Programmation en C
Cours 3

Licence Maths-Info
Aix-Marseille Université
2011-2012

Valentin Emiya
valentin.emiya@lif.univ-mrs.fr

30 janvier 2012

Lundi dernier

- Les fonctions
  - Syntaxe et sémantique
  - Passage de paramètre
- Les tableaux
  - Syntaxe, sémantique
  - Stockage en mémoire, passage de paramètre
  - Les tableaux à 2 dimensions
- La compilation séparée :
  - Diviser son programme en plusieurs fichiers
  - L'outil *make* pour automatiser la compilation

Aujourd'hui

- Compilation et make
  - Rappel : compilation séparée, dépendances, make
  - gcc et ses options : precompilation, compilation, assemblage, édition de liens
- La précompilation : les directives
- Les types
  - `struct, union, enum`
  - définir un nouveau type avec `typedef`
- L'outil *gdb* pour « débuguer »

Aujourd'hui

- Compilation et make
  - Rappel : compilation séparée, dépendances, make
  - gcc et ses options : precompilation, compilation, assemblage, édition de liens
- La précompilation : les directives
- Les types
  - `struct, union, enum`
  - définir un nouveau type avec `typedef`
- L'outil *gdb* pour « débuguer »

Exemple de fichier makefile

```
CC = gcc
COPT = -W -Wall -ansi -pedantic

executable: eratosMain.o eratosMin.o eratosSet.o eratosInit.o
	$(CC) $(COPT) -o executable $^

eratosMain.o: eratosMain.c eratosthene.h
	$(CC) $(COPT) -c $<

eratosMin.o: eratosMin.c eratosthene.h
	$(CC) $(COPT) -c $<

eratosSet.o: eratosSet.c eratosthene.h
	$(CC) $(COPT) -c $<

eratosInit.o: eratosInit.c eratosthene.h
	$(CC) $(COPT) -c $<
```

fichier makefile : erreurs fréquentes

- Nom du fichier : "makefile", sans extension (pas de makefile.txt), sinon :

```
make: *** No rule to make target `nom_cible'.  Stop.
```

- Attention aux fautes de frappe, sinon :

```
make : *** No rule to make target `affichage_tableau.c',
needed by `affichage_tableau.o'.  Stop
```

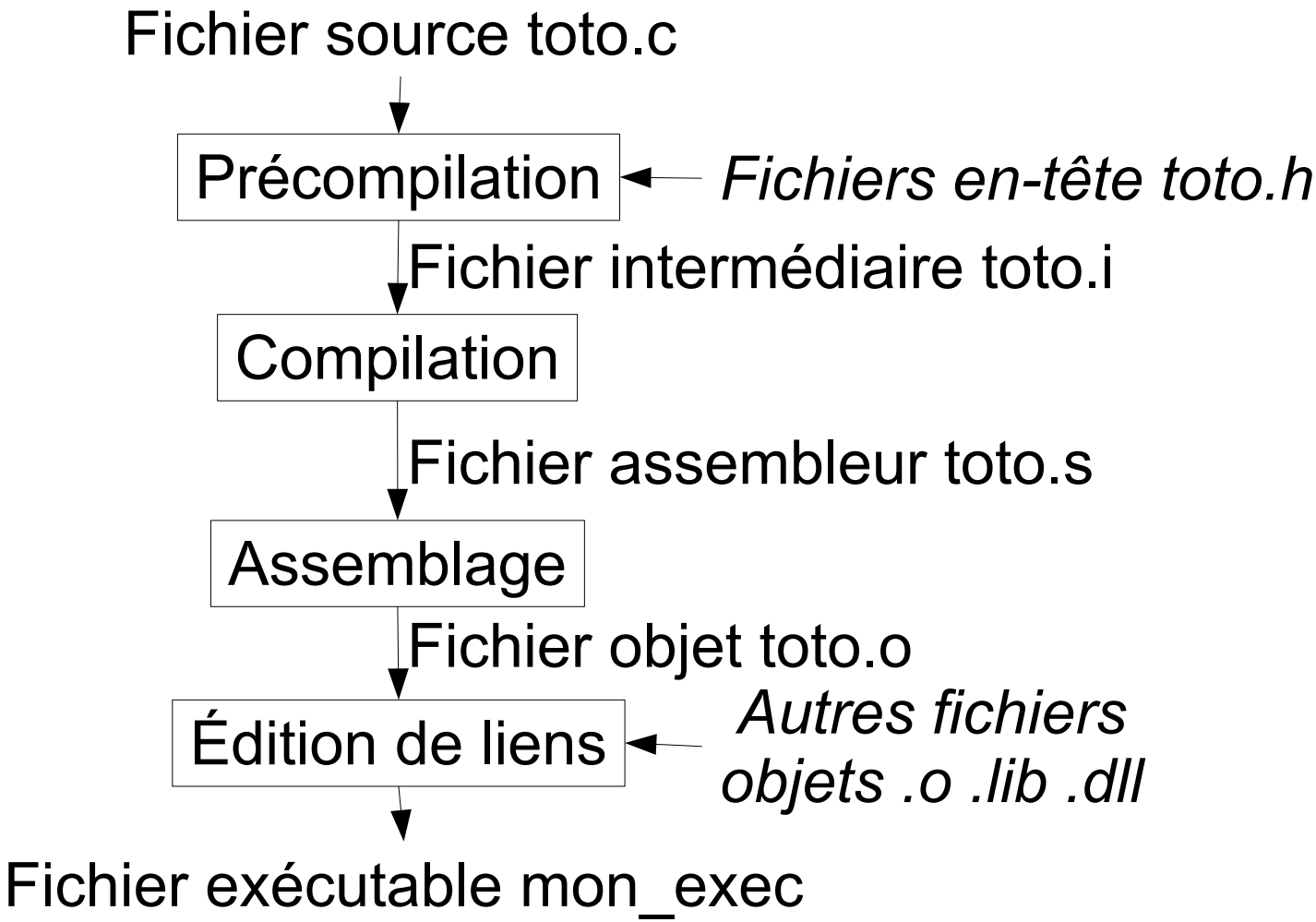
- Ne pas oublier la tabulation avant la commande (et attention à l'éditeur de texte utilisé), sinon

```
makefile:7: *** missing separator.  Stop.
```

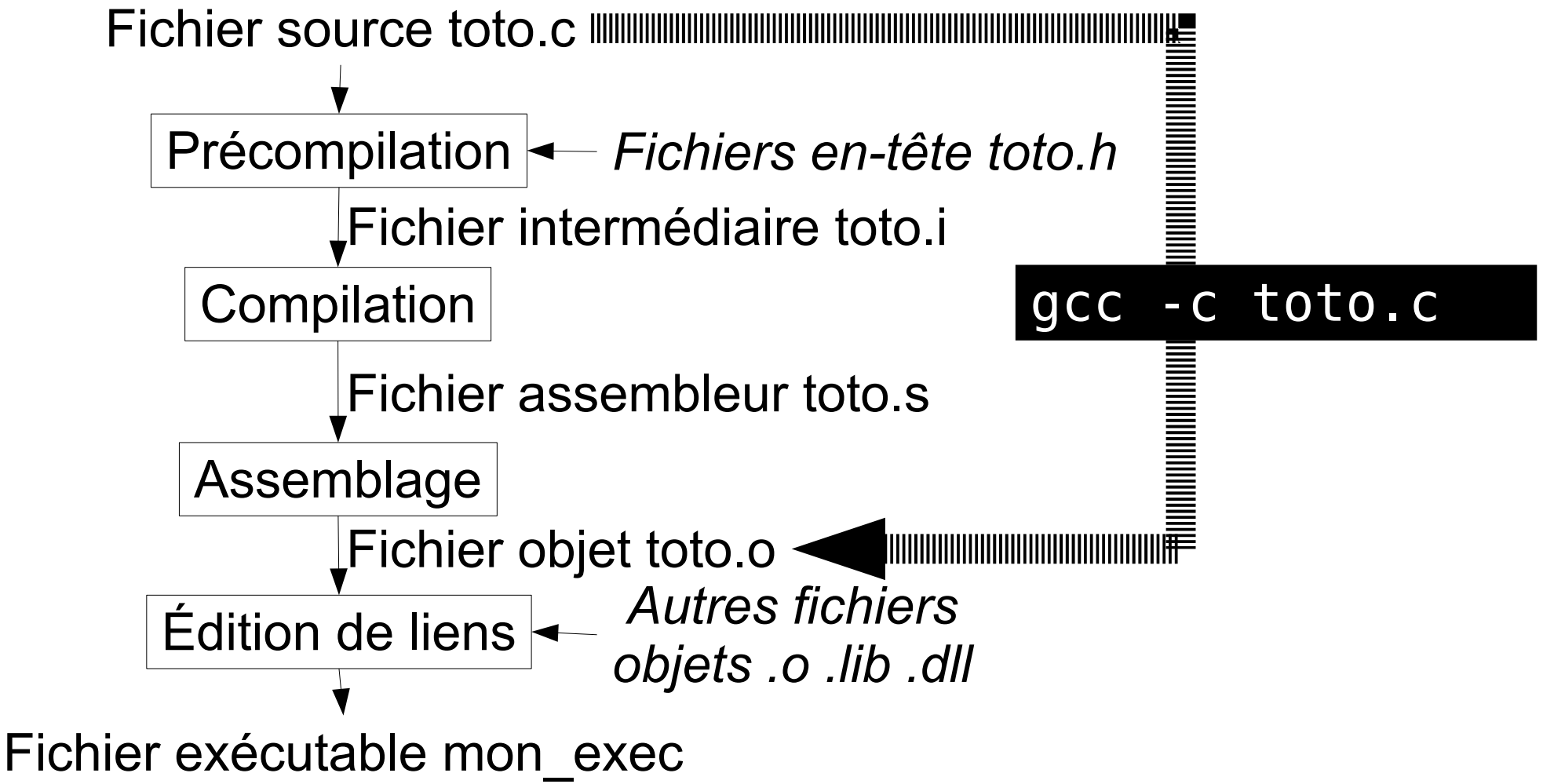
- Ne pas oublier de se placer dans le répertoire du makefile pour lancer `make nom_cible`, sinon

```
make: *** No rule to make target `nom_cible'.  Stop.
```

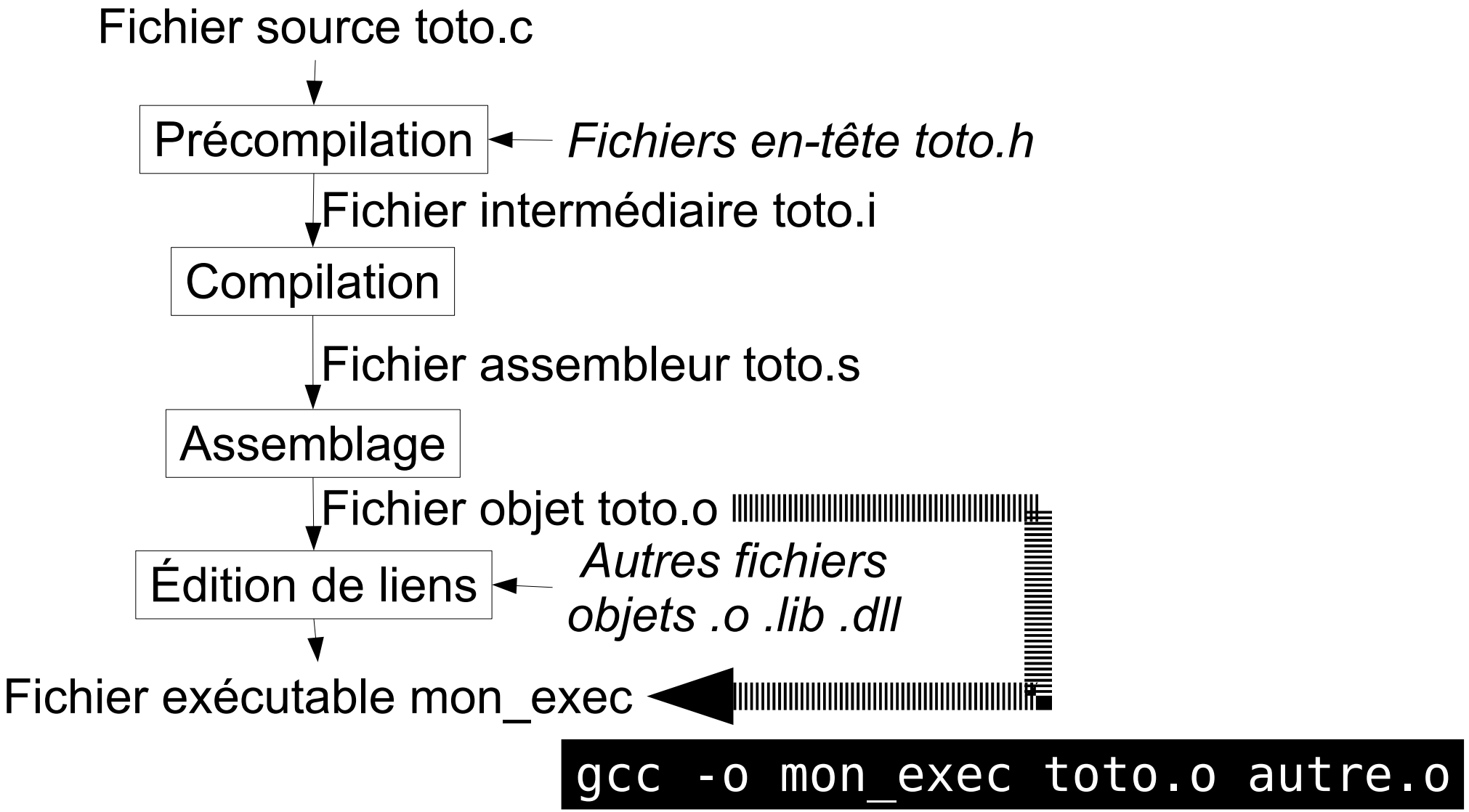
Compilation



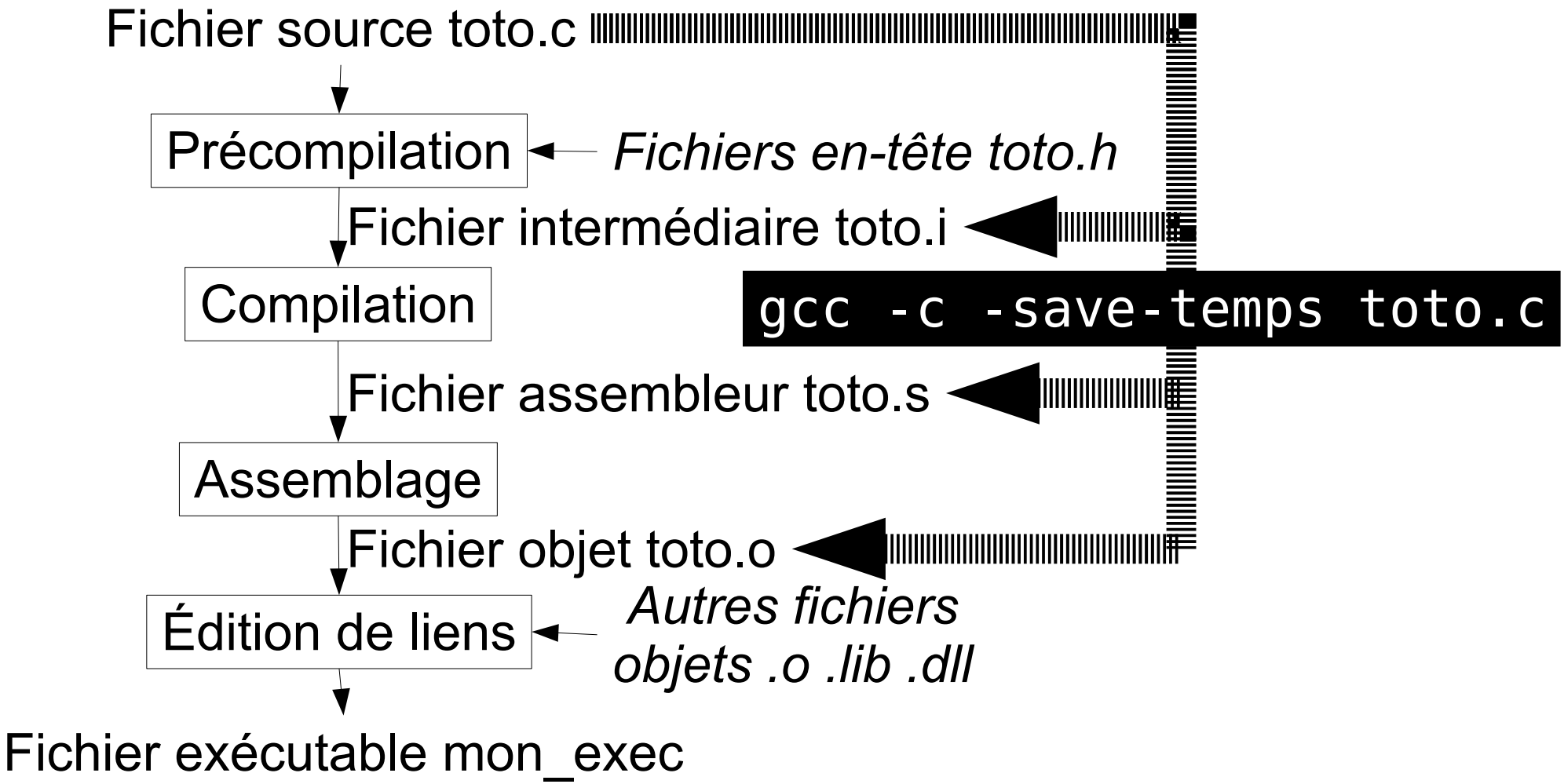
Compilation : obtenir le fichier objet



Compilation : obtenir l'exécutable



Compilation : garder les fichiers .i, .s



Aujourd'hui

- Compilation et make
  - Rappel : compilation séparée, dépendances, make
  - gcc et ses options : precompilation, compilation, assemblage, édition de liens
- La précompilation : les directives

`#define, #include, #if, #ifdef, #ifndef...`

- Les types
  - `struct, union, enum`
  - définir un nouveau type avec `typedef`
- L'outil *gdb* pour « débuguer »

Pratique du C
Les directives au préprocesseur
Types composés
Définition de nouveaux types

Licence Informatique — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Semestre 5 — 2009-2010

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

Précompilation : que renvoie ce programme ?

Fichier zero.h :

```
#define N 3
#define F(X) (X)*(X)
#define G(X) X*X
#define H(X) X+=200
```

Fichier zero.c

```
#include "zero.h"

int main (void)
{
    /* ceci est un commentaire */
    int a,b,c,d;
    a = N;
    b = 2*N;
    c = F(a+b);
    d = G(a+b);
    if (c!=d)
        H(d);
    return d;
}
```

Réponse : regardons le fichier précompilé zero.i

Comment ?

Par exemple, lançons la compilation de zero.c avec l'option `-save-temps` permettant de garder les fichiers intermédiaires

```
$ gcc -c -save-temps zero.c
```

Autre possibilité : affichage de la sortie du précompilateur avec

```
$ gcc -E zero.c
```

zero.h :

```
#define N 3
#define F(X) (X)*(X)
#define G(X) X*X
#define H(X) X+=200
```

zero.c

```
#include "zero.h"
int main (void)
{
    /* ceci est un commentaire */
    int a,b,c,d;
    a = N;
    b = 2*N;
    c = F(a+b);
    d = G(a+b);
    if (c!=d)
        H(d);
    return d;
}
```

zero.i :

```
int main (void)
{
    int a,b,c,d;
    a = 3;
    b = 2*3;
    c = (a+b)*(a+b);
    d = a+b*a+b;
    if (c!=d)
        d+=200;
    return d;
}
```

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

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

`#if` *expression_constante*
lignes à insérer si
expression_constante vraie
`#else`
lignes à insérer si
expression_constante fausse
`#endif`
	`#ifdef` *identificateur*
	lignes à insérer si
	identificateur est défini
	`#else`
	lignes à insérer si
	identificateur n'est pas défin
	`# 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 la compilation

```
#ifndef MAMACRO
#error "MAMACRO inconnue"
#endif
```

Fichier .h : l'usage systématique de #ifndef.

Où ? Dans les fichiers .h

Quoi ? Si le fichier s'appelle toto.h, le débuter/terminer par

```
#ifndef TOTO_H
#define TOTO_H
... /* contenu du fichier */
#endif
```

Quand ? Toujours (tous les fichiers .h systématiquement)

Pourquoi ? Pour éviter les inclusions infinies lorsque plusieurs fichiers .h contiennent un `#include` des autres

fichier1.h :

```
#ifndef FICHIER1_H
#define FICHIER1_H

#include "fichier2.h"

...

#endif
```

fichier2.h :

```
#ifndef FICHIER2_H
#define FICHIER2_H

#include "fichier1.h"

...

#endif
```

Macro prédéfinie du préprocesseur

Il existe un certain nombre de macro prédéfinies :

- __**FILE**__ correspond au nom du fichier source ;
- __**func**__ 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.c
     int main (void) {
  int a = 2 ;
  return 0;
}
```

Bonnes pratiques de programmation en C

- Découper intelligemment le code en plusieurs fichiers
- Utiliser les fichiers .h pour déclarer les types, fonctions, etc.
- Faire le diagramme des dépendances et le `makefile` dès que possible
- Utiliser systématiquement `#ifndef` dans les fichiers .h

Aujourd'hui

- Compilation et make
  - Rappel : compilation séparée, dépendances, make
  - gcc et ses options : precompilation, compilation, assemblage, édition de liens
- La précompilation : les directives
- Les types
  - `struct, union, enum`
  - définir un nouveau type avec `typedef`
- L'outil *gdb* pour « débuguer »

Exemple introductif

```
struct mastructure {
  char o ;
  int six ;
};
```

Une structure contenant deux éléments :

un `char` et un `int`.

Une *structure* est le regroupement de plusieurs variables de types différents dans une même entité.

- cet objet est composé d'une séquence de membres de types divers ;
- chaque membre porte un nom interne à la structure ;
- le type des membres peut être quelconque (imbrication) ;
- les membres sont stockés de manière contiguë en mémoire ;
- déclaration :

```
struct identificateur_du_modèle
{
   type liste_identificateur_de_membre ;
   type liste_identificateur_de_membre ;
   ...
} ;
```

- déclaration de variable d'un type structure :
  `struct` *identificateur_de_modèle liste_identif_variable* ;
- définition et déclaration simultanées de variables :

  `struct` *identificateur_de_modèle* {
  type *liste_identificateur_de_membre* ;
  type *liste_identificateur_de_membre* ;
  ...
  } *liste_identificateur_de_variable* ;
- le nommage de la structure est alors facultatif ;
- accès à un membre : opérateur . de sélection de champs
  identificateur_de_variable . *identificateur_de_membre* ;

```
struct mastructure {
      char o ;
      int  six ;
} ;
struct mastructure mavariable ;
mavariable.o='o' ; mavariable.six = 6 ;
```

Une spécificité du compilateur gcc

La norme ISO ne permet pas de faire de l'initialisation des structures lors de leurs déclarations.

Mais le compilateur gcc prévoit tout de même cette possibilité :

```
struct complexe {
  int re ;
  int im ;
} foo = {
  .im = 2,
  .re = 1
} ;
```

Plus canoniquement, l'initialisation peut se faire en donnant la liste entre { } de constantes :

```
struct complexe {
  int re ;
  int im ;
} foo = { 1, 2 } ; /* il faut respecter l'ordre */
```

Exemple de représentation en mémoire d'une structure

```
struct adresse {
  int num;
  char rue[40];
  long int code;
  char ville[20];
};

struct personne {
  char nom[20];
  char prenom[25];
  int age;
  struct adresse adr;
} bibi = {
  .nom = "Moi",
  .prenom = "Idem",
  .age = 100,
  .adr.num = 39,
  .adr.rue = "Publique",
  .adr.ville = "Lille",
  .adr.code = 59000 };
```

```
.globl bibi
        .data
        .align 32
        .type   bibi,@object
        .size   bibi,120
bibi:
        .string "Moi"
        .zero   16
        .string "Idem"
        .zero   20
        .zero   3
        .long   100
        .long   39
        .string "Publique"
        .zero   31
        .long   59000
        .string "Lille"
        .zero   14
```

Ne pas confondre C et ses héritiers

Attention : C n'est pas un langage orienté objet et donc, il n'y a pas de constructeur en C.

Il n'y a pas d'initialisation "générique" associée à un type. Le code suivant n'est pas du C valide :

```
struct adresse
{
  int num = 36 ;
  char rue[40] = "Quai des Orf\'evres";
  long int code = 75001;
  char ville[20] = "Paris" ;
};
```

Copie et affectation d'une structure comme un tout

Contrairement aux tableaux, l'affectation

```
#include "les_definitions_des_transparents_precedents"
struct personne bobo;
int main(void){
  bobo = bibi ;
  return 0 ;
}
```

est possible et provoque une copie physique des données de l'espace mémoire associé à `bibi` dans celui associé à `bobo`.

En conséquence, on peut :

- passer des structures en argument de fonction (copie) ;
- utiliser une structure comme valeur de retour de fonction ;
- mais C étant un langage de bas niveau, les structures ne se comparent pas.

Un type `union` permet :

- de définir une variable qui peut contenir à des moments différents des objets de type et de taille différents ;
- la manipulation de différents types de données dans un même espace mémoire.

La manipulation des unions est semblable à celle des structures :

- syntaxe similaire à celle des structures :

```
union identificateur_d'union
{
type liste_identificateur_de_champs ;
type liste_identificateur_de_champs ;
                    ...
} liste_identificateur_de_variable ;
```

- accès à un champs :
  identificateur_de_variable.*identificateur_de_champs*

Exemple de représentation en mémoire d'une union

Les champs potentiels sont stockés de manière superposé en mémoire.

```
union nombre{

  int entier ;

  struct complexe {
    float re ;
    float im ;
  } comp_var ;

  char symbol[20] ;

}  bar = { .entier = 999 } ;
```

```
.globl bar
        .data
        .align 4
        .type   bar,@object
        .size   bar,20
 bar:
        .long   999
        .zero   16
        .text
```

Exemple d'affectation en mémoire d'une union

```
union nombre{

  int entier ;

  struct complexe {
    float re ;
    float im ;
  } comp_var ;

  char symbol[20] ;

}  foo,bar={.entier=999};

int main(void)
{
   foo=bar ;

   return 0 ;
}
```

```
        .data
        .size   bar,20
bar:
        .long   999
        .zero   16
foo:    .zero   20
        .text
.globl main
main:   ......
        movl    bar, %eax
        movl    %eax, foo
        movl    bar+4, %eax
        movl    %eax, foo+4
        movl    bar+8, %eax
        movl    %eax, foo+8
        movl    bar+12, %eax
        movl    %eax, foo+12
        movl    bar+16, %eax
        movl    %eax, foo+16
        ......
```

- Syntaxe : *type-énuméré* :
  $\Rightarrow$ `enum` *identificateur* { *liste-d-énumérateurs* }

  liste-d-énumérateurs :
  $\Rightarrow$ *liste-d-énumérateurs*$_{option}$ *énumérateur*

  énumérateur :
  $\Rightarrow$ *identificateur*
  $\Rightarrow$ *identificateur* = *expression-constante*
- Sémantique :
  - type dont les valeurs possibles font partie des *énumérateurs* ;
  - *identificateur* dans *énumérateur* : constante entière ;
  - nom d'un *identificateur* : distinct d'une variable ordinaire ;
  - valeur entière nulle au départ et incrémentée pour chaque nouvel *identificateur* ;
  - spécifier une valeur (*expression-constante*).

```
enum {VRAI, FAUX} test=FAUX; /* contraire aux
                                conventions du C */
enum mois_m { jan=1, feb=2, mar, avr, may, jun, jul,
              aug, sep, oct, nov, dec};
enum mois_m mavariable ;
```

```
enum {VRAI,FAUX} test=FAUX;
int main(void){
  test = VRAI ;
  return 0 ;
}
```

```
.globl test
        .data
        .align 4
        .type   test,@object
        .size   test,4
test:
        .long   1
        .text
.globl main
        .type main,@function
main:
        ...
        movl    $0, test
        ....
```

Exemple de définition de type

Types définis par l'utilisateur

- ajoute un nom désignant un type existant ;
- lisibilité : utilisé pour les structures complexes ;
- portabilité : paramétrer un programme (`size_t`) ;
- syntaxe : identique à celle d'une variable

  `typedef` *type* *identificateur_de_type*
- ne crée pas un nouveau type, plutôt un synonyme ;

```
typedef enum mois_m mois_t; /*d\'efinit pr\'ec\'edement*/

mois_t mois;

typedef enum {FALSE, TRUE} bool_t; /* conforme \`a la
                                      norme C        */
bool_t b, btab[30] ;

typedef struct point { int x; int y;} point_t;
typedef struct rectangle {point_t P1, P2;} rectangle_t;
rectangle_t carre_unite = {{ 0, 0}, { 1, 1}};
```

```
typedef struct point {
      int x;
      int y;
} point_t;
typedef struct rectangle {
      point_t P1;
      point_t P2;
} rectangle_t ;

rectangle_t carre_unite = {{ 0, 0},
                           { 1, 1}};

int main(void)
{
  return 0 ;
}
```

```
        .file   "typedef.c"
.globl carre_unite
        .data
        .align 4
        .type carre_unite,@object
        .size carre_unite,16
carre_unite:
        .long   0
        .long   0
        .long   1
        .long   1
        .text
.globl main
.type main,@function
main:
        ........
```

Aujourd'hui

- Compilation et make
  - Rappel : compilation séparée, dépendances, make
  - gcc et ses options : precompilation, compilation, assemblage, édition de liens
- La précompilation : les directives
- Les types
  - `struct, union, enum`
  - définir un nouveau type avec `typedef`
- L'outil *gdb* pour « débuguer »

Comment débuguer son code ?

« J'ai écrit un nouveau code, ça ne fonctionne pas !

Comment faire ? »

Pas de panique, c'est le lot de tout le monde

- Erreurs de compilation :
  - Regarder la première erreur (les autres sont peut-être la conséquence de la première)
  - La comprendre et aller à la ligne indiquée

```
$ gcc -W -Wall -ansi -pendantic toto.c
toto.c: In function ‘main’:
toto.c:17: error: expected ‘,’ or ‘;’ before ‘printf’
toto.c:13: warning: unused variable ‘m’
toto.c:13: warning: unused variable ‘n’
```

Comment débuguer son code ?

« J'ai écrit un nouveau code, ça ne fonctionne pas !

Comment faire ? »

Pas de panique, c'est le lot de tout le monde

- Erreurs de compilation :
  - Regarder la première erreur (les autres sont peut-être la conséquence de la première)
  - La comprendre et aller à la ligne indiquée
- Warning à la compilation : idem, à éviter !
  (dans ce cas, la compilation aboutit)
- Compilation OK ; erreurs à l'exécution :

l'outils *gdb* entre en scène

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
0 1 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
4       void init (void) ;
(gdb)
```

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:10
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.c:4
4               register int i = 0;
```

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 = {1, 1, 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.

Aujourd'hui

- Compilation et make
  - Rappel : compilation séparée, dépendances, make
  - gcc et ses options : precompilation, compilation, assemblage, édition de liens
- La précompilation : les directives
- Les types
  - `struct, union, enum`
  - définir un nouveau type avec `typedef`
- L'outil *gdb* pour « débuguer »