R 2.04

2025 - 2026

Communication et fonctionnement bas niveau

TD N°1
« Langage C, les étapes du compilateur »

ANNE Jean-François
D'après le TD de F. BOURDON

Le but de ce TD est de se familiariser avec l'architecture bas niveau système et réseau.

« Langage C, les étapes du compilateur »

Notions vues dans ce TD :

Les différentes étapes du compilateur C " cc " ou " gcc ", la compilation conditionnelle, gdb et le désassemblage de code.

Nombre de séance de 2h00 prévu pour faire ce TD : 1,5.

Prochain TD : Déclaration et définitions de fonction : la compilation séparée. Les outils Makefile et la création de librairies statiques et dynamiques.

PS : Les parties correspondant à du travail à faire sont toutes en italiques ; le restant étant du complément au cours.

A). Compilation et préprocesseur (cpp)

Le but de ce TD est de regarder les différentes étapes du compilateur et le rôle qu'elles jouent dans la production du fichier binaire exécutable à partir d'un fichier " source " écrit dans le langage de programmation " C ". Par convention de tels fichiers " source " sont appelés avec l'extension ".c ".

Pour ce TD et les suivants, placez-vous à chaque TD dans un répertoire propre pour ce dernier. Il est important qu'il n'y ait pas d'autres fichiers que ceux du TD afin de bien voir et comprendre les fichiers créés par la chaîne de compilation.

I). Exercice n°1 :

Construire avec l'éditeur de texte " vi ", ou un autre éditeur, le programme **exemple_environ.c** suivant :

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

#define SIZE 100

extern char ** environ;

int main(void) {
  int i = 0;
  char nom[SIZE];
  char * variable;
  for (i = 0; environ[i] != NULL; i++)
    fprintf(stdout, "%d : %s\n", i, environ[i]);
  while (1) {
    fprintf(stdout, " entrez un nom de variable d'environnement, FIN
pour sortir : ");
```

```
        scanf("%s", nom);
        if (!strcmp(nom, "FIN")) break; // sortie de la boucle
        variable = getenv(nom);
        if (variable == NULL)
          fprintf(stdout, "%s : non definie\n", nom);
        else
          fprintf(stdout, "%s : %s\n", nom, variable);
      }
      printf("mon PID =%d, le PID du SHELL =%d\n", getpid(), getppid());
      return 0;
    }
```

- La variable *environ* est définie et chargée automatiquement (« extern ») lors du lancement d'un programme « C ». C'est un tableau de chaînes de caractères (char **) qui contient l'environnement (*variables*) du processus courant. Chaque chaîne a la forme « NOM=VALEUR » (sans blanc de part et d'autre du signe égal) et se termine par un caractère nul (« \0 »). Le tableau lui-même se termine par un pointeur nul (NULL).
- La fonction *getenv* permet de récupérer la valeur d'une variable d'environnement donnée. Il existe les fonctions *putenv* (fait appel à *setenv*) et *setenv* pour modifier la valeur d'une variable d'environnement, ainsi que la fonction *unsetenv*, pour supprimer une variable d'environnement. Les interfaces de ces fonctions sont (pour plus d'informations, regarder le manuel « *man* ») :

  int putenv (char* string) ;

  int setenv (const char* name, const char* value, int overwrite) ;

  int unsetenv (const char* name) ;

  Les fonctions *putenv* et *setenv* renvoient 0 si elles réussissent, ou -1 s'il n'y a pas assez de place pour créer une nouvelle variable. Le paramètre *overwrite* mis à 0 permet de configurer des variables que si elles n'ont pas été initialisées.

- Que fait ce programme ? Faites afficher par le programme les variables HOME, PATH et PWD.
- Dessinez la représentation en mémoire du tableau de chaînes de caractères *environ []*.

II). Exercice n°2 :

Dans la suite des exercices vous devrez observer les fichiers créés à chaque étape par le compilateur, ainsi que leurs droits d'accès. Au fur et à mesure de ces exercices, et sans vous appuyer sur le cours, vous dessinerez la chaîne de compilation avec ses différentes étapes et les fichiers produits associés.

- Compilez (appel au compilateur « cc » ou « gcc ») ce fichier en tapant :

  ```
  prompt> gcc exemple_environ.c
  ```

  On obtient un fichier exécutable nommé **a.out** que l'on peut faire exécuter en tapant son nom. Lancez ce programme.

- Faites :

  ```
  prompt> rm a.out
  ```

- Utilisez la commande suivante afin de n'effectuer que la phase de précompilation.

  ```
  prompt> gcc -E exemple_environ.c
  ```

- Sauvegardez le résultat de la commande précédente dans un fichier « trace.i ».

- Créez le fichier en assembleur *exemple_environ.s* à partir du programme source *exemple_environ.c* en tapant la commande suivante. Vous pouvez examiner ce fichier.

```
prompt> gcc exemple_environ.c -S

prompt> ls exemple_environ.*
```

- Créez le fichier objet *exemple_environ.o* (langage binaire) à partir du fichier assembleur *exemple-environ.s* en tapant :

```
prompt> gcc exemple_environ.s -c

prompt> ls exemple_environ.*
```

  Vous obtiendrez le même résultat avec la commande suivante :

```
prompt> gcc exemple_environ.c -c
```

- Créez un programme exécutable à partir du programme objet en tapant :

```
prompt> gcc exemple_environ.o -o exemple_environ
```

  Vous obtiendrez le même résultat avec la commande suivante :

```
prompt> gcc exemple_environ.c -o exemple_environ
```

- Observez les fichiers créés à chaque étape par le compilateur, ainsi que leurs droits d'accès.
- Utilisez la commande nm pour lister les symboles contenus dans les fichiers objets et exécutables.

```
prompt> nm exemple_environ.o

prompt> nm exemple_environ | grep ' T '

prompt> nm exemple_environ | grep U
```

III). Exercice n°3 :

- Modifiez le programme « exemple_environ » en mettant en commentaire (« // ») les instructions :

```
#include <stdlib.h> et

#include <string.h>
```

- Compilez le programme avec l'option « -Wall » pour obtenir un exécutable. Qu'observez-vous ? La compilation a-t-elle fonctionné (comparez les dates des fichiers) ? Analysez les messages renvoyés par le compilateur. De quelle nature sont-ils ? Quelle est leur signification ?
- Modifiez à nouveau le programme « exemple_environ.c » en retirant les commentaires aux lignes 2 et 3, mis à la question précédente, et en mettant cette fois la première ligne en commentaire :

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

- Compilez le programme. Qu'observez-vous ? La compilation a-t-elle fonctionné ? Expliquez.

IV). Exercice n°4 :

- Nous allons modifier « exemple_environ.c », dans un nouveau fichier « exemple_environ2.c ». Nous allons ajouter la possibilité de modifier les valeurs de certaines variables d'environnement du processus courant. Pour cela vous ferez une boucle (*while*) dans laquelle, à chaque itération, vous

demanderez le nom de la variable d'environnement à changer, puis grâce à un deuxième *scanf()* vous récupèrerez la nouvelle valeur de la variable d'environnement choisie. Le changement effectif de la valeur de la variable sera fait en utilisant la fonction *setenv()*. Pour cela reportez-vous au manuel. Cette boucle sera placée entre la fin du *for* et le début du *while* du programme « exemple_environ.c ». Vous appliquerez votre programme à la modification d'une variable existante, mais aussi à une nouvelle variable. Vous testerez votre programme avec le paramètre overwrite de setenv, mis à 0 puis à 1. Que constatez-vous ?

- Vous pourrez proposer une deuxième version de votre programme en utilisant la fonction *putenv()* à la place de la fonction *setenv()*. Vous regarderez le manuel pour la définition de cette fonction. Elle nécessitera l'usage de fonctions telle que *strcpy()* et *strcat()*.

V). Exercice n°5 :

- Le préprocesseur possède des directives (#) lui permettant de faire des prétraitements sur le fichier source. On peut ainsi définir des pseudo-constantes (#define), des pseudo-fonctions ou macros (#define). On peut aussi inclure des fichiers (#include). On peut enfin faire de la compilation conditionnelle (#ifdef, #else, #endif). C'est ce dernier point que nous allons étudier ici.

- Utiliser les directives de la compilation conditionnelle pour compiler votre programme « exemple_environ2.c » avec ou sans appels à la fonction « printf » pour tester toutes les variables saisies ou retournées dans votre programme. Vous utiliserez la pseudo-constante DEBUG suivant deux possibilités. La première consistera à définir ou non (**#define DEBUG**) cette pseudo-constante dans votre code source. Lorsqu'elle sera définie, vous devez appeler les « printf », sinon vous devez ne pas les appeler. La deuxième solution consistera à compiler votre code avec ou sans l'option « -D**DEBUG** », ce qui reviendra à définir ou non la pseudo-constante DEBUG :

prompt> gcc exemple_environ2.c -DDEBUG -o exemple_environ2

- Expliquez la différence entre les deux méthodes. Après avoir réalisé avec succès les deux méthodes, recompilez votre programme avec l'option « -E » et sans l'option « -o ». Observez le résultat fourni lorsque la pseudo-constante DEBUG est définie et lorsqu'elle ne l'est pas.

Exemple de programme à utiliser :

```
prompt> cat def.h
#ifdef DEBUG
    #define trace(s) printf s#
else
    #define trace(s)
# endif

prompt>

prompt> cat source.c
#define DEBUG
#include "def.h"
#include <stdio.h>

int main()
{
     float r;
     printf("entrez un nombre réel :");
     scanf("%f", &r);
```

```
        trace(("valeur du nombre entré : %f\n", r));
        printf("fin de programme\n");
    }
```

```
prompt>
```

- Regarder la différence de sortie du programme entre l'utilisation du #Define DEBUG et l'option de compilation avec et sans DDEBUG.

B). Utilisation du débogueur symbolique GDB

Cette partie consiste à utiliser l'outil *gdb* afin de voir comment, d'une manière générale, on peut suivre l'exécution pas-à-pas des programmes et explorer l'état des variables, voire les modifier en cours d'exécution.

Avant de pouvoir utiliser gdb il faut compiler votre programme avec l'option -g. Cela permet au compilateur de générer les informations nécessaires au débogage symbolique. Ensuite il suffit de lancer *gdb* avec le nom de votre programme en paramètre et l'option -q pour ne pas voir s'afficher les commentaires liés au démarrage du programme :

```
prompt> gdb prog_test -q

 (gdb)

 …
```

Le débogueur possède un interprète de commandes qui attend que vous lui donniez l'une de ses commandes à exécuter. Pour connaître ces commandes, vous pouvez utiliser le manuel (man). Le principe de fonctionnement de gdb est d'explorer l'exécution d'un programme en l'arrêtant sur des points d'arrêt (break point), afin de pouvoir consulter le contenu des variables, modifier le contenu de ces variables ou encore voir l'état de la pile (stack), correspondant aux différents appels successifs à des fonctions. On peut faire avancer cette exécution ligne par ligne (step/next). Mettre des points d'arrêt (break) fait partie des commandes disponibles dans gdb.

Utiliser le programme prog_test.c :

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

// Fonction qui affiche les éléments d'un tableau
void print_numbers(int *arr, int size) {
    for (int i = 0; i <= size; i++) { // Erreur intentionnelle : dépassement du
tableau
        printf("arr[%d] = %d\n", i, arr[i]);
    }
}

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);
    int a = 10, b = 0;

    printf("Début du programme\n");

    // Appel de la fonction avec un tableau
    print_numbers(numbers, size);
```

```
    // Division par zéro
    printf("Calcul d'une division...\n");
    int c = a / b; // Erreur : division par zéro
    printf("Résultat de la division : %d\n", c);

    printf("Fin du programme\n");
    return 0;
}
```

Exercices :

1. **Points d'arrêt et exploration des variables :**
   - Ajouter un point d'arrêt dans main avant l'appel à print_numbers.
   - Examiner le contenu des variables numbers et size.
2. **Dépassement du tableau :**
   - Suivre la boucle for de la fonction print_numbers pas à pas.
   - Identifier l'erreur de dépassement de tableau lorsque i == size.
3. **Gestion d'une exception (division par zéro) :**
   - Intercepter l'erreur sur la ligne de division par zéro.
   - Modifier la valeur de la variable b pour éviter l'erreur et continuer l'exécution.
4. **Affichage et modification des variables en cours d'exécution :**
   - Afficher les valeurs des variables locales et globales.
   - Corriger une variable directement dans gdb.

I). Exercice n°6

Rentrez le programme addressof.c, ci-dessous, à l'aide d'un éditeur de texte. Ce programme comporte deux erreurs que nous allons trouver avec gdb et l'usage du fichier core.

```
prompt> cat addressof.c


#include <stdio.h >   // 1
#include <stdlib.h >  // 2
#include <unistd.h >  // 3

int main() // 4
{
      char *pointeur = NULL;      // 5
      //char caractere='\0';      // 6
      unsigned int int_var = 5;   // 7
      unsigned int int_div = 0;   // 8
      unsigned int *int_ptr;      // 9

      int_ptr = &int_var;   // 10
```

```
        printf("int_ptr = %p\n", int_ptr);    // 11
        printf("&int_ptr = %p\n", &int_ptr);  // 12
        printf("*int_ptr = %d\n\n", *int_ptr);     // 13
        printf("int_var est localisee a %p et contient %d\n", &int_var,
int_var);  // 14
        printf("int_ptr est localisee a %p, contient %p, et pointe vers
%d\n\n", &int_ptr, int_ptr, *int_ptr);     // 15
        int_var = int_var / int_div;    // 16
        printf("int_var vaut %d\n", int_var); // 17
        fprintf(stderr, "Avant ecriture dans le pointeur\n"); // 18
        pointeur[0] = 'X';    // 19
        fprintf(stderr, "Apres ecriture dans le pointeur\n"); // 20
        return EXIT_SUCCESS;  // 21

}

prompt>
```

Compilez le programme addressof.c avec l'option -g pour pouvoir utiliser gdb. Lancez le binaire.

```
prompt> gcc  -g addressof.c -o addressof

prompt> ./addressof

int_ptr = 0x7ffd21e53af0

&int_ptr = 0x7ffd21e53af8

*int_ptr = 5


int_var est localisee a 0x7ffd21e53af0 et contient 5

int_ptr est localisee a 0x7ffd21e53af8, contient 0x7ffd21e53af0, et
pointe vers 5

Exception en point flottant (core dumped)

prompt>
```

Que constatez-vous ? Cette exécution a-t-elle générée un fichier core à votre nom dans le répertoire :

```
/var/lib/apport/coredump/
```

Lancez la commande ulimit -a dans votre terminal et observez la ligne core file size. Si la valeur sur cette ligne vaut 0, cela signifie que les fichiers core ne peuvent être créés. Dans ce cas il faut lancer la commande ulimit -c unlimited. Vérifier à nouveau la ligne core file size.

```
prompt> ulimit -a

…

core file size              (blocks, -c) 0

…
```

```
prompt> ulimit -c unlimited

prompt> ulimit -a

…

core file size              (blocks, -c) unlimited

….

prompt>
```

Relancez votre binaire et faites ls -lh afin de vérifier si un fichier core a bien été créé, dans le répertoire :

```
/var/lib/apport/coredump/
```

P.S. : *Si votre fichier Core n'a pas été créé, vérifiez que vous n'avez pas de caractères accentués (ou d'espaces) dans l'arborescence de votre fichier C.*

Pour trouver les erreurs (sémantiques et non syntaxiques) du programme, vous allez lancer gdb avec comme premier paramètre -q (pour éviter des commentaires inutiles), puis le nom de votre binaire et enfin le fichier core. Observez le message d'erreur transmis par gdb et en particulier le numéro de la ligne pointé. Placez un point d'arrêt (cf. la liste des commandes de gdb) sur cette ligne avant de lancer l'exécution du programme. En utilisant la commande p pour visualiser le contenu des variables identifiez le problème soulevé par gdb. Corrigez ce problème en modifiant (commande set) la valeur de la variable qui pose problème. Puis allez à l'instruction suivante (commande n). Assurez-vous que le problème a été évité. Continuez l'exécution pas-à-pas (commande n) de votre programme. Que se passe-t-il (deuxième erreur de programmation) ?

Sortez de gdb, corrigez dans votre fichier source la première erreur, sauvegardez le fichier, et relancez une exécution de votre programme hors gdb. Vous pouvez constater qu'un nouveau fichier core vient d'être créé. Il concerne la deuxième erreur. Reprenez la même méthode que pour la première erreur avec gdb. Vous corrigerez le deuxième problème dans gdb en utilisant la commande set. Une fois la terminaison de votre programme normalement atteinte (sans erreur), vous corrigerez cette deuxième erreur dans le fichier source et relancerez une exécution sans gdb qui devrait bien fonctionner.

II). Exercice n°7 :

Nous allons cette fois utiliser gdb pour désassembler du code C vers le langage d'assemblage (assembleur).

Entrez le programme boucle.c suivant avec un éditeur de texte.

```
prompt> cat boucle.c
#include <stdio.h>

int main()
{
      int i;
      for (i = 0; i < 10; i++)
            printf("Hello World!\n");
```

```
}
prompt>
```

Compilez ce programme avec l'option -g et lancez-le dans gdb. Affichez les lignes du programme (commande l) puis désassemblez le code (commande disas main). Essayez de voir la correspondance entre le code écrit en C et le code assembleur correspondant à la machine.

I. Webographie :

- https://bourdon.users.info.unicaen.fr/cours/IUT-1A/index.html
- https://askubuntu.com/questions/966407/where-do-i-find-the-core-dump-in-ubuntu-16-04lts
- https://qastack.fr/programming/17965/how-to-generate-a-core-dump-in-linux-on-a-segmentation-fault
- https://c.developpez.com/cours/mode-emploi-gcc/