Licence informatique & vidéoludisme
Semestre 1

Méthodologie de la programmation

Chapitre 3

La programmation en C

Langage C

Ressources utiles :

- `https://formationc.pages.ensimag.fr/prepa/prof/docs/c-cheatsheet-ensimag.pdf`
- cplusplus.com : `https://cplusplus.com/reference/cstdio/printf/`

Langage C - exemple

```c
/* instruction préprocesseur */
#include <stdio.h>

// Point d'entrée du programme : fonction "main"
int main() {
    printf("Hello, World! \n");
    return 0;
}
```

Langage C - exemple

```c
/* instruction préprocesseur */
#include <stdio.h>

// Point d'entrée du programme : fonction "main"
int main() {
    printf("Hello, World! \n");
    return 0;
}
```

```python
if __name__ == "__main__":
    print("Hello, World! \n")
    return 0
```

Langage C - exemple

```c
/* instruction préprocesseur */
#include <stdio.h>

// Point d'entrée du programme : fonction "main"
int main() {
    printf("Hello, World! \n");
    return 0;
}
```

- tout programme C dispose d'une fonction principale
- toute instruction termine par un ";"
- les blocs sont définis par des accolades
- les commentaires peuvent être définis encadrés par "/*" "*/" (multiligne) ou précédés de "//" (monoligne)

Langage C - Composants élémentaires

- les identificateurs
- les mots-clefs,
- les constantes,
- les chaînes de caractères,
- les opérateurs,
- les signes de ponctuation.

Langage C - Identificateurs

les identificateurs donnent un nom à une entité du programme :

- un nom de variable ou de fonction,
- un type défini par typedef, struct, union ou enum,
- une étiquette.

Un identificateur est une suite de caractères parmi :

- les lettres (minuscules ou majuscules, mais non accentuées),
- les chiffres,
- le “blanc souligné” (_).

les mots-clefs ne peuvent pas être des identificateurs !

Langage C - Mots-clefs

les Mots-clefs sont réservés pour le langage lui-même

- les spécificateurs de stockage

```
1 auto   register   static   extern   typedef
```

- les spécificateurs de type

```
1 char   double   enum   float   int   long   short   signed   struct
2 union   unsigned   void
```

- les qualificateurs de type

```
1 const   volatile
```

- les instructions de contrôle

```
1 break   case   continue   default   do   else   for   goto   if
2 switch   while
```

- divers

```
1 return   sizeof
```

Langage C - Structure d'un programme

- expression
- instruction
- blocs

```
1 [directives au préprocesseur]
2 [déclarations de variables externes]
3 [fonctions secondaires]
4
5 main()
6 {
7     déclarations de variables internes
8     instructions
9 }
```

Langage C - Expressions

une expression est une suite de composants élémentaires *syntaxiquement correcte* :

```
1 x = 0
2 (i >= 0) && (i < 10)
```

Remarque : La valeur rendue par les opérateurs relationnels est de type **int** (il n'y a pas de type booléen en C) : elle vaut 1 si la condition est vraie, et 0 sinon.

Langage C - Instruction / Déclaration

une instruction est une expression suivie d'un **point-virgule** qui signifie : « évalue l'expression »

```
1 if (x != 0)
2     {
3       z = y / x;
4       t = y % x;
5     }
```

une déclaration est instruction composée d'un *spécificateur de type* et d'une liste d'identificateurs séparés par une virgule

```
1 int a;
2 int b = 1, c; // on déclare une variable b qui vaut 1 et une variable c
3 double x = 2.38e4;
4 char message[80];
```

En C, toute variable doit faire l'objet d'une déclaration avant d'être utilisée.

Langage C - Déclaration de fonctions secondaires

les fonctions secondaires

```
type ma_fonction ( arguments )
{
    déclarations de variables internes
    instructions
}
```

exemple :

```
int produit(int a, int b)
{
  int resultat;

  resultat = a * b;
  return(resultat);
}
```

Langage C - Types d'objets

Le type d'un objet définit la façon dont il est représenté en mémoire.

Langage C - Types d'objets

Types entiers

```
1 char
2 short
3 int
4 long
5 long long
```

les entiers peuvent être signés ou non

Langage C - Types d'objets

Entiers non-signés :

```
1 unsigned char /* 1 octet [0, 256[ */
2 unsigned short /* 2 octets [0, 65536[ */
3 unsigned int /* 2 (4) octets */
4 unsigned long /* 4 (8) octets */
5 unsigned long long /* 8 octets */
```

Un entier non signé ne peut contenir que des valeurs positives.

Langage C - Types d'objets

Entiers signés : le signe est porté par le bit de poids fort

```
1 unsigned char /* 1 octet [-128, 128[ */
2 unsigned short /* 2 octets [-32768, 32768[ */
3 unsigned int /* 2 (4) octets */
4 unsigned long /* 4 (8) octets */
5 unsigned long long /* 8 octets */
```

Un entier signé peut contenir des valeurs positives et négatives

Langage C - Types d'objets

Flottants (nombres réels)

```
1 float /* 4 octets [1.2e-38, 3.4e+38] */
2 double /* 8 octets [2.3e-308, 1.7e+308] */
3 long double /* 10 octets [3.4e-4932 to 1.1e+4932] */
```

les différents types correspondent à différents niveaux de précision possibles

Langage C - le type char

N'importe quel élément du jeu de caractères de la machine utilisée

la plupart des machines utilisent désormais le jeu de caractères ISO-8859 (sur 8 bits) :

- les 128 premiers caractères correspondent aux caractères ASCII
- Les 128 derniers caractères sont utilisés pour les caractères propres aux différentes langues (par exemple é, è, etc.)

	déc.	oct.	hex.		déc.	oct.	hex.		déc.	oct.	hex.
	32	40	20	@	64	100	40	`	96	140	60
!	33	41	21	A	65	101	41	a	97	141	61
"	34	42	22	B	66	102	42	b	98	142	62
#	35	43	23	C	67	103	43	c	99	143	63
$	36	44	24	D	68	104	44	d	100	144	64
%	37	45	25	E	69	105	45	e	101	145	65
&	38	46	26	F	70	106	46	f	102	146	66
'	39	47	27	G	71	107	47	g	103	147	67
(	40	50	28	H	72	110	48	h	104	150	68
)	41	51	29	I	73	111	49	i	105	151	69
*	42	52	2a	J	74	112	4a	j	106	152	6a
+	43	53	2b	K	75	113	4b	k	107	153	6b

Langage C - le type char

Un objet de type `char` peut être assimilé à un entier :

```
1 main()
2 {
3   char c = 'A';
4   printf("%c", c + 1);
5 }
```

Langage C - Les Types

Il est possible de transformer un type en un autre.! Les cast sont destructifs

```c
int a = 15 / 6;
// a == 2
float b = (float) 15 / 6;
// b == 2.500000
int c = (int) b;
// c == 2
```

Langage C - Les opérateurs

Affectation : variable = expression

L'affectation effectue une conversion de type implicite

```
1 main()
2 {
3   int i, j = 2;
4   float x = 2.5;
5   i = j + x;     // i vaut 4
6   x = x + i;
7   printf("\n %f \n",x); // affiche 6,5
8 }
```

Langage C - Les opérateurs

Affectation : variable = expression

L'affectation effectue une conversion de type implicite imposé par le type du membre de gauche

```
1 main()
2 {
3   int i, j = 2;
4   float x = 2.5;
5   i = j + x;
6   x = x + i;
7   printf("\n %f \n",x);
8 }
```

Langage C - Les Opérateurs

Opérateurs arithmétiques :

```
1 a + b
2 a - b
3 a * b
4 a / b // attention, même opérateur pour les int et les float
5 a % b
6 a++
7 ++a
8 a--
9 --a
```

Langage C - Les Opérateurs

division entière et flottante : cela dépend du type des *opérandes*

```
1 float x;
2 x = 3 / 2; // x vaut 1
3 x = 3 / 2.; // x vaut 1,5
```

Langage C - Les Opérateurs

Opérateurs logiques booléens

```
1 // AND
2 a && b
3 // OR
4 a  b
5 // XOR
6 a != b
7 // NOT
8 !(a && b)
```

l'évaluation se fait de gauche à droite :

```
1 int a=12
2 int b=3
3 if ((a >= 0) && (a <= 9) && !(b == 0))
```

le troisième clause n'est pas évaluée car le seconde renvoie 0

Langage C - Les Opérateurs

Comparateurs bit à bit :

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Langage C - Les Opérateurs

```
1  // bitwise AND
2     11001000
3  &  10111000
4  ------------
5  =
6
7  // bitwise OR
8     11001000
9   \  10111000
10 ------------
11 =
```

Langage C - Les Opérateurs

```
1  // bitwise XOR
2     11001000
3  ^  10111000
4  ----------
5  =
6
7  // bitwise NOT
8  ~  11001000
9  ----------
10 =
```

Langage C - Les Opérateurs

l'opérateur d'adresse & appliqué à une variable retourne l'adresse-mémoire de cette variable. La syntaxe est : &objet

Langage C - Les Opérateurs

```
1  // bitwise XOR
2     11001000
3  ^  10111000
4  ----------
5  =
6
7  // bitwise NOT
8  ~  11001000
9  ----------
10 =
```

Langage C - Les branchements conditionnels

```
1 if (expression_1 )
2   instruction_1
3 else if (expression_2 )
4   instruction_2
5     ...
6 else if (expression_n )
7   instruction_n
8 else
9   instruction_n+1
```

chaque instruction peut être un bloc d'instructions

Langage C - l'opérateur ternaire

condition ? expression-1 : expression-2

```
1
2 a = (test expression) ? b : c;
3 // équivalent à :
4 if (test expression) {
5     a = b
6 } else {
7     a = c
8 }
```

Langage C - boucles

boucle while

```
1 while (expression)
2   instruction
```

- Tant que *expression* est vérifiée ($\neq 0$), *instruction* est exécutée.
- Si *expression* est nulle au départ, **instruction ne sera jamais exécutée**.
- *instruction* peut évidemment être une instruction composée. Par exemple, le programme suivant imprime les entiers de 1 à 9.

```
1 i = 1;
2 while (i < 10)
3   {
4     printf("\n i = %d",i);
5     i++;
6   }
```

Langage C - boucles

boucle do while :

```
1 do
2   instruction
3 while (expression);
```

- ici, *instruction* sera exécutée tant que *expression* est non nulle. *instruction* est donc toujours exécutée au moins une fois.
- par exemple, pour saisir au clavier un entier entre 1 et 10 :

```
1 int a;
2 do
3   {
4     printf("\n Entrez un entier entre 1 et 10 : ");
5     scanf("%d",&a);
6   }
7 while ((a <= 0)  (a > 10));
```

Langage C - boucles

boucle for

```
for (expr 1 ;expr 2 ;expr 3)
  instruction
```

```
// for
for (init variant; test expression; update variant) {
// code
}
```

Langage C - boucles

boucle for

```
1 for (init variant; test expression; update variant) {
2     // code
3 }
```

Que se passe-t-il ?

1. Le variant de boucle est initialisé.
2. l'expression est ensuite évaluée.
3. Le contenu de la boucle est exécuté.
4. Le variant de boucle est incrémenté
5. Retour à 2

```
1 for (i = 0; i < 10; i++)
2   printf("\n i = %d",i);
```

Langage C - Le switch

```
1  switch (expression )
2    {
3    case constante_1:
4      instructions 1
5      break;
6    case constante_2:
7      instructions 2
8      break;
9      ...
10   case constante_n:
11     instructions n
12     break;
13   default:
14     instructions n+1
15     break;
16   }
```

Branchement non conditionnel break

break permet de sortir d'une boucle et d'exécuter l'instruction suivant la fin de la boucle

```
1  main()
2  {
3    int i;
4    for (i = 0; i < 5; i++)
5      {
6        printf("i = %d\n",i);
7        if (i == 3)
8          break;
9      }
10   printf("valeur de i a la sortie de la boucle = %d\n",i);
11 }
```

affiche :

Branchement non conditionnel break

break permet de sortir d'une boucle et d'exécuter l'instruction suivant la fin de la boucle

```
1  main()
2  {
3    int i;
4    for (i = 0; i < 5; i++)
5      {
6        printf("i = %d\n",i);
7        if (i == 3)
8          break;
9      }
10   printf("valeur de i a la sortie de la boucle = %d\n",i);
11 }
```

affiche :

```
1  i = 0
2  i = 1
3  i = 2
4  i = 3
```

Branchement non conditionnel "continue"

continue permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle.

```
1  main()
2  {
3    int i;
4    for (i = 0; i < 5; i++)
5      {
6        if (i == 3)
7          continue;
8        printf("i = %d\n",i);
9      }
10   printf("valeur de i a la sortie de la boucle = %d\n",i);
11 }
```

affiche :

Branchement non conditionnel "continue"

continue permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle.

```c
main()
{
  int i;
  for (i = 0; i < 5; i++)
    {
      if (i == 3)
        continue;
      printf("i = %d\n",i);
    }
  printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

affiche :

```
i = 0
i = 1
i = 2
i = 4
```

Fonctions

Comme en Python il est possible de déclarer des fonctions.

- En C le type des variables retournées doit être spécifié.

```
1 type_de_retour nom(paramètres) {
2     // corps de la fonction
3 }
```

Programme en C

Un programme en C est une suite de déclaration de fonctions et variables

- Une fonction spéciale main est le point d'entrée du programme.
- La fonction main renvoie un entier :
  - si le programme termine normalement, celui-ci doit être 0 ;
  - sinon, celui-ci est un code d'erreur.
- La fonction main reçoit deux arguments :
  - un entier égal au nombre d'arguments reçus par le programme,
  - un tableau de chaînes de caractères avec les valeurs des arguments.

```
1 int main (int argc, char *argv[])
2 {
3     // le programme
4     return 0;
5 }
```

Langages de programmation

langage interprété vs. langage compilé

- langage *interprété*
  1. **code source** + **données d'entrée** → **interpréteur** → **données de sortie**
  - traduction en code machine « à la volée » (++ temps d'exécution)
  - Ex : Python, Javascript, PHP, Ruby, etc.
- langage *compilé*
  1. **code source** → **compilateur** → **code binaire** (fichier *exécutable*)
  2. **code binaire** + **données d'entrée** → **SE** → **données de sortie**
  - traduction du code *une fois pour toutes* (−− temps d'exécution)
  - indiqué pour les problématiques de temps réel (JV, aérospatiale, SE)
  - Ex : C, C++, Ada, etc.

la différence ne tient pas tant aux langages eux-mêmes mais à *la manière* dont on les utilise

Compilation en C

1. traitement par le préprocesseur : le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source ...).
2. compilation : traduit le fichier généré par le préprocesseur en **assembleur**, (suite d'instructions du microprocesseur)
3. assemblage : transforme le code assembleur en un fichier binaire : instructions directement compréhensibles par le processeur. Le fichier produit est le **fichier objet**
4. édition de liens : un programme est souvent séparé en plusieurs fichiers source (librairies notamment), il faut donc lier entre eux les différents fichiers objets. Le fichier produit est l'**exécutable**

Compilation en C

1. traitement par le préprocesseur : le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source ...).
2. compilation : traduit le fichier généré par le préprocesseur en **assembleur**, (suite d'instructions du microprocesseur)
3. assemblage : transforme le code assembleur en un fichier binaire : instructions directement compréhensibles par le processeur. Le fichier produit est le **fichier objet**
4. édition de liens : un programme est souvent séparé en plusieurs fichiers source (librairies notamment), il faut donc lier entre eux les différents fichiers objets. Le fichier produit est l'**exécutable**

Compilation avec gcc

- gcc : GNU Compiler Collection
- gcc [options] fichier.c [-llibrairies]
- fichier objet par défaut : a.out (on peut le modifier avec l'option -o)

Sources

- Cours de Pablo Rauzy (lien)
- Cours de Jean-Pascal Palus (lien)
- Cours d'Anne Canteaut