Initiation à la programmation avec le langage C

Alain Leaper - GEBULL *

15 juillet 2013

Sommaire

I Préambule à l'édition 1 **5**

1 Introduction **5**

2 Rappel des objectifs **5**

3 Premier bilan, vers une édition 2 ? **5**

II Support des séances **6**

4 Séance 1 **6**
4.1 Rapide tour d'horizon, les grandes familles . . . 6
Les langages interprétés . . . 6
les langages compilés . . . 6
Autre approche . . . 6
4.2 Installation du compilateur gcc . . . 6
4.3 Minimum de "survie" sous unix/gnuLinux . . . 6
4.4 Saisie d'un fichier source . . . 6
4.5 Compilation d'un fichier source . . . 7
4.6 Exécution du fichier (exécutable) ainsi obtenu . . . 7
4.7 Un exemple interactif . . . 7
4.8 Erreurs non détectables par le compilateur . . . 8

5 Séance 2 **9**
5.1 Vrai ou Faux ? . . . 9
5.2 Traiter une alternative . . . 9
5.2.1 Une alternative s'exprime avec l'opérateur if . . . 9
5.2.2 Opérateurs de comparaison . . . 9
5.2.3 Opérateurs logiques . . . 9
&& et logique . . . 9
|| ou logique . . . 9
5.3 instruction et bloc d'instructions . . . 10
5.4 L'indendation . . . 10
5.5 Programme exemple . . . 10
5.6 Forme "ésotérique" d'une alternative . . . 10
5.7 Choix multiples . . . 10

*Gâtine Et Bocage Utilisateurs de Logiciels Libres

6 Séance 3 **11**
6.1 Numération à l'aide d'éléments binaires . . . 11
6.2 Nombres signés et non signés . . . 11
6.3 Les types entiers en C . . . 12
6.4 Quel format pour quel type ? . . . 12

7 Séance 4 **13**
7.1 La notation hexadécimale . . . 13
7.2 Exemples d'utilisation . . . 13
7.3 Nombres non entiers . . . 13
7.4 Déclarations de variables . . . 14
7.5 Accès aux éléments d'un tableau . . . 14
7.6 Les opérateurs arithmétiques du langage C . . . 14

8 Séance 5 **16**
8.1 Instructions itératives (boucles) . . . 16
8.1.1 boucle while . . . 16
8.1.2 boucle do while . . . 16
8.1.3 boucle for . . . 16
8.1.4 rupture dans une boucle . . . 16
8.2 Forçage de type . . . 16
8.3 Opérations au niveau bits . . . 17
8.3.1 liste de ces opérateurs . . . 17
8.3.2 Table de vérité du & (et) . . . 17
8.3.3 Table de vérité du | (ou inclusif) . . . 17
8.3.4 Table de vérité du ^ (ou exclusif) . . . 17
8.3.5 Table de vérité du ~ (négation) . . . 17
8.3.6 forme abrégée de ces opérateurs . . . 17

9 Séance 6 **18**
9.1 Adresses mémoire . . . 18
9.2 Classes de stockage . . . 18
9.3 Le modificateur const . . . 18
9.4 Pointeurs . . . 18
9.5 Règles de nommage . . . 18
9.6 Initialisation . . . 19
9.7 Usage . . . 19
9.8 Relation entre pointeurs et tableaux . . . 19
9.9 Chaîne de caractères . . . 19
9.10 Tableaux à plusieurs dimensions . . . 19

10 Séance 7 **20**
10.1 Utilité des fonctions . . . 20
10.2 Déclaration d'une fonction . . . 20
10.3 Définition d'une fonction . . . 20
10.4 Appel d'une fonction . . . 20
10.5 Passage d'arguments par valeur, par adresse . . . 20
10.6 Classe de stockage automatique . . . 21
10.7 Arguments de la fonction main() . . . 21
10.8 Fonctions ou macros ? . . . 21
10.9 Modificateur inline . . . 22

11 Séance 8 **23**
11.1 Variables structurées . . . 23
11.2 Création de types avec typedef . . . 23
11.3 Création d'un type structuré, sans utiliser typedef . . . 23
11.4 Déclaration de variables . . . 23
11.5 Création d'un type structuré, en utilisant typedef . . . 24
11.6 Les unions . . . 24
11.7 Les énumérations . . . 24
. . . 25

12 Séance 9 **26**
12.1 Utiliser plusieurs fichiers . . . 26
12.2 Découpe formelle . . . 26
12.3 Visibilité des variables et des fonctions . . . 26
12.4 Découpe fonctionnelle . . . 27
12.5 Quelques options de gcc . . . 27
. . . 28
12.6 grep : un utilitaire qui vous veut du bien ! . . . 28

13 Séance 10 **29**
13.1 Retour sur les pointeurs . . . 29
13.1.1 Accès aux champs d'une structure avec un pointeur . . . 29
Déclaration d'un pointeur sur un type structuré : . . . 29
Initialisation d'un pointeur sur un type structuré : . . . 29
Atteindre un champ : . . . 29
13.1.2 Pointeurs de fonctions . . . 29
13.1.3 Tableaux de pointeurs de fonctions . . . 30
13.1.4 Fonctions qui retournent un pointeur de fonction . . . 30
13.1.5 Pointeur de fonction utilisé en tant qu'argument d'une fonction . . . 30
13.2 Automates . . . 30

14 Séance 11 **32**
14.1 Opérations sur les fichiers . . . 32
14.1.1 Création d'un flux . . . 32
14.1.2 Ouverture d'un fichier . . . 32
14.1.3 Suppression de l'association d'un flux (monFlux) à un fichier (monFichier) . . . 32
14.1.4 Écriture dans un fichier (ouvert) . . . 32
14.1.5 Lecture dans un fichier (ouvert) . . . 33
. . . 33
. . . 33
14.1.6 Connaître la position courante . . . 33
. . . 33
14.1.7 Modifier la position courante . . . 33
14.1.8 Erreur sur un flux . . . 33
14.1.9 Unix et les fichiers . . . 33
14.2 Variables de classe de stockage dynamique . . . 34
14.2.1 Accès à ces variables . . . 34
14.2.2 Allouer de la mémoire . . . 34

15 Séance 12 **35**
15.1 Le debugger gdb . . . 35
15.2 L'utilitaire make . . . 35
15.3 Graphisme et langage C . . . 36

III Documents disponibles **37**
15.4 Code source 37
15.4.1 Répertoire exemples 37
15.4.2 Répertoire bonus 37
15.5 Debugger GDB 37

Première partie

Préambule à l'édition 1

1 Introduction

Cette initiative se déroule dans le cadre du développement participatif et du partage des connaissances qui est un des principes fondateurs du logiciel libre.
Les outils utilisés sont ceux disponibles dans le cadre du logiciel libre, à savoir :

- le système d'exploitation Unix/gnuLinux (peu importe la distribution)
- un éditeur (gedit, vi, ...)
- le compilateur gcc
- le debugger [1] gdb

Le présent document est, bien sûr, libre et utilisable selon la GPL [2].

2 Rappel des objectifs

. découvrir un autre aspect de l'informatique : "pas seulement acheter ses cravates sur internet"
. contre les idées reçues :
l'ordinateur, " Il veut pas, il sait pas, il peut pas, ..."
Il faut au moins BAC +25 pour comprendre comment ça marche.

3 Premier bilan, vers une édition 2 ?

En dehors d'une amélioration de la présentation (grâce à LaTeX !) et du présent préambule, je ne compte pas apporter de changement au document original qui a servi de support aux séances qui se sont déroulées en 2012/2013 [3].
Tenant compte des remarques des participants, une évolution paraît souhaitable sur les points suivants :

1. utilisation de la ligne de commande Unix/gnuLinux (déplacement dans l'arborescence...).
2. insister d'avantage sur les bases de la programmation.
3. plus d'execices.

Si cette première expérience devait se renouveler, une édition 2 serait à envisager avec les modifications :

- moins entrer dans les particularités du langage C.
- une petite introduction au langage interprété Python.
- des séances plus réalistes compte tenu de la duréee d'une séance (une heure).

Quoiqu'il en soit, cette édition 1 pourrait toujours être utile aux accros. du langage C, en raison de la puissance qu'il tire de l'utilisation des pointeurs...

1. je me refuse à utiliser la « francisation » débogueur, qui n'a aucun sens !
2. General Public License
3. le mercredi soir, au Centre Socio-culturel de Bressuire.

Deuxième partie

Support des séances

4 Séance 1

4.1 Rapide tour d'horizon, les grandes familles

Il n'est pas question d'essayer de lister tous les logiciels de programmation. Il en existe beaucoup dont j'ignore jusqu'à l'existence ! Toute classification est, plus ou moins, arbitraire. En voici une, pour situer le langage C parmi les langages "portables".

Les langages interprétés BASIC, les shells (bash ...), Python, PHP, Perl, awk ...

les langages compilés Pascal, Fortran, ADA, langage C et C^{++} ...

Autre approche

. Java est un cas un peu spéciale, le code java s'exécute sur une machine virtuelle java
. certains langages possèdent une extension orientée objets comme PHP, Python, C (C^{++}) ou sont uniquement objet (Java, Smalltalk).

L'assembleur est un langage non portable, dédié à un type de microprocesseur.

4.2 Installation du compilateur gcc

Il suffit d'installer la paquet gcc
– soit à l'aide d'un gestionnaire graphique, synaptic, par exemple.
– soit avec la ligne de commande dans un terminal :
$ apt-get install gcc

4.3 Minimum de "survie" sous unix/gnuLinux

===> faire un dessin pour expliquer l'arborescence des répertoires (directories)
Ouvrir un terminal, l'invite (prompt) peut prendre différentes formes (en général $).
Une ligne n'est interprétée (ie exécutée) qu'après la frappe de la touche Entrée.

$ cd nomRep -> on se déplace dans nomRep, si celui-ci est "sous" le répertoire actuel.
$ cd .. -> on remonte d'un "cran" dans l'arborescence, possible : cd ../../.. autant qu'il faut...
$ pwd -> indique le répertoire actuel.
$ cd /home/nomUtilisateur/unSousRep -> place dans unSousRep (si existe...).
$ ls -> liste ce qui se trouve sous le répertoire courant, l'option -l (ls -l) donne plus d'infos...

4.4 Saisie d'un fichier source

Une fichier texte comportant l'extension .c est reconnu comme étant un fichier source du langage C.
La saisie d'un fichier source doit se faire avec un éditeur de texte, exemple gedit, jamais avec un traitement de texte.
Un exemple de fichier source est donné par ex1.c. Une archive, exemples.tgz, contenant tous les fichiers exemples est disponible. [4]

4. soit sur GEBULL.com, soit en me le demandant : gargamel79@orange.fr

```
/* ex1.c tout ceci est du commentaire ... mettez tout ce que vous souhaitez ... la date par exemple
Ce programme utilise la fonction printf() qui fait partie de la bibliothèque (library) standard du langage C. */
//voici une autre façon de faire des commentaires (une seule ligne)

#include <stdio.h> //inclusion du fichier stdio.h pour pouvoir utiliser la fonction printf()
/*l'appel de cette fonction permet l'affichage d'une chaîne de caractères (string), son prototype est contenu dans stdio.h */

int main(void) //indique le début du programme, void -> on ne passe pas d'argument, retourne un entier signé (int)
{
printf("boujour tout le monde!!! \n"); //ne pas oublier le ';' pour séparer les instructions
/* \n signifie qu'il ne s'agit pas de la lettre 'n' mais de la commande : aller à la ligne (line feed) dont la valeur est 10 */
return 0; //l'entier de valeur 0 indique la fin normale du programme
}
```

4.5 Compilation d'un fichier source

$ gcc nomDuFichier.c

Il y a cependant quelques options à connaître :

-Wall : donne un maximum d'avertissements (warnings) sur les anomalies détectées par le compilateur ; pas les erreurs.

-o nomFichier : permet d'indiquer le nom souhaité pour le fichier exécutable. Si omis, on génère un fichier a.out par défaut (ce qui est un comble, puisque le format de ce fichier exécutable est ELF!!). Essayer la commande file : $ file a.out

Dans notre exemple, cela donne :

$ gcc -Wall -o ex1.elf (ou tout autre nom, l'extension .elf n'est nullement obligatoire).

4.6 Exécution du fichier (exécutable) ainsi obtenu

$ ./nomExe (si nomExe existe dans le répertoire courant).

Le '.' indique "là où on se trouve"

===> faire un dessin : L'éditeur est chargé en RAM, par le système s'exploitation(SE) pour réaliser la saisie. La sauvegarde du fichier source se fait sur le DD.

Le compilateur gcc est chargé en RAM (par le système s'exploitation), s'exécute, si pas d'erreur génère un fichier exécutable sur le DD.

Le fichier exécutable est chargé en RAM par le SE et s'exécute, toujours en RAM. La libération, éventuelle, de la RAM est à charge du SE.

4.7 Un exemple interactif

Entrée d'une chaîne sur demande du programme, source : ex2.c

Lors de son éxécution, un programme peut réserver des emplacements dans la méoire (RAM).

Ces emplacements sont appelés **variables** et sont repérés par leur nom (identifiant).

Ici le programme réserve une variable de type tableau de caractères, c'est à dire une suite de caractères pour y ranger la chaîne de caractères entrée par l'utilisateur.

L'identifiant est prenom. Tous les caractères ne sont pas permis pour constituer un identifiant. ex2-3.c donne un exemple de ce qui se passe si un caractère invalide est entré dans un identifiant (utlisation d'un caractère accentué).

Identifiants valides : a-z, A-Z, 0 à 9 et _ (underscore)

Cependant un chiffre n'est pas accepté en tant que premier caractère, comme dans ex2-4.c .

4.8 Erreurs non détectables par le compilateur

A partir de l'exécutable obtenu avec ex2.c, essayez d'entrer un prénom composé (exemple Anne Gaëlle).

le compilateur est sympa. : il vous indique les erreurs de syntaxe, mais jamais les fautes de conception -> bug

Pour remédier à cette situation, si les prénoms composés doivent être acceptés, il faut modifier notre programme. C'est ce qui est fait dans ex2-2.c

Bug ou pas bug ?
On peut dire sans beaucoup se tromper que tous les programmes ont des bugs. Ceux-ci se révèlent, ou non, selon l'usage que l'on en fait.

Plus on teste, moins il en reste !

5 Séance 2

5.1 Vrai ou Faux ?

Une variable de type boolean ne possède que deux valeurs : VRAI (THRU), FAUX (FALSE). Contrairement à d'autres langages le type "boolean" n'est pas défini par le langage C.

En langage C, tout ce qui n'est pas "faux" est "vrai". Une condition fausse résulte de l'évaluation d'un entier nul. En conséquence, toutes les autres valeurs (différentes de zéro) sont reconnues comme une condition vraie.

5.2 Traiter une alternative

5.2.1 Une alternative s'exprime avec l'opérateur if

"**if**" signifie "si" et "**els**e" signifie "sinon", else est optionnel

```
if (condition)
  instruction1;
 else
  instruction2;
```

En langage C, les parenthèses servent (entre autres) à forcer l'évaluation d'une expression.
Si condition est vraie, instruction1 est exécutée, sinon (condition fausse) instruction2 est exécutée.

5.2.2 Opérateurs de comparaison

L'évaluation d'une condition se fait à l'aide d'opérateurs :

== Si les valeurs situées à gauche et à droite du signe == sont égales, alors la condition est vraie.Dans le cas contraire, la condition est fausse.

> Si les valeurs situées à gauche du signe > est supérieure à celle de droite, la condition est vraie.Dans le cas contraire, la condition est fausse.

>= Si les valeurs situées à gauche du signe > est supérieure ou égale à celle de droite, la condition est vraie.Dans le cas contraire, la condition est fausse.

< Si les valeurs situées à gauche du signe > est inférieure à celle de droite, la condition est vraie.Dans le cas contraire, la condition est fausse.

<= Si les valeurs situées à gauche du signe > est inférieure ou égale à celle de droite, la condition est vraie.Dans le cas contraire, la condition est fausse.

!= Si les valeurs situées à gauche et à droite du signe != sont différentes, alors la condition est vraie.Dans le cas contraire, la condition est fausse.

ATTENTION, l'opérateur == ne doit pas être confondu avec l'opérateur d'affectation = (simple =).
= affecte à la partie située à gauche la valeur située à droite du signe =.

Remarquer qu'en C, la vérification d'une valeur non nulle s'écrira, le plus souvent :
if (valeur) : équivalent à : if (valeur != 0)
Alors que la vérification d'une valeur nulle s'écrira, le plus souvent :
if (!valeur) : équivalent à : if (valeur == 0)

5.2.3 Opérateurs logiques

Il est possible de combiner plusieurs conditions pour évaluer une condition "finale".

&& et logique :
condition1 && condition2 :est vraie si condition1 et condition2 sont toutes les deux vraies, fausse dans le cas contraire.

|| ou logique :
condition1 || condition2 :est fausse si condition1 et condition2 sont toutes les deux fausses, vraie dans le cas contraire.

5.3 instruction et bloc d'instructions

Entre le if et le else, une seule instruction peut ne pas être suffisante. On utilise alors un bloc d'instructions limité par des accolades { et } .

```
if␣(condition)
␣␣{
␣␣instruction_1;
␣␣instruction_2;
␣␣..............
␣␣}
␣else
␣␣{
␣␣autreInstruction_1;
␣␣autreInstruction_2;
␣␣..............
␣␣}
```

5.4 L'indendation

Chacun a ses petites manies...L'indentation que j'utilise n'est pas une règle appliquée par tous. Cependant la découverte de Python m'a conforté dans cette voie. Dans Python c'est l'indentation qui limite un bloc d'instructions, il n'y a même pas de limiteur de bloc !
Remarquer l'indentation du 5.3, j'y ai fait figurer les espaces pour bien mettre en évidence à qui appartiennent le else et chaque bloc.
Pas d'économie de papier, rendons le code le plus clair possible. Dans le cas de if imbriqués, on s'y retrouve toujours.

5.5 Programme exemple

L'exemple ex3.c "demande" le temps qu'il fait. Seulement 3 réponses sont possibles :

1. soleil
2. pluie
3. couvert

Toute autre réponse est invalide. Le programme donne un conseil s'il trouve une réponse valide.

L'exemple ex3-2.c peut être compilé sans erreur, il est cependant parfaitement illisible (par moi en tout cas !).

5.6 Forme "ésotérique" d'une alternative

```
condition ? instruction_1 : instruction_2
si condition est vraie, instruction_1 est exécutée
sinon, condition fausse, instruction_2 est exécutée
```

5.7 Choix multiples par utilisation d'une expression entière : l'instruction switch

Dans **switch(sélecteur)**, sélecteur doit être une expression entière. Les différents cas possibles sont énumérés par :
case valeur :
break renvoie à la fin du switch : toute instruction est exécutée tant qu'un break n'est pas rencontré.
default : est utilisé si aucune correspondance n'est trouvée, en général traitement d'erreur...

Le programme ex3-3.c demande de rentrer un prix, nombre entier de 0 à 10. Il renvoit un commentaire selon la valeur entrée.

6 Séance 3

6.1 Numération à l'aide d'éléments binaires

Jusqu'ici, il a été question de manipulation de chaînes de caractères et de nombres (entiers), sans trop s'intéresser à leur représentation dans l'ordinateur. Celle-ci fait appel aux fameux "bits" (de l'anglais binary digit) dont on sait qu'il existent en cohortes de méga ou de giga bits à l'intérieur de l'ordinateur.

Un élément est dit binaire s'il ne possède que deux états. Par exemple un robinet fermé (l'eau ne coule pas) peut conventionnellement être représenté à l'état 0 et s'il est ouvert (l'eau coule) être représenté à l'état 1. Cela ne veut pas dire que les états intermédiaires n'existent pas, pour passer de l'état fermé à l'état ouvert mais l'on décide d'en faire abstraction.

Considérons maintenant 8 ampoules électriques alignées :
- si l'ampoule est allumée on la représente par un '**X**' et on lui associe l'état 1 (par exemple)
- si l'ampoule est éteinte on la représente par un '**O**' et on lui associe l'état 0

La valeur de chaque ampoule dépend de sa position, selon la figure suivante :

```
O  X  O  O  O  X  O  X

I  I  I  I  I  I  I  I--1 (2^0)
I  I  I  I  I  I  I-----2 (2^1)
I  I  I  I  I  I--------2x2=4 (2^2)
I  I  I  I  I-----------2x2x2=8 (2^3)
I  I  I  I--------------2x2x2x2=16 (2^4)
I  I  I-----------------2x2x2x2x2=32 (2^5)
I  I--------------------2x2x2x2x2x2=64 (2^6)
I-----------------------2x2x2x2x2x2x2=128 (2^7)
```

Dans le cas représenté ici, si on fait le compte des ampoules allumées, on obtient :

64 + 4 + 1 = 69

Si maintenant, on fait abstraction qu'il s'agit d'ampoules électriques et qu'on représente l'état binaire associé à chaque état physique allumé/éteint, on obtient :
0 1 0 0 0 1 0 1
Un octet (byte en anglais) comprend 8 éléments binaires (8 bits). Dans le cas de la figure il permet de représenter le nombre 69.

Quel est le plus grand nombre que l'on puisse représenter avec un octet (toutes les ampoules allumées) ?
1 1 1 1 1 1 1 1
128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 2^8$ - 1
D'où la règle : avec n bits on peut obtenir la valeur 2^n - 1

Avec un octet, il est possible de représenter n'importe quel nombre entier de 0 à 255.
Pour représenter des nombres plus grands, il suffit d'accoler les octets.
Ainsi avec 4 octets (4 x 8 = 32 bits) on peut aller jusqu'à 2^{32}-1 = 439 4967 295.

6.2 Nombres signés et non signés

En langage C, les nombre entiers peuvent être signés ou non signés. Pour les nombres signés, le bit le plus à gauche permet de tenir compte du signe.

La représentation se fait à l'aide de la méthode dite "complément à 2".
L'avantage de cette méthode est qu'elle comporte une représentation unique pour 0.
Voici la règle pratique : en partant de la droite vers la gauche, on ne change pas les 0 ni le 1er1 rencontrés, ensuite on change les 1 en 0 et les 0 en 1.
Exemple : changer le signe de +4 : 00000100 -> 11111100 (et bien sûr, ça marche aussi dans l'autre sens)

Pour un octet représentant des nombres signés, les valeurs vont de +127 à -128.

6.3 Les types entiers en C

Voici les types entiers signés qui existent en langage C :
char , short int, int, long int, long long int

Et voici les non signés :
unsigned char , unsigned short int, unsigned int, unsigned long int, unsigned long long int

Par commodité, sauf pour int, on omet souvent "int", ce qui donne :
char , short, int, long , long long
unsigned char , unsigned short, unsigned, unsigned long, unsigned long long

6.4 Quel format pour quel type ?

Quel est le format (nombre d'octets) de chacun de ces types ? La seule chose prévue par les normes successives (K&R, ANSI, C88, C99) c'est :
char <= short <= int <= long <= long long

Autrement dit le format dépend de la plateforme utilisée !

L'exemple ex4.c, à l'aide de l'opérateur sizeof() permet de répondre à cette question...

7 Séance 4

7.1 La notation hexadécimale

Un octet (byte) est formé de 8 éléments binaires (bits).
par exemple, le nombre : 01001101 en représentation binaire a en représentation décimale 77

Il n'est pas très commode, pour un être humain d'énoncer une suite de 0 et de 1.
Pour palier à ce problème, on divise l'octet en 2 parties de 4 bits chacun (2 quartets)
dans l'exemple précédent 0100 1101
Et on a recours a une notation dite hexadécimale (base 16) au lieu de la notation décimale (base 10).

De 0 à 9 : mêmes symboles que la notation décimale, pour un quartet :

0000 -> 0
0001 -> 1
0010 -> 2
0011 -> 3
0100 -> 4
0101 -> 5
0110 -> 6
0111 -> 7
1000 -> 8
1001 -> 9

Pour la suite (10 à 15 décimal) on utilise les symboles A, B, C, D, E, F (ou a, b, c, d, e, f), pour un quartet :

1010 -> A (ou a) soit 10 en notation décimale
1011 -> B (ou b) soit 11 en notation décimale
1100 -> C (ou c) soit 12 en notation décimale
1101 -> D (ou d) soit 13 en notation décimale
1110 -> E (ou e) soit 14 en notation décimale
1111 -> F (ou f) soit 15 en notation décimale

Ainsi l'exemple précédent (77 en notation décimale) donne, en notation hexadécimale
4 D
0100 1101

En langage C, pour indiquer qu'il s'agit d'une notation hexadécimale, le chiffres sont précédés de '0X' (ou de '0x').
0X4D ou 0x4D ou 0x4d ou 0X4d

7.2 Exemples d'utilisation

L'exemple ex5.c

. déclare une variable de type entier signé (int)
. demande d'entrer un nombre entier, notation décimale, range la valeur entrée dans cette variable.
. affiche ce nombre (la variable est accédée par son adresse) en notation hexadécimale.

La variante ex5-2.c permet de faire le travail inverse (notation hexadécimale -> notation décimale)

7.3 Nombres non entiers

D'autre types permettent de travailler sur des valeurs non entières (valeurs approchées des nombres réels). Selon la précision désirée on a le choix entre des variables de type :

float (%f)
double (%f)
long double (%lf)

On peut les manipuler sous la forme 4.156 (attention point '.' pas virgule ',') ou sous la forme "mantisse exposant" 5.21e4 qui signifie 5.21 10^4 (10 puissance 4). On peut employer e ou E.
Il s'agit de nombres signés.

Pour connaître la taille en octets de ces types, il suffit d'adapter l'exemple ex4.c ...
C'est fait dans bonus3.c...

L'exemple ex6.c permet de calculer une circonférence à partir du diamètre, entré sous forme d'un float.
L'intérêt de la définition d'une macro est de pouvoir facilement changer la valeur de PI sans avoir à aller rechercher cette valeur dans tous le fichiers où elle est utilisée.
exemple passer de 3.141 à 3.1416
L'exemple ex6-2.c n'utilise pas de macro...
Une autre solution pour définir une constante est d'utiliser le mot "const". Exemple :
const float 3.141 ;
====> Expliquer la différence entre substitution de texte et réservation mémoire...

7.4 Déclarations de variables

Tous les types rencontrés jusqu'à maintenant peuvent participer à la déclaration de variables soit sous forme "simple", comme dans :

```
unsigned int var1 , maVar, autreEnt ;
double v1, salaire, autreD ;
```

Soit sous forme "composée", déjà vu pour les tableaux de caractères, comme dans : char unNom[25] ;
Ou comme :

```
int tab1[40] ; Un tableau de 40 nombres entiers signés.
long double tLong[12] ; Un tableau de 12 valeurs non entières (de type long double).
```

7.5 Accès aux éléments d'un tableau

L'accès aux valeurs d'un tableau est réalisé grâce à un index commençant à 0, accès à la première valeur rangée dans le tableau. Un index est une valeur entière toujours positive ou nulle.
L'exemple ex6-3.c montre ce mécanisme...
L'exemple ex6-4.c donne un exemple d'utilisation d'un tableau de float ou de double.

ATTENTION : **Le compilateur ne fait aucune vérification sur la validité d'un index !**

7.6 Les opérateurs arithmétiques du langage C

En plus des opérateurs permettant de réaliser les quatre opérations bien connues :

```
+ : addition
- : soustraction
* : multiplication
/ : division
```

L'opérateur modulo % donne le reste de la division entière, ainsi : 8 % 3 = 2

Le langage C offre d'autres opérateurs un peu plus "exotiques"

```
var++ : poste incrémentation, la variable var est d'abord utilisée puis augmentée de 1
++var : pré-incrémentation, la variable var est d'abord augmentée de 1 puis utilisée
var– : poste décrémentation, la variable var est d'abord utilisée puis diminuée de 1
–var : pré-décrémentation, la variable var est d'abord diminuée de 1 puis utilisée
```

Ces opérations sont souvent utilisées pour manipuler les index de tableaux.

Forme abrégée des affectations :

```
var += 3 ; signifie var = var + 3 ;
var -= 3 ; signifie var = var - 3 ;
var *= 3 ; signifie var = var * 3 ;
var /= 3 ; signifie var = var / 3 ;
var %= 3 ; signifie var = var % 3 ;
```

L'exemple ex6-5.c utilise ces opérations. Il est conseillé de regarder le code source en même temps que le résultat de l'exécutable...

Les opérations arithmétiques (ainsi que la plupart des opérations) s'évaluent de gauche à droite. C'est le premier opérande rencontré qui fixe le type.
L'exemple ex6-6.c montre que le résultat obtenu n'est pas toujours celui souhaité.
Le forçage de type (cast) est parfois nécessaire...

==========> Distribuer un tableau récapitulatif des opérateurs avec leur priorité.

8 Séance 5

8.1 Instructions itératives (boucles)

8.1.1 boucle while

While (condition) instruction ;

Répète une instruction (ou un bloc d'instructions) tant qu'une condition est vraie.
L'exemple ex7.c utilise une itération pour donner la valeur numérique associée aux caractères A à Z , selon le codage ASCII [5].
==========> Distribuer un tableau du codage des caractères en ASCII

8.1.2 boucle do while

do instruction ; **while** (condition) ;
Ne pas oublier le ';' final !!!

Répète une instruction (ou boc d'instructions) tant que la condition est vraie. La différence réside dans le fait que l'instruction est exécutée au moins une fois, même si la condition est fausse dès le départ. ex7-2.c illustre ce cas ...

8.1.3 boucle for

for (initialisation ; condition ; actualisation) instruction ;
La différence avec une boucle while est que la condition initiale (en général initialisation d'une variable) et la mise à jour de celle-ci (actualisation) est présente dans l'instruction elle-même, en plus de la vérification de la condition de sortie.

En fait une boucle while pourrait s'écrire sous la forme :
for (; condition ;) instruction ;
Avec l'initialisation et l'actualisation laissés vides. De la même manière, il est toujours possible d'écrire une suite d'intructions, comportant une boucle while, équivalente à une boucle for...

L'exemple ex7-3.c utilise une boucle "for" pour visualiser la valeur numérique de caractères. Ne pas laisser d'espace entre les caractères. Essayer des caractères accentués...
En bonus, l'exemple ex7-4.c supporte les espaces mais introduit 2 caractères supplémentaires :
. la valeur 0 qui termine toute chaîne de caractères (invisible)
. le caractère '\n' (line feed) qui provoque un retour à la ligne (visible !) dont la valeur est 0xA (10 en décimal).

8.1.4 rupture dans une boucle

L'instruction **break** (déjà vue pour l'instruction switch) est utilisée pour sortir d'une boucle avant la fin "normale".
L'exemple ex7-5.c illustre cette possibilité.

Il existe une autre instruction de sortie "sauvage" : **continue**.
Elle ramène à la condidion de la boucle, sans exécuter les instructions qui suivent continue.
Elle est beaucoup moins utilisée que break ...

8.2 Forçage de type

Il est possible de changer le type d'une variable, **si on est sûr de ce que l'on fait !**
Cette opération (cast) est réalisée en faisant précéder la variable du nouveau type souhaité, entre parenthèses.
(nouveau type) variable
L'exemple ex7-6.c montre l'utilisation d'un "cast".
Essayer cet exemple, sans le "cast". **Faire Ctrl + c pour tuer le programme !**

5. American Standard Code for Information Interchange

8.3 Opérations au niveau bits

Certains opérateurs permettent de modifier un entier au niveau bit (bitwise operators).

8.3.1 liste de ces opérateurs

et : opérateur &
ou inclusif : opérateur |
ou exclusif : opérateur ^
négation : opérateur ~
décalage à droite : opérateur »
décalage à gauche : opérateur «

Ces opérations sont utilisées, en particulier, lors d'actions vers les périphériques.
L'exemple ex7-7.c utilise les opérateurs &, |, ».

8.3.2 Table de vérité du & (et)

0 & 0 -> 0
1 & 0 -> 0
0 & 1 -> 0
1 & 1 -> 1

8.3.3 Table de vérité du | (ou inclusif)

0 | 0 -> 0
1 | 0 -> 1
0 | 1 -> 1
1 | 1 -> 1

8.3.4 Table de vérité du ^ (ou exclusif)

0 ^ 0 -> 0
1 ^ 0 -> 1
0 ^ 1 -> 1
1 ^ 1 -> 0

8.3.5 Table de vérité du ~ (négation)

1 -> 0
0 -> 1

C'est un opérateur unaire.
exemple : var1 = ~ var2 ;

8.3.6 forme abrégée de ces opérateurs

Comme pour les opérateurs aritmétiques, lorsque l'opérande résultat (à gauche) est impliqué à droite du signe = il existe une forme abrégée.
L'exemple ex7-7.c montre les deux écritures possibles, abrégée ou non.

& -> &=
| -> |=
^ -> ^ =

~ n'a pas de forme abrégée (puisque unaire).Pour inverser tous les bit de var -> var = ~ var ;
Ne pas confondre & avec && ni | avec || (opérateurs logiques)
Remarquer qu'il n'existe pas d'opérateur logique pour le OU exclusif...

9 Séance 6

9.1 Adresses mémoire

Toute variable, quelque soit son type occupe un "certain nombre" d'octets consécutifs en mémoire vive (RAM). L'octet de départ permet d'accéder à une variable par son adresse.

```
octet 1  <-- adresse d'une variable de n octets
octet 2
octet 3
.......
octet n
```

En lange C le symbole & (ampersand) devant une variable permet d'indiquer qu'il s'agit de son adresse, et non de sa valeur.

Exemple : int var1 = 33 ;
déclaration de la variable de type int (entier signé) avec initialisation à la valeur 33. &var1 désigne l'adresse de var1, qui se trouve "quelque part" en mémoire. Par exemple la fonction scanf() utilise une variable par l'intermédiaire de son adresse.

9.2 Classes de stockage

La place d'une variable en mémoire est définie par sa classe de stockage. On distingue les classes :

. automatique, c'est à dire dans la pile (stack). Toutes les variables vues jusqu'à présent étaient de classe automatique. Elles sont dites locales puisqu'une fois sortie de la fonction qui les utilise, elles sont détruites. Ces notions seront reprises lors de l'étude des fonctions.

. statique, une zone mémoire leur est réservée. Voir l'exemple ex8.c Elles sont dites globales car elles restent accessibles (sauf restriction) depuis différentes fonctions. Elles sont réservées lors du lancement du programme selon des emplacements prévus par la compilation et le lien. Leur durée de vie est celle du programme.

. dynamique, elles prennent place dans une zone dite tas (heap). Réservation par des fonctions telles que malloc(), calloc()... Elles sont réservées / libérées (fonction free()) lors de l'exécution du programme. Elles seront vues ultérieurement.

9.3 Le modificateur const

L'accès à toute variable peut être limité à la seule lecture.
La valeur est fixée une fois pour toute à l'initialisation (par le compilateur).
const unsigned ctrMax=255 ;
const float PI = 3.1416 ;

9.4 Pointeurs

L'accès à une adresse peut se faire par l'intermédiaire d'un pointeur. Celui-ci contient une valeur qui est l'adresse de la variable.
Un pointeur possède un type qui dépend de la variable qu'il pointe.
La déclaration d'un pointeur utilise le symbole '*' (astérique) de la manière suivante :
nomDuType * nomDuPointeur ;
Exemples :

```
char * ptCaract ;
int * pEntier ;
float * ptValApproche ;
unsigned char * ptrToto ;
```

9.5 Règles de nommage

Les règles de nommage des pointeurs sont identiques à celles des variables. caractères : a à z, A à Z, 0 à 9, _

9.6 Initialisation

On affecte un pointeur (de type correspondant) à l'adresse d'une variable par :
nomPointeur = &nomvariable ;
Opération dite de référenciation.

9.7 Usage

L'accès à la valeur d'une variable dont l'adresse est contenue dans un pointeur se réalise par une opération dite de déférenciation.
valeur = * nomPointeur ;
L'exemple 8-2.c déclare des pointeurs (automatiques ou statiques) et montre leur utilisation.

===> il sera, sous doute, nécessaire de construire d'autres exercices selon les questions...

9.8 Relation entre pointeurs et tableaux

Lors d'exemples précédents, il a été vu des expressions telles que :

```
unsigned prix ;// type entier non signé
scanf("%u", &prix) ;//& signifie : adresse de la variable prix
```

Par contre, on pourrait se demander pourquoi :

```
char prenom[20] ; //prenom est un tableau de 20 caractères (char)
scanf("%s", prenom) ; //la chaîne fournie par l'utilisateur et transférée dans le tableau
```

et non :
scanf("%s", &prenom) ;
puisque la fonction scanf() attend, en 2ième argument, une adresse.
En fait, un tableau est déjà un pointeur. **Il ne faut donc pas mettre de'&' pour désigner son adresse**.
L'exemple ex8-3.c illustre l'utilisation de pointeurs sur un tableau.
L'exemple ex8-4.c montre la possibilité d'initialiser un tableau ainsi qu'un piège de l'auto-incrémentation.

9.9 Chaîne de caractères

Une chaîne de caractères (ASCII) est un tableau particulier, terminé par zéro '\0'.

L'exemple ex8-5.c effectue l'écriture d'une chaîne dans le "code". Tout se passe bien tant que l'accès se fait en lecture. La tentative d'écriture, changer le "m" de "monde" en "M" échoue (décommenter la dernière ligne, avec nouvelle compilation) en écriture : il est possible possible d'écrire dans une zone de données (data) mais pas dans une zone de code. Il n'est pas exclus que certains compilateurs refusent une telle tentative...
L'exemple ex8-6.c initialise une chaîne dans une tableau (zone des données). L'écriture devient possible.
L'exemple ex8-7.c montre une façon de faire, moins pénible pour une chaîne longue...

9.10 Tableaux à plusieurs dimensions

L'exemple ex8-8.c donne un exemple de tableau à plusieurs dimensions.

10 Séance 7

10.1 Utilité des fonctions

Jusqu'à présent tout le code des programmes exemples était contenu dans une seule entité : la fonction main().

Les fonctions permettent une découpe fonctionnelle des problèmes à traiter ainsi que la possibilité de réutiliser le même code si besoin (diminution de la taille des programmes). Intégrée dans une bibliothèque, une fonction permet une approche "boîte noire" : l'utilisateur n'a pas à se soucier de son fonctionnement de manière intime, juste connaître ce qu'elle attend (arguments) et ce qu'elle renvoie.

Il faut signaler, cependant quelques inconvénients :
- . chaque appel consomme du temps processeur : mémorisation des arguments, de l'adresse de retour, récupération de la valeur retournée par le programme appelant...
- . si la pile (stack) est mal dimensionnée, des appels consécutifs trop nombreux, en général des appels récursifs, amènent à un débordement de la pile hors de la zone prévue.

10.2 Déclaration d'une fonction

typeRetour nomFonction (typeArg1 argument_1, typeArg2 argument_2, ...) ;
C'est aussi ce qu'on appelle le **prototype** de la fonction.
S'il la fonction ne renvoie rien, le prototype prend la forme :
void nomFonction(typeArg1 argument_1, typeArg2 argument_2, ...) ;
Le type renvoyé par défaut (aucun type notifié explicitement) est int. C'est une très mauvaise pratique !

10.3 Définition d'une fonction

```
typeRetour nomFonction(type argument_1, type argument_2, ......)     <----- entête
{                                                                   <----
instruction;                                                           I
...............                                                        I
instruction qui utilise un ou des paramètre(s);                        I corps
..........                                                             I
return valeur; //valeur de typeRetour, s'il y a lieu                  I
}                                                                    <---
```

10.4 Appel d'une fonction

```
valeur = nomFonction(argument_1, argument_2, ......);
   ou
nomFonction(argument_1, argument_2, ......); //rien n'est renvoyé, ou n'est pas utilisé
   ou
nomFonction(); //pas d'argument
   ou
valeur = nomFonction();
```

Exemples Dans l'exemple ex9.c la fonction nommée calculCube retourne le cube d'un entier non signé passé en argument. Ici elle est utilisée par la fonction main().
Dans l'exemple ex9-2.c une fonction volSphr() utilise une autre fonction...

10.5 Passage d'arguments par valeur, par adresse

Passage par valeur : c'est le cas des exemples ex9.c et ex9-2.c. On passe aux fonctions la **valeur** du rayon...

Il est possible, et même quelque fois indispensable de passer l'**adresse** d'une variable plutôt que sa valeur.
Il est à noter que la valeur de retour peut être, elle aussi, l'adresse d'une variable. L'exemple ex9-3.c montre que le passage par valeur ne produit pas le résultat attendu (permutation des valeurs de varA et varB). Il faut utiliser l'exemple ex9-4.c avec passage par les adresses des 2 entiers. Remarquer la façon dont on écrit le passage par adresse...
Une autre, pas très bonne, façon de faire serait d'utiliser des variables globales (ex9-4-1.c). Mais alors la fonction permut() devient dépendante de son environnement.

> Lorsque l'on passe un tableau en tant qu'argument, il s'agit toujours d'une adresse !

Comme cela a déjà été vu, l'identifiant d'un tableau correspond à une adresse...

10.6 Classe de stockage automatique

Comme cela a été dit dans la séance 6, les variables déclarées à l'intérieur d'une fonction sont de classe de stockage automatique. C'est à dire quelles prennent place dans une zone mémoire particulière : la pile (stack).
On parle de variables locales car elle ne peuvent être accédées que lors du déroulement de la fonction.
Les arguments sont également placés sur la pile (ainsi que l'adresse de retour), avant le saut à l'adresse de la fonction. la valeur de retour est récupérée depuis la pile par le programme appelant. Après chaque utilisation, la pile est "nettoyée" pour être prête à un nouvel usage (le pointeur de pile -stack pointer- est remis à la valeur précédant l'appel de la fonction).

10.7 Arguments de la fonction main()

Jusqu'à présent la fonction main(), qui est invoquée lors du lancement d'un programme, a été utilisée sans argument. Deux arguments peuvent lui être passés :

. le premier est un entier (int) qui permet de compter le nombre d'arguments passés sur la ligne de commande.
  Sa valeur est égal au nombre d'arguments +1
  On le désigne habituellement par argc (mais ce n'est nullement obligatoire).
. le second est un tableau de chaînes de caractères.
  On le désigne habituellement par argv (mais ce n'est nullement obligatoire).

Compiler ex10.c et lancer le programme en ne lui donnant aucun argument. Constat :
le nombre d'argument donne 1. Dans ce cas la dimension du tableau est 1, la chaîne pointée par tableau[0] est le nom du programme.
Dans l'exemple ex10-2.c, 3 arguments sont attendus sur la ligne de commande...
Dans l'exemple ex10-3.c un argument de type char est attendu sur la ligne de commande.

10.8 Fonctions ou macros ?

Une macro avec paramètres, par son aspect, ressemble beaucoup à une fonction. Il s'agit cependant de quelque chose de fort différent :

. il n'y a pas de saut à une adresse (pas d'utilisation de la pile).
. le code est dupliqué autant de fois que la macro est appelée.

L'exemple ex11.c donne un exemple de la définition et de l'utilisation de la macro _arrondi(x) .
J'ai placé un '_' en tête de son identifiant mais ce n'est pas obligatoire.

Elle utilise la fonction floor() qui donne la valeur entière immédiatement inférieure.
Ainsi floor(4.856) donne 4.0
La macro _arrondi(x) a pour but de donner l'arrondi de manière "intelligente" :
6.501 -> 7
6.499 -> 6
cas limite (par choix) 6.5 -> 6
La fonction floor() se trouve dans la bibliothèque mathématique, non présente par défaut. Elle doit être invoquée explicitement sur la ligne de commande par l'option -lm :
$ gcc -Wall ex11.c -lm -o nomExe

Il n'y a pas de vérification sur le type des paramètres avec les macros.
De plus, elles sont connues pour avoir des effets indésirables (side effect, traduit parfois par effet de bord).
ex11-2.c donne un exemple d'un tel problème.

10.9 Modificateur inline

Avec le modificateur "inline", introduit depuis la norme C99, on obtient, en quelque sorte, une entité intermédiaire entre fonction et macro. :

. pas d'utilisation de la pile, le code est dupliqué.
. vérification sur les types, pas d'effet de bord.

L'exemple ex11-3.c fait la même chose que ex11-2.c mais avec une fonction en ligne (inline).
Cette fois tout se passe comme prévu.

En conclusion, n'utiliser les macros que si l'on est sûr qu'elles ont été parfaitement testées ; pour de nouveaux développements, préférer (si nécessaire) les fonctions en ligne.

11 Séance 8

11.1 Variables structurées

Une structure permet de rassembler des données non homogènes (contrairement à un tableau).
Elle se déclare en utilisant le mot réservé "struct". l'intérieur d'une structure est constituée de champs. chaque champ est typé (n'importe quel type).
Plusieurs syntaxes sont possibles...

```
struct
{
type1_t nomChamp1 ;
type2_t nomChamp2, nomChamp3;
.......
} nomStructure;
```

L'accès aux champs (lecture ou écriture) est réalisé de la façon suivante :
nomStructure.nomChamp ;
ex12.c donne un exemple de déclaration et d'utilisation.
Les structures supportent une initialisation, même incomplète, comme dans l'exemple ex12-2.c, à condition de respecter l'ordre des champs et qu'il n'y ait pas de "trou".
Cependant, la norme C99 permet d'utiliser une initialisation moins restrictive, voir ex12-3.c.
Remarquer que le champ ancienneté (anciennete) qui n'a pas été initialisé est à 0. Est-ce le cas avec tous les compilateurs ?. Ne pas trop s'y fier...

11.2 Création de types avec typedef

D'une manière générale, il est possible de déclarer un nouveau type (personnel) en plus de ceux créés par le langage C (int, float, char, ...), en utilisant le mot "typedef".

Exemple, pour rendre le code plus clair, on désire définir un type booléen :
typedef unsigned char Booleen ;
Ce nouveau type peut désormais participer aux déclarations de variables :
Booleen autorise = VRAI, finRecherche, monflag = FAUX ;
En admettant que VRAI et FAUX aient été définis auparavant...

Note : il existe essentiellement 2 écoles (mais rien n'est obligatoire).
- . faire commencer l'identifiant par une lettre majuscule (mode C++) exemple : Booleen
- . faire suivre par _t : exemple booleen_t (mode classique du C)

11.3 Création d'un type structuré, sans utiliser typedef

Il existe un moyen de déclarer un type structuré, de manière implicite (sans utiliser typedef).

```
struct TypeStruc
{
type1_t nomChamp1 ;
type2_t nomChamp2, nomChamp3;
.......
};
```

11.4 Déclaration de variables

Il est possible de déclarer des variables (structurées) avec ce type.
struct TypeStruc varStr_1, varStr_2, varStr_3 ;
Remarque : le mot réservé "struct" doit continuer d'être utilisé.
Dans ex12-4.c le type et les variables sont locales à main() (variables automatiques), c'est juste pour changer un peu...

11.5 Création d'un type structuré, en utilisant typedef

Il est possible de déclarer un type structuré en utilisant typedef, comme pour n'importe quel autre type.
L'exemple ex12-5.c est une variante de ex12-4.c et montre comment procéder.
Noter que pour la définition des variables (var_1 ...) le **mot "struct" n'est plus utilisé**. Dans cet exemple, on peut se poser la question de savoir **à quoi sert** "MonType" ?
L'exemple ex12-6.c montre que dans ce cas il peut parfaitement être omis.

Il existe pourtant des cas où c'est impossible. Par exemple si le type structuré contient un pointeur sur ce même type (cas des listes chaînées de structures).
Voici comment procéder :

```
typedef struct CeType
{
int cetEntier;
char ceCaract;
struct CeType * suivant;
struct CeType * precedent;
float unFlotant;
} MonType;
```

======> éventuellement, expliquer ce qu'est une liste chaînée.

Comme pour tout type de variable, il est possible d'avoir des tableaux de structures.
L'exemple ex12-7.c utilise un tel tableau.
Le cas de pointeurs sur une structure sera vu ultérieurement. (séance 10)

Une structure fait partie de ce qu'on appelle, en langage C, une valeur à gauche (de left value, Lvalue).
C'est à dire qui peut figurer à gauche du signe =.
En particulier, une affectation sur des structures de même type est licite.
De même, elle peut figurer en tant qu'argument d'une fonction, ainsi qu'en valeur de retour.
L'exemple ex12-10.c illustre ces possibilités.

11.6 Les unions

Une façon de partager une même zone mémoire est d'utiliser les unions. Cela ressemble un peu aux structure, mais c'est tout à fait différent. Le mot réservé est "union".

```
union
{
type_1 champ_1;
type_2 champ_2;
} varUnion;
```

Les champs se **recouvrent** mutuellement (dans une structure ils sont l'un derrière l'autre).
Dans l'exemple ex12-8.c une variable union est déclarée directement, puis une autre après une déclaration de type union.

Les caractères accentués, le "big-endian" en face d'un "little-endian" constituent deux des sept plaies de l'informatique !

11.7 Les énumérations

Dans le même genre d'idée, regroupement d'entités à l'aide d'un même identifiant, il existe les énumérations de constantes pour fixer les valeurs possibles d'une variable entière.
Le mot réservé est "enum".
Une variable déclarée avec ce type est censée ne prendre que les valeurs définies dans l'énumération, cependant

aucune vérification n'est faite.
L'exemple ex12-9.c montre comment ça marche et qu'aucune vérification n'est faite si l'on sort des valeurs prévues dans l'énumération...

Une faiblesse peut, parfois, se muer en avantage...
Ainsi, pour une déclaration de type Booleen, tout ce qui n'est pas faux est vrai !
typedef enum {FAUX, VRAI} Booleen ;
Booleen drapeau = FAUX ;

12 Séance 9

12.1 Utiliser plusieurs fichiers

Toute classification est arbitraire, il est cependant souvent utile de séparer les problèmes pour mieux les résoudre. La découpe en fichiers et en fonctions répond à ce besoin.

Chacun a un point de vue plus ou moins personnel, voire ses petites manies...
Il n'en reste pas moins vrai que le premier travail à effectuer est de mettre "noir sur blanc" ce que l'on désire exactement réaliser, les limites, les évolutions possibles...
Les **spécifications** sont d'autant plus utiles lorsque plusieurs interlocuteurs sont impliqués.

L'exemple donné dans ./bonus/portParal a pour objectif d'introduire la notion de découpe fonctionnelle et des moyens pouvant être mis en oeuvre en langage C.

NB : si une station ne comporte pas de port parallèle, prendre ..../bonus/plaque_1.0 qui est semblable quand à la découpe mais ne demande pas de matériel spécifique.
On y trouvera une ébauche de spécification (specif.txt).

12.2 Découpe formelle

Par commodité, il est préférable de séparer les fichiers sources des binaires.
C'est réalisé dans ./bonus/portParal/src et ./bonus/portParal/bin
Cela permet de travailler avec des fenêtres différentes et ainsi d'avoir un rappel facilité des commandes précédentes.
De plus la gestion du logiciel s'en trouve grandement améliorée.

Dans le répertoire contenant les binaires, il est classique d'utiliser l'utilitaire make ...
Par souci de simplicité j'ai utilisé le script compil.sh pour la compilation et la passe de lien. Il aurait été possible de fusionner ces 2 phases, sans faire apparaître explicitement les fichiers .o (fichiers objets) ...

Remarquer l'option -I de gcc pour fixer le chemin des fichiers inclus (personnels). Une alternative serait d'utiliser des " " dans le fichier lui-même. Exemple :
#include "../inc/definitions.h"
razObj.sh et raz.sh ne sont là que pour faire le ménage.
En ce qui concerne les fichiers source, j'ai pour habitude de séparer les .c d'avec les .h (headers). Ces derniers sont destinés à recevoir différentes déclarations (définitions de type, de prototypes, de macros). Le .h est traditionnel mais n'a rien d'obligatoire...
Pour ces fichiers remarquer #ifndef ... et #define..... Le but est d'éviter d'avoir à gérer des inclusions répétées.
#ifdef est une autre directive de **compilation conditionnelle**.
Dans tous les cas, #endif doit clore la partie de code impacté.

12.3 Visibilité des variables et des fonctions

La porté des variables automatiques est d'emblée limitée à la fonction qui les déclare, on parle de variables locales à la fonction.
Le mot clé "**static**" permet de limiter la visibilité (fonction ou variables statiques) au fichier qui les déclare. Il en est ainsi des fonction vidClav() et init().
Au contraire, il peut être souhaitable que des variables aient une portée "globale" c'est à dire qu'elles soient accessibles par différentes fonctions, voire des fonctions utilisées hors d'un même fichier.
Une variable (identifiant unique) ne doit être déclarée qu'une fois. C'est dans ce but que le mot réservé "**extern**" est employé. Exemple :
extern unsigned long tpPhase, tpCycles ;
Remarquer que le mot "extern" n'est pas utilisé pour les fonctions, il suffit que le prototype soit connu (par inclusion).

Remarquer également que le fichier où la fonction est définie inclus son prototype. Ce n'est pas obligatoire, mais on s'assure ainsi que l'entête de la fonction est conforme à son prototype.

> Il est recommandé, autant que faire se peut, de limiter l'usage des variables globales.

Les problèmes rencontrés peuvent être, par exemple :

. modification intempestive non voulue, surtout dans le cas de plusieurs concepteurs
. même identifiant entre une variable locale et une variable globale (ceci n'est pas signalé par le compilateur). A l'exécution, le comportement peut ne pas être celui souhaité...

Un petit garde-fou, peut être de les déclarer toutes dans un **fichier spécifique**, ici varGlob.c.

12.4 Découpe fonctionnelle

. lance.c contient la fonction main() nécessaire à l'exécution de tout programme en C. Tout le code pourrait être contenu dans cette fonction mais ce serait contraire à la conception modulaire qui facilite la compréhension et la maintenance des programmes.
. saisie.c ne s'occupe que de l'interface avec l'utilisateur pour la saisie des paramètres nécessaires à chaque exécution.
. gestPp.c se charge de l'ouverture et la fermeture du port parallèle. L'utilisateur ne désire pas forcément de se charger de ce genre de problème...
. utilPp.c prend en charge l'écriture vers l'extérieur des accès DATA du port parallèle. C'est typiquement la partie susceptible d'être modifiée par l'utilisateur.

12.5 Quelques options de gcc

-v : (verbose) donne des informations sur tout ce qui concerne la compilation.

-ON : N un nombre de 0 à 3, définit le niveau d'optimisation. -O0 -> aucune optimisation.

-g : pour que le fichier exécutable contienne toutes les infos. de debugging. (gdb)

-c : arrête après la phase de compilation. Seuls les fichiers objet (.o) sont générés.

-o : attribue le nom du fichier exécutable (sinon, par défaut, c'est a.out).

-Ichemin : pour ajouter un chemin de recherche des fichiers inclus

-Lchemin : pour ajouter un chemin de recherche des bibliothèques (libraries)

-lbibliothèque : pour ajouter une bibliothèque au lien.
exemple -lm ajoute la bibliothèque mathématique
Remarque : il ne faut pas faire figurer ni 'lib' ni .so
exemple /usr/lib/libsqlite3.so -> -L/usr/lib -lsqlite3 (et non -llibsqlite3.so)

-DMACRO : définit MACRO comme si celle-ci avait été définie par #define MACRO. La différence est que MACRO est connue de tous les fichiers source.

-S : génère un fichier assembleur source (ne génère pas d'objet .o). Voir bonus/assembleur...

-E : arrête après la phase de préprocesseur. Est utilisée pour voir le développement des macros.
Pour y voir quelque chose, s'utilise souvent avec l'utilitaire tr pour supprimer les lignes vides :
gcc -E source.c |tr -s '\n' 1> fichier

-mtune= ou **-march**= pour spécifier un type de CPU. exemple -mtune=i486

-std=c99 : si on veut bénéficier des nouvelles possibilités du C99...

Il existe une option **-static** qui est destinée à la passe de lien (rien à voir avec "static" dans un fichier source, vu ci-dessus).

Par défaut, le lien se fait avec des bibiothèques dynamiques (en .so - shared objects ? -). Les bibiothèques sont hors de l'exécutable, elles sont partagées et appelées selon les besoins des différents programmes. Avec **-static**, le lien se fait avec des bibiothèques statiques (en .a - archive -). Dans ce cas la bibiothèque est incluse dans le fichier exécutable. Il y a une différence de taille évidente si on fait la manip. dans les 2 cas.

12.6 grep : un utilitaire qui vous veut du bien !

Mais où donc a été défini xyz ?
grep 'xyz' *
Si une recherche récursive dans les sous répertoires est nécessaire : option -r (Gnu seulement)
grep -r 'xyz' *

13 Séance 10

13.1 Retour sur les pointeurs

13.1.1 Accès aux champs d'une structure avec un pointeur

Comme toute variable (automatique, statique, dynamique) une structure possède une adresse.
Soit la déclaration de variables structurées suivante :

```
 struct TypeStruct
   {
   type1 nomChamp_1;
   type2 nomChamp_2;
   .......
   };
struct TypeStruct var1, var2, ...;
```

Déclaration d'un pointeur sur un type structuré : struct TypeStruct * nomPtr ;

Initialisation d'un pointeur sur un type structuré : nomPtr = &var2 ;
Il est, bien sûr, possible d'opérer en une seule étape :
struct TypeStruct * nomPtr = &var2 ;

Atteindre un champ : Pour atteindre un champ avec ce pointeur, on pourrait écrire (déférentiation séance 6) :
* (nomPtr).nomChamp_2 ;
Attention les **parenthèses** sont **obligatoires** car l'opérateur '.' est plus prioritaire que '*'.

C'est pourquoi il existe une écriture plus synthétique (sans * ni parenthèse) :
nomPtr->nomChamp_2 ;
L'exemple ex13.c utilise cette notation.

Dans les fonctions telles que printf() ou scanf() qui font intervenir le format, il ne faut pas perdre de vue le type des arguments attendus, en particulier lorsqu'il s'agit de champs de structures. ex13-2.c montre un aperçu de ce genre de problèmes...

13.1.2 Pointeurs de fonctions

Une fonction possède une adresse, celle-ci est donc susceptible d'être contenue dans un pointeur (de type adéquat).

Soit la déclaration de prototype d'une fonction :
Type_r nomFonction(Type_1 arg1, Type_2 arg2, ...) ;

Déclaration d'un pointeur pour un tel prototype :
Type_r **(* nomPointeur)**(Type_1 , Type_2 , ...) ;

La fonction nomFonction ayant été définie par ailleurs, deux écritures sont possibles pour la définition (affectation) du pointeur nomPointeur :
nomPointeur = &nomFonction ;
nomPointeur = nomFonction ;

Appel d'une fonction par un pointeur, deux écritures sont possibles :
valeur = **(* nomPointeur)** (arg1, arg2, ...) ;
ou
valeur = **nomPointeur** (arg1, arg2, ...) ;
valeur est de type Type_r (valeur peut être omis, si non utilisée dans le programme...)

L'exemple ex14.c présente un cas (simpliste) de déclaration, de définition, d'utilisation d'un pointeur de fonction.

13.1.3 Tableaux de pointeurs de fonctions

Comme pour tout ce qui concerne les tableaux, les données doivent être homogènes, c'est à dire des pointeurs de même type au niveau de la déclaration.

Type_r **(* nomTableau [TAILLE])** (Type_1 arg1, Type_2 arg2, ...) ;
Type_r est le type de retour des fonctions dont le pointeur est contenu dans le tableau de taille TAILLE.
Type_1 arg1, Type_2 arg2, ... sont les arguments de ces fonctions.

L'exemple ex14-2.c présente l'utilisation d'un tableau de pointeurs de fonctions. Chaque fonction est dédiée à une opération particulière (addition, multiplication, division, modulo). Le tableau permet, grâce à l'index, de réaliser l'opération souhaitée.
L'exemple ex14-3.c est une variante qui utilise une **déclaration de type** pour les pointeurs de fonctions (qui retournent un int et dont les arguments sont deux int).

13.1.4 Fonctions qui retournent un pointeur de fonction

Ces fonctions retournent une adresse (le contenu d'un pointeur). Il est cependant nécessaire de faire figurer le type de la fonction "pointée" ; c'est à dire son type de retour ainsi que le type de ses arguments (peuvent être remplacés par "void" s'il y a lieu).

Type_r(* nomFonction(Type_1 arg1, Type_2 arg2, ...))(TypePoint_1 argP_1, TypePoint_1 argP_1,...) ;

- Type_r : type de retour de la fonction pointée
- Type_1 arg1, Type_2 arg2, ... sont les arguments de la fonction nomFonction
- TypePoint_1 argP_1, TypePoint_1 argP_1,... sont les arguments de la fonction pointée.

L'exemple ex14-4.c est un peu "tiré par les cheveux". Il a simplement pour but d'illustrer le cas où différents types sont utilisés...
L'exemple ex14-5.c, avec l'utilisation d'une **définition de type** (typedef) sur la fonction pointée, simplifie l'écriture du prototype de la fonction appelante.

13.1.5 Pointeur de fonction utilisé en tant qu'argument d'une fonction

Le prototype de la fonction utilisée en tant qu'argument doit figurer dans la fonction "utilisatrice" afin qu'il soit connu lors de l'appel.
L'exemple ex14-6.c est certes un exercice de style ! Il met en place prototypes et appels...
Une variante avec définition de type utilisant typedef est réalisée dans ex14-7.c .

typedef : encore un ami qui vous veut du bien

13.2 Automates

Les pointeurs (fonctions, tableaux) sont largement utilisés dans les programmes basés sur des états, événements, branchement, également connus sous le nom d'automates.
bonus4.c donne un exemple d'automate avec tableau à une dimension. Le nombre d'événements significatifs par état est en principe limité (programme interactif).

bonus5.c donne un exemple d'automate avec tableau à deux dimensions.
Tous les événements sont attendus par chaque état. C'est le cas d'événements imprédictibles (par exemples états de capteurs).

14 Séance 11

14.1 Opérations sur les fichiers

14.1.1 Création d'un flux destiné à accéder à un fichier régulier (mémoire de masse, disque etc contenant des données).

Exemple :
FILE * monFlux ;
monFlux est un pointeur sur une structure de type FILE.
La structure pointée par monFlux est créée par le système, elle contient tous les champs nécessaires à la manipulation d'un fichier.

14.1.2 Ouverture d'un fichier

la fonction fopen() permet d'associer ce flux à un fichier.
Exemple :
monFlux = **fopen**("./monFichier","ab") ;
Si le retour (monFlux) de fopen() vaut NULL, l'opération d'ouverture a echoué.
Le premier argument de fopen() est une chaîne de caractères contenant le nom du complet du fichiers (chemin + nom).
Le deuxième argument de fopen() précise le mode d'ouverture. Cela peut être :

"w" : écriture (write) seule. Celle-ci s'effectue à partir du début du fichier. Si le fichier existe déjà, il est **écrasé**, sinon il est **créé**.
"r" : lecture (read) seule. Le fichier **doit exister** (sinon échec).
"a" : **ajout** (append) écriture en fin de fichier, s'il existe, sinon il est créé.
"r+" indique que la **lecture et l'écriture** sont possibles. Idem pour a+, w+.
Associé à ces possibilités "b" indique que les "caractères" de valeur 0x0A et 0x0D (line feed, carriage return) doivent être traités comme les autres.

14.1.3 Suppression de l'association d'un flux (monFlux) à un fichier (monFichier)

int resultat = **fclose**(monFlux) ;
Si resultat égale 0, l'opération a réussi.
D'éventuels tampons provisoires sont vidés dans le fichier, qui est désormais fermé (il existe en mémoire de masse mais n'est plus associé à un flux). Le flux (monFlux) garde la valeur précédente.
La tentative de supprimer une association déjà supprimée provoque une erreur système (voir exemple ex15.c).

14.1.4 Écriture dans un fichier (ouvert)

Il existe plusieurs solutions pour écrire dans un fichier.

En générale la fonction **fwrite**() permet d'écrire des **blocs de données**, par exemple des nombres. Mais rien n'empêche de l'utiliser pour écrire une chaîne.
nbBlocsEcrits = fwrite(adrBlocs, tailleUnBloc, nbBlocs, flux) ;
l'opérateur sizeof est souvent utilisé pour connaître la taille d'un bloc.
L'exemple ex15.c réalise l'écriture d'une chaîne dans un fichier.
L'exemple termine par une double suppression d'association qui provoque une **erreur système**. Mettre cette ligne en commentaire si on veut éviter le problème.

Une autre technique consiste à utiliser des fonctions permettant le **formatage**.
ex15-1.c utilise **fputs()** pour écrire une chaîne dans un fichier.
ex15-2.c utilise **fprintf()** pour l'écritue formatée, ici un entier.
N'importe quel éditeur peut être utilisé pour vérifier ce que l'on vient d'écrire...

14.1.5 Lecture dans un fichier (ouvert)

Pour la lecture il existe le pendant des fonctions d'écriture.
ex15-3.c utilise **fread**() pour la lecture d'un fichier "binaire".
Noter l'utilisation de **freopen**() pour la fermeture/réouverture...

La fonction **fscanf**() permet une lecture formatée à partir d'un flux.
ex15-4.c donne un exemple de son utilisation ainsi que l'utilité des fonctions **fflush**() pour forcer l'écriture sur le fichier disque et de **rewind**() pour revenir en début de fichier.

La fonction **fgets**() associée au flux **stdin** permet de limiter les entrées clavier inappropriées, voire malveillantes.
ex15-5.c réalise un tel filtrage entre la 1iere et la 2ieme entrée.
Essayer,par exemple, 45kklfdldgldldglrgotretoieziezireiikfkfkdfllezepeppezpr
45 est reconnu, grâce à la boucle avec **getchar**() on vide le clavier si le caractère de saut de ligne ('\n') ne se trouve pas dans le tampon tabEnt[].

14.1.6 Connaître la position courante

ex15-6.c utilise la fonction **ftell**() pour connaître la position courante dans le flux.
La fonction **feof**() retourne vrai (!=0) si la fin de fichier est atteinte, c'est à dire que la dernière opération de lecture a échoué.

ex15-7.c donne un exemple d'utilisation de **fgetpos**() pour mémoriser la position courante, pour la retrouver ensuite avec **fsetpos**().

14.1.7 Modifier la position courante

En plus des fonctions vues précédemment, il existe la fonction **fseek**().
ex15-8.c montre son utilisation, le déplacement peut être positif ou négatif (si cela a un sens).
Les 3 constantes prédéfinies indiquent le point de référence.

- SEEK_END : fin de fichier
- SEEK_SET : début de fichier
- SEEK_CUR : position courant

14.1.8 Erreur sur un flux

Remarque générale :
lorsqu'une erreur se produit sur un flux Pour pouvoir continuer à l'utiliser il faut supprimer l'indication d'erreur avec :
clearerr(flux) ;

14.1.9 Unix et les fichiers

Attention à ne pas confondre les fonctions précédentes avec open(), read, write(), close(), qui sont des appels systèmes pour agir sur des fichiers ne correspondant pas toujours à des fichiers réguliers mais plutôt à des éléments physiques tels que clavier, ports, terminal (fichiers /devices/....).
open() renvoie un descripteur de fichier (file descriptor), en fait un entier, qui permet d'accéder à un fichier.
Exemple, accès au fichier /dev/ttyS0 qui correspond au 1e port série :
fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY) ;
Noter que stdin, stdout, stderror correspondant respectivement aux descripteurs de fichiers 0, 1, 2, sont des flux (ie des pointeurs de structures FILE) qui sont créés par le système (donc n'ont pas à être créés par l'utilisateur).
L'ouverture d'un fichier par un flux exerce une influence sur la liste des descripteurs de fichiers, bonus6.c met cela en évidence...
Rien n'empêche d'accéder à un fichier régulier avec des fonctions de la famille open()... C'est ce qui est fait dans bonus7.c.
Cependant l'accès à un fichier régulier (mémoire de masse) et optimisé, par une gestion de tampons (buffers), avec un flux utilisant les fonctions de la famille fopen() etc ...

14.2 Variables de classe de stockage dynamique

14.2.1 Accès à ces variables

L'accès se fait toujours par l'intermédiaire d'un pointeur dans une zone mémoire appelée tas (heap).
Avantages :

. pouvoir allouer et libérer de la mémoire en cours d'exécution du programme.
. seul la partie qui "possède" le pointeur a accès aux variables dynamiques.

Inconvénient :
à force d'allouer et de libérer, on peut se retrouver avec un tas "fragmenté".
Le système d'exploitation gère, en principe, cette situation de manière optimale...

14.2.2 Allouer de la mémoire

La fonction **malloc**(), alloue le nombre d'octets demandé.
ex-16.c écrit puis relit la mémoire qui est ensuite libérée par **free**().
A noter la "bonne précaution" après un free().
On peut allouer une variable de n'importe quel type, y compris une structure, avec malloc().
ex16-2.c montre l'allocation d'une table de structures et son utilisation.

Fonction **calloc**() : en fait il y a peu de différence avec malloc().
ex16-3.c transpose ex16-2.c avec calloc().

La fonction **realloc**() permet de redimensionner une allocation de mémoire précédente.
Il y a cependant certaines précaution à prendre :

. la nouvelle allocation doit inclure la taille de l'allocation précédente.
. il est préférable d'utilisé un pointeur intermédiaire, sinon en cas de problème sur la nouvelle allocation, il est impossible de libérer la mémoire déjà allouée (problème de fuite de blocs).

ex16-4.c redimensionne un tableau de 8 entiers en tableau de 12 entiers.

ex16-5.c a pour but de tester la mémoire disponible (tas), attention cela dure "un certain temps" ! Sur ma station : compteur mémoire = 2281697269
Attention, il a été constaté une élévation importante de la température avec certains processeurs en faisant cet exercice ! Donc : Ctrl C si nécesaire...

15 Séance 12

15.1 Le debugger gdb

Tout d'abord, il faut charger le paquet éponyme gdb.
Voici un petit exemple, aller dans ..../bonus/plaque_1.0/bin
Exécuter compDebug.sh, l'option -g permet d'avoir les infos. de debugging.
Lancer GDB pour l'exécutable obtenu :
gdb plaque_1.0-elf
On se retrouve dans le debugger avec le prompt gdb (>), qui attend une commande.

Dans un autre terminal ouvrir, par exemple, .../bonus/plaque_1.0/src/dialHm.c (pour visualiser le source).

Dans le debugger, entrer un point d'arrêt, par exemple, à la ligne 100
b dialHm.c :100
Puis r (run) pour lancer l'exécution.
r
Le programme s'arrête puisqu'il attend des entrées utilisateur. Entrer "s" puis un nombre, le programme s'arrête à la ligne 100 . Taper p passTrou.x -> valeur quelconque
Faire n (next)
Taper p passTrou.x -> on retrouve la valeur entrée précédemment.
Taper s (step) -> on passe à l'instruction suivante....
Pour quitter le debugger, entrer q (quit).
Puis y pour confirmer.

Ceci n'est qu'un tout petit exemple des possiblités de GDB.
J'ai mis dans ..../bonus une petite documentation qui en dit un peu plus, sans pour autant prétendre faire le tour de la question.
doc_GDB.txt est une doc. personnelle qui est fournie "as it is"...

15.2 L'utilitaire make

Charger, si nécessaire, le paquet éponyme make.
Par défaut il utilise un fichier de commande nommé Makefile. L'option -f permet de forcer un fichier avec un autre nom.
make ;
ou
make -f monMake ;

Ce fichier de commandes, principalement, contient des groupes de 3 entités :
- la cible (target)
- les dépendances (dependencies)
- la règle (rule) associée. Elle indique les actions. La ligne doit impérativement **commencer par une tabulation**.

```
cible : dépendances
    règle
```

La ligne de commande peut, éventuellement, indiquer un cible :
make -f monMake maCible ;
Si aucune cible n'est indiquée sur la ligne de commande, c'est la 1ere cible qui est exécutée.
Voici un exemple d'utilisation avec quelques commentaires

```
# makefile pour compiler des fichiers sources (.cpp)et obtenir un exécutable
# les fichiers objets sont générés (et conservés) dans ce répertoire.

.PHONY :  raz
NOMEXEC = servFic_2.0

CC=g++
LINK = g++

REPSRC = ../src
REPINC = ../inc
FLAGS= -Wall -g -O0 -I $(REPINC) -c
LISTOBJ = syst1.o seq.o main.o portSerie.o  \
          recN0.o servN2.o recN1.o emN1.o emN0.o

vpath %.cpp $(REPSRC)

%.o: %.cpp
        $(CC) $(FLAGS) $^ -o $@
$(NOMEXEC) : $(LISTOBJ)
        $(LINK) $(LISTOBJ) -o $@
raz:
        rm  -f $(NOMEXEC) *.o
```

.PHONY : raz indique simplement que cette cible n'est pas un nom de fichier
vpath %.cpp indique où chercher les fichiers d'extension .cpp
%.o : %.cpp indique que toute cible .o dépend de fichiers .cpp
noter la tabulation avant la règle : compiler tous les fichiers .cpp pour obtenir les .o indiqués dans LISTOBJ
les fichiers .o auront le même nom ($@) que leur .cpp respectifs
La seconde cible dépend de LISTOBJ
La rège indique qu'il faut faire le lien des fichier .o pour générer un exécutable de même nom que la cible (-o $@)
La 3eme cible ne dépend de rien, sa règle fait le ménage...
On l'invoque par :
make raz (si le nom de ce fichier est Makefile dans le répertoire courant)

15.3 Graphisme et langage C

EZ-Draw est un logiciel libre qui permet de faire des fenêtres ...(mode graphique) à partir du langage C.
Il peut être facilement récupéré sur le WEB.
Personnellement j'utilise, sous Debian, un ligne de commande du genre :
gcc -Wall -o monExe -I../inc ../src/monFichier.c ../src/ez-draw.c -L/usr/lib -l X11 -lXext
dans ../src on trouve : ez-draw.c et monFichier.c
et dans ../inc on trouve : ez-draw.h

Troisième partie
Documents disponibles

15.4 Code source

Les fichiers sources des exemples sont contenus dans deux répertoires, dans l'archive codeSource.tgz
C'est une division un peu arbitraire.
La présentation sous forme de fichiers texte (.c) améliore la lisibilité, grâce à la colorisation syntaxique.
L'archive peut être obtenue sur demande [6]. Elle devrait être aussi disponible sur le site GEBULL [7].

15.4.1 Répertoire exemples

Illustre la syntaxe des instructions employées lors des séances.

15.4.2 Répertoire bonus

Contient quelques exemples complémentaires ainsi que des informations n'ayant pas un rapport direct avec l'initiation à la programmation.

- Le sous répertoire **plaque_1.0** peut donner quelques idées sur la façon d'organiser un projet...
- Le sous répertoire **portParal** n'a d'intérêt que pour une station équipée d'un port parallèle.
- **EZ-Draw-1.0** permet la création de fenêtre à partir du langage C, merci à Edouard Thiel. [8]
- sousrépertoire **assembleur**, si on veut voir à quoi cela ressemble dans le cas des processeurs 'i86.

15.5 Debugger GDB

Une documentation personnelle sur l'utilisation du debugger gdb est fournie "telle qu'elle", c'est à dire sous forme d'un fichier texte... Elle se trouve dans le répertoire bonus.

6. gargamel79@orange.fr
7. www.gebull.org
8. Edouard.Thiel@lif.univ-mrs.fr