Partie B

Rappels : Langage C

IUT A

Cours n° B.1

Rappels de C, structure d'un programme

Présentation générale

- ☞ Programme C $\equiv$ ensemble de fonctions
- ☞ Langage défini par une norme. Nous utiliserons la norme ISO C90 (également appelée ANSI C).
- ☞ Les fonctions (leur code source) sont réparties dans un ou plusieurs fichiers textes
- ☞ Une des fonctions **doit** se nommer `main` :
  c'est le code de cette fonction qui sera exécuté par le programme

➠ **Le langage C est n'est pas un langage de très haut niveau :**

- Accès bas niveau à l'architecture des machines (code souvent rapide)
- Typage faible
- En général peu de contrôle de cohérence :

les compilateurs considèrent que le programmeur est intelligent et qu'il sait ce qu'il fait ! ! !

Variables

- ☞ Définir une **variable** c'est :
  1. réserver une zone mémoire pour des données d'un certain type
  2. nommer cette zone pour faire référence à son contenu via un identificateur
- ☞ Un **type** correspond principalement à la taille des données manipulées
  L'opérateur `sizeof()` *permet de connaître la taille (en octets) occupée par une variable ou un type.*

Peu de types primitifs :

Symbole	Taille occupée	Données représentées
`char`	1 octet	entier
`int`	dépend de la machine	entier
`float`	dépend de la machine	nombre en virgule flottante simple précision
`double`	dépend de la machine	nombre en virgule flottante double précision

Qualifications

Il est possible de préfixer un type par un ou plusieurs mots réservés de façon à préciser la manière dont une variable doit être considérée (pour l'arithmétique par exemple).

- `short` et `long`
- `signed` et `unsigned`
- `const`
- `extern`
- `static`
- `register`
- ...

Généralités

- Syntaxe des fonctions :

```
type_de_retour nom (liste_des_paramètres) { code }
```

- **Toute fonction doit être déclarée avant d'être utilisée** (possibilité de simplement prototyper)
- Les structures de contrôle sont, dans l'ensemble, identique à celles de JAVA
- Les opérateurs sont, dans l'ensemble, identique à ceux de JAVA
- **Les définitions de variable doivent être placées AVANT les instructions** (dans les fonctions ou en dehors des fonctions)
- **Une définition de variable n'initialise pas le contenu de la variable** (valeur inconnue avant première affectation)

```c
int une_variable_globale;

void fait_pas_grand_chose(void);

int main (void)
{
  if (une_variable_globale == 3)
  {
    fait_pas_grand_chose();
  }
  return 0;
}

void fait_pas_grand_chose(void)
{
  char c;

  c = 7;

  une_variable_globale = (int) c;
}
```

Affichage formaté

```
int printf(const char *format, ...);
```

- Définition via l'utilisation des headers :

```
#include <stdio.h>
```

- Le format représente une suite d'instructions permettant de définir **quoi** et **comment** imprimer.
- Le format peut inclure :
  - des caractères classiques
  - des séquences d'échappement (`\n`, `\t`, etc)
  - des spécificateurs de conversions :
    - `%d` affichage comme un entier
    - `%c` affichage comme un caractère
    - `%s` affichage comme une chaîne de caractères

À chaque spécificateur doit correspondre un paramètre du bon type

```c
#include <stdio.h>

int
main (int argc, char ** argv)
{
  int i = 66;

  char c = 'C';

  char *s = "Bonjour tout le monde";

  printf ("avec %%d : %d\n", i);
  printf ("avec %%c : %c\n", i);
  printf ("avec %%d : %d\n", s);
  printf ("avec %%c : %c\n", s);
  printf ("avec %%s : %s\n", s);
}
```

Chaîne de compilation

Pour passer du fichier source au fichier exécutable plusieurs étapes sont nécessaires :

① Utilisation d'un préprocesseur ..... `cpp`
transformation du source par remplacement textuel en C pur

② Utilisation d'un compilateur ..... `cc1`
transformation du source en code assembleur

③ Utilisation d'un assembleur ..... `as`
transformation du source assembleur en langage machine (objet)

④ Utilisation d'un éditeur de liens ..... `ld`
assemblage des différents objets et ajout du code de démarrage

gcc

Toutes ces étapes sont généralement masquées par l'outil de développement qui permet de générer le programme exécutable en deux étapes :

① `gcc -c essai.c` ..... étape ① à ③

② `gcc -o essai essai.o` ..... étape ④

➠ On dit souvent que l'outil de compilation (gcc) est un *wrapper*

plus de détails dans le manuel : `gcc(1)`

Options de compilation

Chaque compilateur possède son jeu de fonctionnalités supplémentaire étendant la norme C ANSI (norme ISO C90).

Ces fonctionnalités peuvent être désactivées afin d'assurer un source *portable*.

`gcc` comporte un **très** grand nombre d'options, dont :

- `-ansi`
  Désactive les fonctionnalités de `gcc` ne respectant pas la norme ANSI
- `-pedantic`
  Rejette tous programmes ne respectant pas strictement la norme ANSI
- `-Wall -W`
  Active un grand nombre d'avertissement
- `-Werror`
  Transforme tous les avertissements en erreur
- `-fno-builtin`
  Désactive la gestion implicite de certaines fonctions par le compilateur
- `-g`
  Ajoute les informations utiles aux déboggueurs dans les fichiers générés

Préprocesseur

- Le préprocesseur est un outil permettant de transformer du texte via des remplacements
- Il offre, entre autre, la possibilité dans un fichier source :
  - de définir des *macros* qui seront remplacées par leur valeur partout dans le fichier
    ```
    #define MACRO valeur
    ```
  - d'inclure le contenu d'un autre fichier
    ```
    #include "fichier"
    #include <fichier>
    ```
  - de tester l'existence d'une macro
    ```
    #ifdef MACRO
       /* cette partie restera si MACRO est definie */
    #else
       /* cette partie restera si MACRO n'est pas definie */
    #endif
    ```

Préprocesseur

- main.c

```
int main(int argc, char **argv)
{
#include "simpleIncluded.c"
```

- simpleIncluded.c

```
  printf("cet include \"casse\" la syntaxe C\n");
}
```

Ces deux fichiers génèrent un programme C valide

Préprocesseur

```
# 1 "badInclude.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "badInclude.c"

int main(int argc, char **argv)
{
# 1 "simpleIncluded.c" 1
  printf("cet include \"casse\" la syntaxe C\n");
}
# 5 "badInclude.c" 2
```

Attention, le préprocesseur est purement textuel, il est indépendant de la syntaxe C. Ceci est considéré un mauvais exemple de programmation

Compilation

Votre C :

```
int main() {
   int x = 42;
   int y = 5 + x;
   return 0;
}
```

Ressemble à ça en assembleur (la signification exacte importe peu ici) :

```
  push %rbp
  mov  %rsp,%rbp
  movl $0x2a,-0x8(%rbp) ; met 42 dans x
  mov  -0x8(%rbp),%eax  ; prépare l'addition (42 dans EAX)
  add  $0x5,%eax        ; exécute l'addition (5+EAX)
  mov  %eax,-0x4(%rbp)  ; met le résultat dans y
  mov  $0x0,%eax        ; valeur 0 à retourner
  pop  %rbp
  retq
```

Éditions des liens

- ☞ Pour créer un programme il faut créer un fichier (exécutable) contenant tous les objets (variables ou fonctions) qu'il manipule.
- ☞ Quand un objet est défini dans un fichier source le compilateur le définit aussi dans le fichier objet correspondant (l'adresse de l'objet est connu dans ce fichier).
- ☞ Quand un objet est déclaré dans un fichier source il n'est pas obligatoirement défini dans le fichier objet correspondant.

➠ Dans le fichier final toutes les adresses des objets doivent être connues.

Pour uniquement déclarer un objet il faut le qualifier avec `extern` dans le fichier.
Pour **définir** un objet il faut le déclarer et donner sa définition (pour une fonction le code source doit être présent dans le fichier).

Éditions des liens

La définition d'un objet est nécessaire à l'**exécution** du programme : on ne peut pas exécuter une fonction si son code ne fait pas partis du programme, on ne peut pas mettre une valeur dans une variable si il n'y a pas d'espace mémoire réservé pour celle-ci

La déclaration d'un objet est nécessaire pour la **vérification de la correction** du programme (au moment de la compilation). Le compilateur vérifie quand on appelle une fonction qu'on a bien déclaré qu'elle existait, même si on ne l'a pas définie

Compilation d'un exécutable