Programmation impérative en C

Cours 1. Présentation du langage

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 2 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Sommaire

Organisation de l'UE (partie C)

- 5 cours
- 6 séances de TP
- 2 notes :
  - Une note de contrôle continu (= 50%)
    - Un QCM en amphi (≈ 25%)
    - Un projet (≈ 25%)
    - Interros surprises en amphi. (≈ ?%)
  - Un contrôle terminal (= 50%)

- **Programmation impérative en Python**

  Je pars du principe que :

  - vous savez déjà programmer ;
  - les `if`, `for`, `while`, `return` n'ont plus de secret pour vous.

- **Système 1**

  Je pars du principe que :

  - la ligne de commande vous est familière.
  - les `ls`, `cd`, `mv`, `cp` n'ont plus de secret pour vous.

- ***The C Programming Language***

  B. W. Kernighan, D. M. Ritchie, *Prentice Hall*, 1978

- **♥ Le langage C — C ANSI ♥**

  B. W. Kernighan , D. M. Ritchie, *Masson - Prentice Hall*, 1994, 2e édition
  Traduit par J.-F. Groff et E. Mottier

- **Langage C — Manuel de référence**

  Harbison S.P., Steele Jr. G.L., *Masson*, 1990
  Traduit en français par J.C. Franchitti

Sommaire

- La trinité des barbus :
  - Kenneth Thomson
  - Denis Richie
  - Brian Kernighan
- Unix et C partagent la même histoire :
  - 1969 : **Unix** par K. Thomson puis D. Ritchie
  - 1969 : **Langage B** par K. Thomson
  - 1972 : **Langage C** par D. Ritchie
  - 1978 : sortie du **K&R** (Kernighan et Ritchie), la bible du langage C.
  - 1983 : D. Ritchie et K. Thomson recoivent le prix Turing pour Unix.
  - 1983 : *Reflection on Trusting Trust* (discours de Thomson)
  - 1987 : **GCC** : Richard Stallman, « père » du logiciel libre (un barbu)
- Évolution du langage :
  - 1978 : « norme » K&R (caduque)
  - 1990 : norme **C ANSI** (utilisée durant ce cours)
  - 1999, 2011, 2023 : normes C99, C11, C23

Ritchie (debout) et Thomson (assis)

- PDP-11 : ordinateur sur lequel à été développé Unix et C.
- Où est l'écran ?
  - Il n'y en a pas, seulement une imprimante.
  - d'où le mot `print` (imprimer) pour afficher du texte

Points forts

- Langage conceptuellement simple :
  - peu de concepts ;
  - il est assez facile d'avoir une bonne vision globale du C.
- Très bas niveau :
  - gestion manuelle de la mémoire ;
  - le développeur est libre et fait ce qu'il souhaite.
- *Lingua Franca* des langages de programmation :
  - très nombreuses bibliothèques logicielles ;
  - tous les langages ont une interface avec C pour pouvoir les utiliser.
- Le C est un des langages les plus rapides :
  - Fortran est aussi très rapide (calcul scientifique) ;
  - C++ (inspiré du C mais avec de l'objet) ;
  - Rust (nouveau mais très prometteur) ;
  - Java (un peu moins rapide que les précédents, mais reste très rapide).

- Langage conceptuellement simple :
  - peu de concepts ;
  - pas de paradigme objet, fonctionnelle ou autre ;
  - pas d'exceptions ;
  - typages faibles (on peut toujours convertir).
- Très bas niveau :
  - Gestion manuelle de la mémoire (fuite mémoire, accès aléatoires)
  - le développeur est libre et fait ce qu'il souhaite (dont des bétises)
- Portabilité limitée (mais meilleur que celle de l'assembleur) :
  - Difficile d'écrire un programme qui marche pour toutes les plateformes
  - Le compilateur a une grande liberté dans l'interprétation du langage.
- Syntaxe pédante et ultra concise

- Aide à comprendre comment fonctionne un ordinateur :
  - comment est gérée la mémoire ;
  - comment sont codées les données ;
  - comment fonctionnent les autres languages.
- Encore beaucoup utilisé.
- Fait partie de la culture générale de tout informaticien.
  - c'est le « Aristote » des langages de programmation
    - brillant en son temps
    - peut-être un peu trop influent et un peu trop longtemps…
  - Histoire (résumée) des langages de programmation :
    - Histoire de comment on a voulu réparer les défauts de C…
    - …tout en étant le plus proche possible de C (mais avec de l'objet).
    - Les langages trop éloignés de C ont longtemps été snobés.

Sommaire

- Une instruction est une opération de base.

*.c
```
x = 3 ;                          /* affectation */
printf("Hello world\n") ;  /* appel d'une procèdure */
t[x] = 7 ;                /* modification d'un tableau*/
```

- les instructions se terminent par le symbole « ; »
  - on peut mettre plusieurs instructions sur une ligne.
  - le retour à la ligne équivaut à un espace.

*.c
```
x =
3 ; printf("Hello world\n") ;  t[x]
= 7 ;
```

- Le but reste quand même d'être lisible...
  - Chaque instruction sur une seule ligne.
  - Pas plus d'une instruction par ligne.

- En C ANSI les commentaires commencent par `/*` et se terminent par `*/`.
- Les commentaires peuvent tenir sur plusieurs lignes.

*.c
```
x = 3 ; /* voici
un long commentaire vachement intéressant
et là c'est la fin */
y=2 ;
```

- En C99 et C++, on peut commenter une fin de ligne avec `//`.

*.c
```
x = 3 ; // voici un petit commentaire
y = 2 ;
```

- Le cours étant sur C ANSI on n'utilisera pas `//`.

- En C, les blocs sont encadrés par des accolades : `{ }`

*.c
```c
if (x == 3) {
  printf("x vaut 3\n");
} else {
  printf("x vaut autre chose\n");
}
```

*.c
```c
if ( test ) {
  instructions
}
```

*.c
```c
if ( test ) {
  instructions
} else {
  instructions
}
```

*.c
```c
if ( test ) {
  instructions
} else if ( test ) {
  instructions
} else if ( test ) {
  instructions
} else { /* Facultatif */
  instructions
}
```

- En C, on retrouve la boucle `while`.

*.c
```
i = 0;
while ( i < 10 ) {
  i = i + 1;
}
printf("Fini\n")
```

- La variante `do ... while` permet de passer au moins une fois dans la boucle.

*.c
```
i = 11;
do {
  i = i + 1 ;      /* i prend la valeur 12 */
} while ( i<10 ); /* On quitte la boucle après un passage */
printf("i vaut: %d\n",i)
```

- En C, la boucle `for` n'est qu'une autre façon de faire un `while`

*.c
```c
expr1 /*Initialisation*/
while (expr2) {/*Condition*/
  …
  expr3 /* Incrémentation */
}
```

*.c
```c
for (expr1; expr2; expr3) {
  …
}
```

- Exemple : Affichons « `Hip Hip Hip Hourra !` »
  - Faisons varier `i` parmi `0, 1, 2`
  - Trois passages dans la boucle.

*.c
```c
i=0;
while (i<3) {
  printf("Hip ");
  i++; /* i=i+1 */
}
printf("Hourra !\n");
```

*.c
```c
for (i=0; i<3; i++) {
  printf("Hip ");
}
printf("Hourra !\n");
```

- On peut utiliser les 3 opérations sur les booléens : `&&` `||` `!`
  - Correspond au ET, au OU et ou NON.
- On peut utiliser les 4 opérations sur les flottants : `+` `*` `-` `/`
- On peut utiliser les 5 opérations sur les entiers : `+` `*` `-` `/` `%`

*.c
```c
int a = 15;
float x = 15.;

a = a/2;  /* vaut 7   de type int   : division entière */
x = x/2;  /* vaut 7.5 de type float : division décimale */
```

- Le C permet de raccourcir certaines affectations :

*.c
```c
x += 3;  /* x = x+3   */
y %= 255; /* y = y%255 */
```

- Les opérateurs ++ et --

*.c
```c
x++ ; /* équivalent à « x = x+1 » */
x-- ; /* équivalent à « x = x-1 » */
```

- En C les affectations sont des expressions.

```c
y = (x=3) + (z=18) /* vaut 3 + 18 */
```
*.c

- En C, x++ est aussi une expression
  - `x++` vaut l'ancienne valeur de `x` avant l'affectation
  - `++x` vaut la nouvelle valeur de `x` après l'affectation

```c
x=3
y = 2 + (x++) /* = 2 + 3  */

x=3
y = 2 + (++x) /* = 2 + 4 */
```
*.c

- Que vaut « `x+=(x=3)+(++x+x++)` » ?
  - Non définie (et heureusement !)
  - N'abusez pas de ces écritures !

Sommaire

- Les types entiers (version simplifiée sans garantie).
  - on peut ajouter `signed` ou `unsigned` avant le type.
  - par défaut les types sont signés sauf `char` (dépend de la machine).
  - `unsigned char` (entre 0 et 255) et `signed char` (entre -128 et 127)
  - N'utilisez `char` (sans précision) que pour de l'ASCII (entre 0 et 127)

*.c
```c
char c;  /* 1 octet */
short s; /* 2 octets : abréviation de « signed short int» */
int i;   /* 4 octets : abréviation de « signed int» */
long l;  /* 8 octets : abréviation de « signed long int» */
```

- Les types flottants (forcément signés).

*.c
```c
float f;         /*  4 octets */
double d;        /*  8 octets */
long double l;   /* 16 octets */
```

- Autres types (que l'on verra plus tard dans le cours) :

*.c
```c
char *s ="Je suis une chaîne de caractère ";
int faux = 0; /* Il n'y a pas de booléen de C */
int vrai = 1; /* ou n'importe quel entier différent de 0 */
```

Remarques sur les char

- En C, pas de différence entre caractère et entier (entre 0 et 127).
  - Les écritures `65` et `'A'` représente le même nombre.
  - C'est la table ASCII qui sert de référence.
- Attention à ne pas confondre les caractères et les chaînes :

*.c

```c
/* char : guillemets simples */
char lettre1 = 'B'
char lettre2 = 66

/* chaînes : guillemets doubles
 *         : déclaration avec « * » avant le nom de variable
 */
char *chaine = "On écoute le prof !"
char *lettre = "A"

if ('A' == "A") {
  printf("Tous le monde à 20/20 dans l'UE de C\n");
} else {
  printf("Désolé, mais ça ne marche pas\n");
}
```

- Les tailles indiquées en octets sont indicatives !
- La norme du C ANSI ne garantie rien !
  - Un `char` est représenté sur un *byte* (dépend des systèmes)
  - Un *byte* étant la plus petit taille de variable
  - On a forcément `char` ≤ `short` ≤ `int` ≤ `long`
  - On a forcément `char` ≤ `float` ≤ `double` ≤ `long double`
- Dans les versions modernes de C (depuis 1999) :
  - il existe aussi le type `long long int`
  - `short` et `int` sont au moins sur 2 octets (16 bits).
  - `long` est au moins 4 octets (32 bits)
  - `long long` est au moins sur 8 octets (64 bits).
- Posix (la norme Unix) impose les `char` sur 8 bits.
- On peut faire des choses plus propre avec la bibliothèque `stdint`
  - On le verra plus tard dans le cours (`uint32_t` ou `int_fast8_t`)
  - En pratique on utilisera le type `int` (jusqu'à des valeurs de 2 milliards)

- Le C travaille toujours selon une arithmétique modulaire.
  - Les `unsigned char` étant codé sur un octet, on travaille modulo 256.

*.c
```c
unsigned char c1 = 200;      /* vaut 200 */
unsigned char c2 = 2 * c1 ; /* vaut 144 */
```

- Le C fait des conversions automatiques selon le contexte.
  - `x` est de type `int`, `y` de type `long`
  - `x+y` sera donc de type `long` (*cast* implicite) et vaudra $2^{32} + 16$
  - mais en sauvegardant dans `b` (*cast* implicite vers `int`), `b` vaudra 16.

*.c
```c
int  x = 17; long y = 4294967295; /* 2^32 - 1 */
long a = x+y; /* 4294967312 */
int  b = x+y; /* 16 */
```

- On peut forcer C à faire des conversions avec un *cast* explicite
  - « `x + (int) y` » sera un `int` de valeur 16
  - `c` sera donc un `long` de valeur 16 (*cast* implicite)

*.c
```c
long c = x + (int) y; /* 16 */
```

- En C (et contrairement à Python), chaque variable possède un type.
  - Pour C on parle de typage statique
  - Pour Python on parle de typage dynamique
  - En Python les données sont typées, pas les variables.

- Avant d'utiliser une variable il faut la déclarer (= lui donner un type).

*.c
```c
int i;              /* Déclaration sans initialisation */
int j;
int a, b, c;        /* Déclarations multiples */
int somme = i + 3; /* Déclaration avec initialisation */
```

- Les arguments d'une fonction sont déclarés dans sa signature.

*.c
```c
int addition(int a, int b) {
  return a+b; /* a+b est bien de type int*/
}
```

- Pour définir une fonction, on doit indiquer :
  - Le type de retour (avant le nom de la fonction)
  - Le type de chaque argument (avant son nom)
  - `type_retour nomfonction(type1 arg1, …)`

*.c
```c
int addition(int a, int b) {
  return a+b; /* a+b est bien de type int*/
}
```

- Chaque nouvelle variable doit être déclarée au début de la fonction.
  - C ANSI : si une déclaration se situe après une instruction ⇒ erreur !
  - C ANSI : `for (int i, i<=n, i++)` est interdit.

*.c
```c
int addition(int a, int b) {
  int s;
  s = a+b ;           /* instruction */
  int resultat = s; /* déclaration : erreur !!! */
  return s;
}
```

- Calculons la somme des entiers de 1 à n compris.

$$\texttt{somme}(\texttt{n}) = \sum_{\texttt{i}=1}^{\texttt{n}} \texttt{i} = 1 + 2 + \ldots + \texttt{n}$$

*.c

```c
int somme(int n) {
  int resultat ;     /* En tout trois variables sont */
  int i;             /* déclarées : n, i et resultat */
  /* Fin des déclarations */

  /* Début des instructions*/
  resultat = 0;
  for (i=1; i<=n; i++) {
     resultat += i; /* resultat = resultat + i */
  }

  return resultat; /* correspond bien au type de sortie*/
}
```

- Le programme C exécute toujours la fonction `main`.
  - Elle doit donc toujours être définie.
  - Le type `void` indique que `main` ne prend pas d'argument.
  - La fonction `main` doit obligatoirement retourner un entier
    - `0` si tout c'est bien passé.
    - Une autre valeur sinon (cf. cours d'Unix)

*.c
```c
#include <stdio.h> /* Nécessaire pour utiliser printf */

int main(void) {
    printf("Hello world !\n");
    return 0;
}
```

- Pourquoi `Hello World` ? D'après Brian Kernighan et selon wikipédia :
  - « *What I do remember is that I had seen a cartoon that showed an egg and a chick and the chick was saying, "Hello, world".* »
  - « Ce dont je me rappelle est d'avoir vue une bande dessinée montrant un œuf et un poussin et le poussin disait "Bonjour, monde" ».

- Mettons notre fonction `somme` dans un fichier complet : « `exemple.c` »

*.c

```c
#include <stdio.h> /* Nécessaire pour utiliser printf */

int somme(int n) {  /*somme(n) = 1+2+3+…+n */
  int resultat;
  int i;          /* déclaration de resultat et i */

  resultat = 0;
  for (i=1; i<=n; i++) {
     resultat += i; /* resultat = resultat + i */
  }
  return resultat;
}

int main(void) {
  int s;
  s = somme(10);
  printf("1+2+…+9+10 = %d\n", s);
  return 0;
}
```

Sommaire

- Pour compiler :
  - On se place dans le répertoire contenant notre fichier `*.c`
  - On utilise le programme `gcc` (GNU C Compiler)
  - Le résultat est un exécutable (cf cours Unix) appelé `a.out`
  - On peut l'exécuter en précédant son nom par `./`

SHELL
```
olivier@valrose:~/dossier $ ls
exemple.c
olivier@valrose:~/dossier $ gcc exemple.c
olivier@valrose:~/dossier $ ls
a.out exemple.c
olivier@valrose:~/dossier $ ./a.out
1+2+…+9+10 = 55
olivier@valrose:~/dossier $
```

- Pour donner un (vrai) nom à notre programme : option `-o nom`

SHELL
```
olivier@valrose:~/dossier $ gcc exemple.c -o exemple
```

- Voici un code (`toto.c`) fautif...

  ...compilant sans problèmes !
- Ajoutons des avertissements :
  - `-Wall` : tous les *warnings* !
  - `-pedantic` : soyons pédant
  - `-ansi` : norme C ansi

*.c

```c
#include <stdio.h>

int main(void) {
  int x;
  /* x=3; */
  printf("x=%d\n",x);
  /* return 0; */
}
```

SHELL

```
olivier@valrose:~ $ gcc toto.c -o toto
olivier@valrose:~ $ ./toto
x=0
olivier@valrose:~ $ gcc -Wall -pedantic -ansi toto.c -o toto
toto.c: In function ‘main’:
toto.c:8:1: warning: control reaches end of non-void function
[-Wreturn-type]
    8 | }
      | ^
toto.c:6:5: warning: ‘x’ is used uninitialized in this function
[-Wuninitialized]
    6 | printf("x=%d\n",x);
      | ^~~~~~~~~~~~~~~~~
```

- Il peut être pénible de retaper toujours la même commande.
  - On cherche à éviter de se répéter : c'est la raison d'être de l'informatique !
  - il existe un outil merveilleux pour automatiser des tâches...
  - ... c'est le script shell (c'est même sa raison d'être) !

compiler.sh

```
#!/bin/bash
echo "Compilation de l'exercice 1"
gcc -Wall -pedantic -ansi exercice1.c -o exercice1
echo "Compilation de l'exercice 2"
gcc -Wall -pedantic -ansi exercice2.c -o exercice2
```

SHELL

```
olivier@valrose:~ $ ls
exercice1.c    exercice2.c    compiler.sh
olivier@valrose:~ $ chmod u+x compiler.sh
olivier@valrose:~ $ ls
exercice1.c    exercice2.c    compiler.sh
olivier@valrose:~ $ ./compiler.sh
Compilation de l'exercice 1
Compilation de l'exercice 2
olivier@valrose:~ $ ls
exercice1  exercice1.c  exercice2  exercice2.c  compiler.sh
```

- À chaque fois, on recompile tout : peu efficace et lent.
  - Il existe un outil plus adapté : Makefile (après les vacances)

Sommaire

- Pour afficher des chaînes on utilise la commande `printf` avec :
  - `%d` pour les entiers (écriture décimale)
  - `%f` pour les flottants
  - `%s` pour les chaînes de caractères.

*.c
```c
int jour = 27;
char *mois = "octobre";

printf("Bonjour monde");
printf("Nous sommes le %d du mois %s\n",jour, mois);
printf("Deux tiers valent : %f\n", 2./3.);
printf("Deux tiers valent : %.2f\n", 2./3.);
```

- Ce qui donne :

stdout
```
Bonjour mondeNous sommes le 27 du mois octobre
Deux tiers valent : 0.666667
Deux tiers valent : 0.67
```

- Attention de ne pas oublier le `'\n'` !

`%[indicateur][largeur][.précision][taille]type`

- Les types de bases : ♥
  - `%d` : entier (décimal et signé)
  - `%f` : flottant
  - `%c` : caractère
  - `%s` : chaîne de caractères
  - `%p` : pointeur

- Entiers (*unsigned*) :
  - `%u` décimale (base 10)
  - `%o` octal (base 8)
  - `%x` héxadécimal (base 16)

- Flottant :
  - `%f` par défaut
  - `%e` écriture scientifique
  - `%g` le plus court entre les deux

`%[indicateur][largeur][.précision][taille]type`

- Pour les entiers : (`h` = *short* et `l` = *long*)

	`short`	`int`	`long`
signé	`%hd`	`%d`	`%ld`
non signé	`%hu`	`%u`	`%lu`

- Pour afficher les flottants : (`L` = *Long*)

	`float`	`double`	`long double`
Décimale	`%f`	`%f`	`%Lf`
Scientifique	`%e`	`%e`	`%Le`

`%[indicateur][largeur][.précision][taille]type`

- Pour un flottant : nombre de chiffres après la virgule (arrondi)
- Pour une chaîne largeur max de la chaîne.

*.c

```c
float euro = 6.55957;
char *msg = "ABCDEFGH";

printf("%f\n", euro);    /* 6.559570 */
printf("%.2f\n", euro); /* 6.56 */

printf("Message = %s\n",msg);    /* Message = ABCDEFGH*/
printf("Message = %.2s\n",msg); /* Message = AB*/
```

`%[indicateur][largeur][.précision][taille]type`

- Indicateur :
  - `+` : affiche le signe (symbole « `+` » ou « `-` »)
- Largeur : permet d'aligner le texte en ajoutant des symboles.
  - si `indicateur` vaut « `-` » alignement à gauche avec des espaces
  - si `indicateur` vaut « ␣ » alignement à droite avec des espaces
  - si `indicateur` vaut « `0` » alignement à droite avec des zéros
  - si `indicateur` vaut « `+` » alignement à droite avec des espaces (après avoir ajouter le signe)

*.c

```c
printf("DÉB:%+d:FIN\n", 23);     /* DÉB:+23:FIN */
printf("DÉB:%+d:FIN\n", -23);    /* DÉB:-23:FIN */

printf("DÉB:%-10d:FIN\n", 23);   /* DÉB:23        :FIN */
printf("DÉB:%+10d:FIN\n", 23);   /* DÉB:       +23:FIN */
printf("DÉB:% 10d:FIN\n", 23);   /* DÉB:        23:FIN */
printf("DÉB:%010d:FIN\n", 23);   /* DÉB:0000000023:FIN */
```

`%[indicateur][largeur][.précision][taille]type`

- Pour la largeur et la précision on peut mettre un symbole « * »
- Dans ce cas, il faut préciser ces valeurs en argument du `printf`

*.c

```c
int i = 12;
float f = 6.66957;

printf(":%+*d:\n",10,i);          /* :       +12: */
printf(":%+10d:\n",i);            /* :       +12: */

printf(":%*.*f:\n",10,2,f);       /* :      6.56: */
printf(":%10.*f:\n",2,f);         /* :      6.56: */
printf(":%10.2f:\n",f);           /* :      6.56: */
```

- La fonction getchar permet de lire un caractère sur l'entrée standard
- On lit jusqu'à tomber sur '\n' ou EOF

*.c

```c
#include <stdio.h>

int lire_valeur(void) {
    int taille = 0;
    char c;
    printf("longueur> ");
    while ( (c=getchar()) != '\n' && c!= EOF ) { /* L1 */
        printf("%c : %d\n", c, taille);
        taille ++;
    }
    printf("Vous avez entré %d caractères\n", taille);
    return 0;
}

int main(void) {
    for (;;) lire_valeur(); /* L2 */
    return 0;
}
```

- Prenons un moment pour comprendre les lignes L1 et L2.

Sommaire

- Pour cette UE, nous utiliserons `gcc`, `make` et un terminal.
- Sous GNU/Linux et OS X :
  - Rien à faire tout est disponible et fonctionne par défaut, normalement…
- Sous Windows :
  - stratégie 1 : installer Linux :)
  - stratégie 2 : installer WSL (sous système Linux pour Windows)
  - stratégie 3 : installer Linux sur une machine virtuelle (cf Système 1)
  - stratégie 4 : Allez-voir sous Moodle.
- Ne connaissant pas Windows, je ne peux pas trop vous aider.
  - Mais vous pouvez vous entraider :
    - forum Moodle
    - Discord

Il est fondamental que vous installiez C sur votre machine pendant ces vacances !

- Je ne suis pas un grand fan des IDE (durant l'apprentissage)
  - une IDE vous simplifie le travail en faisant le travail à votre place.
  - pédagogiquement, c'est sous-optimale.
- Le mieux est de mettre les mains dans le cambouis.
  - on compile à la main en ligne de commande
  - on utilise des éditeurs de texte simples.
- Quels éditeurs ?
  - ♥ GNU/Emacs ♥ : libre, sophistiqué et puissant (pour les *happy few*)
  - Autres éditeurs de texte libres sous GNU/Linux
    - vim : un autre éditeur historique surpuissant (je préfère Emacs)
    - gedit : l'éditeur de texte par défaut au PV. Un peu simple mais suffisant.
  - çapuecestpaslibre
    - VSCode (Microsoft) : votre préféré (bien intégré avec WSL)

Brian Kernighan

« Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as you can, you are, by definition, not smart enough to debug it. »

Déboguer est deux fois plus compliqué qu'écrire le code dans un premier temps. Ainsi, si vous avez écrit le code en y mettant toute votre ingéniosité, vous n'êtes par définition pas assez intelligent pour le déboguer.

Merci pour votre attention
Questions

?

Cours 1 — Présentation du langage