Le debugger CODE::BLOCK

Ce cours/TD/TP est à réaliser à l'aide de code::block
Disponible gratuitement ici : https://codeblocks.org/downloads/26
Souc Windows , télecharger codeblocks-xx.yymingw-setup.exe (xx.yy est le numéro de la dernière version)

code ::block est préinstallé sur les distributions Linux type Debian (Ubuntu, Mint ...)

Cette version de code::block dispose du compilateur c/c++ mingw sur windows et gcc sous Linux ainsi que du debugger GDB (Gnu Debugger) qui permet de poser des points d'arrêt, d'avancer un programme en pas à pas et de consulter les variables.

Prérequis :
Connaissances élémentaires du langage C. main, fonctions, variables, types, printf.
Cration d'un projet dans code::block en langage C.

1 Prise en main du debugger

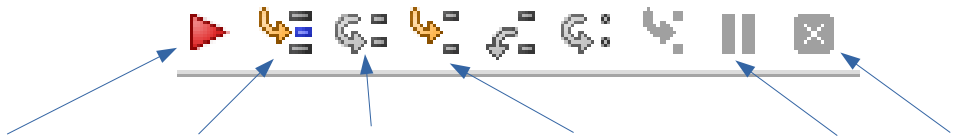
Ecrire le programme p1.c ci dessous, dans code::block qui va permettre de visualiser les divers zones d'enregistrements des variables en mémoire.
Les variables locales (ici xloc et yloc), sont déclarées dans une fonction elles ne sont accessibles que depuis la fonction ou elles sont déclarées (ici main). Les variables globales en dehors des fonctions (ici iglo et jglo) sont accessibles de n'importe quelle partie du programme.

```
char mess[]="Bonjour"; // une chaine
int iglo;      // variables globales
int jglo;

int main()
{
   char xloc,yloc;      // variables locales
   iglo=0x12345678; // les entiers sont codés sur 32bits
   jglo=0xABCDEF09;
   xloc=0x11;     // les char sont codés sur 8 bits
   yloc=0x22;
   return 0;
}
```

Assurez-vous que la syntaxe est bonne, compiler-executer … il n'y a rien à voir puisque le programme ne fait rien. Le debugger va permettre de visualiser les variables dans la mémoire de l'ordinateur.

Commandes du debbuger



Lancer debug / executer / executer une ligne / entrer dans la fonction / pause / arrêt debug

Placer un point d'arrêt sur la ligne iglo=0x12345678 ; pour cela cliquer à droite du numéro de ligne, un point rouge apparaît. Lancer le debugger puis en cliquant sur le triangle rouge … l'exécution s'arrête sur le point d'arrêt (triangle jaune).

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   char mess[]="Bonjour"; // une chaine
5   int iglo;        // variables globales
6   int jglo;
7
8   int main()
9   {
10      char xloc,yloc;        // variables locales
11      iglo=0x12345678; // les entiers sont codés sur 32bits
12      jglo=0xABCDEF09;
13      xloc=0x11;       // les char sont codés sur 8 bits
14      yloc=0x22;
15      return 0;
16  }
```

Cliquer sur Next Line (executer une ligne) et dérouler le programme en pas à pas jusqu'à la fin.
Relancer à nouveau le debugger jusqu'au point d'arrêt.

Pour observer les variables cliquer sur le cafard →

Faire apparaître les fenêtres « Memory dump » et « Watches»

Waches affiche les valeurs actuelles des variables, remarquer la distinction entre les locales et les globales

Watches

Function arguments		
Locals		
xloc	0 '\000'	
yloc	0 '\000'	
iglo	0	int
jglo	0	int
mess	"Bonjour"	char [8]

Dérouler le programme en pas à pas jusqu'à return 0 ;,

La fenêtre **« Watches »** affiche les valeurs des variables à la fin du programme.
Remarque : la fenêtre « Watches » affiche les variables en décimal, un clic-droit – Properties permet un affichage hexadecimal.

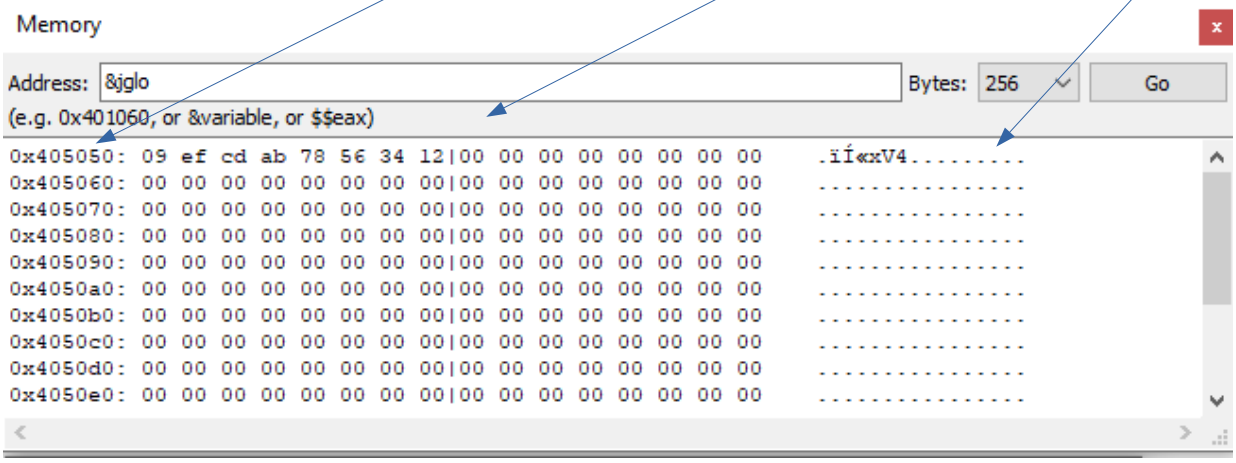
Remarque : Il est possible de modifier la valeur d'une variable dans Watches par un double clic sur sa valeur. L'accès est possible hors de la zone "locals »

Watches

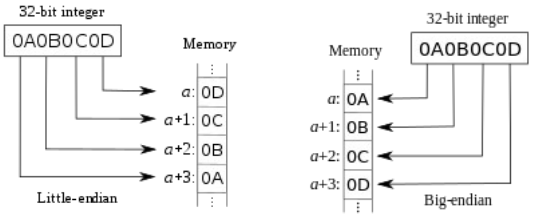
Function arguments		
Locals		
xloc	17 '\021'	
yloc	34 '"'	
iglo	0x12345678	int
jglo	0xabcdef09	int
xloc	0x11	char
yloc	0x22	char
mess	"Bonjour"	char [8]

Memory permet de visualiser (dumper) le contenu de la mémoire.
Pour visualiser la variable iglo, entrer dans la barre « Address » **&iglo.**

La fenêtre **« Memory »** ressemble alors à celle-ci
Memory est divisé en trois parties, à gauche les adresses, au centre les valeurs en hexadécimal, à droite les valeurs en ASCII.

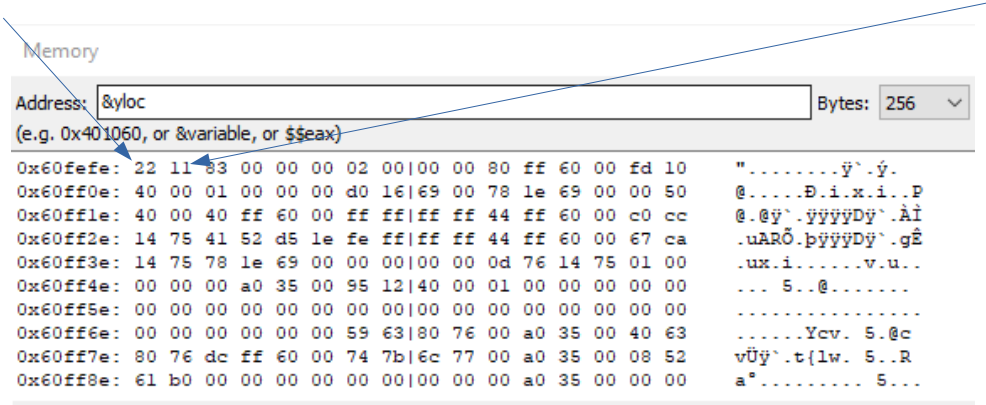


Les adresses sont codées sur 32 bits donc quatre octets. L'octet de poids fort étant ici égale à 0, il n'est pas représenté, (0x405050 est en fait 0x00405050).
Les données sont affichées par groupes de 16 octets ainsi la première ligne affiche les contenus des adresses 0x405050 à 0x40405F
Les variables sont rangées l'une après l'autre mais « à l'envers » . Ce type de stockage s'appelle « little endian »

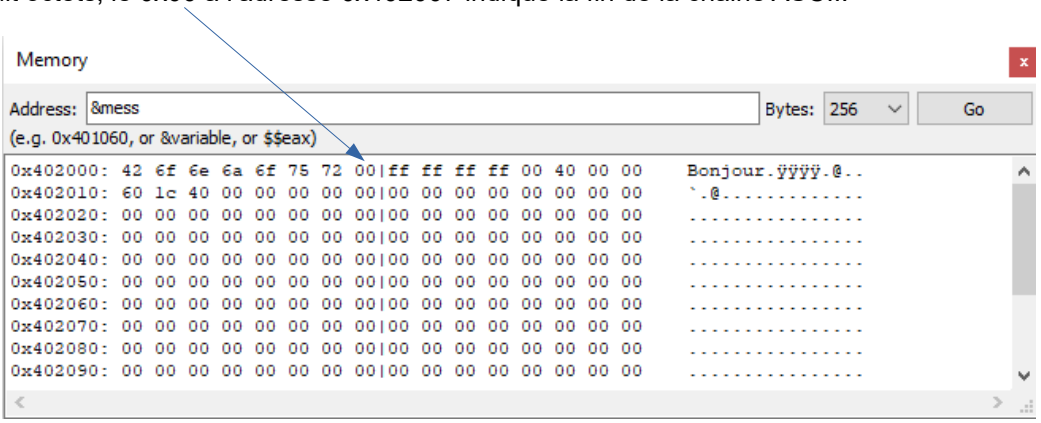


iglo est rangée à l'adresse 0x405050 (quatre octets) (iglo = 0xabcdef09)
jglo est rangée à l'adresse 0x405054 (quatre octets) (jglo = 0x12345678)

Dans la barre d'adresse de « Memory » entrer l'adresse de yloc, relever alors l'adresse et la valeur de xloc et yloc



Faire de même pour la chaîne de caractère mess, qui ici est rangée aux adresses 0x402000 à 0x402007, soit huit octets, le 0x00 à l'adresse 0x402007 indique la fin de la chaine ASCII.



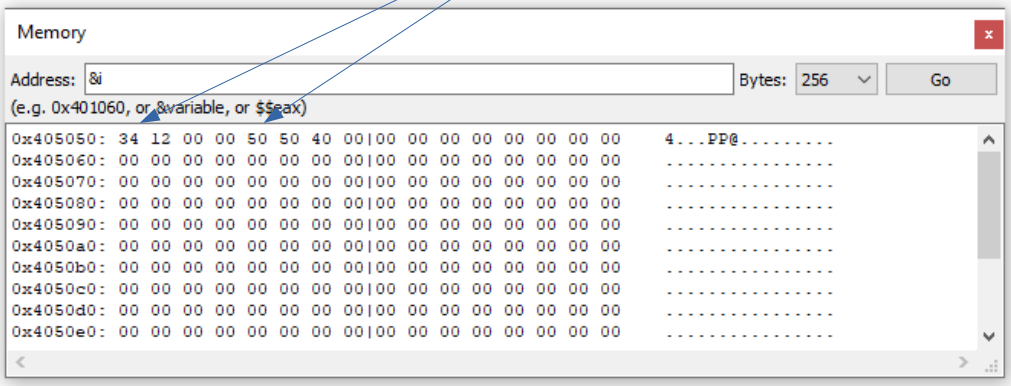
Les variables de type pointeurs en C/C++

Les pointeurs sont des variables qui désigne l'adresse d'une variable.
Les pointeurs ont toujours la même taille (la taille d'une adresse) mais peuvent pointer sur des variables de types différents.
On peut créer des pointeurs sur des char, int, float , tableau etc.…

Exemple :
soit p un pointeur sur des entiers courts (short 16 bits), pour le déclarer on ajoute une * devant le nom
Soit i un entier court (short 16bits)

Le programme ci-dessous permet avec le debugger de visualiser i et p

```
1  #include <stdio.h>
2
3  short int i;
4  short int *p;
5
6  int main(void)
7  {
8      i=0x1234;
9      p=&i;
10     printf("%p\n",p);
11     return 0;
12 }
13
```



i se trouve à l'adresse 0x00405050 et contient 0x1234 (codage little endian)
p se trouve à l'adresse 0x00405054. Ici une adresse est composée de quatre octets, donc p occupe quatre octets.
p contient 0x00405040 qui est bien l'adresse de i, on dit que **p pointe sur i.**

2 Exercice , prise en main

Dérouler le programme suivant en pas à pas, observer l'évolution des variables dans les fenêtres watches et memory.
Remarquer en particulier dans la fenêtre memory la recopie du message

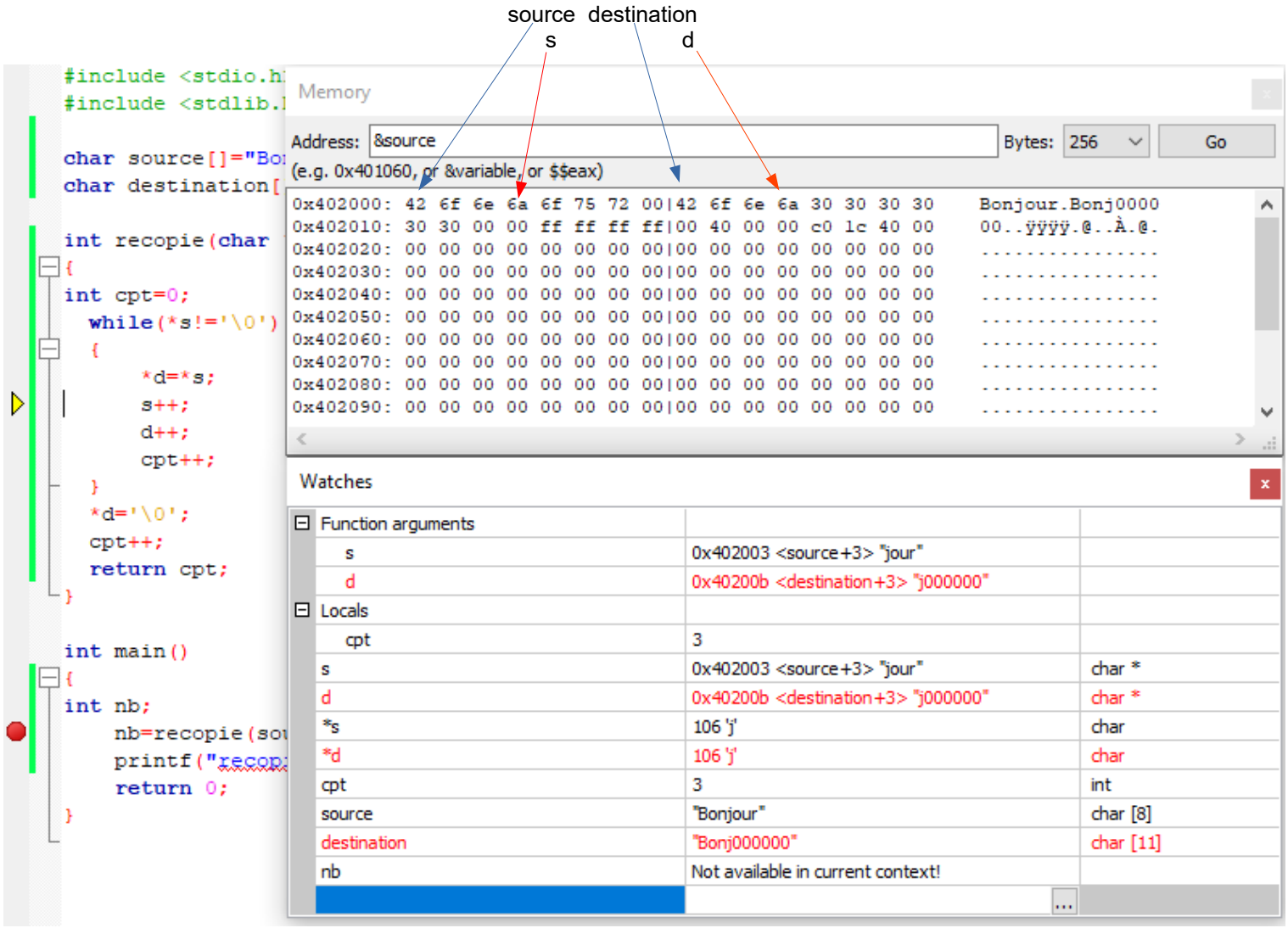
```
#include <stdio.h>
#include <stdlib.h>

char source[]="Bonjour";
char destination[]="0000000000";

int recopie(char *s, char *d)
{
int cpt=0;
  while(*s!='\0')
  {
      *d=*s;
      s++;
      d++;
      cpt++;
  }
  *d='\0';
  cpt++;
  return cpt;
}

int main()
{
int nb;
    nb=recopie(source,destination);
    printf("recopie de %d caracteres %s vers %s\n",nb,source,destination);
    return 0;
}
```

Le debugger CODE::BLOCK



s et d sont des pointeurs (des adresses) sur des char.
*s et *d représentent les contenus de ces adresses
Ici s pointe sur le caractère ASCII 0x6A 'j' du tableau source et d pointe sur le caractère ASCII 0x6A 'j' du tableau destination

3 Fonctionnalités avancées du debugger

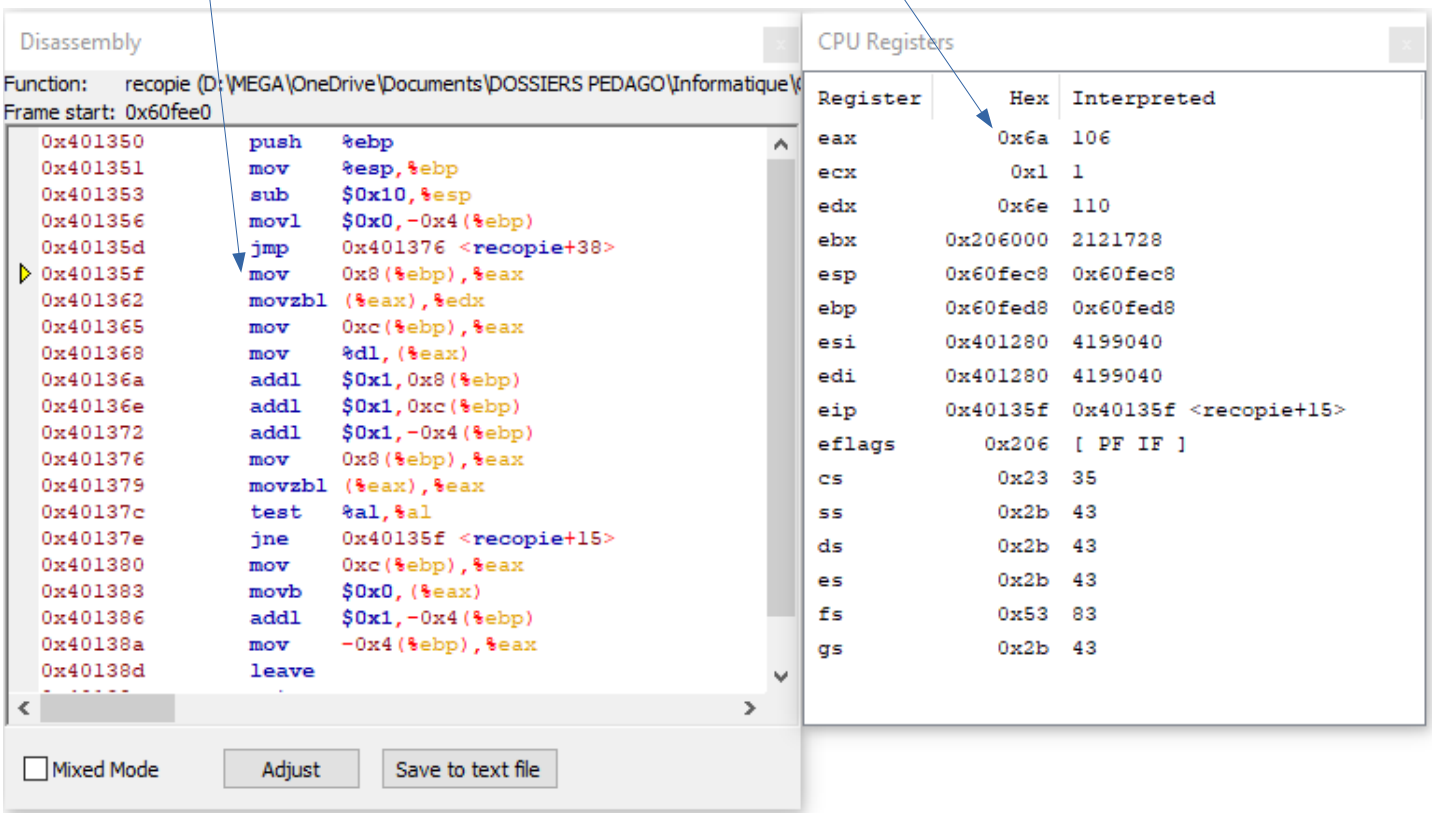
Le debugger permet d'observer et modifier le programme en profondeur, jusqu'au code machine.

Placer un point d'arrêt sur le while de la fonction recopie.
Ajouter les fenêtres de debug Disassembly et CPU Registers.

Executer le programme jusqu'au point d'arrêt, la fenêtre Disasembly affiche le code de la fonction en langage assembleur et la fenêtre CPE Register l'état des registres du processeur.

Le pas à pas permet de visualiser le déroulement du programme et l'évolution des registres.
En observant les registres vous pourrez visualiser les passages des données des tableaux dans ces registres.

Exemple : chargement du code ASCII 0x65 soit 'j' dans le registre eax.



4 Gestion de la pile d'appel des fonctions.

Le programme ci-dessous possède une fonction qui calcule le factoriel d'un nombre.
(https://fr.wikipedia.org/wiki/Factorielle)

La fonction factorielle est récursive, elle s'appelle elle-même. Ce type de programmation est très efficace du point de vue de la taille du code mais dangereux du point de vue de l'occupation mémoire RAM. La fenêtre « Call stack » du debugger permet de visualiser les appels récursifs.

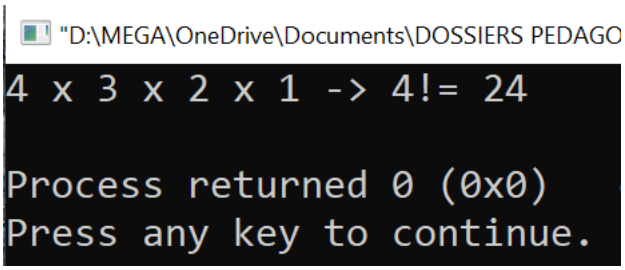
```
#include <stdio.h>
#include <stdlib.h>

unsigned long factoriel(unsigned int n)
{
  printf("%d ",n);
  if (n == 1 || n == 0) return 1;
  printf("x ");
  return(n*factoriel(n – 1));  // appel recursif ici
}

int main()
{
unsigned int nb;
unsigned long fac;
    nb=4;
    fac=factoriel(nb);
    printf("-> %d!= %ld\n",nb,fac);
    return 0;
}
```

Call stack

Nr	Address	Function	File	Line
0		factoriel (n=1)	D:\M...	8
1	0x401377	factoriel(n=2)	D:\M...	7
2	0x401377	factoriel(n=3)	D:\M...	7
3	0x401377	factoriel(n=4)	D:\M...	7
4	0x4013a3	main()	D:\M...	15



Placer un point d'arrêt sur nb=4 ; et dérouler le programme en pas à pas afin d'observer l'évolution des appels récursifs.