Université Joseph Fourier
GRENOBLE
UFR IM²AG

INF352
Éléments de programmation en C

Présentation très incomplète du langage mettant l'accent sur les notions importantes au sein de l'UE via l'utilisation d'exemples

Structure d'un programme

Un programme en C est constitué d'un ensemble de définitions de :

- types
- variables (globales)
- fonctions

il doit comporter une fonction nommée `main`.

```
#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

Compilation

Traduction du code source en un programme exécutable

```
gcc -o programme -Wall -Werror -g hello_world.c
```

Options utiles :

- `-o nom` : nom du fichier exécutable
- `-Wall` : afficher plus d'avertissements
- `-Werror` : considérer les avertissements comme des erreurs
- `-g` : inclure les infos de deboguage

Quelques compilateurs : `gcc`, `llvm-gcc`, `clang`, `icc`

Fonction

Une fonction est constituée de :

- paramètres formels, locaux à la fonction (passage par valeurs)
- définitions de variables locales
- séquences d'instructions

Prototype : permet de déclarer une fonction avant sa définition

```
int addition(int, int); // prototype

int main() {
    int a, b;
    scanf ("%d %d", &a, &b);
    printf("%d %d %d\n", a, b, addition(a,b));
    return 0;
}

int addition(int a, int b) { // définition
    a += b;
    return a;
}
```

Variable

Une variable est typée :

- entiers : `char`, `short int`, `int`, `long int`, variantes `unsigned`
- rationnels : `float`, `double`
- pointeurs
- type composés à partir des types de base

`extern` : permet de déclarer une variable (globale) avant sa définition

```
extern int a;

int main() {
    for (int i=0; i<10; i++) {
        a++;
        printf("%d ", a);
    }
    printf("\n");
    return 0;
}

int a=0;
```

Entrée / sorties

Deux fonctions essentielles :

- `printf` : affiche selon un format donné la valeur des paramètres
- `scanf` : lit selon un format donné la valeur à donner aux paramètres
  - implique un passage par adresse qui n'existe pas en C
  - les arguments sont précédés de l'opérateur `&`

C'est le format qui dit comment interpréter la valeur donnée

```
int main() {
    int i;

    scanf(" %d", &i);
    printf("%d %x %c\n", i, i, i);
    return 0;
}
```

Format

Chaine de caractères, contenant des :

- parties constantes à afficher ou atendues lors de la lecture
- directives de traitement, une par argument suivant

Formats classiques :

- entiers : `%d` et `%i` (décimal), `%x` (hexadécimal), `%c` (caractère)
- variantes : `%ld` (entier long), `%u` et `%lu` (non signés)
- rationnels : `%f` (float), `%lf` (double)
- chaines de caractères : `%s`

Cas particulier pour `scanf` :

- `" "` est une séquence quelconque de séparateurs
- `%s` lit un seul mot (jusqu'au prochain séparateur)

Tableau

Séquence d'éléments d'un type existant

- éléments accessibles par leur position dans la séquence (indice)
- un tableau est toujours manipulé par adresse (pas de copie)

```
void lire_tableau(int t[], int n) {
    for (int i=0; i<n; i++) {
        scanf(" %d", &t[i]);
    }
}
int main() {
    int a[10];

    lire_tableau(a, 10);
    for (int i=0; i<10; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
    return 0;
}
```

Chaîne de caractères

Tableau de caractères contenant le caractère `'\0'` (convention)

- tableaux passés par adresse, pas de `&` pour `scanf("%s", s);`
- attention aux débordements !

```
int my_strlen(char s[]) {
    int i=0;

    while (s[i] != '\0')
        i++;
    return i;
}

int main() {
    char s[10];

    scanf("%s", s);
    printf("%d\n", my_strlen(s));
    return 0;
}
```

Dans la bibliothèque standard (`strlen, strcpy, strcmp, ...`)

Deboguage avec gdb

Le *debugger* gnu permet :

- d'executer un programme dans un environnement de *debug*
  - `gdb ./ex_chaine` pour entrer dans l'environnement
  - `run` qui peut être suivie d'arguments pour exécuter
- de visualiser
  - le code source :
    `list ligne, list fichier:ligne, list fonction`
  - l'état de la mémoire :
    `print expression, display expression, undisplay numero`
- de suivre précisément l'exécution avec
  - des informations détaillées en cas d'erreur
  - des points d'arrêt qu'on peut gérer avec :
    `break ligne, break fichier:ligne, break fonction,`
    `infos breakpoints, delete numero`
  - la possibilité d'exécuter pas-à-pas avec `step` et `next`

Structure

Type composé de plusieurs champs :

- chaque champ est d'un type défini et nommé
- opérateur . pour l'accès aux champs d'une variable

```
struct complexe {
    double r, i;
};

struct complexe addition(struct complexe a, struct complexe b) {
    a.r += b.r; a.i += b.i;
    return a;
}

int main() {
    struct complexe a, b, c;
    a.r = 2.0; a.i = 1.3;
    b.r = 1.1; b.i = 0.5;
    c = addition(a, b);
    printf("%lf %lf %lf", a.i, b.i, c.i);
    return 0;
}
```

Type

Défini par `typedef` à partir de toute construction :

```
typedef struct {
    double r, i;
} complexe;

complexe addition(complexe a, complexe b) {
    a.r += b.r; a.i += b.i;
    return a;
}

int main() {
    complexe a, b, c;
    a.r = 2.0; a.i = 1.3;
    b.r = 1.1; b.i = 0.5;
    c = addition(a, b);
    printf("%lf %lf %lf\n", a.i, b.i, c.i);
    return 0;
}
```

Abstraction plus poussée

```
typedef struct { double r, i; } complexe;

complexe creer_complexe(double r, double i) {
    complexe c; c.r = r; c.i = i; return c;
}

double img(complexe c) { return c.i; }

complexe addition(complexe a, complexe b) {
    a.r += b.r; a.i += b.i; return a;
}

int main() {
    complexe a, b, c;
    a = creer_complexe(2.0, 1.3);
    b = creer_complexe(1.1, 0.5);
    c = addition(a, b);
    printf("%lf %lf %lf\n", img(a), img(b), img(c));
    return 0;
}
```

Programmation modulaire

Consiste à séparer une partie indépendante du programme :

- interface connue dans tout le programme placée dans un fichier `.h`
- implémentation confinée dans un fichier `.c`

Multiples avantages :

- distribution du travail aisée, selon le découpage en modules
- dialogue entre développeurs simplifié, via une interface documentée
- compilation séparée possible
- evolutivité, changement de l'implémentation d'un module possible
- test simplifié, par module, de manière incrémentale

Interface

Déclarations (types, variables externes, prototypes) placées dans un fichier d'entête (`.h` pour *header*)

`complexe.h`

```
#ifndef __COMPLEXE_H__ // directive du préprocesseur
#define __COMPLEXE_H__ // pour éviter l'inclusion multiple
typedef struct {
    double r, i;
} complexe;

// constructeur
complexe creer_complexe(double, double);

// accesseur
double img(complexe);

//operation
complexe addition(complexe, complexe);
#endif
```

Implementation

Dans un fichier source (`.c` comme le langage), définitions des fonctions et variables du programme annoncés dans l'interface

`complexe.c`

```
#include "complexe.h"

complexe creer_complexe(double r, double i) {
    complexe c;
    c.r = r; c.i = i;
    return c;
}

double img(complexe c) {
    return c.i;
}

complexe addition(complexe a, complexe b) {
    a.r += b.r; a.i += b.i;
    return a;
}
```

Programme principal

Le programme principal n'a besoin que de l'interface pour pouvoir utiliser les complexes et être traduit

main.c

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

int main() {
    complexe a, b, c;
    a = creer_complexe(2.0, 1.3);
    b = creer_complexe(1.1, 0.5);
    c = addition(a, b);
    printf("%lf %lf %lf\n", img(a), img(b), img(c));
    return 0;
}
```

Compilation séparée

Deux étapes pour compiler :

- traduction et optimisation du code de chaque fichier source
- regroupement de l'ensemble en un unique programme

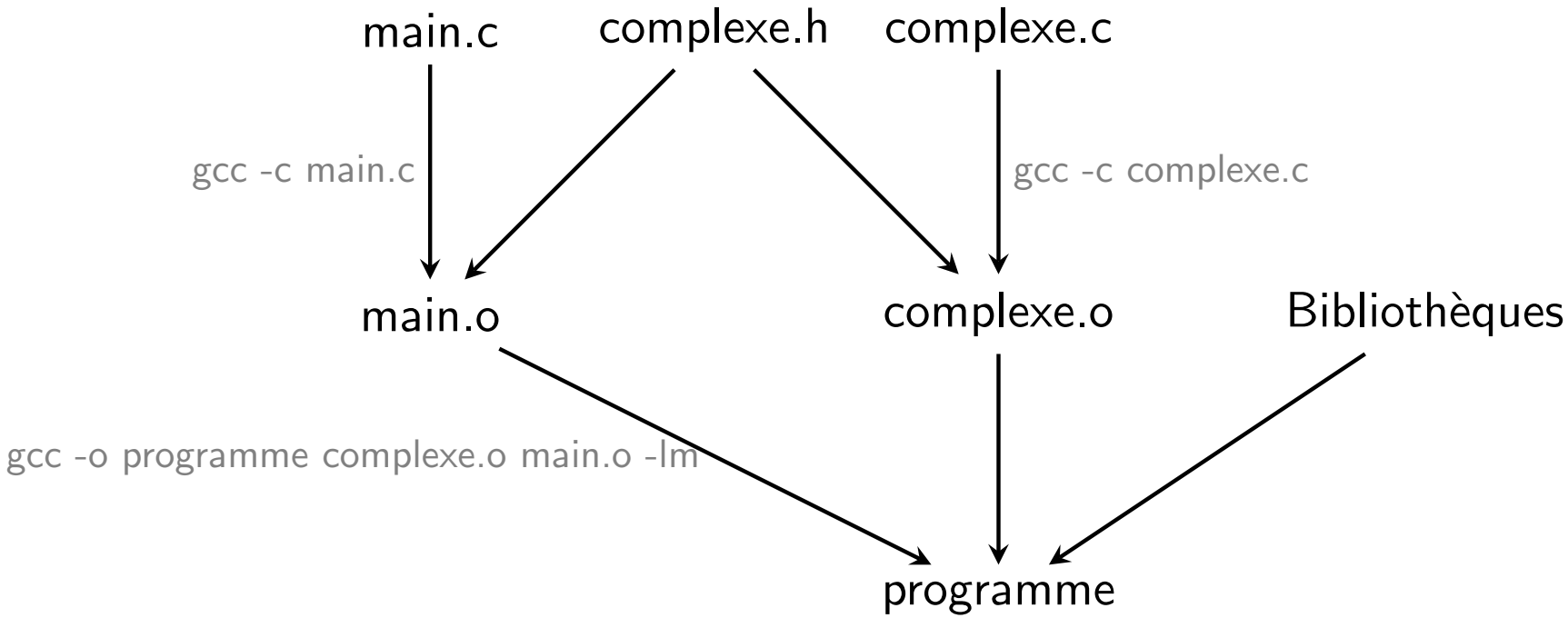
Avec gcc :

- traduction d'un fichier source `.c` en fichier objet `.o`
  `gcc -Wall -Werror -g -c complexe.c`
- édition de liens entre les fichiers objet `.o` et bibliothèques
  `gcc -o programme complexe.o main.o -lm`

Eviter de traduire inutilement :

- un `.o` ne dépend que du `.c` associé et des `.h` qu'il inclus
- la traduction et l'optimisation sont lentes

Compilation d'un programme



Plusieurs commandes nécessaires à chaque mise à jour $\rightarrow$ automatisation

make

Outil d'automatisation de processus de construction :

- on exprime les règles de fabrication pour chaque étape
- ne réexécute que les commandes nécessaires en cas de modification

Règles contenues dans un fichier nommé `Makefile` :

```
cible: liste des dépendances
    commande pour fabriquer la cible
```

Pas d'ordre particulier, mais la première règle est la cible par défaut

Exemple

Makefile

```
programme: complexe.o main.o
    gcc -o programme complexe.o main.o -lm

complexe.o: complexe.c complexe.h
    gcc -Wall -Werror -g -c complexe.c

main.o: main.c complexe.h
    gcc -Wall -Werror -g -c main.c
```

Utilisation :

```
> make
gcc -Wall -Werror -g -c complexe.c
gcc -Wall -Werror -g -c main.c
gcc -o programme complexe.o main.o -lm
> make
make: ‘programme’ is up to date.
> touch main.c; make
gcc -Wall -Werror -g -c main.c
gcc -o programme complexe.o main.o -lm
```

Documentation

La documentation des différentes fonctions C disponibles sur le système est accessible via la section 3 des pages de manuel :
`man 3 nom_de_fonction`
On y trouve en particulier, dans la section *synopsis* :

- les noms des fichiers système à inclure
- le prototype de la fonction cherchée
- les bibliothèques avec lesquelles il faut faire l'édition de liens

La documentation sur le langage C lui même se trouve dans :

- des livres comme "Le langage C" de B.W. Kernighan et D.M. Ritchie, édité chez Dunod, ISBN-10 : 2100715771 ISBN-13 : 978-2100715770
- des documentations en ligne comme
  `http://inf352.forge.imag.fr/Introduction_ANSI_C-B.Cassagne.pdf`