Les bases du langage C

Michael Mrissa
michael.mrissa@univ-pau.fr

Université de Pau
et des Pays de l'Adour

Note de l'enseignant

Cours initialement créé par A. Aoun, A. Benzekri, J.-M. Bruel, puis repris par Nicolas Belloir (merci pour les sources LaTeX)

Introduction

Un langage de programmation *de haut niveau* doit être traduit vers le langage machine.

Les langages interprétés

- Traduction instruction par instruction.
- Le programme est traduit à chaque exécution.
- Scheme, Shell Unix, PHP, Javascript ...

Les langages compilés

- Traduction des programmes dans leur ensemble.
- Le programme est traduit en une seule fois pour générer un binaire exécutable sur un processeur
- C, C++, Pascal ...
- Efficacité du programme, propriété intellectuelle ...

Un langage de programmation *de haut niveau* doit être traduit vers le langage machine.

Les langages interprétés

- Traduction instruction par instruction.
- Le programme est traduit à chaque exécution.
- Scheme, Shell Unix, PHP, Javascript ...

Les langages compilés

- Traduction des programmes dans leur ensemble.
- Le programme est traduit en une seule fois pour générer un binaire exécutable sur un processeur
- C, C++, Pascal ...
- Efficacité du programme, propriété intellectuelle ...

Un langage de programmation *de haut niveau* doit être traduit vers le langage machine.

Les langages interprétés

- Traduction instruction par instruction.
- Le programme est traduit à chaque exécution.
- Scheme, Shell Unix, PHP, Javascript ...

Les langages compilés

- Traduction des programmes dans leur ensemble.
- Le programme est traduit en une seule fois pour générer un binaire exécutable sur un processeur
- C, C++, Pascal ...
- Efficacité du programme, propriété intellectuelle ...

Interprété vs compilé

- Indépendance des processeurs ?
- Installation d'un interpréteur obligatoire ?
- Garantie de comportements identiques ?
- Coût de compilation/interprétation ?
- etc. . .

Le langage C, un langage de programmation

- **structuré**
- **compilé**

Compilation d'un programme en langage C

1. **Traitement par préprocesseur** : transformations textuelles.
2. **Compilation** : transformation du code source en assembleur.
3. **Assemblage** : transforme le code assembleur en binaire (fichier objet).
4. **Edition de lien** : lie les différents fichiers objets.

Le langage C, un langage de programmation

- **structuré**
- **compilé**

Compilation d'un programme en langage C

1. **Traitement par préprocesseur** : transformations textuelles.
2. **Compilation** : transformation du code source en assembleur.
3. **Assemblage** : transforme le code assembleur en binaire (fichier objet).
4. **Edition de lien** : lie les différents fichiers objets.

Structure d'un programme C

Niveaux d'écriture

Il existe 3 niveaux dans l'écriture d'un programme en langage C

- Fichiers,
  - contient un ensemble de fonctions,
  - une seule s'appelle "main" (point d'entrée du programme).
- Fonctions,
  - contient un seul bloc.
- Blocs.
  - contient un ensemble d'instructions,
  - est délimité par { }.

Chaque composante du programme possède ses propres variables.

Exemple de composante

Fichier	Fonction	Blocs

Variables
Fonction1
Fonction2
	Nom_fonction (Paramètres)
	Déclaration Paramètres ;
	Bloc_fonction
		{
		Variables_bloc ;
		<liste instructions>
		}

Exemple

```
#include <stdio.h> //preprocesseur
int main () {
        printf ("Hello word");
}
```

Exemple

```c
#include <stdio.h> //preprocesseur
#define TVA 19.6

// fonction
float calcule_TVA(float prixHT){
  return ((prixHT * TVA) / 100);
}

int main () {   // point d'entrée du programme
        float HT;
        scanf ("%f", &HT);          // saisie prix HT
        HT = HT + calcule_TVA (HT); //calcul prix TTC
        printf ("prix TTC = %f\n", HT); // affichage
        printf("\n");               //Retour à la ligne
        return (0);                 // Point de sortie
}
```

Identificateur

- Suite de caractères permettant de reconnaître une entité du programme (Variables, Fonctions, ...),
- Est composé d'un ou plusieurs caractères.
  - Le premier parmi [A..Z] [a..z] [_],
  - Les suivants parmi [A..Z] [a..z] [_] [0..9].

Exemple

```
nombre_max, NbMax, NombreMax1 ...
```

Le langage C possède un vocabulaire dont les mots ne peuvent pas être pris comme des identificateurs d'un programme.

Les mots réservés

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

Remarque

Le mot clé entry n'est actuellement pas utilisé, et réservé pour de futures utilisations. Certains compilateurs réservent aussi des mots clé comme pascal, fortran et asm.

Variables, types et déclaration

Une variable

- est l'association d'un identificateur à une zone mémoire contenant la valeur de cette variable,
- possède une classe d'enregistrement, un type,
- doit être déclarée et être allouée avant son utilisation.

Type de variable (1/3)

Une déclaration s'accompagne de la définition d'un type pour cette entité.

Type de donnée	Signification	Taille (octets)	Plage de valeurs
char	caractère	1	-128 à 127
unsigned char	caractère non signé	1	0 à 255
short int	entier court	2	-32768 à 32767
unsigned short int	entier court non signé	2	0 à 65.535

Type de variable (2/3)

Type de donnée	Signification	Taille (octets)	Plage de valeurs
int	entier	2 [a]	-32768 à 32767
int	entier	4 [b]	-2147483648 à 2147483647
unsigned int	entier non signé	4	0 à 4294967295
long	entier long	4	-2147483648 à 2147483647
unsigned long	entier long non signé	4	0 à 4294967295

a. sur processeur 16 bits
b. sur processeur 32 bits

Type de variable (3/3)

Type de donnée	Signification	Taille (octets)	Plage de valeurs
float	réel en simple précision	4	$3,4 * 10^{-38}$ à $3,4 * 10^{38}$
double	réel en double précision	8	$1,7 * 10^{-308}$ à $1,7 * 10^{308}$
long double	flottant double long	10	$3.4 * 10^{-4932}$ à $3.4 * 10^{4932}$

Syntaxe pour la déclaration d'une variable

Déclaration

```
<classe d'enregistrement> <type> <nom>;
```

Il est possible pour certaines classes de variables de donner une valeur au moment de la déclaration. Exemples :

Exemple

```
static long hexa = 0xFFL;
extern float reel = 0., pi = 3.14159;
```

Exemple I

```
#include <stdio.h>          /* Fichier d'entête */
#define TAILLE 20           /* Constante littérale */
int somme;                  /* Variable du fichier */
void main()                 /* Fonction principale */
{                           /* Début du bloc de la fonction main */
  int a;
  int b;                    /* Variables du bloc */

 /* Affichage d'un message, et saisie au clavier de deux enti
  printf("Entrez deux entiers séparés par un blanc\n");
  scanf("%d%d",&a,&b);
  if (a >= TAILLE)
  {     /*Début du bloc */
    int c;                  /*Variable du bloc */
    c = a + TAILLE;         /*Calcul de la valeur de la variable c */
    a = 2 * c;              /*Calcul de la valeur de la variable a */
```

Exemple II

```
    } /*Fin du bloc */
    somme = a + b;      /*calcul de la somme */
    /*Affiche la valeur de la variable somme */
    printf("La somme des deux entiers est : %d \n", somme);
} /*Fin du bloc de la fonction main */
```

Instructions, expressions, opérateurs, et constantes

Définition

Une instruction est une action élémentaire du langage qui se termine obligatoirement par le caractère ' ;'. Les instructions sont exécutées séquenciellement de la première à la dernière sauf dans le cas d'instructions de contrôle.

Remarque

Dans le langage C on distingue plusieurs catégories d'instructions : les instructions nulles, les instructions expressions, les instructions composées ou les instructions de contrôle.

L'instruction nulle

Elle est représentée par un point-virgule seul. Elle est utile pour positionner une étiquette, ou pour munir une boucle d'un corps vide.

Exemple

```
;
```

L'instruction expression

Généralement les instructions expressions sont des opérations d'affectation ou des appels à des fonctions. La plupart des instructions sont des instructions expression sous la forme :

Exemple

```
<expression>;
```

L'instruction composée ou bloc

On utilise une instruction composée (bloc) là où on n'en voudrait qu'une. Généralement à l'intérieur d'instructions de contrôle (itérative, conditionnelle, autre).

Exemple

```
{
    < liste de déclarations >
    < liste d'instructions >
}
```

Les instructions de contrôle

Ce sont des instructions qui changent l'ordre d'exécution séquentielle des instructions, elles permettent les choix, les itérations, les déroutements.

Les expressions

Les expressions sont la combinaison d'opérateurs, d'identificateurs et de constantes. Certaines règles sont à connaître dans l'écriture d'expressions.

- Une constante, un identificateur sont des expressions.
- Une chaîne est une expression.
- L'appel d'une fonction est une expression.
- Une expression entre parenthèses est une expression.
- La combinaison d'expressions par un opérateur est une expression.
- Une expression suivit d'une expression entre crochets est une expression.
- Une '*Lvalue* suivie d'un '.' et identificateur est une expression.

Lvalue

- Nom anglais, vient de "Left value".
- Une *Lvalue* est une expression qui fait un renvoi vers une zone mémoire que l'on peut manipuler.
- La *Lvalue* persiste après la ligne qui la mentionne.
- Concrètement, c'est une "case mémoire".
- Un identificateur de variable est une *Lvalue*
- Certains opérateurs donnent comme résultat une *Lvalue* par exemple l'opérateur * sur un pointeur (déréférencement).

Opérateurs arithmétiques

−	− <exp>
*/%	<exp1> OP <exp2>
− +	<exp1> OP <exp2>

Exemple

```
a = b - 12;
b = a * c;
b = b * ( 12 + c);
```

Opérateurs logiques et relationnels

!	!<exp>
&& ||	<exp1> OP <exp2>
== != > < >= <=	<expl> OP <exp2>

Exemple

```
a = !b;
a = c == b;
a = b && c;
```

Opérateurs d'affectation

=	<Lvalue> = <exp>
OP =	<Lvalue> OP= <exp>

Exemple

```
a += 12;
```

Erreur classique

Le programmeur veut affecter b à a.

```
/* il aurait du écrire : */ a = b;
/* il a écrit : */ a == b;
```

Opérateurs incrémentation, décrémentation

++	++<Lvalue> ou <Lvalue>++
--	--<Lvalue> ou <Lvalue>--

Exemple

```
a = i++;
```

Opérateurs de traitements binaires

```
~          ~<exp>
>> <<      <exp> OP <exp2>
& | ^      <exp1> OP <exp2>
```

Exemple :

```
unsigned a = 0xF0F0, b = 2, c = a >> b;

/* c == 0011 1100 0011 1100 soit 0x3C3C */
```

Opérateurs d'indirection

```
*          *<exp>
&          &<Lvalue>
```

Opérateur "sizeof"

```
sizeof
sizeof <exp>
sizeof (<nom_type>)
```

Opérateurs d'évaluation

```
()          (<exp>)  ,          <exp1>,<exp2>
```

Exemples

```
i = (j = 2, 1);
for(i = 1, j = 1; i <= LIMITE; i++, j = j + 2){;}
```

Priorité des opérateurs

- Priorité 1 (la plus forte) : ()
- Priorité 2 : ! ++ --
- Priorité 3 : * / %
- Priorité 4 : + -
- Priorité 5 : < <= > >=
- Priorité 6 : == !=
- Priorité 7 : &&
- Priorité 8 : ||
- Priorité 9 (faible) : = += -= *= /= %=

Même priorité dans la même classe (de 1 à 9)

Dans le code : évaluation de gauche à droite **dans l'expression**

Priorité des opérateurs

Attention : pour a = b = 1 ; on a une différence entre

- a = b++ ; //a est à 1
- a = ++b ; //a est à 2
- mais ++ est bien prioritaire sur l'affectation, et la priorité est bien respectée
- Explication : pré-incrémentation et post-incrémentation
  - b++ incrémente b et renvoie **l'ancienne valeur de b**
  - ++b incrémente b et renvoie **la nouvelle valeur de b**
- En résumé : écrire du code clair et lisible
- séparer les instructions (b++ ; a = b ; ou l'inverse) peut être utile
- Pour aller plus loin : `http://c-faq.com/expr/index.html`

Les constantes

Décimales	10	-32768L
Octales	03677	0177
Hexadécimales	0xFF	0xA1L
Réelles	3.14159	-1,234e18
Caractères	'A'	'\n'
Chaînes	"Bonjour"	"Oui"

Les instructions de contrôle

Format

```
if ( <expression> )
    <instruction1>
[else
    <instruction2>
]
```

Fonctionnement

- L'instruction1 est exécutée *si* le résultat de l'expression est VRAI,
- sinon c'est l'instruction2 qui est exécutée.
- La partie *else* de l'instruction est facultative.

Exemple

```
if (a==b)
    a = a + 1;
else
    b = b + 1;
```

```
if (a>b) {
    b = b + 1;
    a = b;
}
```

Erreur classique

Le programmeur veut tester si a > b :

```
/* Ce qu'il a écrit */ if (a > b);
/* Ce qu'il doit écrire */ if (a > b)
```

Le programmeur veut tester si a > b :

```
/* Ce qu'il doit écrire */
if (a > b)
    { if ( x > y ) x = y ;}
else
    ...
```

```
/* Ce qu'il a écrit */
if (a > b)
    if ( x > y ) x = y ;
else
    ...
```

Remarque

Dans le langage C, nous avons :

VRAI == différent de zéro
FAUX == égal à zéro
"En C tout est vrai sauf 0 qui est faux"

donc : if (a != 0) $\Longleftrightarrow$ if (a)

Itération 'tant que...faire' : *while*

Format

```
while ( < expression > )
  < instruction >
```

Fonctionnement

- L'instruction est répétée tant que le résultat de l'expression est VRAI.
- S'utilise lorsque le nombre d'itération n'est pas déterminé avant le départ de la première itération.
- La condition est évaluée avant le départ de l'itération.

Exemple

```
while ( a != 10 ) {
    sum = sum + a;
    a = a + 1;
}
```

Remarque

Dans l'exemple ci-dessus :

- si a vaut 0 il y aura 10 itérations ;
- si a $>$ 10 il y aura un nombre indéterminé d'itérations.

Itération 'pour...faire' : *for*

Format

```
for ( <exp. init.>; <exp. condition>; <exp. évolution> )
        < instruction >
```

Fonctionnement

- L'instruction est répétée tant que le résultat de l'expression condition est VRAI.
- S'utilise lorsque le nombre d'itérations à effectuer est exactement défini.

Exemple

```
for (sum=0, i=0; i != 10; i = i+ 1) {
    a = 2 * i;
    sum = sum + a;
}
```

Equivalence entre la boucle *for* et *while*

Exemple

```
i=0;
while (i < 10) {
    sum = sum + i;
    i= i + 1;
}
```

```
for(i = 0; i < 10; i++)
    sum = sum + i;
```

Remarque

Dans toutes les formes des itérations, il faut veiller à faire évoluer la condition pendant l'itération.

Exemple

Exemples à **ne pas** suivre !

```
/* itérations infinies */
i = 4;
while (i<10) sum = sum + i;
...
for (i = 0; i < 10; a = a + 1) sum = i + sum;
```

Remarque

- Dans la forme de l'itération for il est possible de supprimer certains éléments.
- Cependant **ces formes sont à éviter** car elles sont la source d'erreurs d'appréciation dans la lecture du code source.

Exemple

```
for(; i<10; i++) /* initialisation effectuée avant */
for(;;); /* boucle infinie */
for(i=0; i++ < 10;) /*évolution & condition groupées */
```

Itération 'faire...tant que' : *do...while*

Format

```
do <instruction>
while ( <expression> );
```

Fonctionnement

- L'instruction est répétée tant que le résultat de l'expression est VRAI.
- Dans ce type d'itération, la condition n'est vérifiée qu'après l'itération, donc l'instruction est au moins exécutée une fois.

Exemple

```
do{
      sum = sum + i;
      i = i + 1;
} while (i<10);
```

Sortie de bloc : *break*

Format

```
break ;
```

Fonctionnement

- Construction pour terminer de façon anormale le déroulement de l'exécution à l'intérieur d'un bloc.
- L'exécution se poursuit à l'instruction suivant immédiatement la fin du bloc.

Exemple

```
i=0; while (i < 20) {
    sum = sum + i;
    if (i==14) /* Arrêt de la boucle lorsque i == 14 */
        break;
    i = i + 1;
}
```

Itération suivante : *continue*

Format

```
continue ;
```

Fonctionnement

Construction pour sauter l'exécution de la fin de l'itération en cours, pour passer à l'itération suivante.

Exemple

```
i=0; while (i<20) {
    if (i == 12) { /* élimine l'itération i == 12 */
        i++;    /* attention à faire évoluer i */
        continue;
    }
    sum += 1;
    i++;
}
```

Branchement inconditionnel : *goto*

Format

```
goto <etiquette>;
```

Fonctionnement

- S'utilise pour dérouter de manière inconditionnelle l'exécution séquentielle des instructions ;
- L'utilisation du goto **n'est pas conseillée** pour la compréhension des programmes.

Exemple

```
i++;
sum+ =i ;
goto monlabel;
...
monlabel: sum /= 4;
```

Sortie de fonction : *return*

Format

```
return [(<valeur>)];
```

Fonctionnement

L'instruction return est utilisée pour terminer l'exécution d'une fonction avec la possibilité de retourner une valeur au contexte qui a appelé cette fonction.

Exemple

```
int echange (int x, int y) {
    ...
    return(0); /* fin fonction, retourne l'entier 0 */
}
```

Choix multiples : *switch...case*

Format

```
switch (< expression entière >) {
    case <constante entière> :
        <instruction 1 >
        ...
        < instruction N >
    ...
  [  default :
        < instruction1 >
        ...
        < instruction N >
  ]
}
```

Fonctionnement

L'instruction switch permet de mettre en place une structure d'exécution qui permet des choix multiples parmi des cas de même type et faisant intervenir uniquement des valeurs constantes entières (char, short, int).

Exemple

	a==1	a==2	a==3	autre
switch(a) {				
case 1 : b++ ;	+1			
case 2 : b++ ;	+1	+1		
break ;	fin	fin		
case 3 : b– ;			-1	
default : b– ;			-1	-1
}			fin	fin
/* resultat final */	b+2	b+1	b-2	b-1

Choix imbriqués : *else if*

Format

```
else if (<expression>)
```

Fonctionnement

C'est la construction à choix multiples, elle est utilisée lorsque, l'évaluation des diverses conditions possibles, ne correspond pas à des constantes entières comme dans l'instruction *switch*.

Exemple

```
if (a<b)
    printf("%d est plus petit que %d",a,b);
else if (a>b)
    printf("%d est plus grand que %d",a,b);
else
    printf("%d est égal à %d",a,b);
```

L'opérateur conditionnel : ?

Format

```
<condition> ? <expression1> : <expression2>
```

Fonctionnement

Le résultat de l'évaluation est conditionné par la condition :

- si < condition > == VRAI résultat est < expression1 >
- si < condition > == FAUX résultat est < expression2 >

Exemple

```
plus_grand = (a > b) ? a : b;
...
while (a != b)
    a>b ? a-- : a++;
```

Les entrées sorties formatées

La fonction *printf*

Format

```
printf("<texte><format>", <nom_variable>);
```

Fonctionnement

La fonction *printf* est une fonction qui permet d'afficher sur la console des messages en fonction du format qui lui est donné.

La fonction *printf*

Exemple

```
int result=0;
char sMot[]="Hello";

   printf("Hello");
   prinf("mot : %s",sMot);
   printf("résultat : %d", result);
```

La fonction *scanf*

Format

```
scanf("<format>", <adresse de la variable>);
```

Fonctionnement

La fonction *scanf* est une fonction qui permet d'entrer à partir de la console des valeurs qui seront stockées à l'intérieur des paramètres. Attention au format !

La fonction *scanf*

Exemple

```
int iNum;
char cLettre;
char sMot[10];

   scanf("%d", &iNum);
   scanf("%c", &cLettre);
   scanf("%s", sMot);
```

Le manuel du développeur

- outil primordial
- pour les appels de fonction
- pour les inclusions de bibliothèques
- pour le debug
- Ex : man 3 printf