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 4 — 2017-2018

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 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.c
     int main (void) {
  int a = 2 ;
  return 0;
}
```

Attributs de fonctions

Le comportement vis à vis des fonctions du compilateur GCC peut être finement manipulé par le biais de la macro à paramètre `__attribute__ ((param))`. Par exemple, on peut utiliser comme paramètre :

- `deprecated` afin de lancer un avertissement si la fonction correspondante est utilisée ;
- `constructor` afin de lancer la fonction correspondante avant le `main` ;
- `destructor` afin de lancer la fonction correspondante après le `main` ;
- `always_inline` afin d'inliner la fonction correspondante quelque soit l'option d'optimisation utilisée.

Ces fonctionnalités ne sont pas standardisées. Consultez la documentation pour plus d'informations.

La compilation et l'exécution du code suivant :

```
#include<stdio.h>
void foo(void) __attribute__ ((constructor)) ;
void bar(void) __attribute__ ((destructor)) ;
int old(int) __attribute__ ((deprecated)) ;
void foo(void) { printf("%s ",__FUNCTION__); return ; }
void bar(void) { printf("%s ",__FUNCTION__); return ; }
int toinline(int dummy) { return dummy+1 ; }
int old(int dummy) { int res ; res = dummy++ ; return res ;}
int main (void) { int tmp ; tmp = old(1) ;
     tmp = toinline(1)  ; printf("%s ",__FUNCTION__) ; retur
```

donnent :

```
% gcc foo.c ; ./a.out ; echo $?
foo.c: In function main:
foo.c:9: warning: old is deprecated (declared at foo.c:8)
foo main bar 2
```

Usage avancé (e.g. assert.h)

On peut convertir l'argument d'une macro en une chaîne de caractères en utilisant l'opérateur #.

Ceci permet des utilisations du type :

```
#include<stdio.h>
#define STRING(x) #x
#define ASSERT(x) if(!x) \
printf("%s not true in file: %s, line: %d,%function %s\n",\
STRING(x),__FILE__,__LINE__,__func__);
int main (void){
        int i = 3 ;
        ASSERT(i==0)
        return i ;
}
```

qui, si le source est `essai.c` produit, comme résultat :

```
% gcc essai.c ; ./a.out
% i==0 not true in file: essai.c, line: 8, function main
```

(à utiliser en phase de conception durant laquelle le message est pertinent).

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_de_modèle_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
        ....
```

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.

Par exemple, la déclaration du nouveaux type `entier` :

```
typedef int entier ;
```

permet les déclarations :

```
entier i,j=2 ;
```

Exemples de définition de type

On peut maintenant substituer des synonymes aux modèles que l'on déclare :

```
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:
   ........
```

Modèle de tableaux

En utilisant `typedef`, il est possible de déclarer un type (modèle) permettant de déclarer des tableaux. Par exemple :

```
typedef int TableauDe10Entiers[10] ;
```

ne déclare pas un tableau de 10 entiers mais permet de construire un synonyme (`TableauDe10Entiers`) utilisable dans la déclaration ultérieure de tableaux. Ainsi, la déclaration

```
TableauDe10Entiers a,b ;
```

correspond à la déclaration de 2 tableaux `a` et `b` de 10 cellules de type `int`.

La construction de ce genre de synonyme se base sur la déclaration classique d'une variable ; l'usage de `typedef` permet de considérer que ce qui aurait été l'identificateur de variable (sans `typedef`) est un identificateur de type (cf. le cours sur les pointeurs de fonctions pour en autre exemple).

Réaliser un codage d'informations en utilisant une suite de shannons

- Par exemple, identificateur dans une table des symboles : mot-clé, externe, statique :

```
#define MOT_CLE 01
#define EXTERNE 02
#define STATIQUE 04
```

  ou bien :

```
enum { MOT_CLE = 01, EXTERNE = 02, STATIQUE = 04};
```

- assignation

```
int drapeaux;
drapeaux |= EXTERNE | STATIQUE; /* Mise a 1 */
drapeaux &= ~(EXTERNE | STATIQUE); /* Mise \‘a 0 */
```

- test

```
if ((drapeaux & (EXTERNE|STATIQUE))==0) /*Test \‘a 0*/
```

Dans cet exemple, seuls 3 shannons sont utilisés sur 32.

Comme alternative, on peut utiliser des champs de lettres binaires

- champ dans une structure dont on spécifie la longueur ;
- considéré comme un petit entier ;
- pas d'adresse pour les membres ;
- exemple

```
struct
{
  unsigned int est_mot_cle : 1;
  unsigned int est_externe : 1;
  unsigned int est_statique : 1;
} drapeaux;
/* Mise \'a 1 */
drapeaux.est_externe = drapeaux.est_statique = 1;
/* Test \'a 0 */
if ( drapeaux.est_externe  == 0 &&
     drapeaux.est_statique == 0    ) ;
```

```
struct {
  unsigned int est_mot_cle : 1;
  unsigned int est_externe : 1;
  unsigned int est_statique : 1;
} drapeaux = {
/* Mise a 1 */
   .est_externe = 1,
   .est_statique = 1
};

int
main
(void)
{
   /* Test \‘a 0 */
   drapeaux.est_externe = 0 ;
   return 0 ;
}
```

```
        .data
.globl drapeaux
drapeaux:
        .byte   6
        .zero   3
        .text
.globl main

main:
  pushl   %ebp
  movl    %esp, %ebp
  subl    $8, %esp
  andl    $-16, %esp
  movl    $0, %eax
  subl    %eax, %esp
  andb    $-3, drapeaux
  movl    $0, %eax
  leave
  ret
```