Université du Luxembourg 2005–2006

Aide Mémoire - langage C

1 Directives de préprocesseur

- inclusion de fichiers
  pour un fichier système :
  `#include <nom_de_fichier>`
  pour un fichier du répertoire courant :
  `#include "nom_de_fichier"`
  ex : `#include <stdio.h>`
- définition de macros (remplacement textuel) :
  sans arguments :
  `#define nom valeur`
  avec arguments :
  `#define nom(`x_1`,...,`x_n`) f(`x_1`,...,`x_n`)`
  ex : `#define carre(x) ((x)*(x))`
- compilation conditionnelle :
  `#if #ifdef #ifundef`
  chacune de ces directives doit être terminée par `#endif`. Les texte entre les deux est pris en compte seulement si la condition est vérifiée.
  ex :
```
#ifdef DEBUG
   printf("Message de debug\n");
#endif
```

2 Commentaires

Entre `/*` et `*/`. Ne peuvent pas être imbriqués.
ex :
```
/* ceci est un commentaire valide */
/* ceci n'est pas un /* commentaire */ valide */
// Ceci est un commentaire sur une ligne
```

3 Structure d'un programme en C

```
/* Declarations globales (variables, prototypes, fonctions) */
...
int main (int argc, char *argv[])
{
  ...
}
```

4 Variables et types

Les types de base les plus répandus sont :
`short int long float double char`
Ils peuvent, entre autres, être prefixés par un ou plusieurs des modificateurs suivants :
`unsigned short long static extern register`

- variable(s) simple(s) :
  `type nom;`
  ou
  `type` $\texttt{nom}_1$`,...,`$\texttt{nom}_n$`;`
  ex : `unsigned int i,j;`

- tableau :
  `type nom[taille];`
  indéxé de `0` à `taille-1`. Accès à un élément `i` par `nom`[`i`].
  ex : `char chaine[25];`

- structure :
  ```
  struct nom_structure
  {
     type1 nom1;
     ...
     typen nomn;
  } nom;
  ```
  Le dernier nom est optionnel et déclare une variable ayant la structure définie comme type. Le champ `nom_structure` est optionnel est permet de nommer la structure pour y accéder ultérieurement avec `struct nom_structure`. Accès à un champ par l'opérateur `.` (point).
  ex :
  ```
  struct point
  {
     int x,y;
  };
  struct point mon_point;

  mon_point.x = 5;
  ```

- type :
  `typedef declaration_de_variable;`
  Le nom utilisé pour la déclaration est le nom du nouveau type défini.
  ex :
  ```
  typedef struct
  {
     int x,y;
  } point;
  point mon_point;

  mon_point.x = 5;
  ```

5 Fonctions

Chacune des fonctions peut être prototypée ou non. Dans tous les cas le prototype se place avant la declaration.

- prototype :
  `type_retour nom(type1 arg1, ..., typen argn);`
  ou
  `type_retour nom(type1, ..., typen);`
- déclaration :

```
type_retour nom(type1 arg1, ..., typen argn)
{
   /* déclaration des variables */
   ...
   /* instructions (corps) */
   ...
   /* si type_retour est différent de void */
   return valeur;
}
```

6 Opérateurs, Expressions

Une expression est constituée d'une valeur, d'une variable, ou de la combinaison de deux expressions par un opérateur arithmétiques ou logiques. Attention, en C, par convention, une valeur nulle est fausse et une valeur non nulle est vraie.

- opérateurs arithmétiques (addition, soustraction/négation, multiplication, division et modulo) :
  `+ - * / %`
- opérateurs logiques (conjonction, disjonction, négation, et comparaisons) :
  `&& || ! < > <= >= == !=`
- opérateurs bit-à-bit (ou, et, non, ou exclusif, décalage à gauche / à droite) :
  `| & ~ ^ << >>`
- pré/post incrémentation/décrémentation :
  `++ --`
  appliqués comme opérateur unaire pré/post-fixes à une variable. Incrémente/décrémente la variable. La valeur de l'expression est soit la valeur de la variable (si l'opérateur est postfixe), soit la valeur de la variable une fois incrémentée/décrémentée (si l'opérateur est préfixe).
- affectations :
  `= += -= *= /= %= |= &= ^= <<= >>=`
  affecte la valeur en partie droite à la variable en partie gauche (`=` simple). Pour la composition d'un opérateur avec l'affectation, affecte le resultat de l'opérateur appliqué à la valeur de la variable en partie gauche et la valeur en partie droite à la variable en partie gauche. La valeur de l'expression est la valeur affectée.

ex :

```
/* les expressions suivantes ont le même effet et la même valeur */
i=i+1;
i+=1;
++i;
/* i++ a le meme effet, mais pas la meme valeur */
```

7 Instructions

Une instruction est soit une expression, soit une structure de contrôle soit un bloc d'instructions délimité par `{` et `}`. Toute instruction doit être suivie d'un `;` (sauf un bloc).
ex :

```
/* une instruction */
i=0;
/* un bloc */
{
  i=0;
  j=1;
}
```

8 Structures de contrôle

- condition :
  ```
  if (expression)
     instruction1;
  ```
  ou
  ```
  if (expression)
     instruction1;
  else
     instruction2;
  ```
  si l'expression est vraie, effectue `instruction`$_1$, sinon effectue `instruction`$_2$ (le cas échéant).

- répétition tant que :
  ```
  while (expression)
     instruction;
  ```
  tant que l'expression est vraie, répette l'execution de l'instruction.

- répétition jusqu'à :
  ```
  do
     instruction;
  while (expression);
  ```
  répette l'execution de l'instruction jusqu'à ce que l'expression soit fausse.

- boucle :
  `for (expression`$_1$`;expression`$_2$`;expression`$_3$`)`
  `instruction;`
  évalue une (unique) fois `expression`$_1$, puis tant que `expression`$_2$ est vraie, execute `instruction` et évalue `expression`$_3$ (dans cet ordre).

ex :
```
for (i=0; i<n; i++)
    printf("%d\n",tableau[i]);
```

9 Pointeurs

- déclaration : `type *nom;`
- adresse d'une variable : `&variable`
- déréférence (accès à la valeur pointée) : `*pointeur`

- déréférence et accès à un champ (pour un pointeur sur structure) : `variable->champ` équivalent à (*variable).champ (les parenthèse sont nécessaires à cause de la précédence).

10 Quelques fonctions système utiles

- dans `malloc.h` (allocation et libération de mémoire) :
  `malloc free`
- dans `stdio.h` (entrées/sorties) :
  `printf scanf fopen fprintf fscanf fclose`
- dans `stdlib.h` (conversion entre types) :
  `atoi atol strtol`
- dans `ctype.h` (test sur les caractères) :
  `isalpha isdigit isalnum`
- dans `string.h` (manipulation de chaines) :
  `strlen strcpy strcmp`

11 Les mots réservés

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

12 Les utilisations de la fonction `printf`

Voici plusieurs exemples didactiques:

```
printf("|%d|\n",14);                    |14|
printf("|%d|\n",-14);                   |-14|
printf("|%+d|\n",14);                   |+14|
printf("|%+d|\n",-14);                  |-14|
printf("|% d|\n",14);                   | 14|
printf("|% d|\n",-14);                  |-14|
printf("|%x|\n",0x56ab);                |56ab|
printf("|%X|\n",0x56ab);                |56AB|
printf("|%#x|\n",0x56ab);               |0x56ab|
printf("|%#X|\n",0x56ab);               |0X56AB|
========================
printf("|%o|\n",14);                    |16|
printf("|%#o|\n",14);                   |016|
========================
printf("|%10d|\n",14);                  |        14|
printf("|%10.6d|\n",14);                |    000014|
printf("|%.6d|\n",14);                  |000014|
```

format	conversion en	écriture
%d	`int`	décimale signée
%ld	`long int`	décimale signée
%u	`unsigned int`	décimale non signée
%lu	`unsigned long`	décimale non signée
%o	`unsigned int`	octale non signée
%lo	`unsigned long`	octale non signée
%x	`unsigned int`	hexadécimale non signée
%lx	`unsigned long`	hexadécimale non signée
%f	`double`	décimale virgule fixe
%lf	`long double`	décimale virgule fixe
%e	`double`	décimale notation exponentielle
%le	`long double`	décimale notation exponentielle
%g	`double`	décimale, représentation la plus courte parmi %f et %e
%lg	`long double`	décimale, représentation la plus courte parmi %lf et %le
%c	`unsigned char`	caractère
%s	`char*`	chaîne de caractères

Table 1: Les différents formats de la fonction `printf`

```
printf("|%*.6d|\n",10,14);                |    000014|
printf("|%*.*d|\n",10,6,14);              |    000014|
========================
printf("|%f|\n",1.23456789012345689e5);     |123456.789012|
printf("|%.4f|\n",1.23456789012345689e5);   |123456.7890|
printf("|%.20f|\n",1.23456789012345689e5);  |123456.78901234567456413060|
printf("|%20.4f|\n",1.23456789012345689e5); |          123456.7890|
========================
printf("|%e|\n",1.23456789012345689e5);     |1.234568e+05|
printf("|%.4e|\n",1.23456789012345689e5);   |1.2346e+05|
printf("|%.20e|\n",1.23456789012345689e5);  |1.23456789012345674564e+05|
printf("|%20.4e|\n",1.23456789012345689e5); |           1.2346e+05|
========================
printf("|%.4g|\n",1.23456789012345689e-5);  |1.235e-05|
printf("|%.4g|\n",1.23456789012345689e5);   |1.235e+05|
printf("|%.4g|\n",1.23456789012345689e-3);  |0.001235|
printf("|%.8g|\n",1.23456789012345689e5);   |123456.79|
```