

# Cours d'Algorithmique-Programmation 2<sup>e</sup> partie (IAP2): programmation impérative et structures de données simples

Introduction au langage C

Sandrine Blazy

**ensiie** - 1<sup>ère</sup> année

24 octobre 2007

**école nationale supérieure d'informatique  
pour l'industrie et l'entreprise**

**ensiie**

Apprentissage de la programmation C, sans oublier les acquis d'IAP1 :

- conception descendante,
- spécifier à l'aide
  - d'interfaces
  - éventuellement de code OCaml
- cas de test
- découpage d'un programme en fonctions paramétrées

Écriture de programmes plus conséquents

Implémenter de différentes façons des structures de données simples

- présentation du langage C,
- comparaison entre OCaml et C,
- structures de données simples (listes, piles, files)
- manipulation d'outils :
  - débogueur,
  - graphes d'appels,
  - documentation automatique
  - profileur
  - ...

- *vieux* langage, *simple* et *efficace* → 1972, Bell Labs (AT&T), B. Kernighan et D. Ritchie, écriture d'Unix
- langage de *bas niveau*, qui produit des programmes efficaces
- langage de *haut niveau*
- langage *compilé*
- langage très *souple* : peu de vérifications et d'interdits, hormis la syntaxe → la tentation est grande d'utiliser cette caractéristique pour écrire le plus souvent des atrocités
- langage *portable*
- langage *modulaire*
- langage *normalisé* en 1989 (norme ANSI), 1990 (norme ISO) et 1999 (1<sup>ère</sup> révision)

- langage impératif (affectation, séquence, itération) typé
- récursivité
- variables locales, variables globales
- toute variable doit être déclarée
- définir avant d'utiliser
- pas d'imbrications de fonctions
- différents modes de passages des paramètres
- tableaux, pointeurs
- langage compilé

# Structure d'un programme C

Directives du preprocesseur

```
/* pour l'instant: - inclusion de bibliotheques  
                  - definitions de constantes */
```

Declarations de types et de variables globales

Declarations de fonctions

```
int main(void)  
{  
    Declarations de variables locales  
    Instructions  
    return 0;  
}
```

# Exemple de programme

```
#include <stdio.h> /* directives au preprocesseur */
#define DEBUT -10
#define FIN 10
#define MSG "Programme de demonstration\n";

/* Interface: carre: int -> int */
int carre(int x) { /* definition de la fonction carre */
    return x * x;
}

/* Interface: cube: int -> int */
int cube(int x) { /* definition de la fonction cube */
    return x * carre(x);
}

int main()          /* programme principal */
{
    /* debut du bloc de la fonction main */
    int i;          /* definition des variables locales */

    printf(MSG);
    for ( i = DEBUT; i <= FIN ; i++ )
    {
        printf("%d  carre: %d  cube: %d\n", i
                , carre(i)
                , cube(i) );
    }
    /* fin du bloc for */
    return 0;
}
/* fin du bloc de la fonction main */
```



- ne jamais placer plusieurs instructions sur une même ligne,
- utiliser des identificateurs significatifs,
- faire ressortir la structure syntaxique du programme,
- laisser une ligne blanche entre la dernière ligne des déclarations et la première ligne des instructions,
- une accolade fermante est seule sur une ligne et fait référence, par sa position horizontale, au début du bloc qu'elle ferme.
- aérer les lignes de programme en entourant par exemple les opérateurs avec des espaces.
- commenter les listings (éviter les commentaires triviaux)

- Définir une variable :
  - définir son type
  - réserver une place en mémoire
  - initialiser la variable
- Syntaxe :
  - `nom_du_type nom_de_la_variable ;`
  - `nom_du_type nom_de_la_variable = valeur ;`
- Exemple :

```
int ma_var = 2;
```

- *surcharge* d'opérateurs :  $2 + 3.5$  est une expression de type *float*
- ☞ règles de conversion implicites
- *hiérarchie* des types : selon l'usage fait d'une variable, une valeur de type caractère peut être considérée comme étant un entier
- ☞ pas d'inférence de types
- un type particulier : *void* désigne le type vide
  - ☞ utile pour les fonctions sans argument ou sans retour
  - ☞ rôle particulier dans l'utilisation des pointeurs
- types élémentaires déjà rencontrés : *int*, *float*, *char*
- pas de type *string* prédéfini (cf. suite du cours)
- l'espace occupé par chaque type dépend de la machine sur laquelle est implémenté le compilateur

- pas de type *bool* prédéfini
- ☞ utilisation du type `int` avec 1 codant *true* et 0 codant *false*
- ☞ une expression est fausse si elle s'évalue à 0, et vraie sinon
- opérateurs sur les booléens : `||`, `&&`, `<`, `>`, mais aussi `==`, `!=`, `!`
- évaluation paresseuse pour `&&` et `||`
  - ☞ attention aux effets de bord
- autres opérateurs sur les entiers : `+`, `-`, `*`, `/`, `%`

- qualificateurs
  - short (en général, 2 octets : compris entre  $-2^{15}$  et  $2^{15} - 1$ ) et long (en général, 4 octets : compris entre  $-2^{31}$  et  $2^{31} - 1$ )
  - ☞ tailles des types entiers définies dans `<limits.h>`
    - signed (par défaut) et unsigned
    - ex. short, unsigned short, unsigned, long, unsigned long
    - `sizeof(short) ≤ sizeof(int) ≤ sizeof(long)`
- Quelques exemples de constantes de type entier
  - 0177 nombre octal valant 127 en base 10
  - 0xff nombre hexadécimal valant 255 en base 10
  - 18
  - 1L (1l), 0x1faL entiers longs
  - 8U (8u) entier non signé
  - 1UL entier long non signé

Type `float` (flottants simple précision : en général, 4 octets)

- Pour obtenir davantage de précision :
  - `double` (double précision, en général, 8 octets)
  - `long double` (précision étendue, plus de 8 octets)
  - ☞ tailles des types réels définies dans `<float.h>`
    - `sizeof(float) ≤ sizeof(double) ≤ sizeof(longdouble)`
- Opérateurs `+`, `-`, `*`, `/`
- Notations : `12.3`, `12.3f`, `12.3d`, `12e-3`, `+12E-3`

## Type char

- notations des constantes : 'a', '\0', '\x0', '\n'
- type implémenté par un entier (ordre ASCII) : 'd'-'a' vaut 3
- mêmes opérations que sur les entiers
- Quelques fonctions (`#include <ctype.h>`) : `tolower`, `toupper`, `isalpha`, `isdigit`, `isxdigit`, `islower`, `isupper`


```
#include <stdlib.h>
```

- `abs: int -> int -> int`
- `div: int -> int -> int`
- `ceil: double -> double -> double`
- `floor: double -> double -> double`

```
#include <math.h> + option -lm
```

- `pow: double -> double -> double`
- `exp: double -> double`
- `log: double -> double`
- `sin: double -> double`
- `sqrt: double -> double`

- Les types entier, caractère, réel sont compatibles entre eux.
- Dans une expression, des valeurs de ces types peuvent être mélangées.

 *Conversions implicites :*

- conversion dans le type le plus fort
  - promotion entière
  - conversion dans une affectation
- Peut produire de l'indétermination ou une perte de précision.

# Type enregistrement

Règle de formation : si  $t_1, t_2, \dots, t_n$  sont des types, ils peuvent être regroupés séquentiellement dans un enregistrement.

- Syntaxe :

```
struct nom_de_structure_facultatif {  
    liste des données contenues dans la structure }  
};
```

- Exemples :

```
struct point {  
    int x;  
    int y;  
    long couleur;  
};  
  
struct Tpoint {  
    int x;  
    int y;  
    long couleur;  
} vpoint;
```

- accès aux champs : `vpoint.couleur`
- initialisation : `vpoint = {1,2,512};`

- Syntaxe :

```
/* Interface
   nom_fonction : type1 → ... typen → type_de_retour */
type_de_retour nom_fonction (type1 arg1, ... typen argn)
{
  déclarations des variables locales à la fonction

  instructions
}
```

- **Procédure** : fonction dont le type de retour est void

- Exemple de fonction

```
/* Interface
   carre: int → int */
int carre(int x)
{
    return x * x;
}
```

puis dans une autre fonction `1+carre(2)*carre(x)`

- Exemple de procédure

```
void affiche_bonjour(void)
{
    printf("Bonjour");
}
```

- Une variable désigne un emplacement de la mémoire
- ☞ Une variable prend différentes valeurs lors de l'exécution d'une fonction
- ☞ Utilisation d'un *environnement* mis à jour en chaque *point de programme*
- Distinction entre *lvalues* et *constantes*
  - *lvalue* : expression qui doit délivrer une variable
  - *constante* : identificateur possédant une valeur qui n'est pas modifiée
- Affectation et instruction de lecture

# Un premier opérateur d'affectation

- *lvalue* désigne un objet en mémoire
- Syntaxe :

```
lvalue = exp
```

- Sémantique :
  - *exp* est évaluée, sa valeur est affectée à la variable désignée par *lvalue*
  - ☞ effet de bord
  - cette valeur est celle de l'expression *lvalue = exp*
- Conversion éventuelle de type (vers celui de *lvalue*)
- Exemples

```
x=1.5  
x=1  
x=y+6  
x=y-z=3
```

# D'autres opérateurs d'affectation

- $x += y$  équivalent à  $x = x+y$
- $--$   $*=$   $/=$   $\%=$
- Opérateurs d'incrémentatation  $++$  et de décrémentatation  $--$ 
  - s'appliquent à une *lvalue* :  $++(x+2)$  n'a pas de sens
  - $x++$  et  $++x$  équivalents à  $x = x+1$
  - $x=y++$  équivalent à  $x = y$  puis  $y = y+1$
  - $x=++y$  équivalent à  $y = y+1$  puis  $x = y$
  - attention aux effets de bord

```
int i=0;
int j=0;
i++ && j++;    /* i=1 et j=0 */
j++ && i++;    /* i=0 et j=1 */
```

- Syntaxe

```
#define ma_constante reste_de_la_ligne
```

- Sémantique : le préprocesseur remplace les occurrences de `ma_constante` par `reste_de_la_ligne`
- Convention : écrire les constantes en majuscules
- Exemples :

```
#define MAX 10  
#define VRAI 0  
#define FAUX 1
```

- Instructions-expressions
- Blocs
- Conditionnelles
- Boucles
- Exécution en séquence : rôle du délimiteur ;

- Syntaxe :

```
exp;
```

- Sémantique : `exp` est évaluée et sa valeur est ignorée
- N'a d'intérêt que si `exp` ; réalise un effet de bord
- Exemples :

```
x + 1;  
y = x + 1;
```

- Syntaxe :

*{liste d'instructions}*

- Sémantique : exécution de chaque instruction de la liste
- Permet de considérer une séquence d'instructions comme étant une seule instruction

# L'instruction if

- Syntaxe :

```
if (exp) instruction1 else instruction2
```

- Sémantique : évaluation de `exp` puis exécution de `instruction1` si sa valeur n'est pas nulle ou de `instruction2` si sa valeur est nulle
- Partie *sinon* optionnelle
- Exemples

```
if (x==y) y=2;  
if (x==y) y=2; else y=6;
```

- Expressions conditionnelles

```
exp1 ? exp2 : exp3
```

- Syntaxe :

```
switch (exp) {  
    case valeur1: instruction1; break; /* ou rien */  
    case valeur2: instruction2; break; /* ou rien */  
    ...  
    case valeurn: instructionn; break;  
    default: instruction0; break;  
}
```

- Sémantique : évaluation comme une valeur entière de `exp` puis évaluation des valeurs des `case` puis exécution de l'instruction associée au `case` dont la valeur correspond à l'expression (ou de l'instruction associée à `default` si aucune des valeurs de `case` ne correspond à l'expression)

- Exemple

```
switch (lettre) {  
    case 'a':  
    case 'A': i=j; break;  
    case 'u': i++; break;  
    case 'z': j=i; break;  
    default: i=j=0; break;  
}
```

## Répétition d'instructions

- tant qu'une condition est vérifiée
- jusqu'à ce qu'une condition soit vérifiée
- un certain nombre de fois

## Boucles infinies

## Équivalences entre les trois formes

## Imbrication des boucles

- Syntaxe :

```
while (exp) instruction
```

- Sémantique : `exp` est évaluée, si sa valeur s'interprète comme vraie (non nulle), `instruction` est exécutée, et ceci tant que `exp` vaut vrai.
- Exemple

```
int x = 0;  
/* affichage des entiers de 0 a 9 */  
while (x != 10) {  
    printf("%d", x);  
    x++;  
}
```

# Les boucles *répéter*

- Syntaxe : `do instruction while (exp)`
- Sémantique : `instruction` est exécutée, puis `exp` est évaluée. Si sa valeur s'interprète comme vraie (non nulle), `instruction` est exécutée à nouveau et ainsi de suite.
- Exemple

```
int x = 1;
int somme = 0
/* somme des entiers de 1 a 10 */
do {
    somme += x;
    x++;
}
while (x <= 10)
```

- Syntaxe : `for (exp1 ; exp2 ; exp3) instruction`
- Sémantique : `exp1` est évaluée (initialisation des variables de la boucle). Puis `exp2` est évaluée et si sa valeur s'interprète comme vraie (non nulle), `instruction` est exécutée, et ceci tant que la valeur de `exp2` s'interprète comme vraie. De plus, `exp3` est évaluée après chaque exécution d'`instruction`.
- Exemples

```
for( ; ; ) /* exp2 qui n'existe pas est supposée vraie */  
; /* boucle infinie ne faisant rien */
```

```
for(i=0; i<n; i++)
```

```
for(i=0, j=n; i<j; i++, j--)
```

# Comparaison entre les trois sortes de boucles

- `for (exp1 ; exp2 ; exp3) instruction` équivaut à

```
{  
exp1;  
while (exp2) {  
    instruction;  
    exp3;  
}
```

- `do instruction while (exp)` équivaut à

```
instruction  
while(exp) {  
    instruction  
}
```

- fait partie de la bibliothèque standard : `#include <stdio.h>`
- affichage de valeurs en les transformant en chaînes de caractères
- *format* décrivant une telle transformation = une chaîne de caractères dans laquelle des caractères représentent les variables à écrire : "%d plus %d vaut %d"
- Syntaxe : `printf(format, exp1, .... expn) ;`
- Sémantique : les `expi` sont évaluées et les valeurs correspondantes sont affichées sur la sortie standard (à l'écran) selon `format`. L'appel à `printf` renvoie le nombre de caractères imprimés.

- commence par % et se termine par un caractère de conversion
- entre les deux on peut écrire dans l'ordre suivant :
  - - cadrage à gauche
  - un nombre minimal de caractères écrits
  - un point
  - un nombre indiquant la précision de la valeur affichée
  - h (resp. l) pour short (resp. long)
- Exemples : ‘‘%d’’ ‘‘%f’’ ‘‘%c’’ ‘‘%s’’ ‘‘%8.4f’’

# “Compilation” d'un programme C

- préprocesseur (*cpp*) :
  - supprime les commentaires
  - prend en compte les lignes commençant par #
- compilateur C : lecture unique du fichier source
- optimiseur de code (*c2*) :
  - élimination du code inutile
  - traduction en instructions propres au processeur
  - optimisation des sauts
- assembleur (*as*)
- éditeur de liens (*ld*) : résolution des références insatisfaites, utilisation de la bibliothèque standard (*libc.a*)

- Compilation

```
gcc -ansi -pedantic -W -Wall -o mon_executable  
mon_prog.c
```

- ☞ `-ansi` : vérification C ANSI sans extensions de langage correspondant au C GNU

- ☞ `-pedantic` : refus de compiler un programme non ANSI

- ☞ `-W -Wall` : maximum de messages (erreurs ou avertissements)

- Autres options

- ☞ `-g` : permet d'utiliser le débogueur

- ☞ `-l nom_bibliotheque` : spécification de bibliothèques (ex. `-lm`)

- Exécution : `mon_executable` (ou `./mon_executable`)

Mode Emacs pour C :

- contrôles de validité du parenthésage
- indentation automatique (ou à la demande)
- commande de compilation
  - ☞ compilation du fichier source le plus récent