

COURS 9

Programmation impérative

Compilation

- Instructions au préprocesseur
- Compilation séparée
- Communication entre modules
- Arguments de la ligne de commande

SOMMAIRE

- Informations pratiques
- Introduction
- Éléments de base
 - Programmer en Langage C – Compilation
 - Structure d'un programme / Règles d'écritures
 - Types de base
 - Constantes/Variables
 - Opérateurs
 - Instructions de contrôle
 - Pointeurs
 - Tableaux
- Fonctions
- Chaînes de caractères
- Pointeurs- Tableaux-Fonctions
- Types Construits
- Entrées – Sorties sur Fichiers
- **Compilation séparée**
- Implémentation de Types Abstraits de Données

Compilation séparée

- Le préprocesseur

- programme standard qui effectue des modifications sur un texte source
- modifie le source d'après les directives données par le programmeur, et introduites par le caractère #.
 - directive du préprocesseur commence par un # et se termine par la fin de ligne
 - Si la directive ne tient pas sur une seule ligne, on peut l'écrire sur plusieurs en terminant les premières lignes par \ qui annule le retour chariot.
 - Ces directives peuvent apparaître n'importe où dans le fichier, pas nécessairement au début.

Compilation séparée/ Pré-processeur

- Le préprocesseur peut rendre plusieurs services
 - l'inclusion de fichier
 - la substitution de symboles
 - le traitement de macro-instructions
 - la compilation conditionnelle

Compilation séparée/ Pré-processeur

Inclusion de fichier .h	
#include	Lorsque le préprocesseur rencontre la commande #include, il remplace cette ligne par le contenu du fichier.
#include <nomFichier.h>	inclusion du fichier nomFichier.h, recherché dans les répertoires systèmes connus du compilateur
#include "nomFichier.h"	inclusion d'un fichier utilisateur nomFichier.h, recherché dans l'espace des fichiers de l'utilisateur

Compilation séparée/ Pré-processeur

Substitution de Symboles	
<code>#define NOM resteDeLaLigne</code>	le préprocesseur remplace toute nouvelle occurrence de NOM par resteDeLaLigne dans toute la suite du code.
<code>#define NB_COLONNES 80</code>	Toute nouvelle occurrence de NB_COLONNES sera remplacé par 80 dans tout le reste du fichier
<code>#define NB_LIGNES 24</code>	Toute nouvelle occurrence de NB_LIGNES sera remplacé par 24 dans tout le reste du fichier
<code>#define TAILLE_TAB NB_LIGNES * NB_COLONNES</code>	Toute nouvelle occurrence de TAILLE_TAB sera remplacé par 80×24 c'est à dire 1920 dans tout le reste du fichier

Compilation séparée/ Pré-processeur

- Macro-instruction
 - permet une substitution de texte paramétrée par des arguments.

Syntaxe	
<code>#define NOM(liste_parametres_formels) resteDeLaLigne</code>	Liste de paramètres formels ↔ liste d'identificateurs séparés par des virgules. Il n'y a pas d'espace entre NOM et (
<code>NOM(liste_parametres_effectifs)</code> Appel de la macro	Préprocesseur remplace l'ensemble <code>NOM(liste_parametres_effectifs)</code> par la chaîne <code>resteDeLaLigne</code> où les occurrences des paramètres formels sont remplacées par les paramètres effectifs.

Compilation séparée/ Pré-processeur

- Macro-instruction

Exemple

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

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



int main(){
    int i=15,j=100,borneInf,borneSup;

    borneInf = MIN(i,j);
    borneSup = MAX(i,j);
    printf("borne inférieure: %d\n",borneInf);
    printf("borne supérieure: %d\n",borneSup);
    return EXIT_SUCCESS;
}
```

borne inférieure: 15
borne supérieure: 100

remplacé par
borneInf= ((i)<(j) ? (i):(j))
borneSup= ((i)>(j) ? (i):(j))

Compilation séparée / Pré-processeur

Pièges à éviter	
<p>Oubli des parenthèses autour des paramètres formels</p> <pre>#define CARRE(a) (a*a)</pre>  <pre>#define CARRE(a) ((a)*(a))</pre>	<pre>int x, y=3; x=CARRE(y+1) ; ⇔ x= y+1*y+1= 2y+1</pre>  <pre>int x, y=3; x=CARRE(y+1); ⇔ x= (y+1)*(y+1)= (y+1)²= y² + 2y +1</pre>
<p>Effets de bords</p> <pre>#define CARRE(a) ((a)*(a))</pre>	<pre>int x, y=3; x=CARRE(++y); ⇔ x= (++y)*(++y) ; //y est incrémenté 2 fois</pre> <div data-bbox="1659 1134 2161 1209" style="border: 2px solid green; padding: 2px; display: inline-block;">x: 25 y: 5</div>

Compilation séparée / Pré-processeur

- Compilation conditionnelle

- but = compiler ou ignorer des ensembles de lignes, le choix étant basé sur un test exécuté à la compilation.

Syntaxe	
<code>#if expression ensembleLignes; #endif</code>	Si l' expression est vraie ensembleLignes est compilé L'évaluation d' expression a lieu au moment de la compilation
<code>#if expression ensembleLignes1; #else ensembleLignes2; #endif</code>	Si l' expression est vraie ensembleLignes1 est compilé, Sinon ensembleLignes2 sera compilé
<code>#if expression1 ensembleLignes1; #elif expression2 ensembleLignes2; #else ensembleLignes3; #endif</code>	Si l' expression1 est vraie ensembleLignes1 est compilé, Sinon si l' expression2 est vraie ensembleLignes2 sera compilé Sinon c'est ensembleLignes3 qui sera compilé

Compilation séparée / Pré-processeur

- Compilation conditionnelle

Exemple

```
#include <stdio.h>

#define AFFICHE 1

int main() {
    int x = 10;

    x = x + 10;
    #if AFFICHE
        printf("x: %d\n", x);
    #endif
    printf("fin du programme\n");
    return 1;
}
```



```
<terminated> tp_ex [C/C++
x: 20
fin du programme
```

Expression **AFFICHE** est vraie
⇔ Ligne compilée

Compilation séparée / Pré-processeur

- Compilation conditionnelle

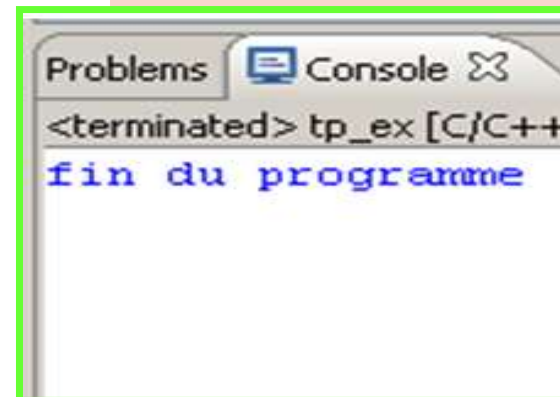
Exemple

```
#include <stdio.h>

#define AFFICHE 0

int main() {
    int x = 10;

    x = x + 10;
    #if AFFICHE
        printf("x: %d\n", x);
    #endif
    printf("fin du programme\n");
    return(1);
}
```



Expression AFFICHE est fausse
⇔ Ligne non compilée

Compilation séparée / Pré-processeur

- Compilation conditionnelle

Autres commandes	Fonctionnalité/exemple
<code>#ifdef</code>	permet de tester si un symbole a été défini par la commande <code>#define</code> <code>#define TEST</code> ... <code>#ifdef TEST</code> <code>#define VALEUR 1</code> <code>#else</code> <code>#define VALEUR 2</code> <code>#endif</code>
<code>#ifndef</code> Ou <code>#if !defined</code>	permet de tester si un symbole n'a pas déjà été défini par un <code>#define</code> <code>#ifndef TEST</code> ou <code>#if !defined (TEST)</code> <code>#define VALEUR 1</code> <code>#define VALEUR 1</code> <code>#endif</code> <code>#endif</code>

Compilation séparée / Pré-processeur

- Compilation conditionnelle

Autres commandes	Fonctionnalité/exemple
<code>defined</code>	<p>opérateur spécial : qui ne peut être utilisé que dans le contexte d'une commande <code>#if</code> ou <code>#elif</code></p> <p>permet d'écrire des tests portant sur la définition de plusieurs symboles</p> <p><code>#ifdef</code> qui ne peut en tester qu'une définition de symbole</p> <p><code>#if defined(SYMBOLE1) defined(SYMBOLE2)</code></p>

Compilation

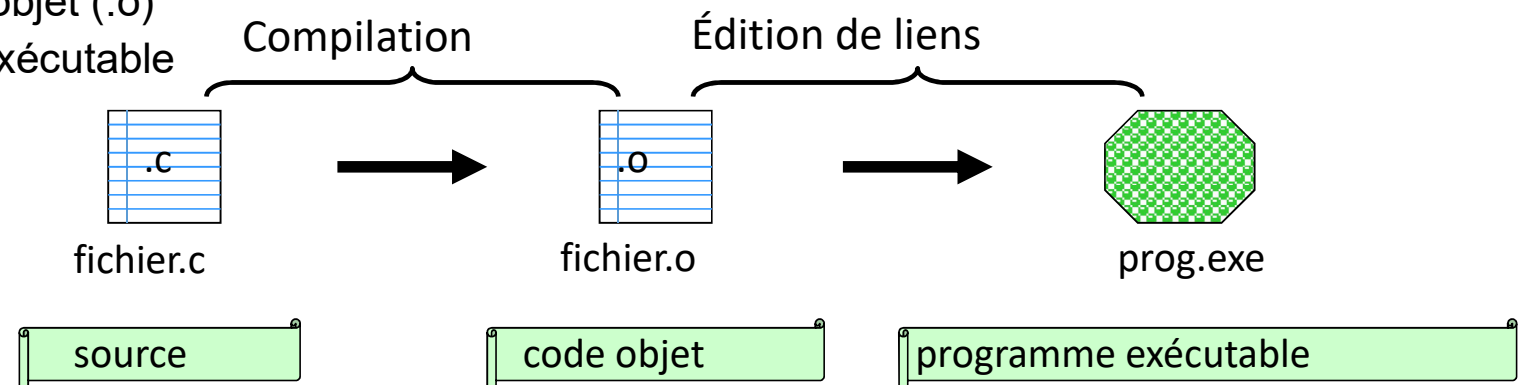
- Compilation simple – Rappel

- Phase de compilation :

- entrée : fichier source (.c)
 - sortie : fichier objet (.o)

- Phase d'édition de liens :

- entrée : fichier objet (.o)
 - sortie : fichier exécutable



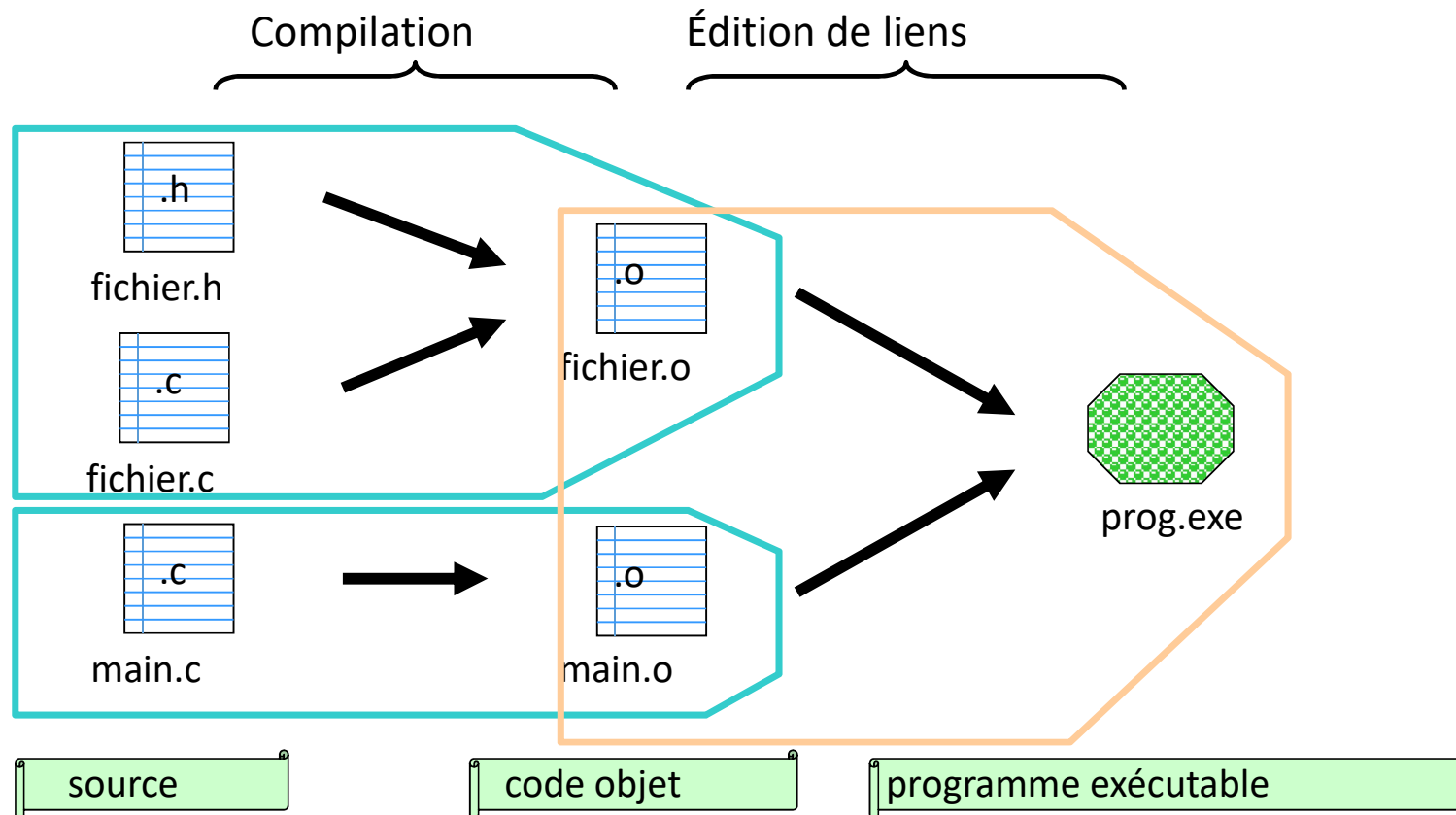
Compilation / Compilation séparée

- Lorsqu'un programme devient grand, il devient intéressant
 - de diviser le code source en plusieurs fichiers .c
 - 👉 il n'est plus nécessaire de recompiler l'ensemble du source à chaque fois mais uniquement ce qui a été modifié.
 - de pouvoir regrouper un ensemble de fonctions et les isoler dans un fichier source
 - 👉 Cet ensemble pourra être réutilisé dans d'autres projets
 - 👉 Notion de librairie

Compilation / Compilation séparée

- Une application peut être conçue comme une collection de modules
 - module = unité de programme mémorisée dans un fichier, qui constitue une unité de compilation
 - Les unités sont compilées séparément
 - 👉 tâche de l'éditeur de liens = rassembler les divers modules objets pour constituer l'application totale.
 - 👉 1 protocole est nécessaire pour la communication entre les divers modules

Compilation /Compilation séparée



Compilation/ Compilation Séparée

- Un fichier objet (.o) est un fichier contenant
 - le code compilé du source correspondant
 - une table des variables et des fonctions exportées définies dans le source
 - une table des variables et des fonctions qui sont utilisées mais non définies dans ce fichier source

Permettront aux modules de communiquer entre eux

Compilation/ Compilation Séparée

- Règles de communication
 - 1 variable ou 1 fonction est partagée entre 2 modules quand elle est utilisée dans ces 2 modules
 - Seules les variables globales à un module peuvent être partagées
 - Toute fonction peut-être partagée
 - 1 variable ou fonction partagée doit faire l'objet d'une déclaration d'importation dans le module où elle est utilisée mais non définie
 - mot clé **extern**

Compilation séparée / Compilation Séparée

- Règles de communication

Importation	
Variable <code>extern int x ;</code>	<p>le traducteur n'engendre pas de place mémoire mais prend uniquement connaissance de l'existence de la définition de cette variable <code>x</code> de type <code>int</code> dans une autre unité de compilation ;</p> <p>la variable importée est visible dans tout le module ou bloc importateur à partir du point de cette déclaration</p>
fonction <code>extern int fonct() ;</code> <code>extern void test(int) ;</code>	<p>déclaration d'importation ne contient que le mot clé <code>extern</code> suivi de la signature de la fonction</p>

Compilation/ Compilation Séparée

- Règles de communication

Illustration

P
r
o
g
r
a
m
m
e

```
module1.c
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int v=23;
5  int x=25;
6
7  extern int val;
8  extern int test(int);
9
10 int main() {
11     int y=6, res;
12     res=test(y+val);
13     return EXIT_SUCCESS;
14 }
```




```
module2.c
1  int val=100;
2
3  int test(int p){
4      extern int v;
5      int i;
6
7      ...
8  }
9  int fonct(float z){
10     ....
11 }
```

Compilation/ Compilation Séparée

- Règles de communication

	Protection de données
Mot clé static	rend un identificateur non exportable ; Cas d'utilisation <ul style="list-style-type: none">• on ne désire pas que les symboles définis dans un module soient accessibles depuis l'extérieur d'un module• pour éviter les conflits entre variables de même nom définies dans des modules différents

Programme

 fichier1.c	 fichier2.c	 fichier3.c
1 <code>static int v=0;</code> 2 <code></code>	1 <code>int v=0;</code> 2 <code></code>	1 <code>extern int v;</code> 2 <code></code>

Compilation/ Compilation Séparée

- En C un module doit être implanté à l'aide de deux fichiers
 - les définitions dans un fichiers source avec comme extension `.c`
 - les déclarations des variables et fonctions exportables dans un fichier d'en-tête avec comme extension `.h`
 - Un module utilisant les fonctions et/ou les variables d'un autre module doit "inclure" le fichier d'en-tête de celui-ci

Compilation/ Compilation Séparée

- Illustration

Programme

main.c	pile.c	pile.h
<pre>1 #include <stdlib.h> 2 3 extern void empiler(double); 4 5 int main(void) { 6 double x ; 7 8 empiler (50) ; 9 return EXIT_SUCCESS; 10 }</pre>	<pre>1 #include "pile.h" 2 3 void empiler(double y) { 4 5 } 6 7 double depiler () { 8 9 }</pre>	<pre>1 void empiler (double) ; 2 3 double depiler(void) ;</pre>

Compilation/ Compilation Séparée

- Fichiers d'entête .h
 - Pour éviter les inclusions multiples
 - Placer le contenu du fichier .h à l'intérieur d'une inclusion conditionnelle

Fichier.h

```
#ifndef ENTETE  
  
#define ENTETE  
  
/* on met ici le contenu de entete.h */  
  
#endif
```

Compilation/ Compilation Séparée

- Fichiers d'entête .h
 - Pour éviter les inclusions multiples
 - Placer le contenu du fichier .h à l'intérieur d'une inclusion conditionnelle

Fichier.h	
<pre>#ifndef ENTETE #define ENTETE /* on met ici le contenu de entete.h */ #endif</pre>	<p>expression ENTETE vaut</p> <p>0 si le fichier n'a pas été inclus</p> <p>1 s'il a déjà été inclus</p>

Compilation/ Compilation Séparée

- Illustration

Programme

```
main.c
1  #include <stdl:
2
3  extern void empiler(double);
4  extern double depiler(void);
5
6  int main(void){
7      double x ;
8      .....
9      empiler (50) ;
10     x = depiler();
11     return EXIT_SUCCESS;
12 }
```

Inclure pile.h

```
pile.c
1  #include "pile.h"
2
3  void empiler(double y){
4      ....
5  }
6
7  double depiler (){
8      ....
9  }
```

```
pile.h
1  #ifndef PILE_HEADER
2
3      #define PILE_HEADER
4      void empiler (double);
5      double depiler(void) ;
6  #endif
```

Compilation/ Compilation Séparée

- Illustration

Programme

```
main.c
1  #include <stdlib.h>
2  #include "pile.h"
3
4  int main(void) {
5      double x ;
6      .....
7      empiler (50) ;
8      x = depiler();
9      return EXIT_SUCCESS;
10 }
```

problème ...

```
pile.c
1  #include "pile.h"
2
3  void empiler(double y) {
4      ....
5  }
6
7  double depiler () {
8      ....
9  }
```

```
pile.h
1  #ifndef PILE_HEADER
2
3      #define PILE_HEADER
4      void empiler (double);
5      double depiler(void) ;
6  #endif
```

problème ... il faut le mot clé **extern** devant **empiler** et **depiler** car elles ne sont pas définies dans main.c

Compilation/ Compilation Séparée

- Illustration

Programme

main.c	pile.c	pile.h
<pre>1 #include <stdlib.h> 2 #include "pile.h" 3 4 int main(void){ 5 double x ; 6 7 empiler (50) ; 8 x = depiler(); 9 return EXIT_SUCCESS; 10 }</pre>	<pre>#define PILE #include "pile.h" void empiler(double y){ } double depiler (){ }</pre>	<pre>1 #ifndef PILE_HEADER 2 #define PILE_HEADER 3 #ifndef PILE 4 #define WHERE_PILE extern 5 #else 6 #define WHERE_PILE 7 #endif 8 WHERE_PILE void empiler (double); 9 WHERE_PILE double depiler(void); 10 #endif</pre>

PILE non définie
`extern void empiler(double);`
`extern double depiler(void);`

PILE définie
`void empiler(double);`
`double depiler(void);`

Arguments de la ligne de commandes

- Langage C offre des mécanismes qui permettent d'intégrer parfaitement un programme C dans l'environnement hôte
 - environnement orienté ligne de commande (Unix, Linux)
- Programme C peut recevoir de la part de l'interpréteur de commandes qui a lancé son exécution, une liste d'arguments
 - ⇔ ligne de commande qui a servi à lancer l'exécution du programme
- Liste composée
 - du nom du fichier binaire contenant le code exécutable du programme
 - des paramètres de la commande

Arguments de la ligne de commandes

- Fonction main reçoit tous ces éléments de la part de l'interpréteur de commandes
- 2 paramètres
 - argc (argument count)
 - nombre de mots qui compose la ligne de commande (y compris le nom de la commande qui a servi à lancer l'exécution du programme)
 - argv (argument vector)
 - tableau de chaînes de caractères contenant chacune un mot de la ligne de commande
 - argv[0] est le nom du programme exécutable

Arguments de la ligne de commandes

- Fonction main reçoit tous ces éléments de la part de l'interpréteur de commandes
- 2 paramètres
 - argc (argument count)
 - nombre de mots qui compose la ligne de commande (y compris le nom de la commande qui a servi à lancer l'exécution du programme)
 - argv (argument vector)
 - tableau de chaînes de caractères contenant chacune un mot de la ligne de commande
 - argv[0] est le nom du programme exécutable

```
int main (int argc, char *argv [ ]) {  
  
}
```

Arguments de la ligne de commandes

- Le programme peut aussi renvoyer un résultat à l'interpréteur de commandes
 - communiquer le bon ou mauvais déroulement de son exécution

fonctions	définies ds <code>stdlib.h</code>
<code>void abort (void)</code>	<ul style="list-style-type: none">• interrompt immédiatement un programme• est utilisée en cas de problème grave• l'implémentation doit décider si abort doit vider les zones tampon et fermer les fichiers éventuellement ouverts
<code>void exit (int etat)</code>	<ul style="list-style-type: none">• Interrompt un programme en vidant les zones tampon et les fichiers ouverts• Paramètre etat informe l'environnement hôte de l'état de terminaison du programme

Arguments de la ligne de commandes

- Paramètre **etat** de la fonction `exit`
 - sous Unix: commande retourne 0 lorsqu'elle a pu s'exécuter avec succès
 - Valeur différente de 0 sinon (valeur particulière pour chaque cause d'échec)
- Pour augmenter l'indépendance vis à vis d'un environnement particulier, la norme ANSI prévoit deux états au moins pour lesquels des constantes sont définies
 - `EXIT_SUCCESS` terminaison normale
 - `EXIT_FAILURE` terminaison anormale