UNIVERSITÉ PARIS 7 - DENIS DIDEROT

Mise à Niveau Langage C III

- Entrées / Sorties
- Passage d'arguments
- Instructions au pré-processeur
- Débugger son programme source

Entrées / Sorties

```
#include <stdio.h>

fopen()   /* Ouverture d'un fichier   */
fprintf() /* Ecriture dans un fichier */
fscanf()  /* Lecture dans un fichier  */
fclose()  /* Fermeture d'un fichier   */

void readFile( char * nom, int * a, int n ){

   FILE * inFile = fopen( nom, "r");
   int i = 0;

   for( i = 0 ; i < n ; i++ ){
        fscanf( inFile, "%d\n", &(a[i]) );
   }
   fclose(inFile);
}
```

Passage d'arguments

```
int main(void){

  blah, blah …
}
```

devient …

```
int main(int argc, char *argv[]){
  int i=0, k=0;

  printf("There are %d arguments
   in the command
   line.\n",argc);

  for( i=0 ; i<argc ; i++ ){
    printf("%d %s\n",i,argv[i]);
  }

  k = atoi(argv[1]);

  printf("The first arg is an
   integer => %d\n",k);
}
```

`argc` : le nombre d'arguments

`argv` : un tableau de chaînes de caractères

NB : les arguments sont des chaînes de caractères, même si elles contiennent des valeurs réelles, entières …

`atoi` : convertit une chaîne ASCII en entier

`atof` : convertit une chaîne ASCII en float

En savoir plus : *`man atoi`*

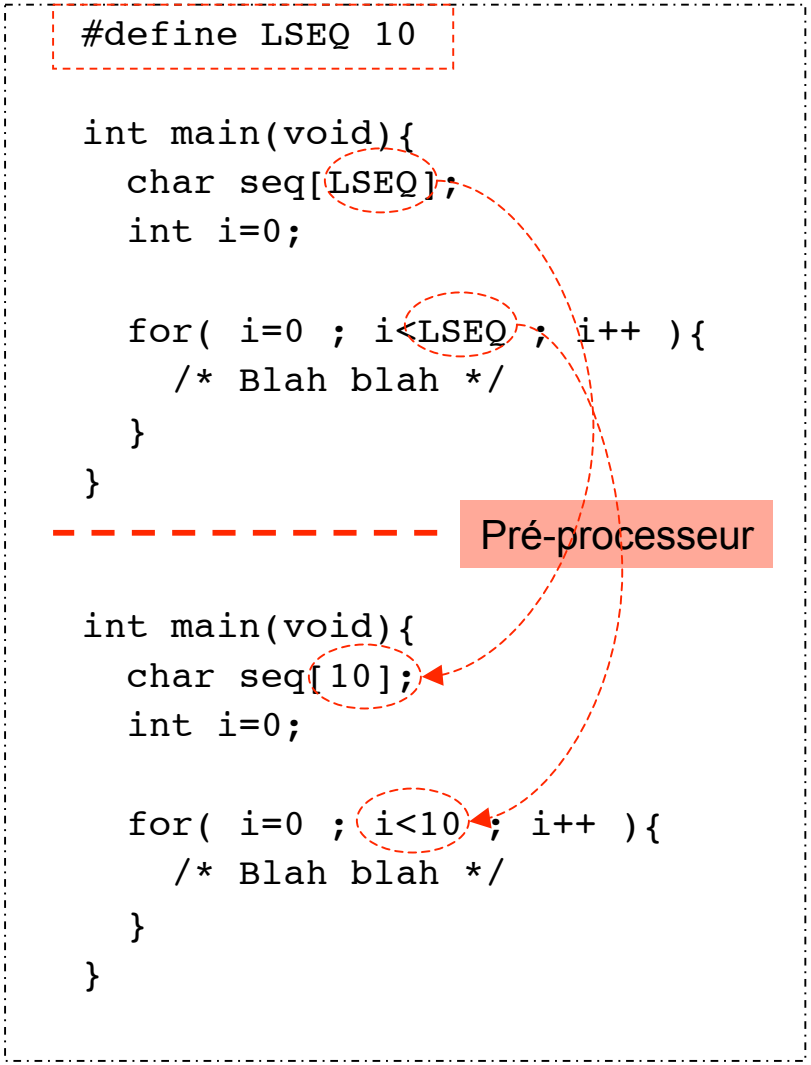
Les instructions au pré-processeur (I)

COMPILATION = 3 étapes

- pré-compilation : ajouter des informations (fichiers en-tête, remplacer les macro-constantes …)
- code-source -> code objet : passer en langage machine le(s) source(s)
- lier tous les objets qui donnent un exécutable final

Vous connaissez déjà :

```
#include <math.h>
```



Les instructions au pré-processeur (II)

- On peut faire des tests sur des macro-constantes. Si le test est vrai, le bloc de code compris entre `#ifdef` et `#endif` sera inclus dans l'exécutable final.

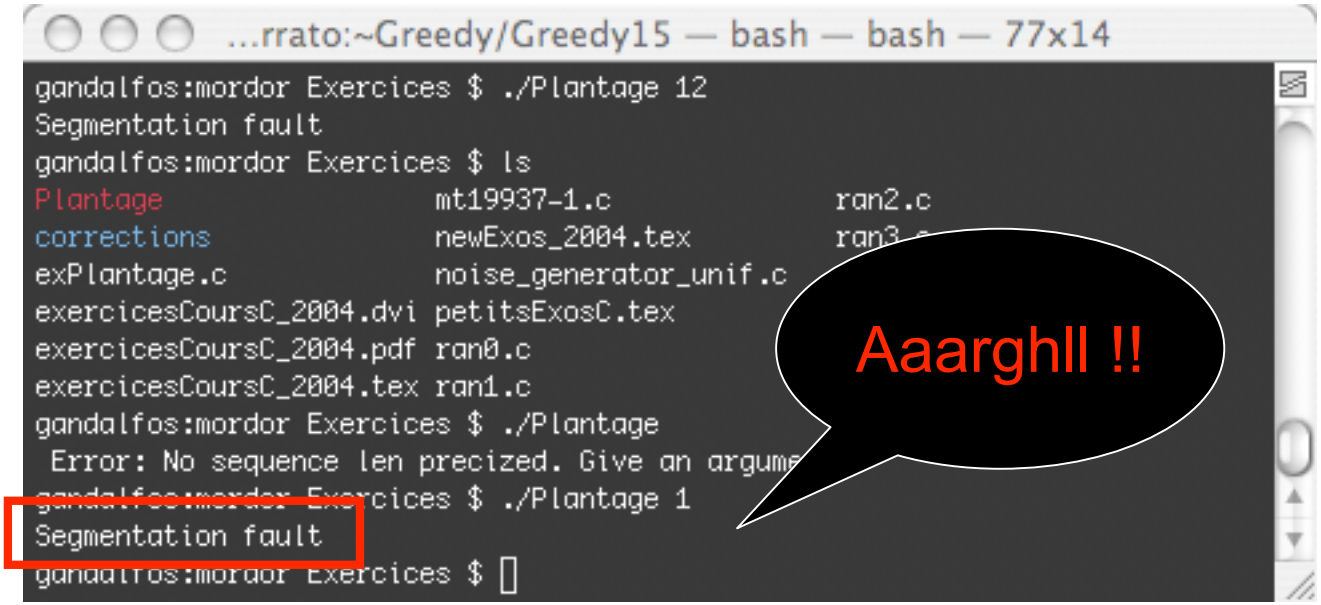
- On peut définir des macro-constantes en ligne de commande via les options de compilation de `gcc`.

```
#define DEBUG
/* #undefine DEBUG */

void myFunction(double x){

#ifdef DEBUG
  printf("[DEBUG: myFunction]
   blah, blah …\n");
#endif

  return exp(x*M_PI);
}
```

```
gcc toto.c -o Toto -D DEBUG
    ou
gcc toto.c -o Toto -D DEBUG=2
```

`-D NAME=DEFINITION`

Mon super programme plante …



D'où vient mon erreur ?

- Ecriture à une adresse non autorisée (*pointeur non alloué, indice d'un tableau > taille tableau, vous ne passez pas une adresse à fscanf() ou scanf() …*)
- Libération de mémoire non allouée par vous
- Appel à une mauvaise adresse mémoire (*printf() ou fprintf() qui tente d'afficher une chaîne alors que vous lui passez un caractère, lecture d'un flux qui n'existe pas car non ouvert …*)
- Appel à une fonction non déclarée ou mal déclarée (*compilation séparée …)*

Debugger son programme …

Plusieurs solutions :

- Afficher des infos :

```
printf("coucou1");
printf("myVar = %d",myVar);
```

- gdb : GnuDeBugger
  - multi-plateformes

- valgrind : uniquement pour Linux.
  - copie et analyse la mémoire du programme
  - exécution très ralentie
  - renvoie les lignes du source où il existe des erreurs et le type de l'erreur
  - ne laisse passer **aucune** erreur

```
valgrind --tool=memcheck commande
```

N'oubliez pas :

- options de compilation :
  - Wall
  - pedantic
  - g : pour gdb

- *`segmentation fault (core dumped)`* : génération d'un fichier `core` décrivant l'état de la mémoire pouvant être donné en entrée de gdb.

GDB : GNU DeBugger (I)

`syntaxe: gdb ./myProg core`

- `run` lancer l'exécution (NB: vous pouvez mettre les arguments, faire des redirections …) Ex:

```
run 1 toto.txt > log.txt
```

- `break` insérer des points d'arrêt dans l'exécution

```
(gdb) break myFunction
(gdb) run
```

marche aussi avec un numéro de ligne:

```
(gdb) break 66
```

- `info` obtenir des infos Ex:

```
(gdb) info break
```

retourne tous les points d'arrêt

- `delete` effacer un point d'arrêt

```
(gdb) delete 3
```

efface le 3ème break

- `enable / disable` activer / désactiver un point d'arrêt

- `if` ajouter une condition au point d'arrêt

(gdb) break myFunction if a==2

- `step` une fois l'exécution stoppée, si la ligne à exécuter contient une fonction à exécuter, `step` vous met en début de fonction

- `next` une fois l'exécution stoppée, si la ligne à exécuter contient une fonction à exécuter, `next` exécute la fonction

GDB : GNU DeBugger (II)

- `continue` poursuit l'exécution jusqu'au prochain point d'arrêt
- `where` affiche la ligne des appels de fonctions qui vous ont amené à l'endroit où le debugger vous a donné la main
- `list` affiche quelques lignes avant et après le point d'arrêt

```
(gdb) list
(gdb) list 40,45
(gdb) list main
```

- `print` affiche le contenu d'une variable

```
(gdb) print myVar
```

- `whatis, ptype` affiche le type d'une variable ; préférer ptype pour les structures

```
(gdb) whatis myVar
```

- `quit` quitter gdb

Bon courage !