Programmation en C
Cours 2

Licence Maths-Info
Aix-Marseille Université
2011-2012

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

23 janvier 2012

Organisation du module

- Confirmation des séances annulées : pas de CM/TD/TP les lundis 26 mars et 2 avril.
- Evaluation : NF = (ET+3*CI)/4
  - ET : examen terminal
  - CI = (2*TPCC + CC)/3 :contrôle intermédiaire
  - CC : contrôle continu (partiel+participation TD)
  - TPCC : TP en contrôle continu (comptes-rendus)
  - Partiel : 20 février (ou 5 mars)

Lundi dernier

- Introduction générale sur le langage C
- Ecrire le premier programme, le compiler et l'exécuter
- Eléments de base du langage
  - Les constantes, les identificateurs
  - Les types
  - Les variables
  - Les expressions
  - Les opérateurs
  - Instructions usuelles
  - Instructions de contrôle

Aujourd'hui

- 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

- 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

Définition d'une fonction : ANSI

Appel à une fonction

Passage de paramètres par copie

Les tableaux

Tableaux passés en paramètre d'une fonction

Exemple de programme : crible d'Ératosthène

Compilation séparée et Make

Exécution pas à pas dans l'environnement gnu debugger

Pratique du C
Fonction – tableau
compilation séparée

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

Semestre 5 — 2009-2010

Syntaxe ANSI : *définition-de-fonction-ANSI* :

> *type-retour* *nom-de-fonction*
> (*liste-de-paramètres-typés*$_{option}$)
> {
> *liste-de-déclarations-locales* $_{option}$
> *liste-d'instructions*
> }

Sémantique :

- *type-retour* : type de la valeur retournée (quelconque),
- *liste-de-paramètres-typés*$_{option}$:
  liste des paramètres formels avec leur type ;
- passage de paramètres *uniquement* par valeur ;
- *liste-de-déclarations-locales* $_{option}$:
  déclaration de variables *locales* à la fonction ;
- *liste-d'instructions* : corps de la fonction.

Syntaxe ANSI : *définition-de-fonction-ANSI* :

type-retour *nom-de-fonction*
(*liste-de-paramètres-typés*$_{option}$)
{
liste-de-déclarations-locales $_{option}$
liste-d'instructions
}

Une fonction retourne toujours une valeur :

- le corps doit contenir au moins une instruction :
  `return` *expression* ;
  sinon le résultat est indéterminé ;
- *expression* qui doit être de type *type-retour* ;
- cette instruction évalue *expression* qui sera la valeur de retour et rend le contrôle d'exécution à l'appelant.

Définition d'une fonction : ANSI

Appel à une fonction

Passage de paramètres par copie

Les tableaux

Tableaux passés en paramètre d'une fonction

Exemple de programme : crible d'Ératosthène

Compilation séparée et Make

Exécution pas à pas dans l'environnement gnu debugger

Définition à la Kernighan et Ritchie

Syntaxe K&R : *type-retour* *nom-de-fonction*
(*liste-d'identificateurs*$_{option}$)
liste-de-déclarations$_{1\ option}$
{
liste-de-déclarations$_{2\ option}$
liste-d'instructions
}

Sémantique : similaire à la norme ANSI

- *liste-d'identificateurs*$_{option}$: liste des paramètres formels sans spécification de type ;
- *liste-de-déclarations*$_{1\ option}$:
  déclaration des types des paramètres formels ;
- les noms doivent être identiques dans
  liste-d'identificateurs et *liste-de-déclarations*$_1$;
- si un paramètre est omis dans *liste-de-déclarations*$_1$:
  son type par défaut est `int`.

Définition d'une fonction : ANSI

Appel à une fonction

Passage de paramètres par copie

Les tableaux

Tableaux passés en paramètre d'une fonction

Exemple de programme : crible d'Ératosthène

Compilation séparée et Make

Exécution pas à pas dans l'environnement gnu debugger

Comparaison ANSI et K&R

Exemple de définition de fonction : norme ANSI

```
int sum_square(int i, int j)
{
  int resultat;
  resultat = (i * i) + (j * j);
  return resultat;
}
```

Exemple de définition de fonction : norme K&R

```
int sum_square(i,j)
  int i,j;
{
  int resultat;

  resultat = (i * i) + (j * j);
  return(resultat);
}
```

Remarques complémentaires

- on ne peut pas définir des fonctions dans des fonctions ;
- `return` est une instruction comme une autre :
  ainsi, elle peut être utilisée plusieurs fois dans le corps d'une fonction

```
int
max
(int a, int b)
{
  if (a > b) return (a); else return(b);
}
```

- répétons que si la dernière instruction exécutée dans une fonction n'est pas un `return`, le résultat retourné est indéterminé.

Dans les transparents du cours, les accolades ouvrantes des bloc d'instructions ne sont pas sur une ligne indépendante uniquement pour permettre la présentation. Ce n'est pas un exemple à suivre.

Définition d'une fonction : ANSI
Appel à une fonction
Passage de paramètres par copie
Les tableaux
Tableaux passés en paramètre d'une fonction
Exemple de programme : crible d'Ératosthène
Compilation séparée et Make
Exécution pas à pas dans l'environnement gnu debugger

Appel à une fonction

- Syntaxe de l'appel à une fonction : *expression-appel* :
  $\Rightarrow$ *nom-de-fonction* (*liste-d'expressions*)
- Sémantique :
  - évaluation des expressions de *liste-d'expressions* ;
  - l'ordre d'évaluation n'est pas fixé par la norme ;
  - résultats passés en paramètres effectifs à la fonction ;
  - le passage se fait par *valeur* ;
  - contrôle d'exécution passé au début de *nom-de-fonction* ;
  - *expression-appel* : valeur retournée par la fonction ;
- Exemples :

```
d = sum_square(a,b) / 2;
c = max(a,b);
```

Procédures : fonctions avec effet latéral

- C ne comporte pas de concept de procédures ;
- Les fonctions peuvent réaliser tous les effets latéraux voulus ;
- En C, une *procédure* est fonction qui ne retourne aucune valeur ;
- "Aucune valeur" a un type de base, le type `void` ;
- Il n'a pas de `return` dans le corps d'une fonction de type de retour void (pour faire cours, d'une procédure) ;
- Exemple d'appel de procédure :

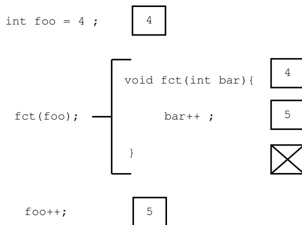
```
#include<stdio.h>
void testzero(int j) {
if(j) return ; /* provoque la sortie */
printf("test positif") ;
}
int main(void) {
testzero(0);
return 0 ;
}
```

En C, les paramètres sont des variables comme les autres.
Un passage d'information se fait par *copie* des paramètres.

```
void fct(int foo){
   foo = 3 ;
   return ;
}
```

```
int main(void){
   int bar ;
   fct(bar) ;
    return bar ;
}
```

À chaque appel de fonction, de l'espace mémoire est créé pour les paramètres et les variables locales (et détruit après l'appel lors du retour à l'appelant).



Aujourd'hui

- 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

Définition d'une fonction : ANSI

Appel à une fonction

Passage de paramètres par copie

Les tableaux

Tableaux passés en paramètre d'une fonction

Exemple de programme : crible d'Ératosthène

Compilation séparée et Make

Exécution pas à pas dans l'environnement gnu debugger

En mémoire, un tableau est un bloc d'objets consécutifs de même type.
Sa déclaration est :

- similaire à une déclaration de variable ;
- il faut indiquer le nombre d'éléments entre `[]`.

Quelques exemples :

```
char s[22]; /* s tableau de 22 caract\`eres */
/* t1 tableau de 10 entiers longs et
   t2 tableau de 20 entiers longs */
long int t1[10], t2[20];
#define N 100
int tab[N/2];
```

Les chaines de caractères

```
|  |'b'|'o'|'n'|'j'|'o'|'u'|'r'|'\0'|  |
```

```
char a [] = "bonjour" ;
char b [] = {'b','o','n','j','o','u','r','\0'};
printf("bonjour") ;
printf("%s\n",a) ;

printf("%s\n",b) ;
```

Une chaine de caractères est un tableau d'éléments de type char terminé par le caractère '\0'.

Définition d'une fonction : ANSI
Appel à une fonction
Passage de paramètres par copie
Les tableaux
Tableaux passés en paramètre d'une fonction
Exemple de programme : crible d'Ératosthène
Compilation séparée et Make
Exécution pas à pas dans l'environnement gnu debugger

Points importants :

- la taille d'un tableau est une constante **qui doit être calculable à la compilation :**

```
char tab[] = "123" ;        .globl tab
                                    .data
int main(){                         .type   tab,@object
                                    .size   tab,4
        return 0 ;          tab:
}                                   .string "123"
```

- les indices dans un tableau commencent en 0 ;

  Les indices d'un tableau de taille `N` vont de `0` à `N-1`.

Définition d'une fonction : ANSI
Appel à une fonction
Passage de paramètres par copie
Les tableaux
Tableaux passés en paramètre d'une fonction
Exemple de programme : crible d'Ératosthène
Compilation séparée et Make
Exécution pas à pas dans l'environnement gnu debugger

L'initialisation d'un tableau se fait :

- par des valeurs constantes placées entre {} séparées par des virgules (,) ;
- si il n'y a pas assez de valeurs : l'espace mémoire restant est soit indéterminé soit mis à 0 ;
- Par exemple : `int t[4] = { 1, 2, 3, 4 };`
- il n'y a pas de facteur de répétition.

Définition d'une fonction : ANSI
Appel à une fonction
Passage de paramètres par copie
Les tableaux
Tableaux passés en paramètre d'une fonction
Exemple de programme : crible d'Ératosthène
Compilation séparée et Make
Exécution pas à pas dans l'environnement gnu debugger

Manipulations élémentaires sur les tableaux

Accès à un élément de tableau par opérateur d'indexation ;

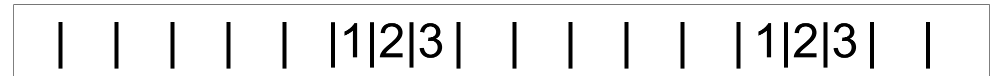
- Syntaxe : *expression* $\Leftarrow$ *nom-de-tableau* [$expression_1$]
- Sémantique :
  - $expression_1$ délivre une valeur entière ;
  - *expression* délivre l'élément d'indice $expression_1$;
  - *expression* peut être une valeur de gauche comme dans l'exemple `x = t[k] ; t[i+j] = x ;`.

L'identificateur `t` n'est pas une variable. Il est associé à une adresse constante correspondant au début de la mémoire allouée au tableau. En mémoire, on a les octets :

	t				
···	1	2	3	4	···

Comparer 2 identificateurs de tableau revient à comparer 2 adresses et non pas les objets stockés à ces adresses. De même, affecter quelque chose à cet identificateur `t = ...` n'a pas de sens.

Exemple :
« égalité » de deux tableaux



↑ a ↑ b

```
int a [] = {1,2,3} ;
int b [] = {1,2,3} ;
printf("%d\n",a==b) ;
/* Rend 0 (faux) */
```

- En mémoire, il y a deux tableaux, à deux adresses différentes, de contenus égaux.
- Le test d'égalité (des adresses) renvoie faux.

Puisque l'identificateur d'un tableau n'est pas une variable, quelle copie est faite lors du passage de paramètre suivant :

```
void fct(int tib[]){
   tib[0] = 1 ;
   return ;
}
```

```
int main(void){
    int tab[2] = { 0, 1} ;
    fct(tab) ;
    return tab[0] ;
}
```

C'est l'adresse qui est copiée. Ceci implique que la fonction principale retourne 1 dans notre exemple.

Dans `fct`, `tib[0]` fait référence à la première *cellule mémoire* définie dans le tableau local à la fonction principale.

Nous étendrons ce principe (passage de paramètre par adresse) aux autres types en utilisant la notion de pointeur.

Passage de paramètre : exemple

```c
void modifie_arg_int(int a){
    a = 5;
}
void modifie_arg_tableau(int t []){
    t[0] = 5;
}
int main(void){
    int mon_entier = 1;
    int mon_tab[3] = {1,2,3};


    printf("a=%d, ", mon_entier);
    printf("t[0]=%d, t[1]=%d, t[2]=%d\n", mon_tab[0], mon_tab[1],mon_tab[2]);

    modifie_arg_int(mon_entier);
    modifie_arg_tableau(mon_tab);

    printf("a=%d, ", mon_entier);
    printf("t[0]=%d, t[1]=%d, t[2]=%d\n", mon_tab[0], mon_tab[1],mon_tab[2]);
    return 0; /* valeur de retour */
}
```

```
$ ./a.out
a=1, t[0]=1, t[1]=2, t[2]=3
a=1, t[0]=5, t[1]=2, t[2]=3
$
```

Tableau en « sortie » d'une fonction

- Lorsqu'une fonction doit « renvoyer un tableau », la création et la manipulation de ce tableau sont difficiles.
- Usage courant par « effet » : passer le tableau en argument, utiliser void comme type de sortie.

```
int[] cree_tableau_uns(int N){
/*PB type de sortie*/
    int n;
    int t[N];
    /* PB N inconnu à la compil */
    for (n=0;n<N;n++){
        t[n] = 1;
    }
    return t;
}
int main(void){
    int u[3]; /* allocation en mémoire */
    u = cree_tableau_de_uns(3);
    /* PB u écrasé; u=... pas autorisé */
    printf("%d %d %d\n",u[0],u[1],u[2]);
    return 0;
}
```

```
void cree_tableau_uns(int t[], int N){
    int n;
    for (n=0;n<N;n++){
        t[n] = 1;
    }
}
int main(void){
    int u[3];
    cree_tableau_de_un(u,3);
    printf("%d %d %d\n",u[0],u[1],u[2]);
    return 0;
}
```

Définition d'une fonction : ANSI
Appel à une fonction
Passage de paramètres par copie
Les tableaux
Tableaux passés en paramètre d'une fonction
Exemple de programme : crible d'Ératosthène
Compilation séparée et Make
Exécution pas à pas dans l'environnement gnu debugger

Tableau bidimensionel

Bien que stockés linéairement, les tableaux peuvent être définis comme multidimensionel :

```
char tab[3][4]={"123","456","789"} ;

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

```
.file   "tableau2d.c"
.globl tab
.data
.type   tab,@object
.size   tab,12
tab:
.string "123"
.string "456"
.string "789"
```

La sémantique est la même que pour le cas monodimensionnel :

```
tab[3][0] = tab[3][0]++
```

Un petit coup d'oeil du coté de l'assembleur

```
        .file   "tableau.c"              char tab[] = "123" ;
.globl tab                               unsigned int i =0 ;
        .data                            int main(){
        .type   tab,@object
        .size   tab,4                            i = tab ;
tab:    .string "123"                            return 0 ;
.globl i                                 }
        .align 4
        .type   i,@object
        .size   i,4   /* Ce code compile en lan\c{c}ant un
i:      .long   0        avertissement~:
        .text            warning: assignment makes integer
        .align 2         from pointer without a cast   */
.globl main
        .type   main,@function
main:   .......
        movl    $tab, i   /* Nous verrons pourquoi lors de
        movl    $0, %eax     l'\'etude des pointeurs     */
        .......
```

Définition d'une fonction : ANSI
Appel à une fonction
Passage de paramètres par copie
Les tableaux
Tableaux passés en paramètre d'une fonction
Exemple de programme : crible d'Ératosthène
Compilation séparée et Make
Exécution pas à pas dans l'environnement gnu debugger

Aujourd'hui

- 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

Algorithme : crible d'Eratosthème

Objectif : déterminer tous les nombres premiers entre 0 et N-1.

Idée :

- parcourir les entiers de façon croissante ;
- le premier candidat rencontré est premier ;
- lorsqu'un nombre est premier, éliminer tous ses multiples et itérer jusqu'à atteindre N.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

Définition d'une fonction : ANSI

Appel à une fonction

Passage de paramètres par copie

Les tableaux

Tableaux passés en paramètre d'une fonction

Exemple de programme : crible d'Ératosthène

Compilation séparée et Make

Exécution pas à pas dans l'environnement gnu debugger

```
   #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;
}

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;
}

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.

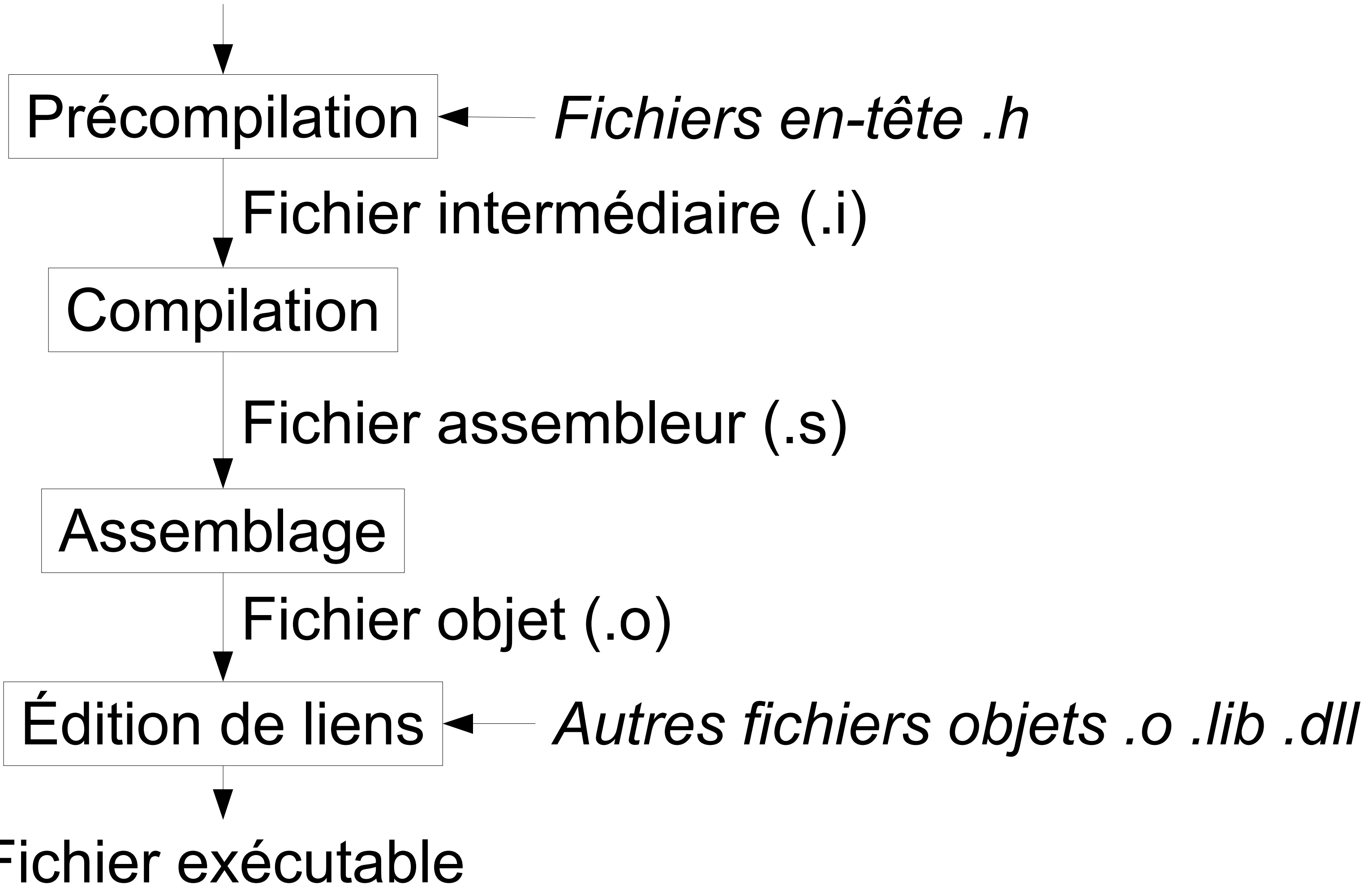
Il faut prendre garde à gérer correctement les *dépendances* entre les différents fichiers.

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).

Compilation



Définition d'une fonction : ANSI
Appel à une fonction
Passage de paramètres par copie
Les tableaux
Tableaux passés en paramètre d'une fonction
Exemple de programme : crible d'Ératosthène
Compilation séparée et Make
Exécution pas à pas dans l'environnement gnu debugger

La fonction principale `eratosMain.c` (déclarer les identificateurs) :

```
#include <stdio.h>
#include "eratosthene.h"
void init (void) ;             /* le prototype des fonctions */
int min_is_candidate(void) ; /* utilis\'ees doit \^etre    */
void set_non_prime(int) ;    /* disponible                 */
int prem[N];      /* la variable globale est d\'efinie ici */
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 ;
   }
```

Fichiers composant notre programme

Il est possible d'obtenir un fichier objet associée 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"
extern int prem [N] ; /* prototype de la variable globale */

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

Obtention d'un exécutable

Au final, on obtient

```
% gcc -c eratosInit.c
% ls
eratosInit.c   eratosMain.c   eratosMin.c   eratosSet.c
eratosInit.o   eratosMain.o   eratosMin.o   eratosSet.o
eratosthene.h
```

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

```
% gcc -o executable eratos*.o
%  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
```

Définition d'une fonction : ANSI
Appel à une fonction
Passage de paramètres par copie
Les tableaux
Tableaux passés en paramètre d'une fonction
Exemple de programme : crible d'Ératosthène
Compilation séparée et Make
Exécution pas à pas dans l'environnement gnu debugger

Définition d'une fonction : ANSI

Appel à une fonction

Passage de paramètres par copie

Les tableaux

Tableaux passés en paramètre d'une fonction

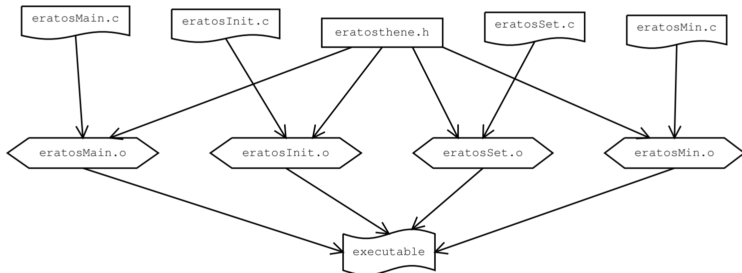
Exemple de programme : crible d'Ératosthène

Compilation séparée et Make

Exécution pas à pas dans l'environnement gnu debugger

Arbre de dépendance

Les opérations précédentes sont modélisées par l'arbre de dépendance :



Utilitaire `make` : syntaxe

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

Automatisation de la compilation :

- Maintenance, mise à jour et régénération de fichiers dépendants ;
- Sources $\longrightarrow$ 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)

```
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:
        rm -f *~ *.o executable
```

Définition d'une fonction : ANSI
Appel à une fonction
Passage de paramètres par copie
Les tableaux
Tableaux passés en paramètre d'une fonction
Exemple de programme : crible d'Ératosthène
Compilation séparée et Make
Exécution pas à pas dans l'environnement gnu debugger

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
	gcc $(OPTIONS) -c eratosInit.c
```

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/
```

Exemple

```
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 $<

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

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

eratosInit.o: eratosInit.c eratosthene.h
	gcc $(OPTIONS) -c $<
```

Aujourd'hui

- 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