Logiciel de base
Première année par alternance

Responsable : Christophe Rippert
Christophe.Rippert@Grenoble-INP.fr

Introduction au langage C99

Introduction

Dans cette partie, on étudie les spécificités du dialecte C99 ainsi que des aspects bas-niveaux du langage qui seront utiles lorsqu'on interfacera du C avec de l'assembleur. On part du principe que vous avez tous déjà programmé en C classique (C ANSI norme ISO89).

Syntaxe de base

Premier programme en C99

On commence par donner un exemple d'un petit programme en C99 :

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

#define PREM_LETTRE 'a'
#define DERN_LETTRE 'z'

// renvoie la position dans l'alphabet de la lettre passée en paramètre
unsigned pos_lettre(char c)
{
    return c - PREM_LETTRE + 1;
}

/*
 * renvoie la lettre suivant dans l'alphabet celle passée en paramètre
 */
char lettre_suiv(char c)
{
    return c + 1;
}

int main(void)
{
    printf("Entrer la lettre par laquelle vous souhaitez commencer : ");
    fflush(stdout);
    char c;
    scanf("%c", &c);
    assert((c >= PREM_LETTRE) && (c <= DERN_LETTRE));
    for (char l = c; l <= DERN_LETTRE; l = lettre_suiv(l)) {
        printf("[%02u] : %c\n", pos_lettre(l), l);
    }
    return 0;
}
```

Ce programme lit une lettre au clavier et affiche toutes les lettres de l'alphabet qui suivent la lettre donnée, avec leur position.

On remarquera :

- qu'on inclut au début du fichier les en-têtes des modules contenant les fonctions de la bibliothèque C que l'on va utiliser : `stdio.h` contient notamment les fonctions d'affichage et de lecture de données, et `assert.h` permet d'écrire des assertions dans le code ;
- qu'on peut définir des constantes grace à pré-processeur C en utilisant la macro `#define` : la valeur de la macro est simplement copiée-collée dans le code losqu'on l'utilise, sans interprétation du compilateur (qui intervient après l'appel au pré-processeur) ;
- qu'il existe deux types de commentaires en C99 : les commentaires classiques du C (`/* ... */`) et les commentaires commençant par `//` et terminés par la fin de la ligne ;
- qu'on défini les fonctions que l'on va utiliser dans le programme principal avant celui-ci, afin de les rendre visibles lors de leur utilisation (en C99, il est possible de définir des fonctions imbriquées dans d'autres fonctions, ce qu'on n'utilisera pas car ça complique en général la compréhension du code) ;
- que la fonction `main` est le point d'entrée du programme (c'est à dire le programme principal) : cette fonction renvoie toujours un `int` qui indique au système si le programme s'est bien exécuté (valeur renvoyée : 0) ou a rencontré une erreur (valeur renvoyée différente de 0), elle peut aussi prendre des paramètres mais dans cet exemple on ne les utilisera pas (d'où le `void` entre parenthèses) ;
- que lorsqu'on fait un affichage avec `printf`, certains systèmes mettent en mémoire tampon la chaine à afficher et ne l'affiche vraiment que lorsqu'elle se termine par un retour à la ligne (caractère `'\n'`) : on peut utiliser la fonction `fflush(stdout)` définie dans `stdio.h` pour forcer l'affichage immédiat sur la sortie standard ;
- que la macro `assert` permet de vérifier des conditions booléennes et stoppe immédiatement le programme en affichant un message d'erreur si la condition n'est pas validée ;
- qu'en C99, on peut déclarer des variables n'importe-où dans le code, y compris dans des boucles (la variable `l` est interne à la boucle `for` et pas visible à l'extérieure de celle-ci) ;
- qu'on peut contrôler l'affichage réalisé par `printf` grace à des directives de formatage comme `%02u` qui force l'affichage de l'entier naturel passé en paramètre sur 2 chiffres, en complétant avec des zéros s'il est plus petit que 10 (liste complète des directives de formatage dans `man printf`).

Ce programme peut-être compilé grace à la ligne de commande :

```
gcc -Wall -Wextra -Werror -m64 -g -std=gnu99 -o mon_prog mon_prog.c
```

qui crée le binaire `mon_prog` à partir du fichier source `mon_prog.c`. Les paramètres passés à GCC ont pour signification :

- `-Wall -Wextra -Werror` : activent le plus d'avertissement possibles, et demande au compilateur de considérer ces avertissements comme des erreurs : le C étant un langage très permissif, il est important de prendre en compte tous les messages affichés par le compilateur pour éviter de se retrouver avec des erreurs à l'exécution difficiles à corriger ;
- `-m64` : active le mode 64 bits, qu'on utilisera en général dans ce cours (on peut aussi utiliser `-m32` pour forcer la génération de code 32 bits) ;
- `-g` : active l'inclusion des informations de mise au point dans le binaire produit, ce qui est indispensable pour utiliser un outil comme GDB par exemple ;
- `-std=gnu99` : active le dialecte C99 qu'on va utiliser dans ce cours, par défaut la plupart des compilateurs implantent le dialecte C89 qui est plus limité.

Note : par défaut, votre environnement est vraisemblablement en français. Si vous préférez avoir les messages d'erreurs de gcc en anglais (souvent beaucoup plus compréhensibles !), ajoutez les deux lignes suivantes dans votre `.bashrc` :

```
export LANG="en_US.UTF-8"
export LC_ALL="en_US.UTF-8"
```

Types de données

Types entiers exacts

Le langage C définit des types entiers classiques comme `int`, `long`, `short`, etc. L'inconvénient de ces types est que leur taille peut dépendre de l'architecture sur laquelle on compile le programme. Par exemple,

l'expression `sizeof(long)` renvoie 4 sur une architecture 32 bits et 8 sur une architecture 64 bits (on peut tester en utilisant les options `-m32` et `-m64` de GCC). Cela rend les programmes peu portables et peut causer des erreurs difficiles à localiser.
Pour éviter ce problème, on utilisera des types entiers de tailles fixes définis dans `inttypes.h` et présentés ci-dessous :

Types signés	Types non-signés	Taille en octets
`int64_t`	`uint64_t`	8
`int32_t`	`uint32_t`	4
`int16_t`	`uint16_t`	2
`int8_t`	`uint8_t`	1

Les types signés sont utilisés pour représenter des entiers relatifs (codés en complément à 2) et les types non-signés servent pour les entiers naturels.
Une conséquence de l'utilisation de ces types est qu'on doit légèrement adapter la chaine de format passée en paramètre des fonctions du type de `printf` lorsqu'on veut afficher des valeurs entières. On n'utilisera plus :

```
printf("entiers signe = %d, non-signe = %u\n", un_int, un_unsigned);
```

par exemple, mais des chaines de format comme :

```
printf("entiers signe = %" PRId32 ", non-signe = %" PRIu32 "\n", un_int32_t,
    un_uint32_t);
```

Il suffit bien sûr d'adapter la taille de la constante `PRIx##` à la taille des variables manipulées (8, 16, 32 ou 64). Les constantes `PRIx##` sont toutes définies dans `inttypes.h`. Il existe aussi des constantes `SCNx##` utilisables avec les fonctions comme `scanf` permettant de lire des valeurs.

Booléens

En C classique, les booléens sont représentés par des entiers (0 représente la valeur fausse et n'importe-quelle valeur différente de 0 représente la valeur vraie).
En C99, il existe un type booléen à part entière, définit dans le fichier `stdbool.h` :

- `bool` est le type des variables booléennes ;
- `true` est la constante représentant la valeur vraie ;
- `false` est la constante représentant la valeur fausse.

Attention, la norme C99 ne précise pas la taille exacte d'un booléen, qui dépend du compilateur (mais en général, `sizeof(bool)` renverra 1 et la valeur du booléen sera codé dans le bit de poids faible de l'octet).
On privilégiera l'utilisation de ce type booléen qui améliore la lisibilité du code produit.

Caractères

Les caractères sont matérialisés par le type `char` du C classique, qui permet de représenter tous les caractères de la table ASCII (128 caractères listés dans `man ascii`). Les caractères sont toujours codés sur 1 octet et peuvent être utilisés comme des entiers (en pratique, un `char` a pour valeur le code ASCII du caractère représenté, c'est à dire un entier entre 0 et 127 inclus).

Tableaux

Un tableau est une structure de données de taille fixe correspondant à un ensemble de valeurs de même type stockées consécutivement en mémoire.
En C classique, la taille d'un tableau doit être connue statiquement à la compilation. En C99, on peut déclarer des tableaux dont la taille est une variable (ce qui ne veut pas dire que la taille du tableau peut changer pendant l'exécution du programme !).
Par exemple, on peut écrire :

```
printf("Entrez la taille du tableau :\n");
uint16_t taille;
scanf("%" SCNu16, &taille);
int32_t tab[taille];
for (uint16_t i = 0; i < taille; i++) {
    tab[i] = 5;
}
```

Attention : la syntaxe Java `int[] tab` n'est pas correcte en C, les crochets doivent impérativement suivre le nom du tableau.

Chaines de caractères

Les chaines de caractères sont des tableaux de caractères terminés par le caractère spécial `'\0'` (c'est à dire le caractère dont le code ASCII vaut 0, à ne pas confondre avec le caractère `'0'` représentant le chiffre 0 et dont le code ASCII vaut 48).
On rappelle qu'il existe de nombreuses fonctions pour manipuler des chaines dans la bibliothèque C standard (en-tête `string.h`), il n'est donc pas utile de ré-inventer la roue lorsque vous avez besoin d'effectuer des opérations de base !

Tableaux à plusieurs dimensions

On utilise parfois des tableaux à plus d'une dimension. En pratique, un tableau déclaré par exemple par `char tab[3][5]` peut être vu comme un tableau de 3 éléments, dont chaque élément est un tableau de 5 caractères.
Il est important de comprendre qu'en mémoire, les tableaux multi-dimensionnels sont applatis puisque la mémoire est un tableau à une dimension, comme illustré par le bout de code ci-dessous :

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

int main(void)
{
    const uint8_t t1 = 3;
    const uint8_t t2 = 5;
    char tab[t1][t2];
    for (uint8_t i = 0; i < t1; i++) {
        for (uint8_t j = 0; j < t2; j++) {
            tab[i][j] = 'a' + i;
        }
    }
    char tab2[t1 * t2];
    memcpy(tab2, tab, t1 * t2);
    for (uint8_t i = 0; i < t1 * t2; i++) {
        printf("%c ", tab2[i]);
    }
    puts("");
    return 0;
}
```

L'exécution de ce programme donne l'affichage : `a a a a a b b b b b c c c c c`.

Equivalence tableau / pointeur

Un tableau est en pratique un pointeur vers la première case du tableau, on peut donc écrire de façon équivalente :

```
int32_t tab[10];
*(tab + 3) = 7;
printf("%" PRId32 "\n", tab[3]);
```

Il est important de comprendre l'arithmétique des pointeurs en C : dans l'exemple ci-dessus, on accède à la case d'indice 3 (c'est à dire la quatrième case vu que les indices commencent toujours à partir de 0) du tableau `tab` en ajoutant 3 au pointeur `tab`. En pratique, si A est l'adresse de `tab`, alors l'adresse de la case d'indice 3 est $A + 3 \times 4$, c'est à dire $A + 3 \times$ `sizeof(int32_t)` puisque chaque case du tableau fait 4 octets de large.
On rappelle aussi que les tableaux sont toujours passés par pointeur à des fonctions C : cela signifie que si on passe un tableau `int32_t tab[1000000]` à la fonction `void fct(int32_t tab[])`, on passe bien sûr une copie du pointeur vers le premier élément du tableau, et pas les 1000000 éléments de celui-ci (on peut d'ailleurs écrire de façon équivalente le prototype de `fct` comme suis : `void fct(int32_t *tab)`). Cela implique que l'opérateur `sizeof` a un comportement différent selon qu'on manipule un tableau de taille connue ou si on manipule le pointeur correspondant.
Par exemple :

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

int32_t tab_glob[10];

void fct(int32_t tab_param[])
{
    int32_t tab_loc[10];
    int32_t taille;
    printf("Entrez la taille du tableau : "); fflush(stdout);
    scanf("%" SCNd32, &taille);
    int32_t tab_var[taille];
    printf("tab_glob : %" PRIu64 ", tab_loc : %" PRIu64 ", tab_var : %" PRIu64 ",
        tab_param : %" PRIu64 "\n", sizeof(tab_glob), sizeof(tab_loc),
        sizeof(tab_var), sizeof(tab_param));
}

int main(void)
{
    int32_t tab[10];
    fct(tab);
    return 0;
}
```

affichera :

```
Entrez la taille du tableau : 10
tab_glob : 40, tab_loc : 40, tab_var : 40, tab_param : 8
```

car :
- le compilateur sait calculer les tailles de `tab_loc` et `tab_glob` qui sont connues statiquement ;
- il sait aussi générer le code lui permettant de calculer la taille de `tab_var` (qui ne sera connue qu'à l'exécution) ;
- par contre, l'exécution affiche 8 comme taille de `tab_param` car les tableaux sont passés par pointeur aux fonctions en C, donc `sizeof` renvoie la taille d'un pointeur.

Structures

Une structure représente un ensemble de données de types potentiellement différents stockés consécutivement en mémoire.

Organisation mémoire

Les différentes normes du C imposent peu de contraintes sur la façon dont les structures sont implantées en mémoire : cela signifie que différents compilateurs pourront organiser les structures de façons différentes (d'ailleurs le même compilateur peut aussi organiser différemment la mémoire en fonction de l'architecture sous-jacente).
Si on prend par exemple la structure suivante :

```
struct une_structure_t {
    uint32_t entier32;
    bool booleen;
    int16_t entier16;
    char caractere;
};
```

On peut se baser sur les contraintes suivantes, implantées par le compilateur GCC :

- l'adresse du premier champ de la structure est toujours égale à l'adresse de la structure elle-même (dit autrement, le champs `entier32` est forcément localisé au début de la structure) ;
- les champs sont placées en mémoire dans le même ordre que leur déclaration (donc `booleen` est forcément à une adresse plus grande que celle de `entier32` et à une adresse plus petite que celle de `entier16`, et ainsi de suite) ;
- les champs sont alignés sur des adresses multiples de la taille du type de données du champs (donc ici : `entier32` est aligné sur une adresse multiple de 4, `booléen` n'est pas aligné (la taille d'un booléen étant 1), `entier16` est aligné sur une adresse multiple de 2 et `caractere` n'est pas aligné).

L'alignement mémoire est une technique permettant d'optimiser les accès mémoire lorsqu'on veut lire ou écrire les champs de la structure. En pratique, cela consiste à rajouter des octets de remplissage (*padding* en anglais) entre deux champs, pour décaler le champs suivant vers une adresse multiple de la taille de son type.
Dans l'exemple ci-dessus (et sur ma machine), le bout de programme ci-dessous :

```
struct une_structure_t ma_struct;
printf("Adresse de la structure : %" PRIu64  "\n", (uint64_t)&ma_struct);
printf("Adresse de entier32     : %" PRIu64  "\n", (uint64_t)&(ma_struct.entier32));
printf("Adresse de booleen      : %" PRIu64  "\n", (uint64_t)&(ma_struct.booleen));
printf("Adresse de entier16     : %" PRIu64  "\n", (uint64_t)&(ma_struct.entier16));
printf("Adresse de caractere    : %" PRIu64  "\n", (uint64_t)&(ma_struct.caractere));
```

provoque l'affichage suivant :

```
Adresse de la structure : 140734733148576
Adresse de entier32     : 140734733148576
Adresse de booleen      : 140734733148580
Adresse de entier16     : 140734733148582
Adresse de caractere    : 140734733148584
```

On voit bien que `entier16` est aligné sur une adresse multiple de 2 et n'est pas placé immédiatement après `booleen`.
Bien sûr, le remplissage dépend de l'ordre des champs, si on déplace certains champs de la structure comme-suis :

```
struct une_structure_t {
    bool booleen;
    uint32_t entier32;
    char caractere;
    int16_t entier16;
};
```

On obtient l'affichage suivant :

```
Adresse de la structure : 140736163233904
Adresse de booleen      : 140736163233904
Adresse de entier32     : 140736163233908
Adresse de caractere    : 140736163233912
Adresse de entier16     : 140736163233914
```

Cette fois-ci, ce sont `entier32` et `entier16` qui sont décalés pour être alignés.
L'opérateur `sizeof` peut être appliqué à un type structure pour afficher la taille mémoire exacte utilisée pour représenter la structure en mémoire (en comptant les octets de remplissage).

Initialisation de structures

On rappelle qu'on peut initialiser les champs d'une structure directement lors de la déclaration comme suis :

```
struct une_structure_t {
    uint32_t entier32;
    bool booleen;
    int16_t entier16;
    char caractere;
};

struct une_structure_t ma_struct = {15, true, -5, 'A'};
```

mais cela impose d'affecter tous les champs et dans l'ordre de leur déclaration. On peut aussi n'affecter que certains champs et dans le désordre en utilisant la syntaxe ci-dessous :

```
struct une_structure_t ma_struct = {.entier16 = -5, .caractere = 'A', .entier32 = 15};
```

Par contre, il n'est pas possible d'utiliser cette syntaxe après la déclaration de la structure et on doit donc affecter ensuite chaque champs séparément : `ma_struct.booleen = true;`.
On peut aussi recopier l'intégralité du contenu d'une structure dans une autre :

```
struct une_structure_t autre_struct;
autre_struct = ma_struct;
```

et passer une structure en paramètre d'une fonction ou en renvoyer en résultat :

```
struct une_structure_t fct(struct une_structure s)
{
    struct une_structure t = {.booleen = !s.booleen, .entier32 = s.entier32 + 5,
        .entier16 = -s.entier16, .caractere = s.caractere + 1};
    return t;
}
```

Ceci dit, comme le compilateur va devoir recopier tous les champs de la structure pour la passer en paramètre, et ensuite allouer une zone mémoire pour la structure à renvoyer et de nouveau recopier les valeurs, il est souvent beaucoup plus efficace de manipuler des pointeurs de structures.

Unions

Une union ressemble beaucoup à une structure, avec la différence que les champs d'une union sont tous placés au même endroit en mémoire :

```
union une_union_t {
    uint32_t entier32;
    bool booleen;
    int16_t entier16;
    char caractere;
};
```

```
union une_union_t mon_union;
mon_union.entier32 = 0x12345678;
printf("entier16 : %" PRIx16 "\n", mon_union.entier16);
```

donne comme affichage : `entier16 : 5678` (ce qui permet de voir au passage que les processeurs Intel sont *little-endian*, c'est à dire place les poids faibles d'un mot en premier dans la mémoire).
Bien sûr, il est très facile d'écrire n'importe-quoi en utilisant des unions, vu qu'elles permettent de cours-circuiter les mécanismes de vérification de types du compilateur. Elles ont cependant leur utilité notamment sur des plates-formes embarquées contraintes avec une quantité de mémoire très limitée.

Champs de bits

Il n'existe pas en C de type bit permettant de manipuler des données de moins de 1 octet, mais on peut utiliser un type de structures particulier (appelé champs de bits) pour cela :

```
struct champs_bits_t {
    uint8_t b7: 1;
    uint8_t b654: 3;
    uint8_t b3210: 4;
};
```

On définit ci-dessus un type de valeur codé sur 1 octet et permettant d'accéder directement au bit de poids fort (`b7`), au 3 bits suivants (`b654`) et au 4 bits de poids faible (`b3210`) de l'entier sur 1 octet, comme illustré dans le bout de code ci-dessous :

```
struct champs_bits_t b = {.b7 = 1, .b654 = 0, .b3210 = 0b1111};
printf("Taille de b : %" PRIu64 ", valeur hexadecimale : 0x%" PRIx8 "\n",
    sizeof(struct champs_bits), (b.b7 << 7) | (b.b654 << 4) | b.b3210);
```

qui affiche `Taille de b : 1, valeur hexadecimale : 0x8f`.
En pratique, ce type de données peut être utile pour modéliser des structures matérielles ou des valeurs de format imposé (comme par exemple des instructions).

Pointeurs

On peut définir en C des pointeurs vers toutes sortes de valeurs. Il est important de se souvenir qu'un pointeur contient une adresse mémoire, c'est à dire une valeur dont la taille est indépendante de la taille de la donnée pointée (mais dépend de l'architecture sous-jacente).
Par exemple, `sizeof` renvoie toujours 8 quand on l'applique à des pointeurs de types `uint64_t *`, `int32_t *`, `char *` ou `bool *` sur une machine 64 bits en utilisant l'option `-m64` de GCC, alors qu'il renvoie toujours 4 si on utilise l'option `-m32` de GCC pour générer du code 32 bits.
L'opérateur `&` permet de récupérer l'adresse d'une variable, et son opposé `*` permet de déréférencer un pointeur pour accéder à la valeur pointée :

```
uint32_t entier = 5;
uint32_t *ptr = &entier;
*ptr = 10;
```

Notez bien la différence entre l'étoile utilisée dans la définition d'un type pointeur et l'étoile représentant l'opérateur de déréférencement.
On utilisera souvent des pointeurs pour permettre la modification des paramètres d'une fonction : on rappelle en effet qu'en C, les paramètres d'une fonction sont toujours passés par copie. Par exemple, la fonction :

```
void fausse_inversion(int32_t a, int32_t b)
{
    int32_t tmp;
    tmp = a;
```

```
    a = b;
    b = tmp;
}
```

appelée dans le bout de code suivant :

```
int32_t x = 4;
int32_t y = 6;
fausse_inversion(x, y);
printf("x : %" PRId32 ", y : %" PRId32 "\n", x, y);
```

n'a aucun effet sur les valeurs de `x` et `y` dans la fonction appelante, alors que :

```
void vraie_inversion(int32_t *a, int32_t *b)
{
    int32_t tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

inverse bien les valeurs originales `x` et `y` lorsqu'on l'appelle dans le bout de code ci-dessous :

```
vraie_inversion(&x, &y);
printf("x : %" PRId32 ", y : %" PRId32 "\n", x, y);
```

On utilisera aussi beaucoup de pointeurs vers des structures, par efficacité (puisque cela évite au compilateur de recopier le contenu de la structure passée en paramètre ou en retour de fonction), ou pour modéliser des types chainés, comme par exemple :

```
struct cellule_t {
    uint32_t val;
    struct cellule *suiv;
};
```

pour définir une cellule d'une liste chainée d'entiers naturels sur 32 bits chacun.
On rappelle une facilité de notation offerte par le langage lorsqu'on manipule des pointeurs vers des structures : si on déclare par exemple une variable pointeur `struct cellule_t *cell`, on peut écrire de façon équivalente : `(*cell).val = 5;` ou `cell->val = 5;` pour affecter le champ `val` de la cellule (notez bien que les parenthèses sont indispensables dans la première notamment car l'opérateur point est prioritaire sur l'étoile).
On rappelle enfin l'importance de l'arithmétique des pointeurs en C : si on définit `uint64_t *ptr`, alors l'opération `ptr++` augmente la valeur de `ptr` (c'est à dire l'adresse vers laquelle il pointe) de 8 et non pas de 1 octet.