CRO: C langage and programming Roots

tanguy.risset@insa-lyon.fr
Lab CITI, INSA de Lyon
Version du January 28, 2019

Tanguy Risset

January 28, 2019

Table of Contents

Les bases du C

- Instruction, expression, programme
- Éléments de base
  - Variable et types de base
  - Opérateurs
  - Fonctions et passage de paramètres

Syntaxe

- La syntaxe est définie par des règles de grammaire.
- Extrait de la grammaire:

```
expression::
        primary
        | expression binop expression
        | expression asgnop expression
        | ....
primary::
       identifier
       | string
       | constant
       | ...
```

```
asgnop::
       =
       | +=
       | ...
binop::
       +
       | -
       | ...
```

- `x = 0 , a + b, a+=1` sont syntaxiquement corrects (respectent la grammaire)
- Un programme syntaxiquement correct n'est pas forcément un programme C valide. Exemple: `1 = 0` est syntaxiquement correct

Expression, Instruction

- Une *expression* est une suite de composants élémentaires syntaxiquement correcte, par exemple: `x = 0` ou bien
`(i >= 0) && (i < 10)`
- En C chaque expression produit:
  - une action (modification possible de l'état des variables en mémoire)
  - un résultat (valeur renvoyée par l'expression)
- Une *instruction* est une expression suivie d'un point-virgule. Le point-virgule signifie en quelque sorte "évaluer cette expression et oublier le resultat".
- Plusieurs instructions peuvent être rassemblées par des accolades { et } pour former une *instruction composée* (ou *bloc*) qui est syntaxiquement équivalent à une instruction. Par exemple,

```
if (x != 0)
{
  z = y / x;
  t = y % x;
}
```


Structure d'un programme C

- Un programme C se présente de la façon suivante :

directives au préprocesseur
déclarations de variables globales
fonctions secondaires

int main()
{ *déclarations de variables internes*
instructions
}

```
#include <stdio.h>

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

  return(0);
}
```

- La fonction `main` est exécutée lors de l'exécution du programme.
- Le *résultat* de la fonction main est le résultat de l'exécution du programme (code d'erreur en général)

Variables

- La notion de variable est très importante en programmation
- Du point de vue sémantique, une variable est une entité qui contient une information :
  - Une variable possède un nom, on parle d'*identifiant*
  - Une variable possède une valeur qui change au cours de l'exécution du programme
  - Une variable possède un type qui caractérise l'ensemble des valeurs qu'elle peut prendre
- Du point de vue pratique, une variable est une manière mnémotechnique pour désigner une partie de la mémoire.
- En C les noms des variables sont composés de la manière suivante: une suite de caractères parmis :
  - les lettres (minuscules ou majuscules, mais non accentuées),
  - les chiffres (sauf en début de nom),
  - le "blanc souligné" (_).

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 langages:
  - les caractères (`char`),
  - les entiers (`int, short, , unsigned long`)
  - les flottants (nombres réels, `float, double`).
  - Il n'y a pas de type booleen, ils sont codés par des `int`
- 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];
```

- Il existe de nombreuses conversions de types implicites

Représentation de l'information

- Le type d'une variable indique au compilateur la manière de stocker la variables en mémoire
- Il est important de connaître comment sont stockées les variables car C propose de nombreuses manipulations au niveau bit.
- La mémoire est une suite de bit structurée en *octets* (8 bits) puis en mots (4 octets, 32 bits).
- L'adresse 1084 doit se lire comme "le 1084^{eme} octet de la mémoire"
- On utilisera trois notations pour représenter les valeurs entières en mémoire.
  - La notation décimale usuelle (10 chiffres): 12_{dec} vaut la valeur 12
  - La notation binaire (2 chiffres): 1100_{bin} vaut la valeur 12
  - La notation hexadécimale (16 chiffres) C_{hex} vaut la valeur 12 (essentiellement pour représenter les adresses).

Les types entiers

mot-clef	taille	dénomination	remarques
char	8 bits	caractère	peut être utilisé comme entier
short	16 bits	entier court	
int	32 bits	entier	
long	>=32 bits	entier long	souvent 64 bits

- En C les entiers signés sont représentés en *complément à 2*, c'est à dire (pour un entier n sur 32 bit):
  - le bit de poids fort (bit 32) représente le signe (0 pour positif, 1 pour négatif)
  - Si l'entier est positif: les 31 autres bits correspondent à la décomposition de l'entier en base 2.
  - Si l'entier est négatif les 31 autres bits correspondent à la décomposition de l'entier $2^{31} - |n|$
  - pour `int n;` on a donc les contraintes: $-2^{31} <= n < 2^{31}$

Les types entier

- Les mots-clef des types peuvent être précédés d'attributs. Par exemple `unsigned int n` indique que l'entier n est positif, la représentation en complément à deux n'est pas utilisée on a alors $0 <= n < 2^{32}$

signed char	$[-2^7; 2^7[$
unsigned char	$[0; 2^8[$
short int	$[-2^{15}; 2^{15}[$
unsigned short int	$[0; 2^{16}[$
int	$[-2^{31}; 2^{31}[$
unsigned int	$[0; 2^{32}[$

- Le fonction `sizeof` calcul la taille d'un type ou d'une expression. Le résultat est un entier égal au *nombre d'octets* nécessaires pour stocker le type ou l'objet. Par exemple ci-dessous `taille` prendra la valeur 2 les deux fois

```
unsigned short x;
taille = sizeof(unsigned short);
taille = sizeof(x);
```


Les types flottants

mot-clef	taille DEC Alpha	Taille PC Intel	
`float`	32 bits	32 bits	flottant
`double`	64 bits	64 bits	flottant double précision
`long double`	64 bits	128 bits	flottant quadruple précision

- Les flottants sont généralement stockés en mémoire sous la représentation de *la virgule flottante normalisée*. On écrit le nombre sous la forme $signe\ 0, mantisse\ B^{exposant}$ $(12,3 \Leftrightarrow 0.123 * 10^2)$. En général, B=2. Le digit de poids fort de la mantisse n'est jamais nul.
- Par exemple dans le standard EE754, en simple précision (32 bits):

signe	Mantisse	Exposant
1bit	8 bits	23 bits

Les caractères

- Un `char` peut contenir n'importe quel élément du jeu de caractères de la machine utilisée.
- un `char` est codé sur un octet ;
- Le jeu de caractères utilisé correspond généralement au codage ASCII (sur 7 bits).
- La plupart des machines utilisent désormais le jeu de caractères ISO-8859-1 (aussi appelée ISO-LATIN-1) dont les 128 premiers caractères correspondent aux caractères ASCII.
- extrait de la table ASCII:

caractère représenté	valeur (dec)	valeur (hex)	· caractère représenté	valeur (dec)	valeur (hex)
!	33	21_{hex}	A	65	41_{hex}
"	34	22_{hex}	B	66	42_{hex}
#	35	23_{hex}	C	67	43_{hex}
...	...	...	...	...	...
0	48	30_{hex}	a	97	61_{hex}
1	49	31_{hex}	b	98	62_{hex}
2	50	32_{hex}	...	...	...

Les constantes

- Les valeurs exprimées dans un programme C sont des *constantes*
- Les constantes entières peuvent être exprimées en décimal (`int i=12`) ou en hexadécimal (`int i=0xA`),
- On peut aussi indiquer qu'elle doivent être stockées sur un `long` (`12L`) ou en `unsigned` (`12U`).
- Les constantes réelles suivent le même principe
- `12.34` (`double`), `12.3e-4`, (`double`), `12.34F` (`float`), `12.34L` (`Long`)
- Les caractères imprimable sont repérentés entre cotes ('): `char a='Z'`
- Les caractères non imprimables sont représentés en code octal (base 8) précédé d'un antislash, Les plus fréquent ont des représentations standard: `\n` nouvelle ligne,
  `\r` retour chariot,
  `\t` tabulation horizontale
  `\f` saut de page,

Constantes et variables

- La notion de constante n'est pas limitée aux valeurs fixes
- Une constante est une valeur non modifiable, par exemple l'adresse d'une variable.
- Il existe une différence fondamentale avec les variables: il n'y a pas de place réservée pour une constante dans la mémoire lors de l'exécution d'un programme.
- C'est le compilateur qui met en dur la valeur de la constante lorsqu'il génère l'instruction:
  `j=i+10` $\Rightarrow$ `add Rj,Ri,#10`

Les opérateurs

- affectation: `variable = expression`
- opérateurs arithmétiques `+,-,*,/,%`,
  `expression-1 op expression-2`
- opérateurs relationnels `>,<, >=,<=,==,!=`
- opérateurs logiques booléens `&&, ||, !`
- opérateurs logiques bit à bit `&, |, ^,~,«,»`
- opérateurs d'affectation composée `+=, -=, *=, /=`, `expression-1 op= expression-2`
  $\Leftrightarrow$
  `expression-1 = expression-1 op expression-2`
- opérateurs d'incrémentation et de décrémentation, `++, - -`
- opérateur de conversion de type `(type) objet`
  `a=(int) c`
- opérateur adresse `&`,
  `a=&b`

Règles de priorité des opérateurs

- les opérateurs sont plus ou moins prioritaires,
- Dans le cas de priorité égales on a un ordre d'évaluation (associatifs à droite ou à gauche)
- ordre de priorité:

opérateurs	associativité
() [] -> .	droite
! ++ – -(unaire) (type) *(indirection) &(adresse) sizeof	gauche
* / %	droite
+ -(binaire)	droite
« »	droite
< <= > >=	droite
== !=	droite
&(et bit-à-bit)	droite
|	droite
&&	droite
||	droite
? :	gauche
= += -= *= /= %= &= ≙ |= «= »=	gauche
,	droite

Table of Contents

Les fonctions

- On peut en C découper un programme en plusieurs fonctions.
- Seule la fonction `main` est obligatoire.
- Même si vous ne définissez pas de fonction, vous utilisez les fonctions des biblothèques C standard (`return`, `printf` par exemple)
- Définition de fonction:

```
type nom (type-1 arg-1,...,type-n arg-n)
{[déclarations de variables locales ]
  liste d'instructions
}
```

```
int factorielle(int n)
{
  int  i, fact=1;
  for (i = 1; i<=n; i++)
     fact *= i;
  return(fact);
}
```

- La première ligne `int factorielle(int n)` s'appelle l'*en-tête* (ou *prototype*, ou encore *signature*) de la fonction.
- Appel de fonction : `x=factorielle(10);`

Passage de paramètres

- Une fonction communique des valeurs avec son environnement appelant à travers les paramêtres et le résultats.
- dans `int factorielle(int n)`, `n` est le *paramètre formel* de la fonction. Il peut être utilisé dans le corps de la fonction comme une variable locale.
- dans `x=factorielle(10);`, 10 est le *paramètre effectif* utilisé lors de cet appel.
- En C, tout ce passe lors de cette appel comme si on exécutait le corps de la fonction *avec la case mémoire pour* `n` *contenant une copie de la case mémoire contenant 10*. On dit que les paramètres sont passés *par valeur*.
- Lorsque l'appel est terminé, la case mémoire de `n` disparait.
- Donc: on peut modifier la valeur du paramètre formel `n` dans le corps de la fonction mais cela ne modifiera pas la valeur du paramètre effectif (10).

Imbrication de 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 ai 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.
- On a plusieurs moyens pour assurer ces contraintes:
  1. Un seul fichier
  2. Déclarer les en-têtes de fonctions en début de fichier
  3. Compilation séparée

Fonctions dans Un seul fichier

```
//fonction factorielle
//calcule n! ou n entier
int factorielle(int n)
{int  i, fact=1;
//double initialisation
for (i = 1; i<=n; i++)
  fact *= i;
return(fact);
}

//fonction main
//affiche 10! à l'ecran
int main()
{int x;
 x=factorielle(10);
 printf("10!=%d\n",x);
 return(0);
}
```

- Compilation:
  `gcc fact1.c -o fact`

Fonction avec déclaration des en-têtes

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

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

//fonction factorielle
//calcule n! ou n entier
int factorielle(int n)
{ int  i, fact=1;
  //double initialisation
  for (i = 1; i<=n; i++)
    fact *= i;
  return(fact);
}
```

- attention au ; après l'en-tête.
- Commande de compilation:
  `gcc fact1.c -o fact`

Deux fichiers

```
//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 main1.c

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

//fonction main
//affiche 10! à l'ecran
int main()
{ int x;
  x=factorielle(10);
  printf("10!=%d\n",x);
  return(0);
}
```

- Commandes de compilation:
  `gcc -c fact3.c -o fact3.o`
  `gcc -c main1.c -o main1.o`
  `gcc main1.o fact3.o -o main1`

Fonctions: La bonne manière de faire

```
//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
```


Récursivité

- Une fonction peu s'appeler elle-même, c'est alors une fonction *récursive*.
- la récursion est une autre manière d'effectuer des boucles.

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

  if (n==1)       //condition d'arret
    fact=1;
  else
    fact=n*factRecurs(n-1);
  return(fact);
}
```

Récursivité

- La récursivité est très importante dans certains paradigmes de programmation (programmation fonctionnelle).
- Elle correspond à une méthode naturelle de décomposition d'un problème en problèmes plus simple:
  - pour calculer $n!$, je cacule $(n-1)!$ (plus simple)
  - puis je calcule $n*(n-1)!$
- Pour s'assurer que. que le programme ne va pas boucler indéfiniement, il est impératif de;
  - Définir la *condition d'arret*: ici c'est `n==1`
  - Définir une quantité qui va croître (ou décroître) lors des appels successifs de la fonction jusqu'à ce que la condition d'arret soit vérifiée, ici cette quantitée est n la valeur du paramètre.
  - On peut formaliser cela avec la notion d'*invariant*

Portée des variable

- Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même durée de vie. On distingue deux catégories de variables.
  - Les variables permanentes (ou statiques)
  - Les variables temporaires
- Chaque variable déclarée a une *portée* (ou durée de vie) qui est la portion de code dans laquelle elle est connue.
- On peut déclarer des variables au début de chaque bloc (début d'une fonction ou portion de code entre accolade). La *portée* (ou durée de vie) de ces variables est limitée au bloc.
- Les blocs sont forcément imbriqués lexicalement, lors d'une utilisation d'une variable n, elle peut avoir été déclarée deux fois dans la suite des blocs englobant. L'utilisation correspond à celle de la première définition rencontrée lorsque l'on remonte dans les blocs (i.e. la dernière effectuée temporellement).

Portée: exemple

```
int n = 10;
void fonction();

void fonction()
{int n = 0;
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{int i;
  for (i = 0; i < 5; i++)
    fonction();
}
----------------
appel numero 1
appel numero 1
appel numero 1
```


Portée: exemple

```
int n;
void fonction();

void fonction()
{ n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{ int i;
  for (i = 0; i < 5; i++)
    fonction();
}
-----------------
appel numero 1
appel numero 2
appel numero 3
appel numero 4
```

Passage de paramêtre par référence

- Tout se passe comme si la fonction `scanf` modifiait un de ses argument.
- En fait comme les argument sont passé par valeurs, on doit passer en paramêtre un *pointeur* sur l'objet à modifier (donc passer l'adresse de l'objet)
- La fonction `scanf` ne peut modifier son argument, mais elle peut modifier l'*objet* pointé par l'argument.
- Ce mechanisme est utilisé systématiquement en C, pour que la fonction modifie un objet, il est passé en argument par référence.

Fichiers

- Les accès à un fichier se font par l'intermédiaire d'un tampon (buffer) qui permet de réduire le nombre d'accès aux périphériques (disque...).
- Pour pouvoir manipuler un fichier, un programme a besoin d'un certain nombre d'informations : l'adresse du tampon, position dans le fichier, mode d'accès (lecture ou écriture) ...
- Ces informations sont rassemblées dans une structure dont le type, `FILE *`, est défini dans `stdio.h`. Un objet de type `FILE *` est un stream (flot).
- Avant de lire ou d'écrire dans un fichier, on l'*ouvre* par la commande `fich=fopen("nom-de-fichier","r")`. Cette fonction dialogue avec le système d'exploitation et initialise un stream `fich`, qui sera ensuite utilisé lors de l'écriture ou de la lecture.
- Après les traitements, on *ferme* le fichier grâce à la fonction `fclose(fich)`.

Table of Contents

Les entrées-sorties

- La librairie standard `libstdio.a` propose des fonctions pour faire des entrées-sorties sur les périphériques standard: l'écran et le clavier.
- Ces fonctions sont `printf` pour écrire et `scanf` pour lire
- Il faut inclure la directive `#include <stdio.h>` au début du fichier si on désire les utiliser.
- Ce sont des fonctions *d'impression formatée*, ce qui signifie que les données sont converties selon le format particulier choisi.

La fonction printf

- 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.
- `%d` indique l'affichage d'un entier signé en décimal.
- `%u` indique l'affichage d'un entier non signé en décimal.
- `%c` indique l'affichage d'un caractère.
- `%f` indique l'affichage d'un réel (double) en décimale.
- exemples:

```
int a;                          int a;
a=10;                           a=10;
printf("a vaut %d \n",a);       printf("a vaut %4d \n",a);
```

- résultats:

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


printf: exemples

- `%x` indique l'affichage d'un entier non signé en hexadécimal.
- exemples:

```
int a;
a=10;
printf("a vaut (en hexa) %x \n",a);
```

```
short a;
a=10;
//conversion en int
printf("a vaut %d \n",a);
```

- résultats:

```
a vaut (en hexa) A
```

```
a vaut   10
```

- Attention ! Il y a une conversion de type:
- exemples:

```
char a;
a=66;
//conversion en int
printf("a vaut %d \n",a);
```

```
short a;
a=66;
//conversion en char
printf("a vaut %c \n",a);
```

- résultats:

```
a vaut 66
```

```
a vaut B
```

Encore printf

- exemples: %f, %e

```
double x = 1e-8 + 1000;
printf("x vaut %f \n",a);
```

```
double x = 1e-8 + 1000;
printf("x vaut %e \n",a);
```

- résultats:

```
x vaut 1000.000000
```

```
x vaut 1.000000e+03
```

- exemples: (conversion vers unsigned)

```
int a;
a=-23674;;
printf("a vaut %d \n",a);
```

```
int a;
a=-23674;;
//conversion en unsigned
printf("a vaut %u \n",a);
```

- résultats:

```
a vaut -23674
```

```
a vaut 4294943622
```


Encore printf

- programme:

```
##include <stdio.h>

int main()
{
  fprintf(stdout,"Ascii[%d]=%c, en octal: %o en hexa: %x\n",
                 '\101','\101','\101','\101');
  fprintf(stdout,"Ascii[%d]=%c, en octal: %o en hexa: %x\n",
                  65,65,65,65);

  return(0);
}
```

- résultat:

```
trisset@hom:~/cours/2005/AGP/cours_tri/transp$ exChar
Ascii[65]=A, en octal: 101 en hexa: 41
Ascii[65]=A, en octal: 101 en hexa: 41
trisset@hom:~/cours/2005/AGP/cours_tri/transp$
```

La fonction scanf

- 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).
- Les formats sont légèrement étendus par rapport à ceux de `printf`
- 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 à affecter.

Fichiers

- on peut ouvrir un fichier sous plusieurs modes: lecture (`"r"`), écriture au début (`"w"`), écriture à la fin (`"a"`). `fopen` retourne 0 en cas d'echec.
- Exemple:

```
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 déscripteur 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`):
  - `stdin` (standard input) : unité d'entrée (par défaut, le clavier, valeur du descripteur: 1) ;
  - `stdout` (standard output) : unité de sortie (par défaut, l'écran, valeur du descripteur: 0) ;
  - `stderr` (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran, valeur du descripteur: 2).
- Il est fortement conseillé d'afficher systématiquement les messages d'erreur sur stderr afin que ces messages apparaissent à l'écran même lorsque la sortie standard est redirigée.

Autres fonction de lecture/ecriture

- Entrée/sorties de caractères
  - `int fgetc(FILE* flot);`
  - `int fputc(int caractere, FILE *flot)`
- Entrée/sorties de chaînes de caractères
  - `char *fgets(char *chaine, int taille, FILE* flot);`
  - `int fputs(const char *chaine, FILE *flot)`
- Entrée/sorties binaires
  - `size_t fread(void *pointeur, size_t taille, size_t nombre, FILE *flot);`
  - `size_t fwrite(void *pointeur, size_t taille, size_t nombre, FILE *flot);`
- positionnement dans un fichier
  - `int fseek(FILE *flot, long deplacement, int origine);`
  - trois valeurs possibles pour origine: SEEK_SET (égale à 0) : début du fichier ,SEEK_CUR : position courante, SEEK_END (égale à 2) : fin du fichier.

fgetc, fputc: exemple

```
##include <stdio.h>
##include <stdlib.h>
##define ENTREE "entree.txt"
##define SORTIE "sortie.txt"
int main(void)
{FILE *f_in, *f_out;
  int c;

  if ((f_in = fopen(ENTREE,"r")) == NULL)
    {
      fprintf(stderr, "\nErreur: Impossible de lire le fichier %s\n",ENTREE);
      return(EXIT_FAILURE);
    }
  if ((f_out = fopen(SORTIE,"w")) == NULL)
    {
      fprintf(stderr, "\nErreur: Impossible d'ecrire dans le fichier %s\n",
      SORTIE);
      return(EXIT_FAILURE);
    }
  while ((c = fgetc(f_in)) != EOF)
    fputc(c, f_out);
  fclose(f_in);
  fclose(f_out);
  return(EXIT_SUCCESS);
}
```

Arguments de la fonction main

- La fonction `main` doit être présente si le programme est utilisé directement (pas comme une librairie), par exemple: appelé depuis le shell.
- En général, la fonction `main` retourne un code d'erreur de type `int`.
- Lorsque l'on compile avec la commande `gcc monProg.c -o monProg`, l'exécution de la command `monProg` dans un shell appelle la fonction `main`
- Si on lance la commande `monProg` avec des arguments (ex: `monProg fichier1.txt`, il est possible de récupérer dans la fonction `main` la valeur de ces arguments sous forme de chaînes de caractères: on utilise les arguments standard de la fonction `main`: `int argc` (*argument count*), `char *argv[]` (*argument values*) (on peut aussi récuperer des variables d'environnement grâce à `envp`)

Exemple de fonction main

- Multiplication de deux entier: `gcc mult.c -o mult`

```
shell> mult 8 32
Le produit de 8 par 32 vaut:  256
```

```
##include <stdio.h>
##include <stdlib.h>
int main(int argc, char *argv[])
{
  int a, b;
   if (argc != 3)
    {
      printf("\nErreur : nombre invalide d'arguments");
      printf("\nUsage: %s int int\n",argv[0]);
      return(EXIT_FAILURE);
    }
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("\nLe produit de %d par %d vaut : %d\n", a, b, a * b);
  return(EXIT_SUCCESS);
}
```

Convention d'écriture en C

- Il existe très peu de contraintes dans l'écriture d'un programme C. Toutefois ne prendre aucune précaution aboutirait à des programmes illisibles.
- On n'écrit qu'une seule instruction par ligne : le point virgule d'une instruction ou d'une déclaration est toujours le dernier caractère de la ligne.
- Les instructions sont disposées de telle façon que la structure modulaire du programme soit mise en évidence.
  - une accolade ouvrante marquant le début d'un bloc doit être seule sur sa ligne ou placée à la fin d'une ligne.
  - Une accolade fermante est toujours seule sur sa ligne.
- Les instructions doivent être indentées afin que toutes les instructions d'un même bloc soient alignées. Le mieux est d'utiliser le mode C d'`emacs` ou de `vi`.
- On commente abondament le code