Plan

Partie 2 : Préprocesseur, makefile, et compilation séparée

Compilation separee

foo.c

```
void foo() {
}
```

- *definit* foo

main.c

```
extern void foo();
int main() {
  foo();
}
```

- *definit* main
- *utilise* foo sans l'avoir defini

Compilation separee

```
cc -c main.c #main.o <- main.c
cc -c foo.c #foo.o <- foo.c
cc -o ex1 main.o foo.o #main.o -> prog <- foo.o
./ex
```

- ex1 contient exactement une definition pour chaque fonction
- chaque fonction utilisee est definie
- il y a un main

```c
#include<stdio.h>
#define IS_NON_PRIME 0
#define IS_PRIME 1
#define IS_CANDIDATE 2
#define N 100

int prem[N];

void init (void)
{
      register int i;
      prem[0]=prem[1]=IS_NON_PRIME;
      for (i = 2; i < N; i = i + 1) prem[i] = IS_CANDIDATE;
      return ;
}

int min_is_candidate (void)
{
      register int i = 0;
      while (prem[i] != IS_CANDIDATE) i = i + 1;
      return i;
}
```

```
void set_non_prime(int start)
{
      register int i = start + 1;
      for (;i < N; i = i + 1)
        if (i % start == 0) prem[i]=IS_NON_PRIME;
      return ;
}

int main(void)
{
      register int next_prime = 1, i;
       init();
      while (next_prime * next_prime < N) {
      next_prime=min_is_candidate();
      prem[next_prime]=IS_PRIME;
      set_non_prime(next_prime);
      }
      printf("Liste des nombres
              premiers inf\\'erieurs \\`a %d\n", N);
      for (i = 0; i < N; i = i + 1)
        if (prem[i] != IS_NON_PRIME) printf("%d ", i);
      return 0 ;
}
```

Nous allons reprendre l'exemple du crible d'Ératosthène pour illustrer la notion de compilation séparée et l'utilitaire de gestion make associé à cette notion.

Objectif : diviser un programme C en plusieurs fichiers afin d'en faciliter la maintenance.

Il faut prendre garde à gérer correctement les *dépendances* entre les différents fichiers.

Pour commencer, on peut regrouper les définitions de macro dans un fichier `eratosthene.h` :

```
#define IS_NON_PRIME 0
#define IS_PRIME 1
#define IS_CANDIDATE 2
#define N 100
```

Un programme doit contenir une fonction principale (main).

La fonction principale `eratosMain.c` (permet entre autre de déclarer les identificateurs) :

```
#include <stdio.h>                                                    1
#include "eratosthene.h"                                              2
void init (void);                /*le prototype des     */          3
int min_is_candidate(void);/*fonction doit \^etre*/                   4
void set_non_prime(int);    /*disponible             */             5
int prem[N]; /*la variable prem est d\'efinie ici*/                   6
int main(void) {                                                      7
     register int next_prime = 1, i;                                  8
     init();                                                          9
     while (next_prime * next_prime < N) {                            10
     next_prime=min_is_candidate();                                   11
     prem[next_prime]=IS_PRIME;                                       12
     set_non_prime(next_prime);                                       13
     }                                                                14
     printf("Liste des nombres                                        15
             premiers inf\\'erieurs \\`a %d\n", N);                   16
     for (i = 0; i < N; i = i + 1)                                    17
         if (prem[i] != IS_NON_PRIME)                                 18
             printf("%d ", i);                                        19
     return 0 ;                                                       20
}
```

Fichiers composant notre programme

Il est possible d'obtenir un fichier objet associé à ce code :

```
% gcc -c eratosMain.c
% ls
eratosMain.c eratosMain.o  eratosthene.h
```

Puis, on peut par exemple faire un fichier par fonction :

```
#include "eratosthene.h"
/* prototype de la variable globale */
extern int prem [N] ;

void init (void)
{ /* la d\'efinition de la fonction init */
   register int i;
   prem[0]=prem[1]=IS_NON_PRIME;
   for (i = 2; i < N; i = i + 1)
        prem[i] = IS_CANDIDATE;
   return ;
}
```

Obtention d'un exécutable

Au final, on obtient

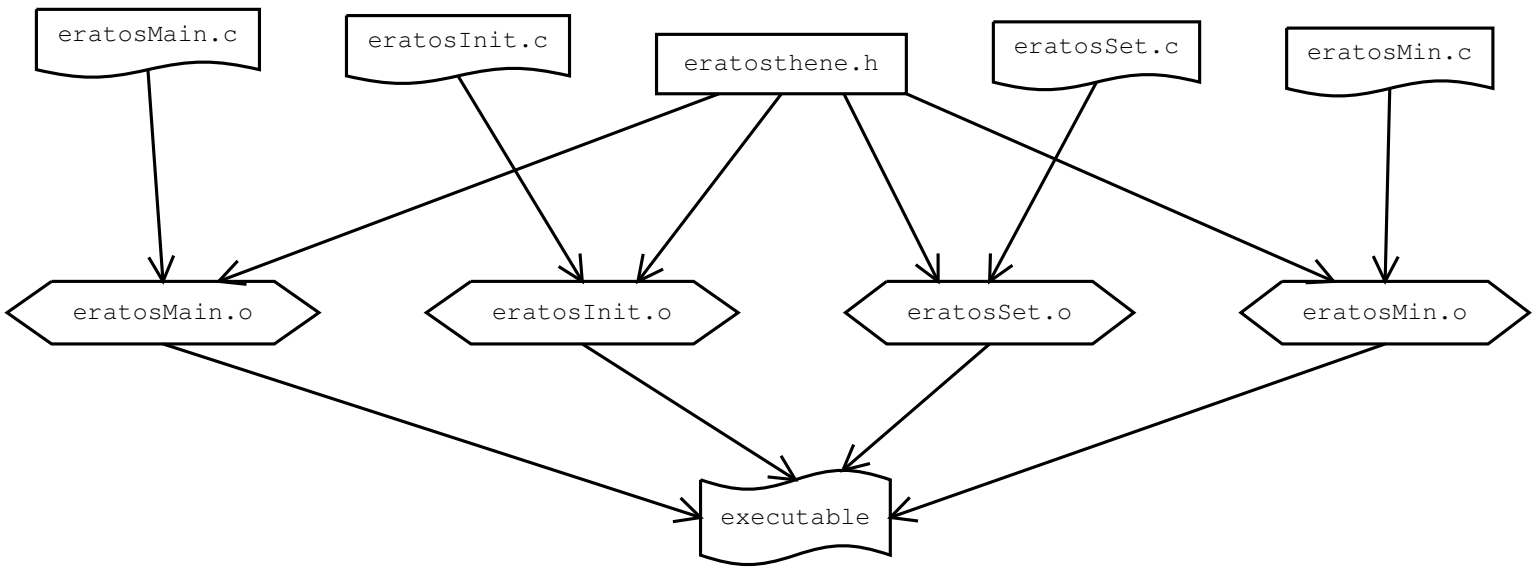
```
% gcc -c eratosInit.c                                                    1
% ls                                                                     2
eratosInit.c  eratosMain.c  eratosMin.c  eratosSet.c                     3
eratosInit.o  eratosMain.o  eratosMin.o  eratosSet.o                     4
eratosthene.h                                                           5
```

Pour conclure, on fait l'édition de lien de ces fichiers objets :

```
% gcc -o executable eratos*.o                                           1
%  executable                                                           2
Liste des nombres premiers inf\'erieurs␣\`a␣100                         3
2␣3␣5␣7␣11␣13␣17␣19␣23␣29␣31␣37␣41␣43␣47␣53␣59                        4
61␣67␣71␣73␣79␣83␣89␣97                                                5
```

Arbre de dépendances

Les opérations précédentes sont modélisées par l'arbre de dépendances (en fait c'est un DAG) :



Utilitaire `make` : syntaxe

Pour les projets importants (le code source de Linux est constitué de 921 fichiers), il faut automatiser les tâches.

Automatisation de la compilation :

- Maintenance, mise à jour et régénération de fichiers dépendants ;
- Sources $\rightarrow$ exécutables ;
- Recompilation quand nécessaire (dates) ;
- Fichier de règles de dérivation (code l'arbre de dépendances)
  `Makefile` ou `makefile`.

Format d'une règle : quoi, pourquoi, comment.

- syntaxe : *target* : *dependencies*
  (tabulation)*commands*
- quoi (*target*) objectif, généralement un fichier ;
- pourquoi (*dependencies*) liste des fichiers/cibles dont dépend *target* ;
- comment (*commands*) commandes à exécuter pour réaliser *target* ;

On peut n'exécuter qu'une *partie* de l'arbre : `%make target`

Exemple (makefile pour un programme C)

```
.PHONY:clean
executable: f1.o f2.o
        gcc -o executable f1.o f2.o
f1.o: f1.c fichier.h
        gcc -c f1.c
f2.o: f2.c fichier.h
        gcc -c f2.c
clean:
```

Utilitaire make : notre exemple

Dans notre cas, on peut écrire le Makefile suivant :

```
OPTIONS = -Wall -ansi -pedantic
OBJETS = eratosMain.o eratosMin.o eratosSet.o eratosInit.o

executable: $(OBJETS)
	gcc $(OPTIONS) -o executable $(OBJETS)

eratosMain.o: eratosMain.c eratosthene.h
	gcc $(OPTIONS) -c eratosMain.c

eratosMin.o: eratosMin.c eratosthene.h
	gcc $(OPTIONS) -c eratosMin.c

eratosSet.o: eratosSet.c eratosthene.h
	gcc $(OPTIONS) -c eratosSet.c

eratosInit.o: eratosInit.c eratosthene.h
```

Algorithme et macros de make

- Pour chaque cible
  - Vérifier les dépendances
    - → Récursion
    - → Date des fichiers de base
  - Si modification

    alors → Lancer les commandes

    sinon → Fichier à jour

`$@` représente le nom complet de la cible courante ;

`$?` représente les dépendances plus récentes que la cible ;

`$<` représente le nom de la première dépendance ;

`$^` représente la liste de toutes les dépendances ;

On peut définir ses propres macros :

```
REP = /etc/ /bin/ /usr/bin/
```

Le dévermineur gdb

L'environnement gdb permet d'exécuter des programmes pas à pas et d'examiner la mémoire du processus en cours.
Pour utiliser gdb, l'exécutable doit avoir été compilé avec l'option -g.
On l'utilise dans un shell en indiquant le fichier à examiner :

```
% gdb executable
GNU gdb 5.3-22mdk (Mandrake Linux)
...... etc.....................
This GDB was configured as "i586-mandrake-linux-gnu"...
(gdb)
```

Ce programme propose une aide en ligne :

```
(gdb) help help
Print list of commands.
(gdb) help quit
Exit gdb.
```

Exécution et examen du code source

Le programme considéré peut être exécuté dans l'environnement gdb :

```
(gdb) run
Starting program: /home/..../executable
Liste des nombres premiers inf\'erieurs \`a 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
61 67 71 73 79 83 89 97
Program exited normally.
(gdb)
```

Lorsque le code source de l'exécutable est disponible la commande `list` permet d'afficher le code source avec chacune de ces lignes numérotées. Dans notre cas :

```
(gdb) list
1       #include <stdio.h>
2       #include "eratosthene.h"
3
```

Placer des points d'arrêt

La commande `break` permet de placer un point d'arrêt sur une instruction du programme source de manière à ce qu'à la prochaîne exécution du programme dans gdb, l'invite du dévermineur soit disponible avant l'exécution de cette instruction.
Une instruction du programme source peut être repérée par le numéro de ligne correspondant ou par un identificateur :

```
(gdb) break 10
Breakpoint 1 at 0x8048353: file eratosMain.c, line 10.
(gdb) break min_is_candidate
Breakpoint 2 at 0x80483f2: file eratosMin.c, line 4.
```

permet de placer deux points d'arrêts aux endroits spécifiés. la commande `info` fournit la liste des points d'arrêts :

```
(gdb) info break
Num Type       Disp Enb Address    What
1   breakpoint keep y   0x08048353 in main at eratosMain.c
2   breakpoint keep y   0x080483f2 in min_is_candidate at
```

Exécution pas à pas

Une fois ceci fait, exécutons notre programme dans gdb :

```
Starting program: /home/.../executable
Breakpoint 1, main () at eratosMain.c:10
10              init();
(gdb)
```

Pour provoquer l'appel `init()`, utilisons la commande `next` :

```
(gdb) next
11              while (next_prime * next_prime < N) {
```

On peut exécuter les instructions associées

```
(gdb) step
init () at eratosInit.c:7
7           prem[0]=prem[1]=IS_PRIME;
```

Pour exécuter les instructions jusqu'au prochain point d'arrêt

```
(gdb) continue
Continuing. Breakpoint 2, min_is_candidate () at eratosMin
```

Affichage du contenu des variables et de la mémoire

Pour afficher le contenu d'une variable, il suffit d'utiliser print

```
(gdb) print prem
$3 = {0, 0, 1, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0,.. etc..
  0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2}
(gdb)
```

On peut provoquer l'affichage à chaque arrêt avec display et le formatter avec printf

```
(gdb) printf "%x\n",prem[1]
1
```

Plus généralement, on obtient l'affichage d'une zone mémoire grâce à la commande :

```
(gdb) x /4xw 0xbffff6a4
0xbffff6a4: 0x00000064 0xbffff6b8 0x0804836b 0x4014cf50
```

Quelques remarques : gdb est un outils très puissant

Remarquez qu'à l'entrée d'une fonction, les paramètres sont indiqués :

```
(gdb) contenu
Continuing.
Breakpoint 1, set_non_prime (start=3) at eratosSet.c:5
5               register int i = start + 1;
```

On peut modifier les valeurs des variables en cours d'exécution :

```
(gdb) set variable start = 0xb
(gdb) print start
$15 = 11
```

Il est possible de tracer l'exécution, de l'interrompre lors d'événements prédéfinis, etc.
Pour plus d'information, utilisez l'aide en ligne de gdb.

Le préprocesseur permet d'inclure dans le code source des fichiers textes complets.

Deux types d'inclusion de fichiers d'entête :

1. `#include <file.h>` : recherche du fichier `file.h`
   - dans les répertoires spécifiés par l'option `-I` du compilateur ;
   - dans le répertoire de la librairie standard (`/usr/include`).

2. `#include "file.h"` : recherche du fichier `file.h`
   - dans le répertoire du fichier qui fait l'inclusion ;
   - comme précédemment ensuite.

Ceci permet d'inclure des prototypes de fonctions, des macros, etc.

Substitution de texte

Le préprocesseur permet de définir des macros constantes et des fonctions sur la base de la substitution de chaîne de caractères.

- macros sans paramètres : `#define A 20` (sans rien ajouter). Attention à l'usage du point virgule (;)
- macros avec paramètres : `#define MAX(a,b) \`
  `((a)<(b)?(b):(a))`
- on peut supprimer une macro par `#undef A`.

Remarques :

- manipulation *purement syntaxique* ;
- toujours utile de parenthéser les paramètres ;
- imbrication possible des macros ;
- pas de blanc entre `MAX` et la parenthèse ouvrante ;
- pas d'effet sur les chaînes de caractères constantes ;
- si la macro nécessite plusieurs lignes, utiliser le `'\'`.

Macro : effet pervers

- Les macros permettent de faire une substitution de texte. Cette substitution est faite par le preprocesseur, avant la compilation.
- Cela peut engendrer des effets inattendus.

Avant preprocesseur

```
#define CARRE(x) (x*x)
int val = CARRE(1 + 1)
```

Apres preprocesseur

```
#define CARRE(x) (x*x)
int val = 1 + 1*1 + 1
```

- Pour eviter ca, parentheser : #define CARRE(x) ((x)*(x))

Macro avec paramètres : attention aux effets latéraux

Considérons l'exemple classique : `#define MAX(a,b) a>b?a:b`.

Supposons que les paramètres soient des expressions incluant des opérateurs de priorité inférieur à `>` et `?` (`MAX( x=y , ++z )` par exemple).

Le résultat est `x = ( y> ++z ? x=y : ++z)` ce qui n'a pas grand rapport avec ce que l'on attendait. Ainsi, on a tout intérêt à définir la macro plus précisément :
`#define MAX(a,b) (((a)>(b))?(a):(b))`.

Mais même dans ce cas, on doit bien remarquer que l'évaluation de cette macro implique une double incrémentation de z qui n'est pas explicite dans l'appel à cette macro.

Directives conditionnelles

Il est possible de conditionner la compilation par :

- l'insertion optionnelle de code

`#if` *expression_constante*	`#ifdef` *identificateur*
lignes à insérer si	lignes à insérer si
expression_constante vraie	*identificateur* est défini
`#endif`	`#endif`

- un test de non définition : `#ifndef` ;
- l'usage de l'alternative

`#if` *expression_constante*	`#ifdef` *identificateur*
lignes à insérer si	lignes à insérer si
expression_constante vraie	*identificateur* est défini
`#else`	`#else`
lignes à insérer si	lignes à insérer si
expression_constante fausse	*identificateur* non défini
`#endif`	`#endif`

Un petit exemple :

```
#ifdef ERREUR          /* Attention \‘a l’utilisation des */
#define SQR(x) x * x /* param\‘etres et aux effets       */
#else                  /* lat\’eraux                      */
#define SQR(x) ((x) * (x))
#endif
a=SQR(4 + 5);  t[i]=SQR(t[i++]);
```

Une macro peut être déclarée depuis le shell lors de la compilation :

```
% gcc -D ERREUR fichiersource.c
```

et ainsi conditionner la compilation du code.

On peut aussi interrompre ou commenter la compilation

```
#ifndef MAMACRO
#error "MAMACRO inconnue"
#else
#warning "MAMACRO est connue"
#endif
```

Macro prédéfinie du préprocesseur

Il existe un certain nombre de macro prédéfinies :

- __**FILE**__ correspond au nom du fichier source ;
- __**FUNCTION**__ correspond au nom de la fonction contenant la ligne courante dans le code ;
- __**LINE**__ correspond à la ligne courante dans le code ;
- __**DATE**__ correspond à la date du préprocessing ;
- __**TIME**__ correspond à l'heure du préprocessing ;
- etc.

Par exemple

```
% nl preprocessing.c
     1  int main (void) {
     2   int a = __LINE__ ;
     3    return 0;
     4  }
```

```
% gcc -E preprocessing
     int main (void)
  int a = 2 ;
  return 0;
}
```