Séance 8 : Découpage modulaire d'un programme, Les fichiers

Langage C : séance 8

Découpage modulaire d'un programme en langage C :

Un programme écrit en langage C est rarement écrit dans un seul module (ou fichier). Le code source s'étend souvent sur plusieurs modules, dont chacun a un rôle bien précis.
Nous allons à travers un exemple comprendre comment décomposer son programme en modules différents et voir comment est construit le programme final (l'exécutable).

Etape 1 : création des modules d'en-tête.

Il s'agit de fichiers ayant une extension « .h » (h pour header). Ces fichiers contiennent des définitions de types et les déclarations des fonctions à développer.

Etape 2 : création des modules de définitions des fonctions.

Ces fichiers reprennent les fonctions déclarées précédemment et les définissent (ajout du corps de la fonction).

Etape 3 : création du programme principal (main).

Le programme « main » appelle les fonctions.

Langage C : séance 8

Découpage modulaire d'un programme en langage C :

Fichier « mesfonctions.h »

```
#ifndef _MESFONCTIONS_H
#define _MESFONCTIONS_H
void mafonction(char * ch);
#endif
```

Fichier « mesfonctions.c»

```
#include ''mesfonctions.h''
void mafonction(char * ch)
{
    printf(''message : %s \n'',ch);
}
```

Fichier « monprogramme.c »

```
#include <stdio.h>
#include ''mesfonctions.h''
int main()
{
    // appel de la fonction
    mafonction(''bonjour '');
    return 0;
}
```

Dans chaque module « .h », on doit commencer par les deux directives au préprocesseur #ifndef et #define pour éviter les inclusions multiples, et terminer par #endif.
Dans les modules de définitions de fonctions (ici le fichier « mesfonctions.c ») , on inclut (directive #include) le module d'en-tête correspondant (ici « mesfonctions.h »).
Dans le programme principal (ici « monprogramme.c »), on inclut le module d'en-tête.

Langage C : séance 8

Le préprocesseur :

Le préprocesseur permet de réaliser une phase de prétraitement du fichier source avant la compilation : inclusion de fichiers, substitution de termes ,…

- **Pour demander à inclure un autre fichier source :**

On utilise la directive « #include » :

#include <stdio.h>

➔ demande l'inclusion du fichier d'en-tête « stdio.h » de la librairie standard du C, contenant toutes les déclarations des fonctions d'Entrées/Sorties.

#include ''monfichier.h''

➔ demande l'inclusion du fichier d'en-tête « monfichier.h » de mon répertoire courant contenant des déclarations de mes fonctions.

on peut écrire aussi : #include ''c:/mesdocuments/monfichier.h''

Langage C : séance 8

Le préprocesseur :

- **Pour demander à définir une expression :**

  On utilise la directive « #define » :

  #define MAX 100 // pas de ; à la fin !

  …

  if (i<MAX) …

  Le préprocesseur remplacera l'expression MAX par sa valeur (100) juste avant la phase de compilation.

  Intérêt : MAX n'est pas une variable, elle n'occupe pas de place en mémoire.

  Inconvénient : MAX n'est pas typée, donc pas de possibilité de vérifier des typages.

- **Les macros :**

  On utilise toujours la directive « #define » :

  #define MAX((a), (b)) ((a)>(b)?(a) : (b))

  …

  if (k != MAX(i,j)) …

Langage C : séance 8

Le préprocesseur :

•La directive d'inclusion conditionnelle :

A chaque directive « #include … », il y a inclusion du contenu du fichier.
Le préprocesseur ne gérant pas les inclusions multiples, si le contenu a déjà été inclus, il y aura une inclusion multiple des mêmes fonctions ➔ échec de l'édition de liens.
Le programmeur doit donc empêcher les inclusions multiples. Il faut donc systématiquement concevoir les fichiers d'en-tête de la façon suivante :
Fichier « monfichier.h » :

```
#ifndef   __MONFICHIER_H__
 #define __MONFICHIER_H__    // on reprend le nom du fichier en majuscule et _
…
//mettre ici toutes les définitions de types et les déclarations des fonctions
…
 #endif    // fin du fichier « monfichier.h »
```

Il n'a plus qu'à faire un **#include ''monfichier.h''** dans les fichiers où l'on souhaite utiliser les types et les fonctions de ce fichier.

Langage C : séance 8

Les variables à travers les modules :

- **Variable globale :**

Rappel : - déclarée en dehors de tout bloc.
- visible de tout le fichier.
- zone mémoire : zone de données.
- durée de vie permanente (celle du main).
- accessible par le main et toutes les fonctions !

- **Variable globale static (ou fonction static):**

idem que variable globale mais rend celle-ci invisible à l'extérieur du fichier.
Une fonction static n'est visible que dans le fichier où elle est déclarée.

- **Variable globale extern (ou fonction extern):**

Indique que l'emplacement réel et la valeur initiale d'une variable, ou du corps d'un sous-programme sont définis ailleurs (autre fichier).

Langage C : séance 8

Les variables à travers les modules :

```
// fichier « fic1.c »
…
int vari=1;  // variable globale
static int varsj=0 ; // variable globale static
static void f1(); // fonction static
…
void f2() {
…
vari=3;     // autorisé
varsj=8;   // autorisé
f1();         // autorisé
}
```

```
// fichier « fic2.c »
…
extern int vari;    // autorisé
extern void f2();  // autorisé
void f3() {
f2();   // autorisé
f1();   // accès interdit
varsj=5;  // accès interdit
vari=10;  // autorisé
}
```

Langage C : séance 8

Les variables à travers les modules :

- **Variable locale :**
  - déclarée dans un bloc.
  - visible uniquement dans le bloc ou les blocs imbriqués.
  - zone mémoire : en pile (stack).
  - durée de vie temporaire : détruite à la sortie du bloc.

- **Variable locale static :**
  - Permet de conserver l'état de la variable entre deux exécutions d'une fonction.
  - La variable n'est initialisée qu'une seule fois.
  - zone mémoire utilisée : celle des variables globales.

Langage C : séance 8

Les variables à travers les modules :

```
// fichier « fic.c »
…
void f1() {
        static int s_i=3;    // variable locale static
        printf(''s_i = %d\n'',s_i);
        s_i ++;
}
int main() {
f1();
f1();
f1();
return 0;
}
```

Affichage obtenu :

s_j = 3
s_j = 4
s_j = 5

Langage C : séance 8

Les fichiers de données :

On peut différencier deux types de fichiers de données utilisés par un programme en langage C :

-**Fichier texte :** il contient des octets dont chaque octet correspond à un caractère. Ce type de fichier peut être créé et modifié par un éditeur de texte. Ce sont des fichiers au format «.txt ».

-**Fichier binaire :** il contient des octets dont on ne peut pas à priori connaître la signification. Seul celui qui y a écrit les octets sera capable de récupérer correctement les données.

Exemple : j'écris dans un fichier le contenu d'une variable « int » (4 octets) et le contenu d'un « char » (1 octet). Ce fichier contient donc 5 octets de données. Si je dois relire ces données dans le fichier, je dois d'abord lire 4 octets puis 1 octet pour retrouver correctement les valeurs de mes données.

Langage C : séance 8

Les fichiers « texte »:

On peut accéder aux contenus des fichiers « texte » avec des fonctions de <stdio.h>.

•Type de variables :

On doit utiliser des variables de type : FILE *

```
FILE * fic=NULL; // on appellera « fic » une variable logique de fichier.
```

• Ouverture de fichier : fonction « fopen ».
•Traitement de lecture : fonctions « fscanf », « fgets », « fgetc ».
•Traitement d'écriture : fonctions « fprintf », « fputs », « fputc ».
•Fermeture de fichier : fonction « fclose ».

Langage C : séance 8

Ouverture de fichier (texte ou binaire):

On doit ouvrir le fichier en spécifiant son nom physique (dans l'arborescence des fichiers) et son mode d'ouverture : lecture, écriture, ajout.
La valeur retournée par la fonction « fopen » permet d'initialiser la variable logique pour accéder au contenu du fichier.
La fonction retourne NULL en cas d'échec (à tester après chaque ouverture).

- la fonction « fopen ».

```
FILE * fic=NULL;    // car fic doit être un pointeur.
fic=fopen( ''nomfichier.txt'' , ''r'' );
```

Nom physique du fichier (avec l'extension !)

Mode d'ouverture possible :
''r'' : lecture, fichier existant.
''w'': écriture (écrasement), création si inexistant.
''a'': écriture à la fin(sans écrasement), création si inexistant.
''r+'': lecture/écriture à la fin, fichier existant.
''w+'': lecture/écriture (avec écrasement), création si inexistant.
''a+'': lecture/écriture à la fin, création si inexistant.

Ensuite : `if (fic != NULL)  // ouverture avec succès !`

Langage C : séance 8

Fermeture d'un fichier (texte ou binaire):

• la fonction « fclose ».

Il ne faut pas oublier de fermer un fichier ouvert par un fopen par la fonction fclose.

```
FILE * fic=NULL;
fic=fopen( ''nomfichier.txt'' , ''r'' );
if (fic !=NULL)
    {
        … traitement
        fclose(fic);            // fermeture du fichier si ouverture réussie uniquement !
    }
else
    {
        printf('' echec ouverture fichier \n'');
        exit (1);               // la fonction exit permet d'arrêter le programme (➔arrêt du main !)
    }
```

Langage C : séance 8

Lecture dans un fichier texte:

• la fonction « fscanf ».

Elle permet de lire un ensemble de valeurs du fichier vers la mémoire.

Syntaxe : int fscanf (FILE * stream, char * chaine, liste d'adresses);

Retourne le nombre d'objets lus.
stream : variable fichier.
chaine et liste adresses comme dans un scanf.

Utilisation :

```
int i;
float f;
if (fscanf(fic, ''%d %f'', &i, &f) == 2)
        printf(''%d %f'', i,f);
```

Langage C : séance 8

Lecture dans un fichier texte:

• la fonction « fgets ».

Elle permet de lire une ligne sous la forme d'une chaîne de caractères du fichier vers la mémoire.

Syntaxe : char * fgets (char * chaine, int n, FILE * stream);

Lit au plus n-1 caractères (ou jusqu'au '\n') du fichier et les place dans chaine.
Rajoute un '\0' en fin de chaine.
Retourne un pointeur sur chaine ou NULL.

Utilisation :

```
char chaine[21];
if (fgets(chaine, fic) != NULL)
        printf(''%s'', chaine);
```

Langage C : séance 8

Lecture dans un fichier texte:

• la fonction « fgetc ».

Elle permet de lire un caractère du fichier vers la mémoire.

Syntaxe : int fgetc (FILE * stream);

Lit un caractère du fichier et le retourne ou EOF en cas d'échec.

Utilisation :

```
char car;
car=fgetc(fic) ;
printf(''%c'', car);
```

Langage C : séance 8

Ecriture dans un fichier texte:

- la fonction « fprintf ».

Elle permet d'écrire sous la forme d'une chaîne de caractères un ensemble de valeurs de la mémoire vers le fichier.

Syntaxe : int printf (FILE * stream, char * chaine, liste de variables);

Retourne le nombre d'objets écrits.
stream : variable fichier.
chaine et liste variables comme dans un printf.

Utilisation :

```
int i=10;
float f=3.14;
if (fprintf(fic, ''%d %f'', i, f) == 2)
        printf(''transfert réussi'');
```

Langage C : séance 8

Ecriture dans un fichier texte:

• la fonction « fputs ».

Elle permet d'écrire une chaîne de caractères de la mémoire vers le fichier.

Syntaxe : int fputs (char * chaine, FILE * stream);

Retourne une valeur non négative si succès ou EOF en cas d'erreur.
chaine : doit se terminer par le caractère '\0'.

Utilisation :

```
char chaine[10]= ''dupont'';
if (fputs(chaine,fic) != EOF)
        printf(''transfert réussi'');
```

Langage C : séance 8

Ecriture dans un fichier texte :

Elle permet d'écrire un caractère de la mémoire vers le fichier.

• la fonction « fputc ».

Syntaxe : int fputc (char c, FILE * stream);

Retourne le caractère c en cas de succès sinon EOF.

Utilisation :

```
char c='A';
if (fputc(c,fic) != EOF)
        printf(''transfert réussi'');
```

Déplacement dans un fichier (texte ou binaire):

- la fonction « ftell ».

Syntaxe : long ftell (FILE * stream);

renvoie la position du pointeur de fichier en nombre d'octets par rapport au début du fichier.

- la fonction « rewind ».

En mode d'accès lecture/écriture sur un fichier, on peut se repositionner en début de fichier avec un appel à la fonction « rewind »:

Syntaxe : void rewind(FILE * stream);

Déplacement dans un fichier (texte ou binaire):

• la fonction « fseek ».

La fonction « fseek » permet de se positionner sur un octet précis dans un fichier.

Syntaxe : int fseek (FILE * stream, long nb, int position);

Déplace le pointeur de déplacement dans le fichier d'un nombre « nb » d'octets plus loin de l'emplacement mentionné par position.
Retourne 0 si déplacement correct, valeur non nulle si erreur.
« position » doit être une des expressions suivantes :

SEEK_SET : début de fichier.
SEEK_CUR : position courante.
SEEK_END : fin de fichier.

Utilisation :

```
fseek(fic, 10, SEEK_SET);   // se déplace sur le 10 ième octets du fichier.
```

Traitement sur les fichiers (texte ou binaire):

• la fonction « remove ».

Le fichier doit être fermé. Cette fonction permet de détruire physiquement un fichier.
On spécifie en paramètre le nom physique du fichier.

```
test=remove(''nomfichier.txt'');
```

Valeur retournée : 0 si succès, -1 sinon.

• la fonction « rename ».

Le fichier doit être fermé. Cette fonction permet de renommer physiquement un fichier.

```
test=rename(''anciennom'',''nouveaunom'');
```

Valeur retournée : 0 si succès, -1 sinon.

Langage C : séance 8

Lecture dans un fichier binaire:

• la fonction « fread ».

Elle permet de lire un groupe d'octets du fichier vers la mémoire.

Syntaxe : size_t fread(void * ptdebut, size_t taille, size_t n, FILE * stream);

Retourne le nombre d'objets lus (valeur du troisième paramètre).
n : nombre d'objets à lire.
taille : dimension en octets d'un objet.
stream : variable fichier.
ptdebut : adresse du bloc mémoire où écrire l'info lue.

Utilisation :

```
int i;
if (fread(&i, sizeof ( int), 1 , fic) == 1)
   printf(''%d'', i);
```

```
int t[10];
if (fread(t, sizeof ( int), 10 , fic) == 10)
        printf(''récupération réussie'');
```

Langage C : séance 8

Ecriture dans un fichier binaire :

• la fonction « fwrite ».

Elle permet d'écrire un groupe d'octets de la mémoire vers le fichier.

Syntaxe : size_t fwrite(void * ptdebut, size_t taille, size_t n, FILE * stream);

Retourne le nombre d'objets écrits (valeur du troisième paramètre).
n : nombre d'objets à écrire.
taille : dimension en octets d'un objet.
stream : variable fichier.
ptdebut : adresse du bloc mémoire où lire l'info.

Utilisation :

```
int i=10;
if (fwrite(&i, sizeof ( int), 1 , fic) == 1)
   printf(''transfert réussi'');
```

```
int t[5]={1,2,3,4,5};
if (fwrite(t, sizeof ( int), 5 , fic) == 5)
            printf(''écriture réussie'');
```