Algorithmique avancée et programmation
Introduction

Rym Guibadj

LISIC, EILCO

Contenu

- Rappels des notions de bases en C
- Complexité et algorithmes de tri
- Structures des données
  - Liste
  - File et Pile
  - Arbre
  - Tas et table de hachage
  - Graphe

Volume horaire

- 6 x 2h Cours
- 7 x 4h TP

Evaluation

- TP
- DS (Devoir Surveillé)
- Examen
- Note module = (TP + 2*DS + 4*Examen) / 7

Origines et normalisation

- Le langage C a été conçu au début des années 1970 par **Dennis Ritchie** et **Ken Thompson** dans les laboratoires Bell
- Objectif : disposer d'un langage souple et efficace pour remplacer l'assembleur dans l'écriture du code pour le système UNIX
- la définition du C a été complétée en 1989 par l'ANSI [1] et reprise par l'ISO [2] en 1990 (C-ANSI, C89, C90)
- l'ISO a émis un nouveau standard en 1999 qui emprunte des éléments au C++ (C99)
- l'ISO ratifie un nouveau standard en 2011 (C11) qui introduit la programmation multi-thread, les expressions à type générique,

1. American National Standards Institute
2. International Organization for Standardization

Caractéristiques du C

- **Procédural** : muni d'instructions de contrôle (if, else, for, while ...)
- **Typé** : tout objet C doit être déclaré avant d'être utilisé
- **Modulaire** : peut être découpé en modules qui peuvent être compilés séparément
- **Universel** : n'est pas orienté vers un domaine d'application particulier
- **Compilé** (et non interprété)

Exemple de programme C

```
#include <stdio.h>
int main ()
{
int result =0 ;
int i;

        for (i = 1; i <= 10; i++)
        {
                result= result+ i;
        }
printf ("%d\n", result);
return 0;
}
```

Déclaration et emplacement

- Déclaration d'une varaiable : **type_varaible nom_varaiable ;**
  Exemple : int a ;
- **Variable globale :** déclarée en dehors de tout bloc et fonction. Elle existe pendant toute la durée de l'exécution
- **Variable locale :** déclarée à l'intérieur d'un bloc (ou une fonction). Elle est créée à l'entrée du bloc et détruite lorsque l'en sort.
- Il est possible de déclarer deux variables de même nom à condition qu'elles ne soient pas déclarées au même niveau d'imbrication

Déclaration et emplacement

```
int a ;
int b;
int main ()
{
        a = 12;
        scanf("%d",&b);

        if(b==1)
        {
        int a ;
        a = 5;
        }

        b = a ;

}
```

les types en C

- **les caractères :** char k = 'a' ; char k1 = 65 ;
- **les entiers :** short i ; int j ; long k ; unsigned int l ;
- **les réels :** float n1 ; double n2 ; long double n2 ;
- **les booléens :** il n'existe pas de type booléen vrai, faux en C. On peut néanmoins utiliser les entiers : la valeur 0 vaut **faux** , toute valeur différente de 0 vaut **vrai**

Entrées / sorties au terminal

La fonction printf : permet d'afficher des informations à l'écran. Le premier paramètre de la fonction printf est une chaîne de caractères composée soit :

- des caractères à afficher tels quels : **printf("Entrez votre nom");**
- des codes de format spécifiés par un % : **printf("%d", a);**

Les codes formats les plus utilisés sont :

Code	Signification
%c	char
%d	int
%f	float
%s	chaîne de caractères

Entrées / sorties au terminal

La fonction scanf : permet de lire un flux d'entrée à partir du clavier.

- Le premier paramètre de la fonction scanf est le code format de la donnée.
- Le deuxième paramètre et l'adresse de l'emplacement ou sera stockée cette donnée.

```
int a ;
scanf ("%d", &a);
char nom[50];
scanf("%s", nom);
```

Les instructions de contrôle

La structure conditionnelle :

```
if (condition)
{
instructionsSiVrai ;
}
```

```
if (condition)
{
instructionsSiVrai ;
}
else
{
instructionsSiFaux ;
}
```

Les instructions de contrôle

Le choix multiple :

```
switch (expression) {
case valeur\_1 : séquence-d-instructions break;
case valeur\_2 : séquence-d-instructions break;
....
case valeur\_n : séquence-d-instructions break;
default : séquence-d-instructions
}
```

Les instructions de contrôle

La boucle **tant que** :

```
while (condition)
{ instructions;}
```

La boucle **répéter** :

```
do {
instructions;
} while (condition);
```

La boucle **pour** :

```
for (initialisation; condition-continuation; variation-compteur)
{ instructions ;}
```

Les pointeurs

Définition : un pointeur contient l'adresse d'une autre variable
Type_pointé * varaible_pointeur ;

```
int a = 5;
int * pt1;

pt1 = &a;          /*le pointeur pt1 contient
                        l'adresse de la variable a */
*pt1= 10 ;       /*la variable pointée par pt1 reçoit 10.
                         Même effet que a= 10*/
```

Tableaux statiques

- **Définition :** un tableau est une collection ordonnée d'éléments de même type rassemblés physiquement les uns à la suite des autres en mémoire.

- **Déclaration :** Type_d_element variable_tableau[taille] ;

- On peut considérer un tableau comme l'adresse en mémoire de son premier élément : le nom du tableau peut être assimilé à un pointeur ;

```
int main()
{
        int t[10];

        t[0] = 12;
        t[9] = 5;
        *t = 2;
        *(t+5) = 1;

}
```

Tableaux à plusieurs dimensions

```
int main()
{
    int i;

        int t1[3][2] ; // un tableau de 3 lignes et 2 colonnes;

        int t1[0][1] = 5; // affecter la valeur 5  à la case
                                                    de la ligne 0 et de l

}
```

chaînes de caractères

Une chaîne de caractère en C est un tableau de caractères à 1 dimension qui contient un caractère spécial marquant la fin de la chaîne $'\backslash 0'$

```
#include<stdio.h>
#include<stdlib.h>
void main() {
char ch[]= "Bonjour";
char *p = ch;

while (*p!='\0')
{
printf("%c",p[i]);
printf("\n");
p++;
}
}
```

chaînes de caractères : fonctions utilitaires

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

Bibliothèque C qui contient les fonctions de manipulation de chaînes de caractères

```
int strlen(const char* cs) ;
```

Retourne la longueur de la chaîne cs.

```
char* strcpy(char* s, const char* ct) ;
```

Copie ct dans s (et retourne s).

```
int strcmp(const char* cs, const char* ct) ;
```

Compare cs avec ct, selon l'ordre du jeu de caractères utilisé.
Retourne une valeur entière à tester :

- négative value si cs < ct (cs avant ct),
- 0 si cs est égale à ct,
- positive si cs > ct (cs après ct)

Les fonctions

Exemple d'une fonction en C

```
int somme(int a, int b) {
int s;
s = a + b;
return s;
}
```

Appel de la fonction :

```
int main ()
{
int i = 3;
int j = 2;
int k = somme(i,j);
printf("%d", k);

return 0;
}
```

Passage des arguments : passage par valeur

La transmission des arguments se fait par copie. La valeur des arguments ne sera pas modifiée

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

void echange(int a, int b) {
int temp = a;
a = b;
b = temp;
printf("apres echange: %d %d\n", a, b);

}

void main() {
int nb1,nb2;
printf("Saisir deux entiers\n");
scanf("%d", &nb1);
scanf("%d", &nb2);
printf("avant appel: %d %d \n", nb1, nb2);
echange(nb1, nb2);
printf("apres appel: %d %d \n", nb1, nb2);
}
```

Passage des arguments : passage par adresse

Si on passe l'adresse de l'argument alors sa valeur peut être changée depuis la fonction

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

void echange(int *a, int *b) {
int temp = *a;
*a = *b;
*b = temp;
printf("apres␣echange:␣%d␣%d\n", *a, *b);

}

void main() {
int nb1,nb2;
printf("Saisir␣deux␣entiers\n");
scanf("%d", &nb1);
scanf("%d", &nb2);
printf("avant␣appel:␣%d␣%d␣\n", nb1, nb2);
echange(&nb1, &nb2);
printf("apres␣appel:␣%d␣%d␣\n", nb1, nb2);
}
```

Structures enregistrements

Une structure permet de manipuler des collections d'objets de types différents

```
struct EnregistrementEtudiant {
int numeroEtudiant;
char nom[80];
char prenom[80];
};

void main() {
struct EnregistrementEtudiant e1, e2;
...
}
```

Structures enregistrements

Simplification d'écriture

```
struct EnregistrementEtudiant {
int numeroEtudiant;
char nom[80];
char prenom[80];
};

typedef struct EnregistrementEtudiant Etudiant;

void main() {
Etudiant e1, e2;
...
}
```

Gestion dynamique de la mémoire

La fonction malloc : permet d'allouer un espace mémoire. Elle prend un argument unique qui correspond à la taille mémoire que l'on souhaite allouer. Elle retourne un pointeur :

- NULL si l'allocation à échouer.
- void * si l'allocation est correcte.

```
#include<stdlib.h>
int i;
int *t;
t = (int *) malloc(100 * sizeof(int));
for (i = 0; i < 100; i++) {
*(t+i) = i;
}
```

Gestion dynamique de la mémoire

La fonction free : permet de libérer l'espace mémoire déjà alloué avec la fonction malloc

```
#include<stdio.h>
#include<stdlib.h>
void main() {
int *p1;
char  *p2;
p1 = (int *) malloc(100 * sizeof(int) );
p2 = (chart *) malloc(50 * sizeof(char) );

free(p1);
free(p2);
```

Gestion des fichiers

- Le C offre la possibilité de lire et d'écrire des données dans un fichier.
- Les accès à un fichier se font par l'intermédiaire d'une mémoire-tampon (buffer).
- Pour manipuler un fichier, un programme a besoin d'un certain nombre d'informations assemblées dans une structure de type FILE * :
  - l'adresse de l'endroit de la mémoire-tampon où se trouve le fichier,
  - la position de la tête de lecture,
  - le mode d'accès au fichier (lecture ou écriture)
- Un structure FILE * est appelé flot de données. Il est défini dans <stdio.h>

fichiers : fonctions utilitaires

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

Bibliothèque C qui contient les fonctions de manipulation de fichier

```
FILE* fopen(const char* filename, const char* mode) ;
```

Ouvre un fichier de nom filename et retourne un pointeur de fichier ou le pointeur nul si échec.

```
int fclose(FILE* stream) ;
```

Ferme le fichier et retourne 0 si succès, EOF si erreur.

```
int fprintf(FILE* stream, const char* format, ...);
```

Convertit les données (selon les indications de la chaîne format) et les envoie sur le flux de sortie stream.

```
int fscanf(FILE* stream, const char* format, ...) ;
```

Effectue une lecture avec conversion, lisant dans stream selon les indications de format.