage C (par rapport à d'autres..)

- Le langage C est inventé à la fin des années 70 pour programmer unix sur la machine PDP11.
- C est un langage de programmation *impératif*
 - On spécifie des *actions* qui ont lieu sur des *données*
 - Contrairement aux langages *objets*, *fonctionnels*, *equationnels*, etc.
- C est un langage *typé*
 - Toute variable doit être *déclarée* avant utilisation avec un *type* et ne sera utilisé qu'avec des valeurs de ce type
 - Contrairement à Python par exemple.
- C permet de (et est fait pour) manipuler directement *les cases mémoires* de la machine.
 - Il fait cela grâce aux *pointeurs*, C est un langage (haut-niveau) de *bas niveau*
 - Contrairement à tous les autres langages haut-niveau par exemple.

Caractéristiques du langage C (2)

- C ne propose en revanche aucune opération qui traite directement des objets de plus haut niveau
 - fichier informatique, liste, table de hachage doivent être construits à *la main* à partir des types de bases.
 - Contrairement à Python par exemple
- Il n'y a pas de gestion implicite de la mémoire
 - il faut allouer et désallouer explicitement la mémoire que l'on veut utiliser pour le programme
 - Il y a des allocations *statiques* (déclarations de variables) ou *dynamiques* (allocation lors de l'exécution).
 - Il n'y a pas de *garbage collecteur* (ramasse miettes)
 - Contrairement à Python, Java par exemple.
- Nécessite très peu de support à l'exécution, juste la `libc` qui est généralement installée sur toute les machines.

Pro/Con du langage C

- Ces caractéristiques en font un langage privilégié pour:
 - La programmation embarquée des micro-contrôleurs (quasiment exclusivement en C).
 - L'écriture de systèmes d'exploitation ou de modules noyaux (Les noyaux Linux et Windows sont en grande partie écrits en C)
 - Les programmes où l'on souhaite maîtriser les ressources matérielles utilisées et les algorithmes implémentés (calcul intensifs, cryptographie, ...)
- Il y a aussi un certain nombre de défauts:
 - Peu de vérification du compilateur ⇒ de nombreux *bugs d'inattention* qui font perdre un temps incommensurable.
 - Pas de gestion d'exception, modularité rudimentaire (pas d'espace de noms, pas de programmation objet).
 - Source de nombreuses failles de sécurité (débordement de tampon).

Quelques règles d'écriture de programmes C

- Ne jamais placer plusieurs instructions sur une même ligne.
- Utiliser des identificateurs significatifs pour les noms de fonctions
- Indentez vos programmes!!!
- une ligne blanche entre la dernière ligne des déclarations et la première ligne des instructions
- Une accolade fermante est seule sur une ligne.
- Aérer les lignes de programme en entourant par exemple les opérateurs avec des espaces
- Utilisez un éditeur pour programmer: Vim, emacs, Atom, sublimeText, vscode
- PAS ECLIPSE!

Table of Contents

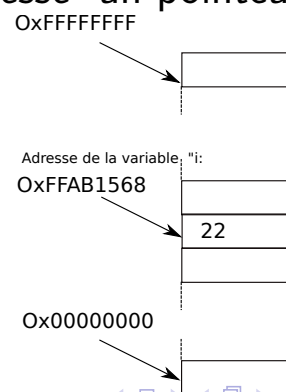
- 1 Introduction
- 2 Rapides rappel d'architectures et de compilation
- 3 Quelques caractéristiques du langage C
- 4 Quelques questions importantes en rapport avec la mémoire
- 5 Quelques notions pour commencer
- 6 Un exemple complet pour résumer
- 7 Quelques notions plus avancées

Valeur ≠ référence

- En informatique il est important de distinguer un *objet* et une *référence*:
 - Une *variable* (ou un objet) est stockée dans la mémoire
 - Une *référence* pointe vers une variable (ou un objet).
 - En C la notion de *pointeurs* est introduite pour faire des références sur les objets.
- En C lorsqu'on déclare une variable, une place est réservée dans la mémoire à l'exécution du programme, en C, on peut manipuler la variable par l'intermédiaire de son *adresse* un pointeur.

```
int main()
{
    int i=22;

    [...]
}
```



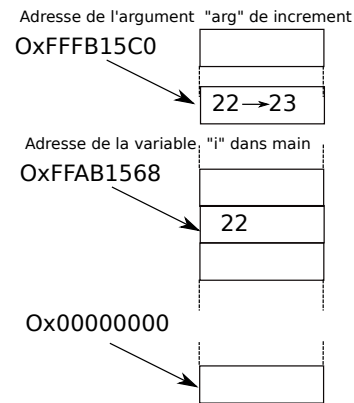
Passage de paramètre par valeur ou par référence

- Supposons qu'une fonction `increment` modifie son argument:

```
void increment(int arg)
{
    arg=arg+1;
}

int main()
{
    int i=22;

    increment(i);
    // ici, i vaut toujours 22
    [...]
}
```



- En C les arguments des fonctions sont passés **par valeur**
- La case de `arg` contient *une copie de la valeur* de la case de `i`

Passage de paramètres en python

- Passage par valeur
- Type entier non mutable

```
def increment(a):
    a=a+1
```

```
i=22
increment(i)
print(i)
# i vaut 22
```

- Passage par référence
- Type liste mutable

```
def allonge(l):
    l.append(4)
```

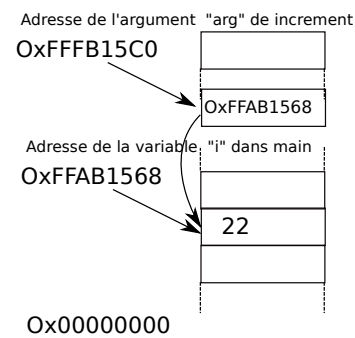
```
l=[1,2,3]
allonge(l)
print(l)
# l vaut [1,2,3,4]
```

Passage “par référence” en C

- En C, on émule un passage par référence, en passant la référence en argument grâce à la notion de *pointeur*.
- opérateur importants: * et &
- l'opérateur & (“adresse”): &i renvoie l'adresse de l'objet i
- l'opérateur * (“déréférencement”): *p renvoie l'objet pointé par le *pointeur* p (i.e. l'objet présent à l'adresse p)
- Attention, l'opérateur * est aussi utilisé dans les déclaration de variables pour spécifier les type “pointeurs”

```
void increment(int *arg) {
    *arg=*arg+1;
}

int main() {
    int i=22;
    increment(&i);
    // ici, i vaut 23
    [...]
```



La notion de pointeur en C

- Un pointeur en C est une variable (qui est donc stockée dans une case mémoire),
- Mais cette variable contient une valeur qui est *l'adresse* d'une autre variable.
- La gestion de la mémoire est beaucoup plus explicite en C:
 - On ne peut pas choisir l'adresse où sera stockée une variable
 - Mais on peut y accéder soit par son nom de variable, soit par son adresse (que l'on récupère par l'opérateur &)
 - La norme du langage C spécifie l'agencement en mémoire de certaines structures de donnée. Par exemple les éléments d'un *tableau* en C (l'équivalent d'une liste en Python), sont *consécutifs en mémoire*.
 - Les fonctions `malloc` (équivalent des fonctions `new` en java ou Python) et `free` permette d'allouer des *blocs* mémoire et de les remplir d'à peu près n'importe quoi.

Valeur et Adresse: exemple

Depuis le passage des adresses sur 64 bits on utilise le format %p pour les adresses

```
int i,j;
i=3;
j=i;
fprintf(stdout,"L'entier i=%d est à l'adresse %p\n",i,&i);
fprintf(stdout,"L'entier j=%d est à l'adresse %p\n",j,&j);
```

affiche

```
L'entier i=3 est à l'adresse 0x7ffd8b130470
L'entier j=3 est à l'adresse 0x7ffd8b130474
```

Exemple d'utilisation d'un pointeur

```
#include <stdio.h>

main()
{
    int i = 3;
    int *p;

    p = &i;
    printf("*p = %d \n",*p);
}
```

imprime *p = 3

objet	adresse	valeur
i	0xBFF434000	3
p	0xBFF434004	0xBFF434000
*p	0xBFF434000	3

Valeur et Adresse: exemple

```
main()
{
    int i = 3, j = 6;
    int *p1, *p2;
    p1 = &i;
    p2 = &j;
    *p1 = *p2;
}
```

après exécution

objet	adresse	valeur
i	0xBFF434000	6
j	0xBFF434004	6
p1	0xBFF434084	0xBFF434000
p2	0xBFF434092	0xBFF434004

```
main()
{
    int i = 3, j = 6;
    int *p1, *p2;
    p1 = &i;
    p2 = &j;
    p1 = p2;
}
```

après exécution

objet	adresse	valeur
i	0xBFF434000	3
j	0xBFF434004	6
p1	0xBFF434084	0xBFF434004
p2	0xBFF434092	0xBFF434004

Comment ne pas s'embrouiller avec les pointeurs

- Dessinez la mémoire !

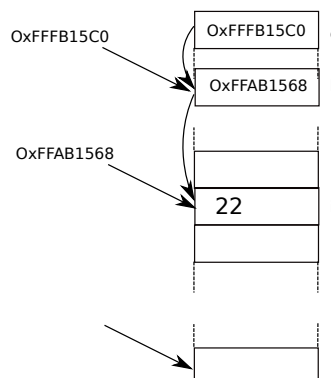
Pour ce programme:

```
int main() {
    int i=22;
    int *p; **q;

    p=&i;
    q=&p;

    print("i vaut %d\n",
          **q);
    [...]
}
```

Dessinez:



ou mieux:

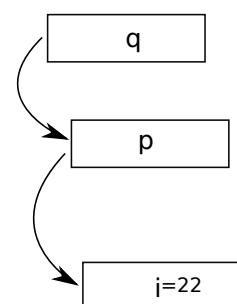


Table of Contents

- 1 Introduction
- 2 Rapides rappel d'architectures et de compilation
- 3 Quelques caractéristiques du langage C
- 4 Quelques questions importantes en rapport avec la mémoire
- 5 Quelques notions pour commencer
- 6 Un exemple complet pour résumer
- 7 Quelques notions plus avancées

Types de base en C

- En C, les variables doivent être *déclarées* avant d'être utilisées, on dit que C est un langage *typé*
- Les types de base en C sont désignés par des *spécificateurs de type* qui sont des mots clefs du langage:
 - les caractères (char),
 - les entiers (int, unsigned int, short, long, ...)
 - les flottants (nombres réels: float, double).
 - Il n'y a pas de type booleen, ils sont codés par des int (0=False)
- Une instruction composée d'un spécificateur de type et d'une liste d'identificateurs éventuellement initialisés séparés par une virgule est une déclaration. Par exemple:

```
int a;  
int b = 1, c;  
double x = 2.38e4;  
char message[80];
```


Les entrées/sorties: printf et scanf

- La syntaxe de la fonction printf est:

```
printf("chaîne de contrôle ",expr-1, ..., expr-n);
```
- La "chaîne de contrôle" contient le texte à afficher et les spécifications de format correspondant à chaque expression à afficher.
- Quelques format: %d → entier, %c → caractère, %f → réel (double).
- exemples:

```
int a;                int a;
a=10;                 a=10;
printf("a vaut %d \n",a); printf("a vaut %x\n",a);
```
- résultats:

```
a vaut 10             a vaut A
```

La fonction scanf (préférez fgetc pour les char)

- La syntaxe de la fonction scanf est:

```
scanf("chaîne de contrôle",&expr-1, ..., &expr-n);
```
- Ici, la "chaîne de contrôle" ne contient que les formats
- Les données à entrer au clavier doivent être séparées par des blancs ou des <RETURN> sauf s'il s'agit de caractères (<RETURN> est un caractère).
- exemple:

```
#include <stdio.h>
main()
{
    int i;
    printf("entrez un entier sous forme hexadecimale i = ");
    scanf("%x",&i);
    printf("i = %d\n",i);
}
```
- Si on entre au clavier la valeur 1a, le programme affiche i = 26.
- Attention à bien donner l'adresse de la variable (&i et non pas i) à affecter (passage de paramètre par "référence").

Fichiers en C

- Un fichier est un *périphérique à accès séquentiel*.
- En C, un fichier sera accédé par une sorte de *curseur* (en lecture ou en écriture) qui peut se déplacer dans le fichier.
- `fopen` retourne le descripteur du curseur (le numéro du *descripteur de fichier*):
- Exemple d'ouverture de fichier en lecture:


```
FILE *fich;
fich=fopen("./monFichier.txt","r");
if (!fich) fprintf(stderr,"Erreur d'ouverture : %s\n", "./monFichier.txt");
```
- Un objet de type `FILE` est quelquefois appelé un descripteur de fichier, c'est un entier désignant quel est le fichier manipulé.
- Trois descripteurs de fichier peuvent être utilisés sans qu'il soit nécessaire de les ouvrir (à condition d'utiliser `stdio.h`):
 - 1: `stdin` (standard input) : unité d'entrée (par défaut, le clavier) ;
 - 0: `stdout` (standard output) : unité de sortie (par défaut, l'écran) ;
 - 2: `stderr` (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran).
- Ces notions (entrée/sortie standard) sont fondamentales dans tout programmes.

Outil pour la programmation : Makefile

- À l'aide d'un fichier de description (en général appelé *Makefile*), l'utilitaire `make` crée la suite de commande qui va compiler votre programme
- Dans un fichier *Makefile*, une *cible* est un objet qui va être produit (par exemple un exécutable) par une règle de production.
- On produit une cible particulière à partir de fichiers particuliers, ces fichiers sont les *dépendances* de la cible.
- Exemple:

```
all: main

main:  main.o
      gcc main.o -o main

main.o: main.c type.h
      gcc -c main.c -o main.o
```

à l'exécution:

```
>make
>gcc -c main.c -o main.o
>gcc main.o -o main
```

Programmation modulaire: un exemple simple

```
//fichier fact3.c
//calcule n! ou n entier
int factorielle(int n)
{
    int i, fact=1;
    for (i = 1; i<=n; i++)
        fact *= i;
    return(fact);
}
```

```
//fichier fact3.h

//Declaration en-tetes
int factorielle(int n);
```

```
//fichier main2.c
#include "fact3.h"

int main()
{
    int x;
    x=factorielle(10);
    printf("10!=%d\n",x);
    return(0);
}
```

- On définit les en-têtes dans un fichier d'en-têtes pour la réutilisation par d'autres programmes.
- Commandes de compilation:
gcc -c fact3.c -o fact3.o
gcc -c main2.c -o main2.o
gcc main2.o fact3.o -o main2

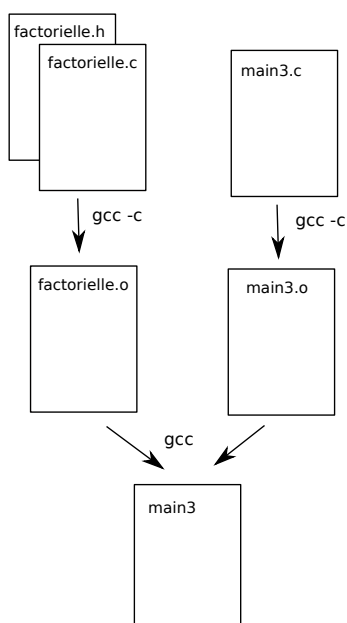
fichiers source .c et .h

- Pour chaque fichier source .c, on définit les signatures (i.e. en-tête) de fonctions dans un fichier d'en-têtes (fichier .h), pour les rendre *publiques* à d'autres fonctions.
- En C on ne peut pas *définir* une fonction à l'intérieur d'une fonction
- En revanche on peut *utiliser* une fonction dans une fonction.
- Pour cela il y a deux contraintes:
 - 1 Au moment de *la compilation*, lorsque le compilateur rencontre l'appel à une fonction factorielle, il faut qu'il ait déjà rencontré l'en-tête de la fonction `int factorielle(int n)`
 - 2 Au moment de *l'édition de lien*, il faut que l'un des codes objet contienne le code compilé de la fonction factorielle.
- Ces contraintes sont assurées en produisant un fichier d'en-tête (.h) pour chaque fichier source (.c)

Table of Contents

- 1 Introduction
- 2 Rapides rappel d'architectures et de compilation
- 3 Quelques caractéristiques du langage C
- 4 Quelques questions importantes en rapport avec la mémoire
- 5 Quelques notions pour commencer
- 6 Un exemple complet pour résumer**
- 7 Quelques notions plus avancées

Première étape: architecture logicielle (après l'étape algo)



fichier Makefile

```
CFLAGS=-Wall
LDFLAGS=-Wall

all: main3

main3: main3.o factorielle.o
    gcc ${LDFLAGS} main3.o factorielle.o -o factorielle

main3.o: main3.c factorielle.h
    gcc ${CFLAGS} -c main3.c -o main3.o

factorielle.o: factorielle.c factorielle.h
    gcc ${CFLAGS} -c factorielle.c -o factorielle.o

clean:
    \rm -f *.o main3
```

Écriture du code: factorielle (recursive)

fichier factorielle.h

```
int factorielle(int n);
```

fichier factorielle.c

```
#include "factorielle.h"

//fonction factorielle
//calcule n! ou n entier>0
//de manière récursive
int factorielle(int n)
{
    int    fact;

    if (n==1)
        fact=1;
    else
        fact=n*factorielle(n-1);

    return fact;
}
```

fichier main3.c

```
#include <stdio.h>
#include "factorielle.h"

int main()
{
    int x;

    x=factorielle(10);
    printf("factorielle(10) = %d\n",x);

    return 0;
}
```

Compilation & Debug

```
trisset@fania:~/cours/2018/CRO/cours_tri/exemples_C$ make
gcc -Wall -c main3.c -o main3.o
gcc -Wall -c factorielle.c -o factorielle.o
gcc -Wall main3.o factorielle.o -o main3
trisset@fania:~/cours/2018/CRO/cours_tri/exemples_C$ ./main3
factorielle(10) = 3628800
trisset@fania:~/cours/2018/CRO/cours_tri/exemples_C$
```

Savoir lire les messages de compilation

fichier main3.c modifié (sans inclure factorielle.h)

```
#include <stdio.h>

int main()
{
    int x;

    x=factorielle(10);
    printf("factorielle(10) = %d\n",x);

    return 0;
}
```

message de compilation

```
trisset@fania:~/cours/2018/CRO/cours_tri/exemples_C$ make
gcc -Wall -c main3.c -o main3.o
main3.c: In function 'main':
main3.c:7:5: warning: implicit declaration of function 'factorielle' [-W
x=factorielle(10);
    ~
gcc -Wall main3.o factorielle.o -o main3
trisset@fania:~/cours/2018/CRO/cours_tri/exemples_C$ ./main3
factorielle(10) = 3628800
```

Savoir lire les messages de compilation

fichier factorielle.c modifié (deux * rajoutées...)

```
[...]
int factorielle(int n)
{
    int    *fact;

    if (n==1)
        fact=1;
    else
        *fact=n*factorielle(n-1);

    return fact;
}
```

```
trisset@fania:~/cours/2018/CRO/cours_tri/exemples_C$ make
gcc -Wall -c factorielle.c -o factorielle.o
factorielle.c: In function 'factorielle':
factorielle.c:11:9: warning: assignment makes pointer from integer without a cast [-Wint-conversion]
    fact=1;
    ~
factorielle.c:15:10: warning: return makes integer from pointer without a cast [-Wint-conversion]
    return fact;
    ~
factorielle.c:13:10: warning: 'fact' may be used uninitialized in this function [-Wmaybe-uninitialized]
    *fact=n*factorielle(n-1);
    ~
gcc -Wall main3.o factorielle.o -o main3
trisset@fania:~/cours/2018/CRO/cours_tri/exemples_C$ ./main3
Erreur de segmentation (core dumped)
trisset@fania:~/cours/2018/CRO/cours_tri/exemples_C$
```

Bonnes pratiques de programmation en C

- Utiliser le flag `-Wall` dans le Makefile.
- Ne **jamais** laisser traîner des Warning non résolus.
- Tester chacune de vos fonctions indépendamment.
- Debugger une seule chose à la fois.
- Consolider la recherche de la source du bug, l'essentiel du temps de perdu en debug vient du fait que l'on s'est trompé sur la source (i.e. la ligne) du bug.
- (Pour l'instant) l'outil de Debug privilégié est le `printf` **MAIS**:
- **Attention**, une instruction `printf` qui s'exécute ne s'affiche pas forcément à l'écran!
- Pour être sur que le `printf` s'affiche, il faut terminer la chaîne de caractère par `\n`:
`printf("La variable i vaut %d\n",i);`

Table of Contents

- 1 Introduction
- 2 Rapides rappel d'architectures et de compilation
- 3 Quelques caractéristiques du langage C
- 4 Quelques questions importantes en rapport avec la mémoire
- 5 Quelques notions pour commencer
- 6 Un exemple complet pour résumer
- 7 Quelques notions plus avancées

Les différentes normes de C

- C est inventé en 1972 par Dennis Ritchie et Ken Thompson
 - Popularisé grâce au livre de Kernighan et Richie (K&R): "The C Programming Language"
 - La première normalisation a abouté en 1989 à la norme C89 dites *Ansi C* (dites aussi C90 ou C ISO), très légères évolutions par rapport au C original de K&R.
 - En 1999, nouvelle normalisation ISO: C99
 - Puis en 2011, le standard C11 est ratifié
- ⇒ On vous conseille d'être compatible C11 (options `-std=c11` de gcc), normalement tous ce que je vous apprend passe à ça..

Optimisation et Warning

- Le niveau d'optimisation (options `-O` de gcc) a une grande influence sur l'efficacité et sur le comportement des programmes faux.
 - Par défaut, le compilateur met l'option `-O2` (niveau 2 d'optimisation)
 - On peut vouloir `-O3` (plus rapide) ou `-O0` (dans le cas de code embarqué)
- `-Wall` indispensable
- Pour avoir un maximum d'information on peut utiliser:
`-Wall -Wextra -pedantic`
- Pendant la phase de test, il est conseillé d'utiliser
`-Werror`
(provoque une erreur dès qu'il y a un Warning).

Utilisation (moderne) des types simples

- Traditionnellement on utilise les types: char, int, short, long ou unsigned.
- On rappelle que les entiers n'ont pas la même taille sur toutes les machines
- char: 8 bits, short: au moins 16 bits, int: au moins 16 bits, long: au moins 32 bits, long long: au moins 64 bits, etc.
- La librairie *standard int type* (<stdint.h>) est de plus en plus préconisée:
 - Entiers signés: int8_t, int16_t, int32_t, int64_t
 - Entiers non signés: uint8_t, uint16_t, uint32_t, uint64_t
 - Utiliser uint8_t plutôt que char si on désigne des octets.
 - La fonction sizeof() renvoie un entier de type size_t qui est, en gros, un entier capable stocker le plus grand index de tableau possible sur la machine. À utiliser impérativement pour faire du code portable.
 - Continuer d'utiliser char si on utilise les fonction de libstring (<string.h>)

programme de la suite du cours

- Le reste des séances CRO se feront sur machine.
- Pour chaque séance vous disposerez
 - D'un sujet de TD/TP
 - De slides de cours expliquant les concepts
 - De sections "pour aller loin" avec solution
- Programme des séances
 - TD1: introduction à C et Makefile
 - TD2: Entrées/Sorties simple, passage de paramètres
 - TD3: Tableaux, structures de contrôle, boucle, E/C Fichiers
 - TD4: Pointeurs, structures et listes chaînées
 - TP1: Pointeurs: tours de Hanoï
 - TD5: Pointeur et arbres
 - TP2: bigInt: tableau et test
 - TD6: listes triées en C (pointeurs avancé)
 - TD7: socket en C (programmation système)
 - TP3: Graphes en C
 - TD8: pointeurs de fonctions