IN3R11-1 – IN3R11-2 – IN3R21

Cours 2 : Compilation d'un programme C, introduction à la syntaxe C/Java

Damien MASSON
d.masson@esiee.fr

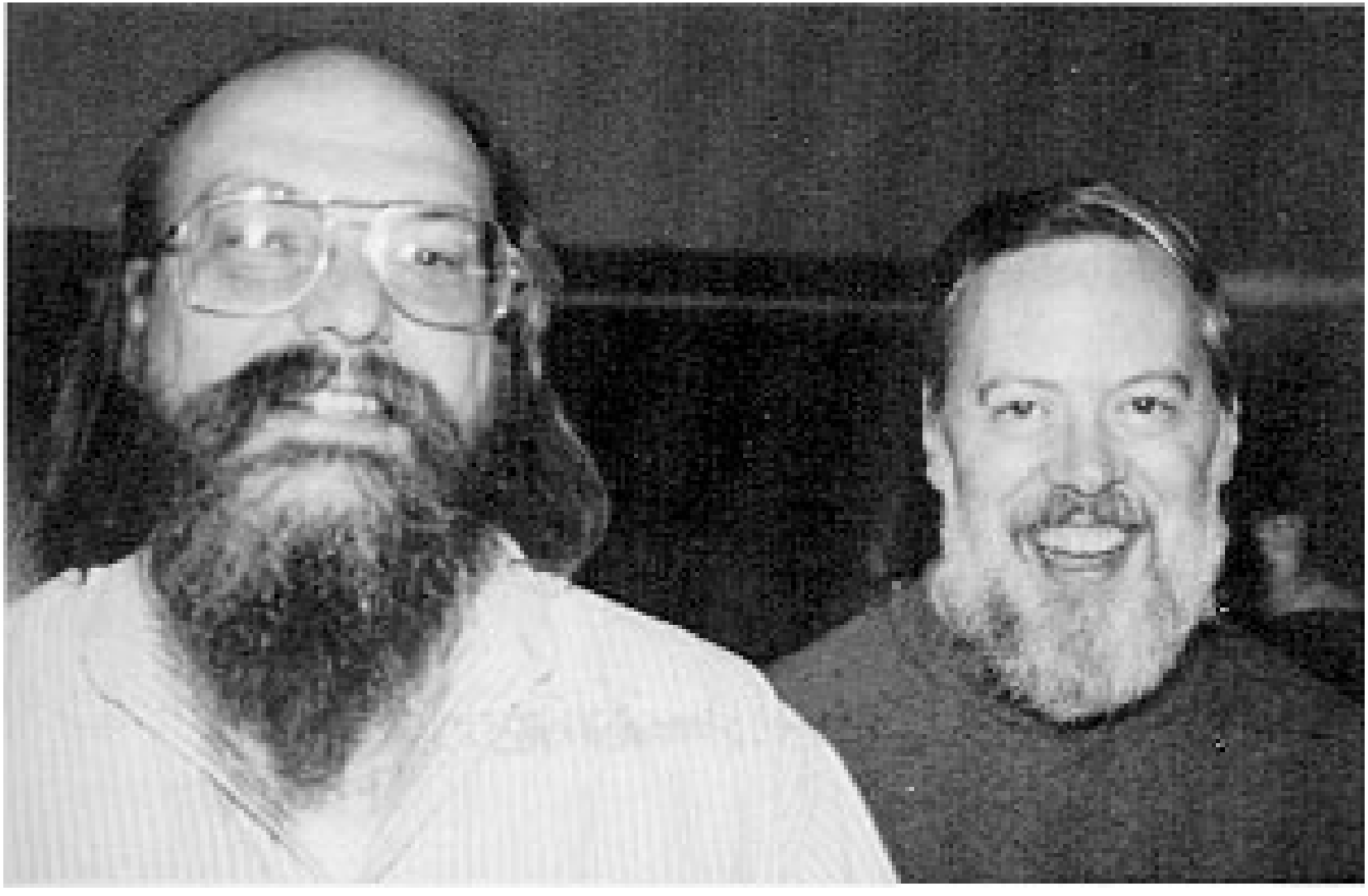
http://esiee.fr/~massond/Teaching/2011-12/

Dernière modification : 11 septembre 2011

Histoire

- besoin d'un langage de haut niveau pour porter UNIX
- C parce qu'évolution de B (évolution de BCPL)
- 1972, laboratoires Bell, Dennis Ritchie et Kenneth Thompson
- 1989 : norme ANSI C (puis ISO C 90)
- 1999 : norme ISO C 99, pas implantée par tous les compilateurs
- langage impératif à la fois haut et bas niveau

Les coupables



Références

- B.W. Kernighan et D.M. Ritchie, Le langage C
- `http://fr.wikibooks.org/wiki/Programmation_C`
- Le cours sur lequel celui-ci est plagié : `http://igm.univ-mlv.fr/~paumier/C`

C ou Java ?

C

- accès direct à la mémoire
- encore très utilisé
- beaucoup d'erreurs liées à la mémoire
- programmation système (Windows, UNIX, MacOS)
- on peut faire de l'objet, mais...
- documentation séparée (man, web, mail au développeur...)

Java

- complètement portable
- gestion automatique de la mémoire
- extrêmement répandu, dans de plus en plus de domaines
- orienté Objet
- puissants IDE
- documentation générée et intégrée au code

Pour les performances, aujourd'hui, c'est pareil. (C plus rapide que Java pour certains programmes, l'inverse pour d'autres...).

Chaine de compilation

- l'humain écrit des fichiers `.c`
- la compilation de ces fichiers produit des fichiers `.o`
- l'édition de liens entre ces fichiers et des librairies produit un exécutable

Les fichiers sources (.c)

- fichiers textes en ASCII
- contiennent des instructions qui seront converties en code assembleur
- ces fichiers sont portables a condition de respecter la norme (pour ce cours ISO C 90)
- programmation modulaire : un fichier par thème

Hello World

HelloWorld.c

```c
/* HelloWorld.c
 * Le classique des classiques */

#include <stdio.h>

int main(int argc,char* argv[]){
   printf("Hello world!\n");
   return 0;
}
```

- `/* commentaires */`
- `include` : inclusion d'une librairie
- `main` : point d'entrée du programme

Compilation et exécution

- Compilation

```
>$gcc HelloWorld.c -c
>$gcc HelloWorld.o -o HelloWorld
```

ou en une étape :

```
>$gcc HelloWorld.c -o HelloWorld
```

- Exécution

```
>$./HelloWorld
Hello world!
```

Un autre exemple

prog.c

```
#include <stdio.h>

#define PI 3.14

float a,b;

float sum(float x,float y) {
  return x+y;
}

int main(int argc,char* argv[]) {
  float c,d;
  a=PI;
  b=1;
  c=sum(a,b);
  printf("%f + %f = %f\n",a,b,c);
  printf("d=%f\n",d);
  return 0;
}
```

```
>$gcc prog.c -o prog
```

En deux fichiers

prog.c

```
#include <stdio.h>

#define PI 3.14

float a,b;

extern float sum(float x,
    float y);

int main(int argc,char* argv
    []){
  float c,d;
  a=PI;
  b=1;
  c=sum(a,b);
  printf("%f + %f = %f\n",a,
      b,c);
  printf("d=%f\n",d);
  return 0;
}
```

sum.c

```
float sum(float x,float y){
  return x+y;
}
```

```
>$gcc -c sum.c
>$gcc -c prog.c
>$gcc sum.o prog.o -o prog

>$gcc sum.c prog.c -o prog
```

En trois fichiers

prog.c

```c
#include <stdio.h>
#include "sum.h"

#define PI 3.14

float a,b;

int main(int argc,char* argv
    []){
  float c,d;
  a=PI;
  b=1;
  c=sum(a,b);
  printf("%f + %f = %f\n",a,
      b,c);
  printf("d=%f\n",d);
  return 0;
}
```

sum.c

```c
float sum(float x,float y){
  return x+y;
}
```

sum.h

```c
float sum(float x,float y);
```

```
>$gcc -c sum.c
>$gcc -c prog.c
>$gcc sum.o prog.o -o prog

>$gcc sum.c prog.c -o prog
```

Sources et Headers

- .c = code : on n'inclut jamais les .c !
- .h = en-tête : on ne compile jamais les .h !
  - définitions des fonctions visibles de l'extérieur du .c
  - définitions de constantes
  - description de la bibliothèque
  - licence du code

Pour éviter les problèmes si plusieurs .c incluent le même .h

```
#ifndef SUM_H_
#define SUM_H_
float sum(float x, float y);
#endif
```

Un beau .h

angles.h

```
/*
 * Copyright (C) 2007
 * Sebastien Paumier <paumier@univ-mlv>
 *
 * This library is free software;
 * you can redistribute it and/or
 * etc.
 *
 */
#ifndef ANGLES_H_
#define ANGLES_H_
/**
 * This library provides implementations
 * of trigonometric functions.
 */
#define PI 3.14
float cosinus(float angle);
float sinus(float angle);
#endif
```

Options de compilations

Obligatoires pour l'unité :

- `-ansi` compatibilité ascendante avec la norme ANSI/ISO
- `-pedantic` garantit la portabilité du code sur tous les compilateurs qui respectent ANSI/ISO
- `-Wall` affiche des avertissements supplémentaires

Options de compilations

Obligatoires pour l'unité :

- `-ansi` compatibilité ascendante avec la norme ANSI/ISO
- `-pedantic` garantit la portabilité du code sur tous les compilateurs qui respectent ANSI/ISO
- `-Wall` affiche des avertissements supplémentaires

```
int main (int argc, char *argv[])
{
        int i, n = argc;
        double x[n];
        for (i = 0; i < n; i++)
                x[i] = i;
        return 0;
}
```

```
>$ gcc -Wall -ansi gnuarray.c
>$ gcc -Wall -ansi -pedantic gnuarray.c
: In function 'main':
:5: warning: ISO C90 forbids variable length array 'x'
```

Quelques conseils utiles

- attention si votre environnement est traduit, les traductions des messages d'erreurs sont souvent approximatives.
- traitez toujours les erreurs de compilation dans l'ordre car une erreur peut en causer une autre.
- indentez votre code (vous même, à l'aide de votre éditeur, avec la commande `indent`...)

Indentation ?

f.c

```
int foo( void)
{int i,j =0 ;
int u;
for (i=0; i < 10 ;i ++)
{j=j+i;
u=u+j;
}
return u;}
```

f.c

```
int foo(void){
  int i, j=0, u;
  for(i=0 ; i<10 ; i++){
    j=j+i;
    u=u+j;
  }
  return u;
}
```

Makefile

- `Makefile` ou `makefile` : fichier décrivant la compilation
- `make` : utilitaire utilisant un `makefile` pour automatiser la compilation
- permet de ne recompiler que ce qui a changé (date des fichiers)
- suite de règles

une règle

cible : dép1 dép2 ...
<TAB> action1
<TAB> action2

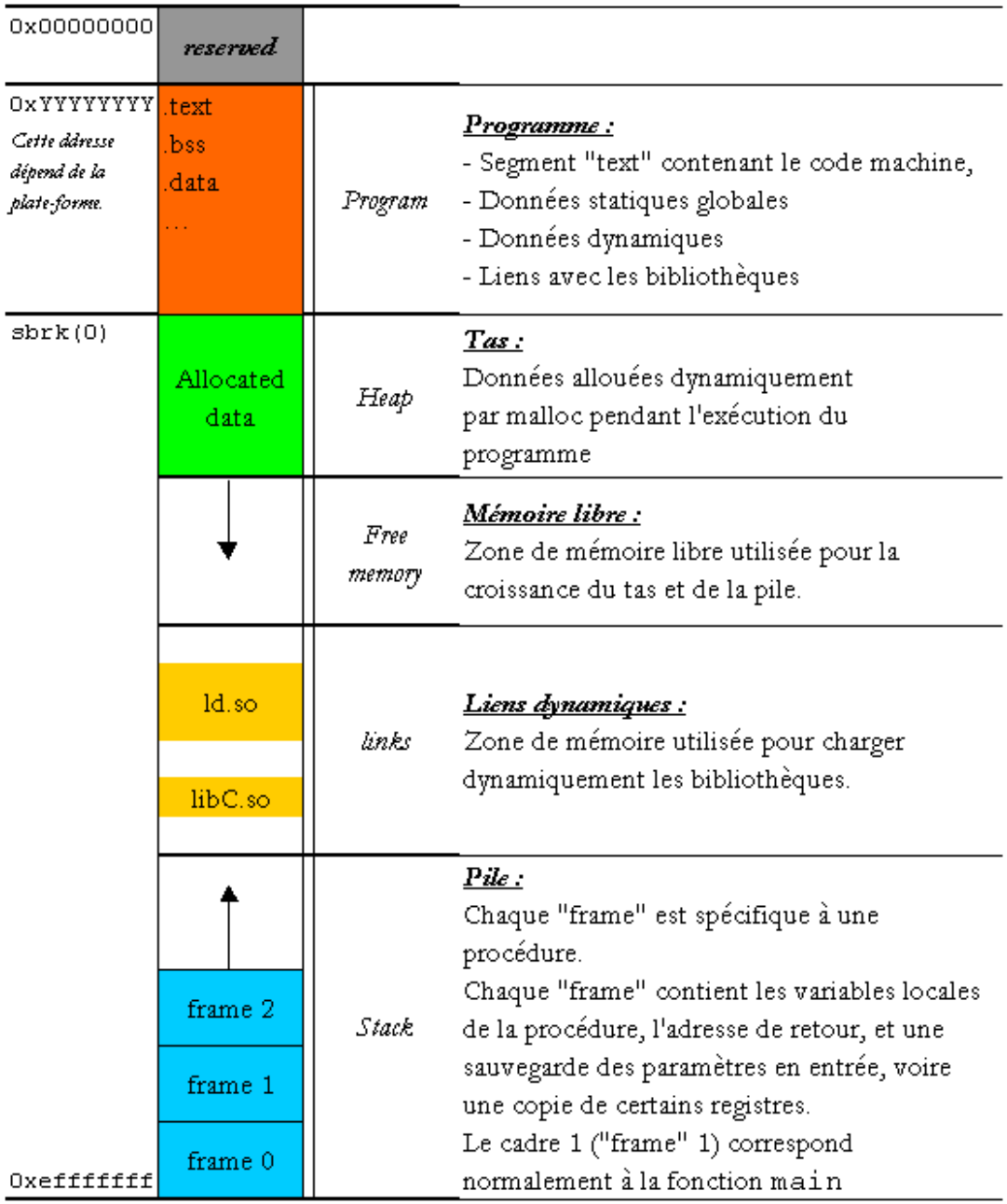
...

Makefile

```
all: trigo
trigo: main.o angles.o
        gcc main.o angles.o -o trigo
        @echo END
clean:
        rm -f trigo main.o angles.o
%.o: %.c
        gcc -c -Wall -ansi $<
```

La mémoire

voir http://ilay.org/yann/articles/mem/



La programmation séquentielle

- 2 modèles de machine : machine à pile ou machine à registres
- instructions = opérations sur la pile ou les registres
- En C, on manipile des variables nommées et typées, qui sont stockées sur une pile
- programme séquentiel : on execute les instructions dans l'ordre !

Instruction

- une instruction en C est une séquence de caractères comprenant des noms de variables et des opérateurs et qui se termine par un ;
- ex : soit deux variables a et b, `a+b;` est une instruction.
- une instruction a de plus une valeur. Dans le cas de a+b ;, cette valeur dépend de la valeur des variables a et b.
- une instruction peut avoir un effet de bord. Ce n'est pas le cas de a+b ; : aucun changement sur la pile (ou autre zone mémoire)
- en revanche, l'expresssion c=a+b ; a la même valeur que a+b ; avec en plus l'effet de bord de changer la valeur de la variable c, c'est a dire le contenu de la pile à l'emplacement nommé c.
- parenthèsage comme en mathématique : d = (c=a+b) ; a un double effet de bord : modifie les variables c et d.
- d=c=a+b ; est ambigue.

La fonction

Son appel correspond à la création d'une nouvelle frame sur la pile.

- elle possède une signature :
  `typeR nomFunc(typeP1 nomP1, typeP2 nomP2, ...);`
- la ligne ci dessus déclare la fonction.
- pour la définire, il faut remplacer le ; par un {
- elle commence par la déclaration de toutes les variables qui seront utilisées
- elle se termine par un {
- on l'appel depuis une autre fonction par son nom, en renseignant une **valeur** (et pas une variable) pour chacun de ses paramètres
- les valeurs passées sont copiées au début de la nouvelle frame sur la pile

TODO : exemple (au tableau pour l'instant)

L'appel de fonction

- C'est une instruction !
- qui peut avoir des effets de bords ou pas (on parle alors de procédure)
- sa valeur est celle du retour (copie de la variable revoyée si assignée)

Main : la fonction première

- c'est le point d'entré dans le programme
- sa signature complère est : `int main(int argc, char** argv);`
- mais on peut écrire : `int main(void);`
- l'entier et le double pointeur servent à transmettre des paramètres au programmes, nous verrons plus tard comment ca fonctionne

Variables

Variable locales :

- l'identificateur commence forcément par une lettre $[a-z][A-Z]$ ou par _
- peut aussi contenir des chiffres
- les mots clés du langage sont réservés
- on évite les noms qui ressemblent à des mots clés d'autres langages (new, class)
- déclaration uniquement en début de bloc $\{\}$: `type nom ;` par exemple : int i ;
- initialisation possible : int i $= 3$;
- déclarations multiples : int i $= 7$, j $= 6$, k ;
- visibilité jusqu'à la fin du bloc.

Variables globales : variables déclarées à l'extérieur d'une fonction. Elles sont accessibles dans toutes les fonctions (à éviter autant que possible).

Variables

Exemple

```c
#include <stdio.h>
int main(){
        int i = 1;
        while(1){
                int i = 0;
                printf("%d\n",i);/*0*/
                break;
        }
        printf("%d\n",i);/*1*/
        return 0;
}
```

Constantes

```
#define identificateur valeur
```

Le préprocesseur, AVANT la compilation, remplace automatiquement toutes les occurences de “identificateur” par “valeur” dans le fichier source (sauf dans les chaînes constantes).

ATTENTION aux noms

```
#include <stdio.h>

#define A 143
int main(int argc,char* argv[]) {
    int A=12;
    printf("%d\n",A);
    return 0;
}
```

Règle : constantes en majuscules, variables locales en minuscules, pas de variables globales.

Constantes

À quoi ca sert ?

mal

```
void foo(int a1[],int a2[]){
    int i;
    for(i=0;i<40;i++) {
        a1[i]=a2[i];
    }
    for(i=2;i<40;i++) {
        a1[i]=a1[i]+a1[i-2];
    }
}
```

bien

```
#define N 40
void foo(int a1[],int a2[]){
    int i;
    for(i=0;i<N;i++) {
        a1[i]=a2[i];
    }
    for(i=2;i<N;i++) {
        a1[i]=a1[i]+a1[i-2];
    }
}
```

Si la valeur change, une seule modification.

Types

- type=taille de zone mémoire + interprétation (signé/non signé, nombre entier/flottant)
- dépendent de la machine (ex. pointeurs=4 octets sur machines 32 bits, 8 sur machines 64 bits)
- seule contrainte du C :
  $sizeof(char) \leq sizeof(short) \leq sizeof(int) \leq sizeof(long)$
- taille du type `type_t` obtenue par `sizeof(type_t)`

Entiers

En général (attention les tailles ne sont pas dans la norme) :

	octet	Signé ?	Valeurs
`signed char`	8 bits	oui	$[-127; 127]$
`unsigned char`	8 bits	non	$[0; 255]$
`char`	8 bits	?	signé ?
`int`	32 bits	oui	$[-(\frac{2^{32}}{2} - 1); \frac{2^{32}}{2} - 1]$
`unsigned int`	32 bits	non	$[0; 2^{32}]$

- la norme C contraint les intervalles des types signés codés sur N bits entre $-(\frac{2^N}{2} - 1)$ et $\frac{2^N}{2} - 1$
- certains compilateurs autorisent la valeur $-(\frac{2^N}{2})$
- problème de portabilité, d'où l'importance de `-ansi`
- le compilateur n'avertit pas en cas de débordement !

Débordement

Exemple

```c
#include <stdio.h>

int main(void){
        unsigned char c1 = 256; /* warning */
        unsigned char c2 = 255;
        c2++; /* pas de warning */
        printf("%d %d\n",c1,c2); /* affiche 0 0 */
        return 0;
}
```

Opérateurs sur les entiers

- addition : $5 + 2 = 7$
- soustraction : $5 - 2 = 3$
- multiplication : $5 * 2 = 10$
- quotient de la division entière : $5/2 = 2$
- reste de la division entirère : $5\%2 = 1$
- $i++$: vaut i et a pour effet d'incrémenter i
- $++i$: vaut $(i + 1)$ et a pour effet d'incrémenter i
- idem avec $i--$ et $--i$

```c
int i = 1;
int a = i++;
printf("%d %d\n",i,a);
/* => 2 1 */
```

```c
int i = 1;
int a = ++i;
printf("%d %d\n",i,a);
/* => 2 2 */
```

Types réels

- `float` : environ 7 chiffres significatifs (non normé)
- `double` : environ 15

Dans les deux cas, les calculs ne sont pas exacts : travail sur des arrondis. À proscrire si le résultat attendu est exact (utiliser deux variables entières).

float.c

```c
#include <stdio.h>
int main(int argc,char* argv[]) {
    float f=0.f;
    int i;
    for (i=0;i<30;i++) {
        f=f+0.1f;
    }
    printf("f=%f\n",f);
    return 0;
}
```

Résultat

```
dm@pc4206b:~/sync/C/tests$
    gcc float.c -o float
dm@pc4206b:~/sync/C/tests$
    ./float
f=2.999999
```

Opérations sur les réels

- + - * : comme pour les entiers
- / : division réelle (avec arrondis)
- l'opérateur provoque des conversion (cast) automatiques vers le type le plus précis

```
int a=...
int b=...
float c = a/b /*division entiere*/
float c = (float)a/b /*ok*/
```

char

- on interprète les char comme des codes de caractères
- pas de problème entre 0 et 127
- au-delà, dépend de l'encodage du système (à éviter)
- on désigne le code du caractère x avec 'x'
- on peut utiliser les opérateurs entiers, par exemple : 'a'+1 vaut 'b' (car les codes sont bien rangés)

Charactères spéciaux :

- '\n' : saut de ligne
- '\r' : retour au début de la ligne
- '\t' : tabulation
- '\\' : \
- '\"' : "

Tableaux

- type nom[taille] ; réserve sur la pile l'espace necessaire pour taille variable de type type
- acces à la n^e par : nom[n] ;
- un tableau a donc une taille fixée, mais cette taille n'est pas accessible par le programme (il faudra la passer en paramètre si le tableau est passé à une fonction)
- ceci n'est qu'une utilisation déguisée des pointeurs, comme on le verra plus tard.

Chaînes de caractères

- contrairement à Java, il n'y a pas de type String en C.
- par convention, la portion d'un tableau de char (*char*[] ou *char*$*$) qui se termine par le caractère '\0' est une chaîne de caractère
- comme en Java, on peut écrire des chaînes constantes entre "
- pas de +, mais concaténation automatique ("aaa""aaa" = "aaaaaa")
- impossible de modifier les chaînes constantes
- fonctions de manipulations des chaînes dans < *string.h* >

Trois lignes équivalentes

```
char * msg = "Hello" ;
char msg[] = "Hello" ;
char msg[] = {'H','e','l','l','o','\0'};
```

Mon type

On peut créer des types avec `typedef`

Type booléen

```
#define TRUE 1
#define FALSE 0
typedef unsigned char boolean;
int main(void){
        boolean b=TRUE;
        ...
        if(b){
          ...
        ...
        return 0;
}
```

La boucle for

- Les boucles permettent de ... boucler dans l'execution séquentielle du programme. Elles permettent donc d'exécuter plusieurs fois une instruction écrite une seule fois.
- il en existe en C trois : for, while et do while. Cependant, on peut toujours s'en sortir en utilisant for, et sa syntaxe est plus complète. Je vous encourage donc à n'utiliser que celle la (au début du moins)
- for (initialisation du compteur ; condition (a priori sur le compteur ?) de continuation ; modification du compteur) { liste d instructions ; }
- Par exemple :

```
for (i=1; i<6; i++) {
        printf("%d", i);
}
```

- sortie immédiate de la boucle : break ;
- saut immédiat au test : continue ;

If...

- il est possible d'écrire une instruction à n'exécuter que si le résultat d'un test est vrai :

```
if (condition) {liste dinstructions;}
```

- la condition doit être entre des parenthèses
- il est possible de définir plusieurs conditions à remplir avec les opérateurs ET et OU (&& et ——)
- Par exemple l'instruction suivante teste si les deux conditions sont vraies :

```
if ((condition1)&&(condition2))
```

  L'instruction suivante exécutera les instructions si l'une ou l'autre des deux conditions est vraie :

```
if ((condition1)||(condition2))
```

- s'il n'y a qu'une instruction, les accolades ne sont pas nécessaires...
- les instructions situées dans le bloc qui suit else sont les instructions qui seront exécutées si la ou les conditions ne sont pas remplies : if(cond){...}else{...}

switch

Cette instruction permet un branchement conditionnel en fonction de la valeur d'une variable :

```
switch (Variable) {
case Valeur1 :
        Liste d instructions;
        break;
case Valeur2 :
        Liste d instructions;
        break;
case Valeur3 :
        Liste d instructions;
        break;
default:
        Liste d instructions;
}
```

Même si on peut toujours écrire le même comportement logique avec une suite de if/else, ce n'est pas équivalent du point de vu du nombre d'opérations effectuées.

printf

fonction d'affichage sur la sortie standard, définie dans `stdio.h`

```
printf("format", arg1, arg2, ...);
```

Les variables sont indiquées par des caractères spéciaux :

- %d : entier
- %c : caractère
- %f : réel
- %s : chaîne de caractère
- %% : pour le caractère %

Exemple :

```
printf("moyenne(%d,%d)=%f\n",i,j,moy);
```

- l'affichage est bufferisé, le buffer est vidé à chaque '\n' ou lorsqu'il est plein
- pour afficher sur la sortie d'erreur standard (ou dans un fichier) utiliser `fprintf(stderr,...)`

scanf

Fonction de saisie au clavier de $< stdio.h >$ qui ressemble à printf, mais :

- '&' devant les variables, sauf les chaînes
- la chaîne de format n'est pas affichée
- les esapces dans la chaîne de format sont ignorés
- la bufferisation se fait par ligne

Exemple :

```
int a,b; char s[64];
scanf("%d␣%d␣%s",&a,&b,s);
```

Explication du '&', voir cours pointeurs.

scanf

- Lors de la saisie, les chaînes sont délimitées par les espaces, les tabulations et les sauts de ligne.
- scanf renvoit un entier ! ! il s'agit du nombre de paramètres saisis correctement par l'utilisateur.

Exemple :

```
int main(int argc,char* argv[]){
    int a,b,c,n;
    n=scanf("%d␣%d␣%d",&a,&b,&c);
/* 4 8 hello */
    printf("%d␣%d␣%d␣%d\n",n,a,b,c);
/* 2 4 8 34345849648 */
    return 0;
}
```

scanf

Attention à ce qui reste dans le buffer

```c
#define NO 0
#define YES 1
int yes_or_no()
{
  char c;
  do {
    printf("y/n ? ");
    scanf("%c",&c);//" %c"
  } while (c!='y' && c!='n');
  if (c=='y') return YES;
  return NO;
}
int main(int argc, char ** argv){
  if(yes_or_no())
    printf("yes\n");
  else
    printf("no\n");
}
```

```
>$ ./a.out
y/n ? r
y/n ? y/n ?
```