Module M1207
Bases de la programmation (Langage C)

IUT de Béziers, dépt. R&T © 2014 -2021
http://www.borelly.net/
christophe.borelly@umontpellier.fr

Contenus du module

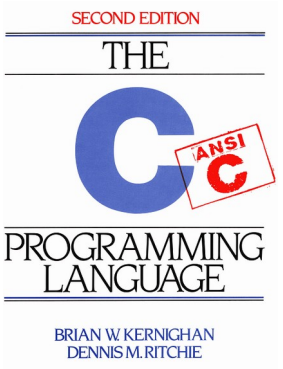
- Proposer une solution logicielle conforme à un cahier des charges simple
  - Concevoir un algorithme

*« Un **algorithme** est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème. »*

  - Transcrire un algorithme dans un langage structuré
  - Compiler, corriger et tester un programme

Histoire du langage C (1)

- Le langage C a été créé entre 1969 et 1973, dans les laboratoires BELL, pour développer le système d'exploitation UNIX.



- Ken THOMPSON et Dennis M. RITCHIE.
- En 1978, Brian W. KERNIGHAN et Dennis M. RITCHIE écrivent : « The C Programming Language »
- http://fr.wikipedia.org/wiki/C_%28langage%29

Histoire du langage C (2)

- Début de la normalisation du langage par l'IEEE en 1987
- 1ère norme en 1989 : ANSI X3-159 (**C89**)
  - American National Standards Institute
- **C90** : ISO/CEI 9899:1990
  - International Organization for Standardization
- **C99** : ISO/CEI 9899:1999
- **C11** : ISO/CEI 9899:2011

Caractéristiques générales

- Langage **compilé**
  - Les instructions du langage C (haut niveau) sont transformées en langage machine (bas niveau) en fonction du type de micro-processeur
- La compilation de déroule en 4 étapes :
  - **Pré-processeur** (remplacements de textes), **compilation** (langage assembleur), **assemblage** (fichier binaire objet), et **édition de liens** (librairies)
- GNU C Compiler : **gcc**

Les extensions de fichiers

- Un programme C s'écrit dans un fichier texte avec l'extension **.c** (fichier source)
- Les fichiers d’en-tête **.h** (header) peuvent être utilisés pour les déclarations

- Les fichiers **.i** ne subissent pas l'étape 1 du processus de compilation (Pré-processeur)
- Les fichiers **.s** contiennent de l'assembleur

Le compilateur GCC

- Outil **HYPER** puissant... (Voir : `man gcc`)
  - **-o** indique le nom du fichier généré (par défaut `a.out` sous LINUX et `a.exe` sous WINDOWS)
  - **-Wall** Affiche tous les erreurs (« warnings »)
  - **-std=c99** (utilise la norme C99)
  - **-O, -O1, -O2, ...** permet d'optimiser le code
  - **-I dir** (i majuscule) indique un répertoire de recherche des fichiers « include »
  - **-lxxx** (l minuscule) ajoute la librairie libxxx.a lors de l'édition de liens
  - ...

Les mots clés (1)

- Les 34 principaux mots clés du langage C :

auto break case char const continue

default do double else enum extern float for

goto if inline int long register restrict return

short signed sizeof static struct switch typedef

union unsigned void volatile while

Premier exemple

```c
#include <stdio.h>
int main()
{
   printf("Bonjour !\n");
   return 0;
}
```

Compilation et exécution

```
cb@pccb$ ls
bonjour.c
cb@pccb$ gcc -Wall bonjour.c -o bonjour
cb@pccb$ ls
bonjour.c bonjour
cb@pccb$ ./bonjour
Bonjour !
```

Les commentaires

- Sur une ligne :

```
// Voici un commentaire sur une seule ligne
```

- Un bloc de commentaires :

```
/* Ceci est un commentaire sur
plusieurs lignes */
```

Les types de donnée

- Les caractères : **char** (8 bits)
- Les entiers : short (16 bits), **int** (32 bits), long (64 bits)
- Les réels : float (4 octets), **double** (8 octets), long double (12 octets)
- L'opérateur **sizeof** permet d'obtenir la taille en octets : sizeof(int), sizeof(x), ...

Types signés/non signés

- Les types entiers sont signés par défaut, mais on peut préciser que les valeurs sont strictement positives à l'aide du mot clé **unsigned**
- char [-128;+127] (entre -2^7 et 2^7-1)
- unsigned char [0;255] (entre 0 et 2^8-1)
- short [-32768;+32767] (entre -2^{15} et $2^{15}-1$)
- unsigned short [0;65535] (entre 0 et $2^{16}-1$)

Syntaxe du C (1)

- Les **identifiants** permettent de donner un nom à un élément du programme (une variable, une fonction, un type de donnée)
  - Ils commencent forcément par une lettre ou _
- Une **expression** permet de représenter une opération simple ou complexe :
  - printf("Hello") ou x=2 ou a>2 && b<=17
- Une **instruction** est une expression ou une suite d'expressions qui se termine par le caractère **;**
- Une suite d'expressions peut s'écrire en utilisant le séparateur **,**
  - a=2 **,** b=a+1;

Syntaxe du C (2)

- On peut définir des **blocs** d'instructions avec les caractères **{** et **}**

```
{
   x=2;
   printf("Hello");
}
```

- Une **déclaration** permet d'indiquer le type de donnée d'une variable (e.g. au début d'un bloc)

```
int x;          // Un entier x
double y,z; // 2 réels y et z
```

Syntaxe du C (3)

- L'**initialisation** permet de fixer la valeur d'une variable :

```
x=2;
y=3.15;
```

- On peut faire la déclaration et l'initialisation en même temps :

```
int x=5;
double y=2.5,z=6e-2;
```

Valeurs entières

- Les nombres entiers peuvent s'écrire de plusieurs façons (décimal, octal ou hexadécimal) :

```
int x=22;   // Décimal (2*10+2)
int y=026;  // Octal (2*8+6)
int z=0x16; // Hexadécimal (1*16+6)
```

- On peut indiquer la taille (l ou L) et le format (u ou U) :

```
int a=0xF1234567L; // -249346713
unsigned int b=0xF1234567UL; // 4045620583
```

Valeurs réelles

- Les nombres réels sont écrits avec la notation mantisse/exposant (par défaut au format double)
  - double reel=3.15e-3; // 3,15 10^{-3}
- On peut spécifier le format float avec la lettre f ou F
- Le format l ou L permet de définir une valeur de type long double (12 octets)

Valeurs caractères (1)

- La valeur d'un caractère (char) se précise entres apostrophes '
  - Exemple : `char c='a';`
- **Exception** : Les caractères **** et **'** s'écrivent en les préfixant du caractère **** soit '\\' et '\''
- On peut aussi écrire un caractère à l'aide de la valeur de son code ASCII :
  - `char a=65,b=0101,c=0x41;`
  - `char d='A',e='\101',f='\x41';`

Constantes caractères (2)

- Certains caractères non imprimables ont une représentation simplifiée :
  - '\n' : Nouvelle ligne (ASCII LF : 10 ou 0x0A)
  - '\t' : Tabulation (ASCII HT : 09 ou 0x09)
  - '\r' : Retour chariot (ASCII CR : 13 ou 0x0D)
  - ...

Conversions de type

- On peut faire des conversions de type de données explicite (casting) quand le type destination est plus petit que celui de départ en précisant le type entre parenthèses :

```
int x=0x4023;    // 4 octets
char y=(char)x; // 0x23 (1 octet)
double z=x; // Pas de conversion nécessaire
```

Les chaînes de caractères

- Une chaîne de caractères se note entre guillemets "

```
"Bonjour !"
```

- Le caractère " dans une chaîne s'écrit \"

```
"(Valeur=\"18\")"  =>   (Valeur="18")
```

- On peut écrire de longues chaînes dans un fichier source en ajoutant le caractère \ en fin de ligne

```
"Une tres tres longue \
chaine"
```

- La déclaration d'une variable : `char *str="Une chaine";`
- Défini **str** comme l'adresse mémoire du début de la chaîne.
- La **fin de la chaîne** est représentée par le code ASCII **0**.

Fonction printf()

- Affichage formaté de données
- Nombre variable de paramètres
- Le caractère % est particulier et permet d'adapter le format des données à afficher
- Il peut y avoir plusieurs « % » dans le format
  - Par exemple **%d** signifie décimal

```
printf("Valeur : %d\n",10); // Valeur : 10
int x=10,y=15,z=20;
printf("Valeurs : %d,%d,%d\n",x,y,z);
     // Valeurs : 10,15,20
```

Principaux formats (entiers)

- **%d** : décimal (int)
- **%ld** : long décimal (long)
- **%u** : décimal non signé (unsigned int)
- **%lu** : long décimal non signé (unsigned long)
- **%o** : octal (int)
- **%lo** : long octal (long)
- **%x** : hexadécimal (int)
- **%lx** : long hexadécimal (long)

Principaux formats (réels)

- **%f** : virgule fixe (double)
  - double x=31.53566565*100; // 3153.566565
- **%lf** : virgule fixe (long double)
- **%e** : notation exponentielle (double)
  - 3.153567e+03
- **%le** : notation exponentielle (long double)
- **%g** : représentation courte (double)
  - 3153.57
- **%lg** : représentation courte (long double)

Autres formats

- **%c** : caractère (char)
- **%s** : chaîne de caractères (char *)
- **%p** : adresse mémoire (pointeur)
- Taille d'affichage :
  - **%10d** Affichage sur 10 digits (cadrée à droite)
  - **%-10d** Affichage sur 10 digits (cadrée à gauche)
  - **%05x** Affichage sur 5 digits hexa (préfixé avec des 0)
  - **%.5f** Affichage 5 chiffres après la virgule
  - **%3.5f** Affichage de 8 caractères (, incluse) avec 5 chiffres après la virgule

Fonction scanf()

- Pour sauvegarder en mémoire une valeur entrée au clavier, on peut utiliser la fonction **scanf()** :

```
int x;
printf("Entrer un entier: ");
scanf("%d",&x);
printf("X: %d\n",x);

char str[20];
printf("Entrer une chaine: ");
scanf("%s",str);
printf("Str: %s\n",str);
```

```
double a;
printf("Entrer un reel: ");
scanf("%lf",&a);
printf("A: %g\n",a);
```

Les opérateurs (1)

- Affectation : **=**
  - int x**=**2;
- Arithmétiques : **+ - * / %**
  - x=2**+**3; // 5
  - x=2**-**3; // -1
  - x=2*****3; // 6
  - x=2**/**3; // 0 (division entière !)
  - x=8**%**5; // 3 (reste de la division entière ou **modulo**)

Les opérateurs (2)

- Relationnels : **> >= < <= != ==**
  - Renvoi une valeur vrai ou faux (i.e. non zéro ou 0)
  - 2>3 Test de supériorité : FAUX
  - 2<=3 Test d'infériorité ou d'égalité : VRAI
  - 2!=3 Test de différence : VRAI
  - 2==3 Test d'égalité : FAUX
- Logiques : **&& || !**
  - x>3 && x<7 ET logique
  - x>3 || x==2 OU logique
  - ! (x>3) Négation logique (x<=3)

Les opérateurs (3)

- Manipulation de bits : **& | ^ ~ << >>**
  - 133 **&** 5 ET bit à bit : donne 5 (0000 0101)
  - 133 **|** 2 OU bit à bit : donne 135 (1000 0111)
  - 133 **^** 5 OU exclusif bit à bit : donne 128 (1000 0000)
  - **~** 133 Complément à 1 : donne 122
    - 1000 0101 donne 0111 1010
  - 16**>>**2 Décalage à droite : donne 4 (divise par 2^2)
    - 0001 0000 devient 0000 0100
  - 1<<3 Décalage à gauche : donne 8 (multiplie par 2^3)
    - 0000 0001 devient 0000 1000

Les opérateurs (4)

- Incrémentation : **++** (ajoute + 1)
  - Pré-fixée : valeur renvoyée = valeur finale
  - Post-fixée : valeur renvoyée = valeur de départ
  - int x=2,y=x**++**,z=**++**x; // x vaut 4 et y vaut 2 et z vaut 4
- Décrémentation : **--** (idem mais - 1)
- Affectation composée : **+= -= *= /= %= &= ^= |= <<= >>=**
  - x**+=**2; est équivalent à x=x+2;
  - Même principe pour les autres opérateurs

Priorités et associativités des opérateurs

- Les priorités permettent de déduire le résultat d'une expression complexe.
  - Exemple : * est plus prioritaire que +, donc 3+4*5 donne 23
- L'associativité est utilisée dans les cas de priorités identiques
  - A gauche : 3<<2<<1 donne 12<<1 soit 24
  - A droite : a+=b+=2 donne a+=b+2
- Les parenthèses permettent de forcer l'évaluation de l'expression
  - x>3 && (x<7 || x>4)

Tableau de priorités/associativités (1)

Opérateur	Priorité	Associativité
`() [] -> . x++ x--`	16	G
`! ~ ++x --x -x +x *x &x sizeof`	15	D
`(conversion de type)`	14	D
`* / %`	13	G
`+ -`	12	G
`<< >>`	11	G
`< <= > >=`	10	G
`== !=`	9	G

Tableau de priorités/associativités (2)

Opérateur	Priorité	Associativité
&	8	G
^	7	G
|	6	G
&&	5	G
||	4	G
?:	3	D
= += -= *= /= %= >>= <<= &= ^= |=	2	D
,	1	G

Les structures de contrôle

- Structures conditionnelles :
  - if/else
  - switch/case
- Les boucles :
  - while
  - do/while
  - for

Structure if/else

- Si la condition est vraie, on exécute le bloc1 d'instructions sinon le bloc2 d'instructions.

```
int x=5,y;
if (x==2) // x est égal à 2 ?
{ // Bloc 1
  y=2;
}
else // x est différent de 2 !!!
{ // Bloc 2
  y=x+2;
}
```

Structure switch/case

- On exécute les instructions suivant la valeur de la variable testée. L'instruction **break** termine une série d'instructions.

```
int x=5,y;
switch (x)
{
   case 1:y=3;break;
   case 2:case 3:y=x+6;break;
   default:y=x;break;
}
```

Boucles

- Permet d'exécuter plusieurs fois un même bloc d'instructions
- Définir une expression de test de fin de boucle
- L’instruction break peut interrompre la boucle
- L’instruction continue permet de « sauter » la suite des instructions jusqu'au test de fin de boucle.

Boucle while

- Tant que la condition est vraie, on exécute le bloc d'instructions.

```
int x=0; // Initialisation
while (x<5) // Test de fin de boucle
{
  printf("x=%d\n",x);
  x++; // Valeur suivante
}
```

Boucle do/while

- Le test de fin de boucle est évalué après le bloc

```
int x=0; // Initialisation
do
{
  printf("x=%d\n",x);
  x++; // Valeur suivante
}
while (x<5); // Test de fin de boucle
```

La boucle for

- Il y a 3 instructions : initialisation, test de fin de boucle et valeur suivante

```
int x;
for (x=0 ; x<5 ; x++){
  printf("x=%d\n",x);
}
```

- Syntaxe C99 avec définition de x dans le for :

```
for (int x=0 ; x<5 ; x++){
  printf("x=%d\n",x);
}
```

Les fonctions (1)

- Les fonctions permettent de réaliser un traitement particulier (réutilisable) dans un bloc d'instructions spécifique
- Un prototype permet de déclarer une fonction (e.g. dans les fichiers .h) en indiquant obligatoirement :
  - Le type de retour (void s'il n'y a pas de valeur de retour)
  - Le nom de la fonction (lettres, chiffres et _)
  - Les arguments (ou paramètres) de la fonction entre parenthèses

```
double calcul(int a,double b);
void affiche();
```

Les fonctions (2)

- La valeur de retour est donnée (si besoin) à l'aide du mot clé **return**.

```
double calcul(int a,double b)
{
  if (a>0) return a+b;
  return b-a;
}
```

- Utilisation : `double y=calcul(5,3.6);`

```
double x;
...
x=calcul(-5,6.254);
```

Adresse mémoire et pointeur

- Toutes les variables sont stockées en mémoire.
- Le caractère **&** permet d'obtenir l'adresse mémoire d'une variable.

```
int x=2;
printf("Adresse de x : %p\n",&x);
```

- Le caractère ***** permet de définir le type **pointeur** (une adresse mémoire).

```
int *pt=&x;
```

Valeur pointée

- Pour obtenir la valeur pointée (le contenu de l'adresse mémoire ou le contenu du pointeur), on utilise à nouveau le caractère *

```
int x=2;
int *pt=&x;
printf("Adresse de x : %p\n",pt);
printf("Valeur de x  : %d\n",*pt);
```

Les tableaux

- Les tableaux sont des pointeurs constants (adresse mémoire non modifiable).
- On utilise les caractères [et] pour la définition et la récupération de la valeur.
- On peut utiliser les caractères { et } pour l'initialisation.

```
int tab[10]; // Tableau de 10 entiers
int tab2[5]={4,5,2,1,3};
int x=tab2[1]; // x vaut 5 (2ème case)
tab2[3]=0; // Met 0 dans la 4ème case
```

Types structurés

- Parfois, il est intéressant de grouper plusieurs variables en un seul élément ou type :

```
struct cbTab
{
  int len;
  char *tab;
};
```

- On accède aux champs, avec le caractère `.` (ou `->` pour les pointeurs)

```
struct cbTab cb;
cb.len=5;
cb.tab=(char *)malloc(cb.len*sizeof(char));
```

Définition d'un nouveau type

- Le mot clé typedef permet de définir un nouveau type de donnée

```
struct cbTab
{
  int len;
  char *tab;
};
typedef struct cbTab cbt;
cbt cb;
cb.len=10;
...
```

Initialisation des structures

- Si on respecte l'ordre des champs d'une structure, on peut l'initialiser ainsi :

```
struct cbPoint{int x,y;};
typedef struct cbPoint point;
point p={5,8}; // p.x=5 et p.y=8
```

- Avec la norme C99, on peut mettre les valeurs dans le désordre avec le nom du champ :

```
point p2={.y=15,.x=2};
```

Fonction principale

- C'est le point d'entrée d'un programme C
- Elle possède plusieurs syntaxes :

```
int main();
int main(void);

int main(int argc,char **argv);
int main(int argc,char *argv[]);

int main(int argc,char *argv[],char *envp[]); // gcc
```

Directives préprocesseur

- Inclusion de fichiers d'en-tête
  - **#include** <stdio.h> // Fichiers « systèmes »
  - **#include** "cblib.h" // Fichiers personnels
- Définition de macros
  - **#define** LEN 50
  - **#define** LSB(x) (x&0xFF)
  - **#define** ADD(a,b) (a+b)
- Suppression de macros : **#undef** LEN

Autre directives

- Directives conditionnelles
  - **#if, #elif**, **#else**, **#endif**

```
#if xxxx
   ...
#elif yyyy
   ...
#else
   ...
#endif
```

  - **#ifdef** (si macro définie)
  - **#ifndef** (si macro non définie)

Macros prédéfinies

- **__FILE__** Nom du fichier
- **__LINE__** Numéro de ligne courante
- **__FUNCTION__** Nom de la fonction courante
- **__DATE__** Date de compilation
- **__TIME__** Heure de compilation

- **NULL** Pointeur de valeur nulle

Différentes zones mémoire

- Il y a principalement 3 zones mémoire différentes pour les variables :
  - Zone des variables globales/statiques
    - Section .data du programme
  - Zone des variables locales/automatique
    - La **pile** (stack)
  - Zone des variables dynamiques
    - Le **tas** (heap)

Variables globales

- Les variables globales sont définies en dehors de toute fonction ou notées **static** :

```
int a=12; // Variable globale (.data)
...
int main()
{
  int b=5; // Variable locale (pile)
  static int c=8; // Variable globale
  ...
}
```

Attributs const et volatile

- Lorsque l'on défini une variable avec l'attribut **const**, on ne peut plus modifier sa valeur par la suite. Cela devient une constante ! (segment mémoire `.rodata`)
  - **const** int WIDTH=50;
- L'attribut **volatile** permet d'indiquer au compilateur de ne pas optimiser l'accès à cette variable.

Variables externes

- S'il on veut utiliser une variable définie dans un autre fichier/librairie, il faut la déclarer **extern** :

```
extern int debug;
```

```
// prg.h
#ifdef PRG_SRC
 int debug=0;
#else
 extern int debug;
#endif
...
```

```
// prg.c
#define PRG_SRC
#include "prg.h"
...
```

```
// main.c
#include "prg.h"
...
```

Variable en registre

- Parfois, il peut être utile de placer une variable dans une zone à accès très rapide pour accélérer les calculs (e.g. un registre du processeur).
- On utilise le mot clé **register** :

```
register int x=0x09080706;
```

```
(gdb) info register
eax            0x1                 1
ecx            0x80018020          -2147385312
edx            0x0                 0
ebx            0x09080706          151521030
```

Allocation mémoire

- Pour les tableaux de taille variable ou dynamique, on peut allouer des zones mémoire (dans le « tas ») avec les fonctions de la librairie stdlib.h
  - // Alloue une zone mémoire de 5 entiers (20 octets)
  - int *p1=(int *)**malloc**(5*sizeof(int));
  - int *p2=(int *)**calloc**(5,sizeof(int)); // Avec mise à 0
- Attention à libérer la mémoire après utilisation
  - **free**(p1);
  - **free**(p2);

Type énumération

- Le mot clé enum permet de définir une liste de constantes entière.

```
enum direction {NORD,EST,SUD,OUEST};

enum booleen {faux=0,vrai=1}; // Avec valeurs
```

- Exemple :

```
enum direction d=EST;
```

Références

- http://fr.wikipedia.org/wiki/C_%28langage%29
- The C programming language – B. KERNIGHAN, B. RITCHIE
- WG14-ISOIEC-9899-n1570
- Programmation en langage C - Anne CANTEAUT (INRIA)