Pratique du C
Fonction – tableau compilation séparée

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

Semestre 5 — 2010-2011

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

type-retour
identificateur-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.

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);
}
```


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

type-retour
identificateur-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.

Définition à la Kernighan et Ritchie

Syntaxe K&R : *type-retour* *identificateur-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 identificateurs 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`.

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.

Appel à une fonction

- Syntaxe de l'appel à une fonction : *expression-appel* :
  ⇒ *identificateur-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 *identificateur-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 (plutôt une valeur indéterminée) ;
- "Valeur indéterminée" 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 :

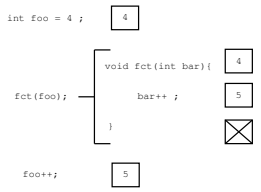
```
#include<stdio.h>
void testzero(int j) {
if(j) return ; /* provoque la sortie */
printf("test positif") ;
}
int main(void) {
testzero(0);
return 0 ;
}
```


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

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

```
int main(void){
   int foo = 5 ;
   fct(foo) ;
   return foo ;
}
```

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).



Les tableaux en C

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];
```


Points importants :

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

```
char tab[] = "123" ;

int main(){

        return 0 ;
}
```

```
        .globl tab
                .data
                .type   tab,@object
                .size   tab,4
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'un tableau lors de sa déclaration

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.

Manipulations élémentaires sur les tableaux

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

- Syntaxe :
  expression ⇐ *identificateur-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.

Passage d'un tableau en paramètre d'une fonction

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.

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"
.globl tab
        .data
        .type   tab,@object
        .size   tab,4
tab:    .string "123"
.globl i
        .align 4
        .type   i,@object
        .size   i,4
i:      .long   0
        .text
        .align 2
.globl main
        .type   main,@function
main:   .......
        movl    $tab, i
        movl    $0, %eax
        .......
```

```
char tab[] = "123" ;
unsigned int i =0 ;
int main(){

        i = tab ;
        return 0 ;
}
```

```
   /* Ce code compile en lan\c{c}ant un
      avertissement~:
      warning: assignment makes integer
      from pointer without a cast  */
```

```
   /* Nous verrons pourquoi lors de
      l'\'etude des pointeurs     */
```



```
#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>
#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 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"
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_NON_PRIME;
   for (i = 2; i < N; i = i + 1) prem[i] = IS_CANDIDATE;
   return ;
}
```


Obtention d'un exécutable

Au final, on obtient

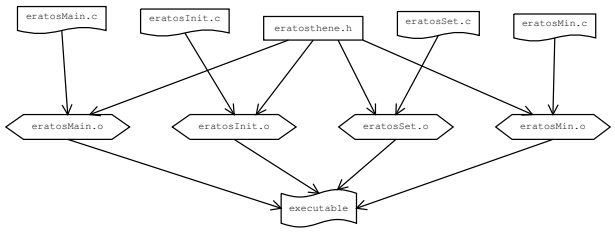
```
% gcc -c eratosInit.c
% ls
eratosInit.c  eratosMain.c  eratosMin.c  eratosSet.c
eratosInit.o  eratosMain.o  eratosMin.o  eratosSet.o
eratosthene.h
```

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
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
```


Arbre de dépendances

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



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


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/
```


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
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;
```


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.

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
```