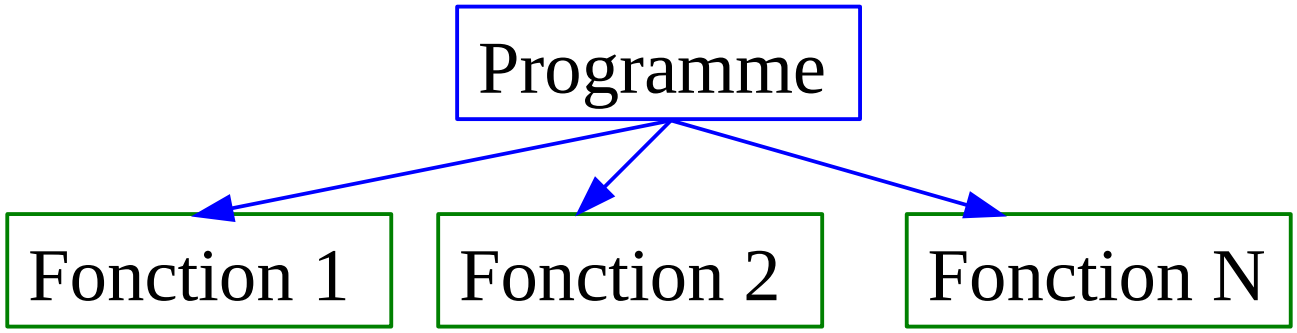
Programmation Impérative II

# Modularité et compilation séparée

minh-anh.tran@u-pec.fr

# Modularité




*relativement indépendantes*

- Intérêt ?
- Etapes pour la génération de l'executable ?
- Influence de ces étapes ?

# Modularité - Idées

Les logiciels sont souvent spécifiques à un type de problème donné. Pour chaque problème, faut-il tout refaire?

Les idées:

- Diviser le travail en plusieurs équipes.
- Créer des morceaux de programme indépendants, réutilisables.
- Eviter de reprogrammer ces morceaux à chaque fois.

Ces idées s'appliquent aussi aux programmeurs individuels. En effet, il est plus facile de décomposer un problème en ses éléments que de le traiter dans sa totalité.

# Modularité - Avantages

- Meilleure lisibilité
- Diminution du risque d'erreur
- Réutilisation des modules déjà existants
- Simplificité de l'entretien
- Favorisation du travail en équipe

# Modularité - Exemple

***Exercice 1 du TD 2.*** Afficher les caractères lus du clavier jusqu'au caractère \n sur une ligne dans l'ordre de lecture et sur la ligne suivante dans l'ordre inverse.

Exemple …

Idée. Découper le programme en sous-programmes indépendants.

- lire_une_ligne
- affiche
- affiche_inverse

# Modularité - Exemple

Idée. Découper le programme en sous-programmes indépendants.

- lire_une_ligne
- affiche
- affiche_inverse

```
int lire_une_ligne (char chaine[]) {...}
void affiche (char chaine[], int n) {…}
void affiche_inverse (char chaine[], int n) {…}
int main (void){
   char s[30];
   int n = lire_une_ligne(s);
   affiche (s, n); affiche_inverse(s, n);
   return 0;
}
```

# Modularité - Exemple

Pour un programme/projet de taille plus grande, on peut:

- Mettre chaque morceau dans un fichier séparé → ***fichiers sources***
- Pour pouvoir vérifier ces morceaux, il faut que le compilateur puisse les compiler indépendamment, sans avoir les autres fichiers du programme → ***compilation séparée***

# Modularité - Exemple

Pour un programme/projet de taille plus grande, on peut:

- Mettre chaque morceau dans un fichier séparé → ***fichiers sources***
- Pour pouvoir vérifier ces morceaux, il faut que le compilateur puisse les compiler indépendamment, sans avoir les autres fichiers du programme → ***compilation séparée***

Dans notre exemple, le code peut être découpé en 4 morceaux:
lire.c aff.c aff_inv.c prog.c

# Modularité

***Les phases du processus de génération des exécutables***

***Préprocesseur:*** traitement des fichiers sources avant compilation

Remplacement de macros, suppression de texte, inclusion de fichiers...

***Compilation séparée*:** compiler séparément les fichiers sources

Fichier source --> fichier en assembleur --> fichier objet
Fichier objet contient: la traduction du code assembleur en langage machine, et les données initialisées + les informations qui seront utilisées lors de la création du fichier executable

***Edition de liens*:** regrouper toutes les données, tout le code des fichiers objets, des bibliothèques, et résoudre des références inter-fichiers

Résultat: fichier image
Par exemple, des fichiers exectutables, des bibliothèques dynamiques

# Modularité

- Toutes ces opérations peuvent être regrouppées en une seule étape par le compilateur.
- Toutefois, il reste possible, à l'aide d'options spécifiques, de décomposer les différentes étapes et d'obtenir les fichiers intermédiaires.

***Problèmes:*** déterminer

1) L'ordre dans lequel les fichiers et les bibliothèques doivent être compilés ?

2) Les dépendances entre fichiers afin de pouvoir regénérer correctement les fichiers images après une modification des sources

# Modularité - Exemple

***Notre exemple.*** prog.c dépend des autres
--> l'ordre de compilation et voici le syntaxe du compilateur gcc:

gcc -c lire.c --> résultat: lire.o
gcc -c aff.c --> aff.o
gcc -c aff_inv.c --> aff_inv.o
gcc -c prog.c --> prog.o
gcc -o prog lire.o aff.o aff.o prog.o --> éxecutable prog

*Options:*

- -c le fichier sera compilé mais l'éditeur de liens ne sera pas appelé.
- -o spécifier le nom de l'executable

*Question:* retaper ces instructions à chaque fois ?

# Modularité

***Solution:*** programme **make** , syntaxe: **make**

***Principe.***

- **make** lit le fichier makefile, dans lequel se trouvent toutes les instructions nécessaires pour compiler un programme.
- puis les exécute si c'est nécessaire: Un fichier qui a été déjà compilé et qui n'a pas été modifié depuis ne sera pas recompilé.

***Remarque.*** **make** se base sur les dates de dernière modification des fichiers pour savoir s'ils ont été modifiés (il compare les dates des fichiers sources et des fichiers produits). La date des fichiers est gérée par le système d'exploitation: il est donc important que l'ordinateur soit à l'heure.

# Modularité - Exemple

***Notre exemple.*** makefile

```
# Le cible et ses dependances
prog: lire.o aff.o aff_inv.o prog.o
	gcc -o $@ lire.o aff.o aff_inv.o prog.o

lire.o: lire.c
	gcc -c lire.c
aff.o: aff.c
	gcc -c aff.c
aff_inc.o: aff_inv.c
	gcc -c aff_inv.c
prog.o: prog.c
	gcc -c prog.c
```

```
# commentaires

cible: prog
dépend de lire.o ...
(à compiler après ses
dépendances)

lire.o dépend de lire.c
...
```

# Compilation séparée en C

*Deux types de fichiers:*

la déclaration de la définition des symboles du programme.

- Fichiers d'en-tête **.h** : toutes les déclarations communes à plusieurs fichiers sources.
- Fichiers **.c** : les codes en langage C.

# Compilation séparée en C

*Déclaration des variables:* Dans un fichier .c (code), les variables qui sont définies dans un autre fichier doivent être déclarées avant leur première utilisation. Exemple:

```
extern int i;
```

*Déclaration des fonctions:*

```
void affiche_inverse (char s[], int n) ;
```