R 2.04

2023 - 2024

Communication et fonctionnement bas niveau

TD N°2

« Déclaration et définitions de fonction : la compilation séparée »

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.

« Déclaration et définitions de fonction : la compilation séparée »

Notions vues dans ce TD :

Compilation séparée, librairies statiques et dynamiques.

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

Prochain TD : Pointeurs et passage des paramètres par valeur.

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

1°). La découpe en fichiers

Quand on commence à écrire de grands programmes C, il est utile de découper ces programmes en plusieurs fichiers. En effet certaines parties sont communes à plusieurs applications. Il est donc intéressant de mettre certaines fonctions dans un même fichier.

On peut imaginer deux stratégies pour intégrer ces fonctions au programme principal.

- La première consiste à inclure dans le programme principal tout le code C des fonctions et à tout recompiler. Cette méthode est coûteuse. Par exemple pour un programme utilisant les fonctions de gestion de l'écran de X-Windows cela demanderait plusieurs heures de compilation.

- La seconde méthode utilise les propriétés de l'éditeur de lien. On procède en plusieurs étapes. On compile les fichiers séparément avec l'option **-c**. Le compilateur crée un fichier **.o** appelé fichier *objet*. Le programme principal est compilé normalement comme si toutes les fonctions étaient définies. Dans une seconde étape on lie les fichiers objets au programme principal. Si l'on ne précise rien le compilateur fait automatiquement la compilation et le lien en se basant sur les extensions des fichiers.

  La commande : **gcc principal.c fonction.o fonction2.o -o principal** demande de compiler le programme *principal.c* avec le compilateur *gcc*, puis de faire le lien avec les fichiers *fonction.o* et *fonction2.o* et de créer un exécutable appelé *principal*.

Bien entendu la fonction *main* doit être uniquement définie dans le fichier *principal.c*, car il s'agit d'indiquer par cette étiquette « main » où se trouve la première instruction à exécuter.

La compilation séparée permet de réutiliser des morceaux de codes (modules objets) dans différents programmes. On peut regrouper des morceaux dans ce que l'on appelle des « librairies » ou encore des « bibliothèques ». Comme nous l'avons vu en cours il existe des librairies statiques et des librairies dynamiques (partageables). Ce sera l'objet de la deuxième partie des exercices proposés.

Pour avoir des informations sur les fonctions comprises dans un fichier objet (exécutable ou non) on peut utiliser la commande Unix **nm**, ou d'autres commandes comme **ldd** et **objdump** dans le cas des librairies dynamiques. Voici un exemple de compilation séparée :

Fichier : Principal.c

```
int main()
{
    printf("programme principal\n");
    fct1();
}
```

Fichier : fonction.c

```
void fct1()
{
    printf("fonction1\n");
    fct2();
}
```

Fichier : fonction2.c

```
void fct2()
{
    printf("fonction2\n");
}
```

Le programme « principal » affiche une chaîne de caractères puis appelle la fonction « fct1 » qui appelle à son tour la fonction « fct2 ». Ces deux fonctions affichent une chaîne de caractères. On compile les fichiers « fonction.c » et « fonction2.c » avec l'option **-c** pour obtenir deux fichiers objets. La commande **nm** va alors donner le résultat suivant (voir le manuel, commande « man nm», pour savoir à quoi correspondent les lettres) :

prompt> nm fonction.o

00000000 t ___gnu_compiled_c
0000000c T _fct1
U _fct2
U _printf
00000000 t gcc2_compiled.

prompt>

Cet objet contient deux variables qui indiquent que l'objet a été créé avec le compilateur **gcc** (*gnu*). Ce programme contient une fonction qui a été définie (« T »). Il s'agit de « fct1 » et deux fonctions (« fct2 » et « printf ») qui ont été utilisées sans être définies (« U »).

Le fichier objet « fonction2.o » est à peu près identique :

prompt> nm fonction2.o

```
00000000 t ___gnu_compiled_c
0000000c T _fct2
U _printf
00000000 t gcc2_compiled.
```

```
prompt>
```

Le fichier objet de l'exécutable principal contient :

```
prompt> nm principal
```

```
00004000 d __DYNAMIC
000040c0 B ___CTOR_LIST__
000040c8 B ___DTOR_LIST__
000023a8 T ___do_global_ctors
00002330 T ___do_global_dtors
00002290 t ___gnu_compiled_c
000022d8 t ___gnu_compiled_c
00002420 t ___gnu_compiled_c
00002330 t ___gnu_compiled_c
00002308 t ___gnu_compiled_c
000023f4 T ___main
000040b8 D __exit_dummy_decl
000040b0 D __exit_dummy_ref
00002330 t __main.o
000040c0 D _edata
000040d0 B _end
000040a8 D _environ
00002768 T _etext
00002420 t _exit.o
000022e4 T _fct1
00002314 T _fct2
000040b4 d _initialized.6
000022a8 T _main
00002290 t cca006231.o
00002020 t crt0.o
000022d8 t fonction.o
00002308 t fonction2.o
00002308 t gcc2_compiled.
00002330 t gcc2_compiled.
00002290 t gcc2_compiled.
000022d8 t gcc2_compiled.
00002420 t gcc2_compiled.
00002020 T start
```

On retrouve dans ce fichier les appels des fonctions « main », « fct1 » et « fct2 ». L'appel à la fonction « printf » n'apparaît pas dans ce listing car l'édition utilise des liens dynamiques (mécanisme par défaut) pour les librairies systèmes.

(Cette partie a été extraite d'un cours réalisé par les enseignants de l'Institut National des Télécommunications - INT – d'Evry).

2°). Déclaration de fonctions

- ***Rappel :***

Le mot-clé « **extern** » pour une fonction est inutile, car il signifie que la fonction est externe à la fonction en cours ; ce qui est impossible, car on ne peut pas définir une fonction à l'intérieur d'une autre fonction.

Le mot-clé « **static** » signifie que la fonction n'est définie qu'à l'intérieur du fichier où précisément elle a été définie.

L'outil « make » et son fichier de configuration « Makefile »

- ***Rappel :***

L'utilitaire « **make** » permet de réaliser la compilation séparée de façon automatique (en s'appuyant sur les dates d'accès aux fichiers), en utilisant un fichier de configuration appelé « **Makefile** ».

cible : dépendances commandes

- *La cible indique le but désiré, par exemple le nom du binaire exécutable ;*
- *Les dépendances mentionnent tous les fichiers dont la règle a besoin pour s'exécuter ;*
- *Les commandes précisent comment obtenir la cible à partir des dépendances (ATTENTION : cette ligne commence impérativement par une tabulation).*

```
prompt> make cible
```

Par défaut, sans « cible », la commande « make » traite le fichier « Makefile » au début.

a) Exercice 1 :

Soit les fichiers suivants :

```
prompt> cat monapp.c
#include "a.h"
#include "b.h"
#include <stdlib.h> // bibliothèque incluant « exit »

int main()
{
      printf("monapp\n");
      f2();
      f3();
      exit(0);
}
prompt>


prompt> cat f2.c
#include "b.h"
```

```
#include "c.h"
#include "e.h"

void f2()
{
      printf("f2\n");
      f21();
      f4();
}
```

```
prompt>
```

```
prompt> cat f3.c
```

```
#include "b.h"
#include "e.h"
#include "f.h"

void f3()
{
      printf("f3\n");
      f31();
      f32();
      f4();
}
```

```
prompt>
```

```
prompt> cat f21.c
```

```
#include "b.h"

void f21()
{
      printf("f21\n");
}
```

```
prompt>
```

```
prompt> cat f31.c
```

```
#include "b.h"

void f31()
{
      printf("f31\n");
}
```

```
prompt>
```

```
prompt> cat f32.c
#include "b.h"

void f32()
{
      printf("f32\n");
}
prompt>


prompt> cat f4.c
#include "b.h"
#include "g.h"

void f4()
{
      printf("f4\n");
      f41();
      f42();
}
prompt>


prompt> cat f41.c
#include "b.h"

void f41()
{
      printf("f41\n");
}
prompt>


prompt> cat f42.c
#include "b.h"

void f42()
{
      printf("f42\n");
}
prompt>
```

- *Créez tous ces fichiers « source » (*.c) dans un nouveau sous-répertoire. Vous créerez également les fichiers « entête » (*.h) en mettant le contenu nécessaire pour une bonne compilation du projet. Notez que la fonction « printf » se trouve dans la librairie « stdio.h ».*
- *Dessinez l'arbre des dépendances entre tous ces fichiers.*

b) Exercice 2 :

- *Ecrivez le fichier « Makefile » (vous pouvez utiliser l'option « -Wall » de « gcc ») utilisé par la commande « make » avec 10 entrées pour la compilation à proprement parler (monapp, monapp.o, f2.o, f3.o, f4.o ...), plus une onzième appelée « clean » pour détruire les fichiers objets (*.o) créés lors de la compilation précédente, ainsi que les fichiers de sauvegarde (*~).*
- *Après avoir lancé la commande « make », exécutez le binaire construit. L'exécution se passe-t-elle bien ?*
- *Faites la commande « make clean », puis la commande « make ». Observez les fichiers construits et les dates de création. Que se passe-t-il si vous refaite « make » ?*
- *Faites la commande « touch b.h » et lancez la commande « make ». Que se passe-t-il ? Pourquoi ?*
- *Faites la commande « touch e.h » et lancez la commande « make ». Que se passe-t-il ?*
- *Sauvegarder votre fichier « Makefile » dans « MakefileSIMPLE », puis utilisez et transformez le fichier « Makefile » donné dans la page des exercices, afin de recompiler votre application avec ce nouveau fichier qui calcule automatiquement les dépendances. Lancez la commande « make » et regardez les fichiers produits. En particulier comparer le contenu des fichiers de dépendances (*.d) avec le contenu de votre ancien « Makefile » renommé « MakefileSIMPLE ».*

c) Exercice 3 :

- *Modifiez uniquement la fonction « f2 », dans le fichier « f2.c », de telle façon qu'elle prenne deux paramètres, le premier étant un entier, alors que le second sera un réel (float) et qu'elle affichera (avec la commande « printf ») la valeur de ces deux paramètres. Lancez « make », puis le binaire exécutable produit. Que constatez-vous ?*
- *Modifiez le « main » (fichier « monapp.c ») de telle façon qu'il appelle la fonction « f2 » en lui passant l'entier « 2 » et le réel « 5.765 » en paramètre. Recompilez votre programme avec l'utilitaire « make » et lancez-le. Que constatez-vous ?*
- *Comment pourrait-on modifier le fichier « monapp.c » pour corriger le problème à l'affichage des paramètres dans « f2 » ? Pourquoi le problème ne se pose pas avec la fonction « f3 » ? Expliquez.*

Pour faire la suite des exercices, placez-vous dans un nouveau sous-répertoire.

d) Exercice 4 :

*La fonction « scanf » permet de récupérer dans un programme courant, et via le clavier, des données en provenance de l'utilisateur. Son prototype est le suivant : « int scanf (const char * format, ...) ». Le problème de cette fonction est qu'elle ne gère pas la mémoire, dans le sens où l'on peut déborder la mémoire sans que la fonction ne réagisse, avec potentiellement l'écrasement de données en mémoire.*

- *Créez un programme qui fait appel à la fonction « scanf » pour entrer une chaîne de caractères dans un tableau de 15 caractères que vous aurez préalablement réservé. Afficher à l'aide de la fonction « printf » le contenu de votre tableau après l'appel à la fonction « scanf ». Que se passe-t-il si vous avez entré un nombre de caractères (sans espaces et tabulations) supérieur à 15 ?*

e) Exercice 5 :

Pour corriger le problème de la fonction « scanf » qui vient d'être soulevé.

- *Vous allez écrire une fonction « saisie1 » qui prend :*
  - *Deux chaînes de caractères en paramètre : la question et la réponse. La première chaine sera utilisée dans « saisie1 » par un « printf » afin de poser une question. La seconde chaine permettra de stocker la réponse à la question ;*
  - *un entier : le nombre maximum de caractères imposés pour la réponse,*

- *donc avec le prototype « int saisie1 (char* question, char* réponse, int lgMaxRep) »*
- *et renvoie le nombre de caractères saisis (vous pourrez créer une fonction int longueur (char* réponse) pour compter le nombre de caractères de la réponse).*

➢ *Cette fonction « saisie1 » va appeler une autre fonction « saisieChaine » avec le prototype « int saisieChaine(char* réponse, int lgMaxRep) » qui demandera la réponse en comptant le nombre de caractères entrés, pour transférer cette réponse dans « saisie1 ».*

➢ *On utilisera la fonction « getc » ou la fonction « getchar ». Ces fonctions lisent un unique caractère « char » et le renvoie une fois converti en entier « int ». En cas d'échec, la valeur renvoyée est « \n » (fin de fichier, généralement définie comme étant égale à -1). On peut l'utiliser comme cela : « int i ; while ((i = getc(stdin))!= EOF) ... ».*

➢ *En résumé la fonction « saisie1 » permet de poser une question et de récupérer la réponse tout en s'assurant que la réponse ne dépasse pas la zone maximale allouée à cet effet. Pour récupérer cette réponse (chaîne de caractères) la fonction « saisie1 » appelle la fonction « saisieChaine » qui remplace la fonction « scanf » en vérifiant le nombre de caractères saisis grâce au troisième paramètre passé dans « saisie1 ».*

➢ *Ecrivez et compilez (option -c) ces deux fonctions « saisie1 » et « saisieChaine ».*

(voir annexes)

f) Exercice 6 :

➢ *Ecrivez un programme « questionnaire.c » qui boucle en appelant autant de fois que nécessaire (égal au nombre de questions) la fonction « saisie1 ». La taille maximale de la réponse (TAILLE_REP_MAX), le nombre de questions (NBR_QUESTIONS), les questions (Q1, Q2, ...) et le prototype de la fonction « saisie1 » seront définis dans un fichier « questionnaire.h » grâce aux instructions du préprocesseur (#define). Vous créerez également le fichier « saisie1.h », qui sera appelé dans le fichier « saisie1.c », dans lequel vous donnerez le prototype de la fonction « saisieChaine », ainsi que celui de la fonction « longueur ». Ces fichiers d'entête vous permettront de compiler l'application simplement, sans avoir recours à la fonction « make ».*

g) Exercice 7 :

➢ *Placez-vous dans un nouveau sous-répertoire dans lequel vous aurez recopié les fichiers de l'application « questionnaire ». Nous allons compiler l'application en incluant (statique) ou non (dynamique) dans notre binaire exécutable, les modules objets des librairies système utilisés.*

➢ *Compilez votre application en statique afin de produire le binaire de nom « questionnaire_sta ». Pour cela lancer la commande :*

```
prompt> gcc -static questionnaire.c saisie1.c saisieChaine.c -
o questionnaire_sta
```

➢ *Maintenant compilez votre application en dynamique afin de produire le binaire de nom « questionnaire_dyn ». Pour cela lancer la commande :*

```
prompt> gcc questionnaire.c saisie1.c saisieChaine.c -o
questionnaire_dyn
```

➢ *Que constatez-vous sur la taille des deux binaires exécutable ? Attention la compilation statique ou dynamique ne concerne ici que la façon dont sont liés les modèles des librairies système au binaire exécutable. Vous pouvez le constater en lançant les commandes suivantes :*

```
prompt> nm questionnaire_sta | grep sais

prompt> nm questionnaire_dyn | grep sais
```

- *Y-a-t-il une différence ? Expliquez.*
- *Par contre existe-t-il une différence en lançant les deux commandes suivantes ? Expliquez.*

```
prompt> nm questionnaire_sta | grep printf

prompt> nm questionnaire_dyn | grep printf
```

- *Pour voir les différences entre ces deux binaires vous pouvez utiliser la commande « ldd » qui liste les bibliothèques partagées/dynamiques d'un exécutable donné :*

```
prompt> ldd questionnaire_sta

prompt> ldd questionnaire_dyn
```

- *Pour voir les librairies dynamiques utilisées par votre binaire, vous pouvez utiliser la commande « objdump » :*

```
prompt> objdump questionnaire_sta | grep NEEDED

prompt> objdump questionnaire_dyn | grep NEEDED
```

h) Exercice 8 :

- *Nous allons maintenant créer une librairie partagée/dynamique pour stocker les modules objets « saisie1.o » et « saisieChaine.o ». Placez-vous dans un sous-répertoire dans lequel vous aurez préalablement recopié les fichiers sources (*.c) et les fichiers entête (*.h).*

  Pour cela il faut compiler le programme de la façon suivante :

```
prompt> gcc -c -fPIC saisie1.c saisieChaine.c
```

- *Puis créer la librairie partagée en y incluant les modules objets du projet :*

```
prompt> gcc -shared -o libSaisie.so saisie1.o saisieChaine.o
```

- *On peut produire le binaire exécutable :*

```
prompt> gcc questionnaire.c -o questionnaire libSaisie.so
```

- *Lancez le binaire produit. Que constatez-vous ? Exécutez la commande « ldd questionnaire ».*
- *Pour remédier au problème de la localisation de la librairie partagée, il faut modifier la variable d'environnement « LD_LIBRARY_PATH » et exporter cette variable modifiée :*

```
prompt> echo $LD_LIBRARY_PATH

prompt> LD_LIBRARY_PATH=`pwd`

prompt> echo $LD_LIBRARY_PATH

prompt> export LD_LIBRARY_PATH
```

- *Faites ces modifications et relancez le programme, puis la commande « ldd questionnaire ». Comparez et commentez la taille du fichier produit par rapport à celles des versions statique et dynamique de l'exercice précédent.*

i) Exercice 9 :

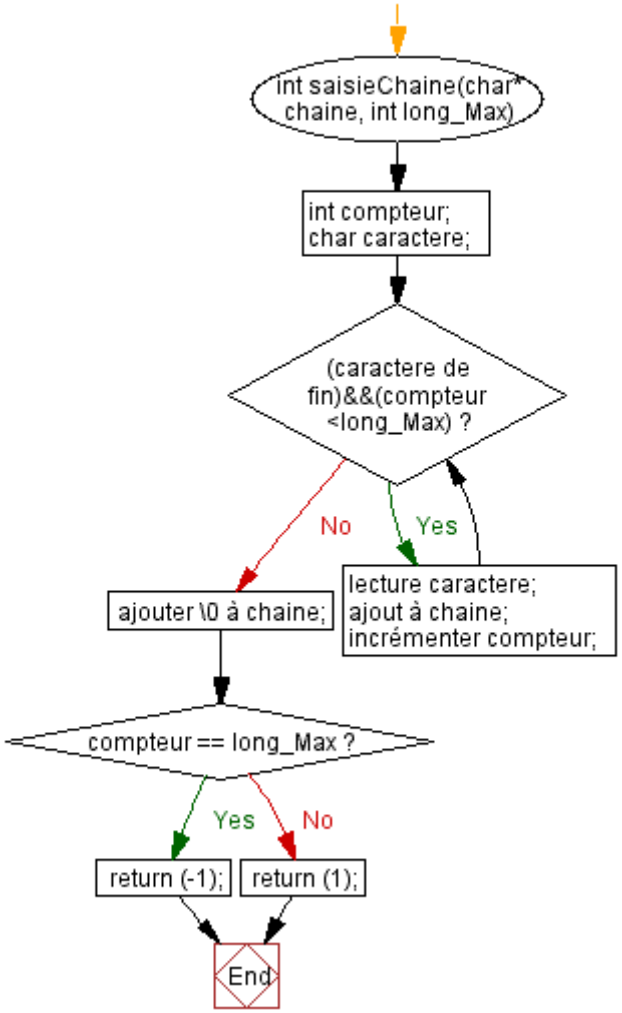
- *On peut également utiliser une librairie statique pour mettre les modules objet « saisie1.o » et « saisieChaine.o ». Pour cela il faut utiliser la commande d'archivage « ar ».*

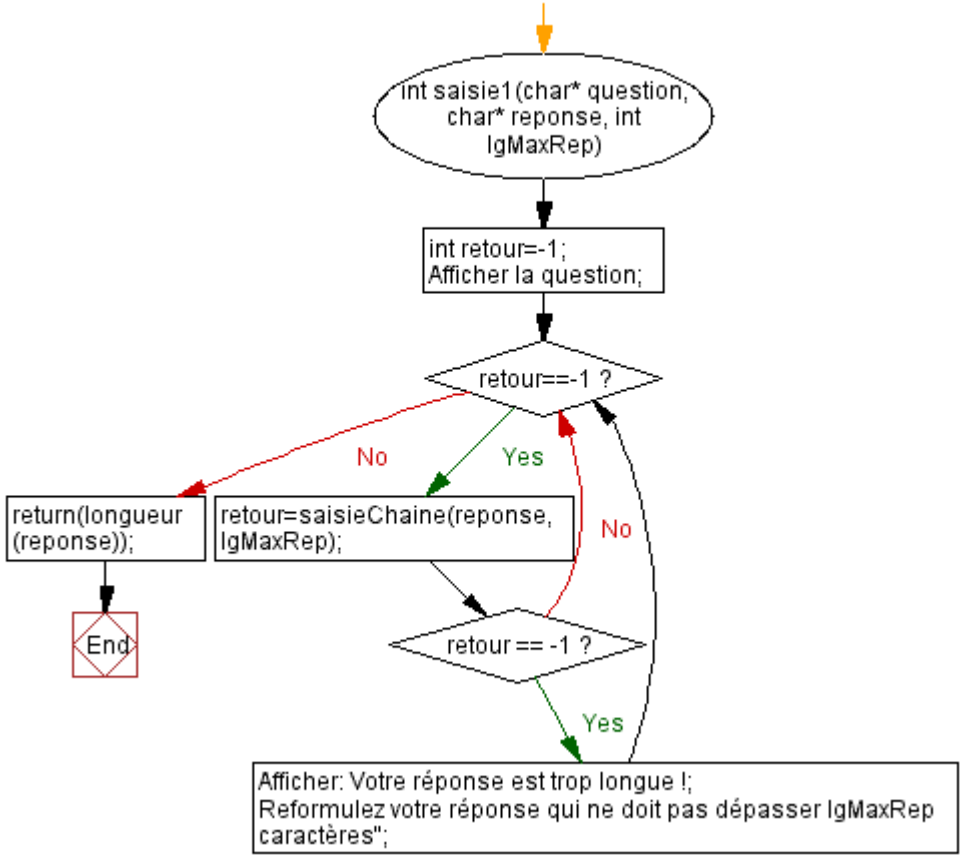
- *Toujours dans le même répertoire construisez une librairie statique avec les deux modules objet « saisie1.o » et « saisieChaine.o ». Pour cela faites les commandes suivantes :*

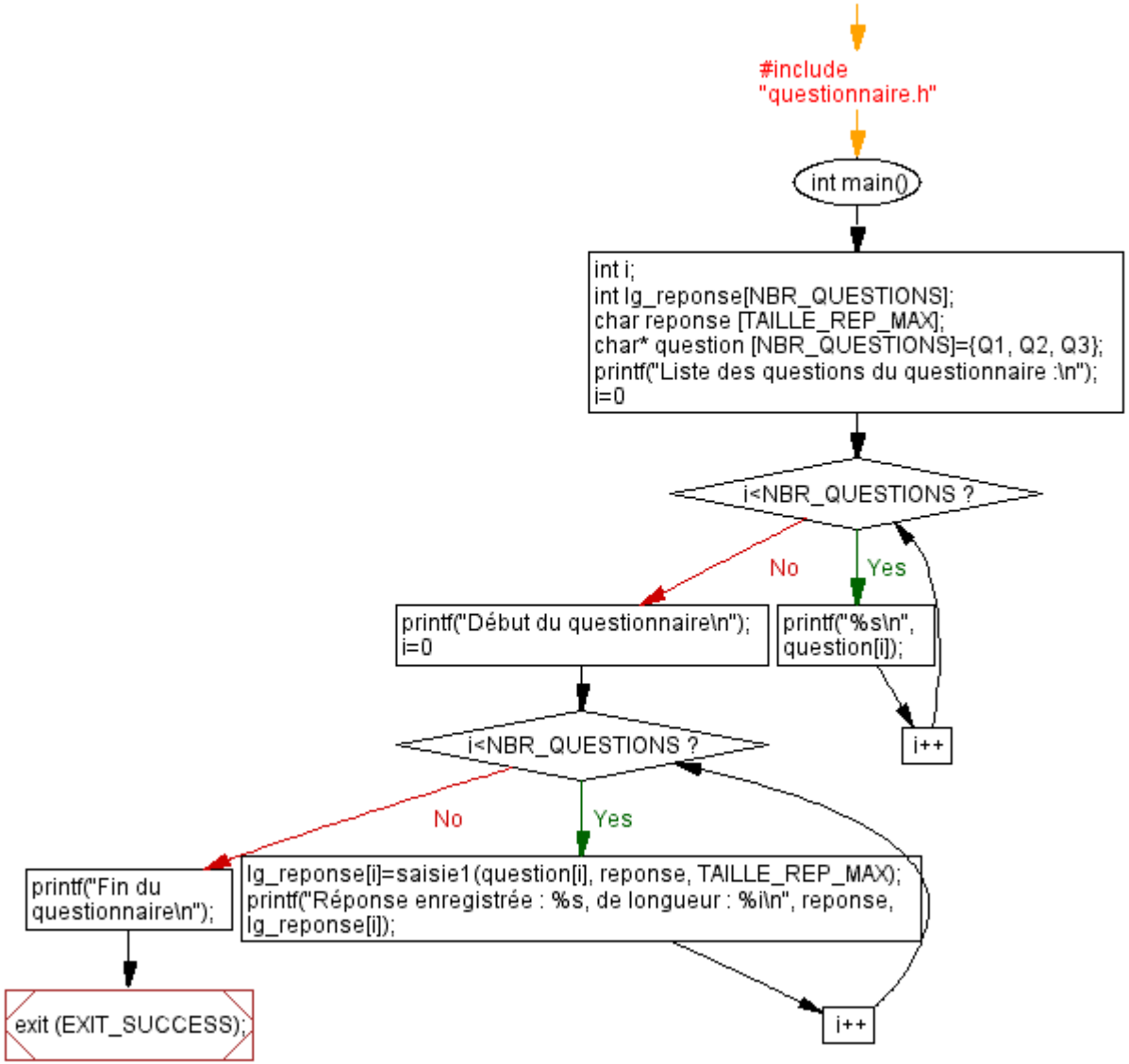
```
prompt> ar r libSaisie.a saisie1.o saisieChaine.o

prompt> ranlib libSaisie.a

prompt> rm *.o

prompt> gcc -static questionnaire.c libSaisie.a -o
questionnaire_sta
```

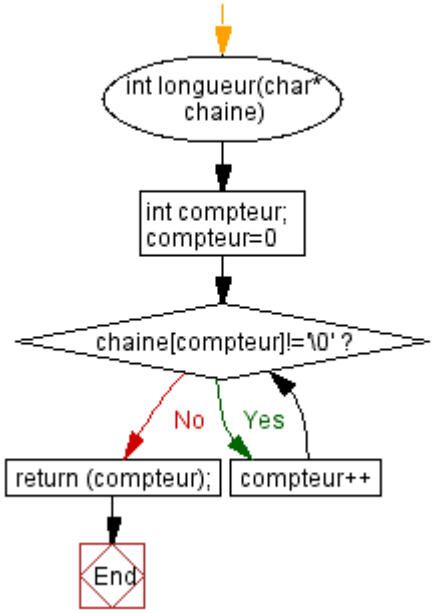
- *Lancez l'exécutable Que se passe-t-il si vous faites la commande « ldd questionnaire_sta » ? Expliquez.*

Annexes :


int saisieChaine(char* chaine, int long_Max)
int compteur;
char caractere;
(caractere de fin)&&(compteur <long_Max) ?
No
Yes
lecture caractere;
ajout à chaine;
incrémenter compteur;
ajouter \0 à chaine;
compteur == long_Max ?
Yes
No
return (-1);
return (1);
End


int saisie1(char* question, char* reponse, int lgMaxRep)
int retour=-1;
Afficher la question;
retour==-1 ?
No
Yes
return(longueur (reponse));
retour=saisieChaine(reponse, lgMaxRep);
End
retour == -1 ?
No
Yes
Afficher: Votre réponse est trop longue !;
Reformulez votre réponse qui ne doit pas dépasser lgMaxRep caractères";


#include "questionnaire.h"
int main()
int i;
int lg_reponse[NBR_QUESTIONS];
char reponse [TAILLE_REP_MAX];
char* question [NBR_QUESTIONS]={Q1, Q2, Q3};
printf("Liste des questions du questionnaire :\n");
i=0
i<NBR_QUESTIONS ?
No
Yes
printf("Début du questionnaire\n");
i=0
printf("%s\n", question[i]);
i++
i<NBR_QUESTIONS ?
No
Yes
printf("Fin du questionnaire\n");
lg_reponse[i]=saisie1(question[i], reponse, TAILLE_REP_MAX);
printf("Réponse enregistrée : %s, de longueur : %i\n", reponse, lg_reponse[i]);
exit (EXIT_SUCCESS);
i++


int longueur(char* chaine)
int compteur;
compteur=0
chaine[compteur]!='\0' ?
No
Yes
return (compteur);
compteur++
End

II. ***Webographie :***

- *https://bourdon.users.info.unicaen.fr/cours/IUT-1A/index.html*
- *https://www.aivosto.com/visustin-fr.html*
-