Programmation en C
Esisar - CS210

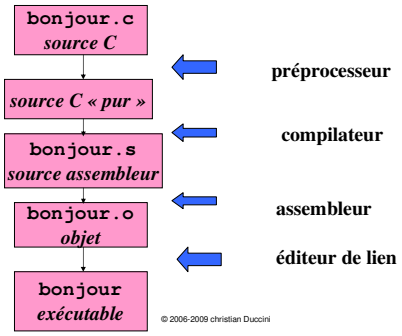
Compilation et édition de liens, exemples de base.

Exemple

Programme affichant sur l 'écran le message :

```
Esisar
bonjour !
```


Compilation du programme simple



Codage C

```
#include <stdio.h>
#include <stdlib.h>
/*déclaration de référence de «dialogue»*/
static void dialogue(void) ;
/*définition de « main»*/
int main(void){
    dialogue() ;
    return EXIT_SUCCESS ;
    }
 /*définition de «dialogue»( ici ou ailleurs!)*/
void dialogue(void)
    {
    printf("Esisar\n") ;
    printf("bonjour !\n") ;
    }
```

Codage C (multi-fichiers)

Le programme écrit précédemment en 1 seul fichier peut être également éclaté en 3 fichiers : « util.h », « bonjour2.c » et « util.c »

fichier « `util.h` » :

```
#ifndef UTIL_H
#define UTIL_H
extern void dialogue(void) ;
#endif
```


Codage C (multi-fichier)

fichier « `bonjour2.c` » :

```
#include <stdlib.h>
#include "util.h"
int main(void)
      {
      dialogue() ;
      return EXIT_SUCCESS ;
      }
```


Codage C (multi-fichier)

fichier « **util.c** » :

```
#include <stdio.h>
#include "util.h"
void dialogue(void)
      {
      printf("Esisar\n") ;
      printf("bonjour !\n") ;
      }
```

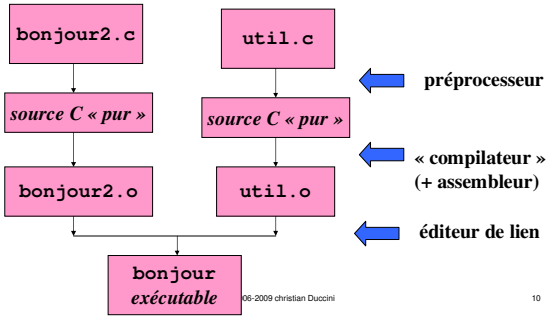

Déclaration de référence, prototype

- Le fichier " **bonjour2.c** " utilise une déclaration de référence (ou prototype, ou signature) de la fonction **dialogue**.
- La (déclaration de) définition de cette fonction étant dans un autre fichier (" **util.c** "), il y a un risque d'incohérence car le compilateur traitera (compilera) chacun des fichiers séparément, éventuellement à des dates (instants) très différents.

Déclaration de référence, prototype

- Il faut alors placer le prototype de la fonction **dialogue** dans un fichier " **util.h** ", et inclure ce fichier *simultanément* dans les deux fichiers sources " **bonjour2.c** " et " **util.c** " pour que le compilateur effectue les contrôles de cohérence.
- L'inclusion du fichier " **util.h** " dans " **util.c** " est *indispensable*, elle permet d'assurer que la déclaration (de référence) de la fonction **dialogue** reste cohérente avec sa (déclaration de) définition.

Compilation séparée



Commandes de compilation (un fichier)

- Commande de base (compilation + édition de lien)

```
$ cc -g -Wall -pedantic bonjour.c -o bonjour
```

- Commentaires :
  - nom commande : **cc** ou **gcc**
  - **bonjour.c** : nom du fichier source à compiler
  - **-o** : option pour indiquer le nom de l'exécutable (par défaut génère un exécutable de nom "a.out")

Commandes de compilation

- options OBLIGATOIRES (je les exige!)
  - **-g** : option pour générer les symboles pour le debug pour « gdb » à la compilation
  - **-Wall** : option d'activation du niveau maximum des messages d'erreurs du compilateur
  - **-pedantic** : conformité "C" standard (enfin presque)

Commandes de compilation (compilation séparée)

- Compilation séparée des 2 fichiers :

```
$ cc -c -g -Wall -pedantic bonjour2.c
$ cc -c -g -Wall -pedantic util.c
```

l'option « **-c** » indique qu 'on appelle le compilateur, mais pas l'éditeur de liens : chacune des commandes va générer le fichier objet « .o » correspondant.

- Edition de liens (pas de « **-c** » !):

```
$ cc bonjour2.o util.o -o bonjour
```


Commandes de compilation (autres formes)

- Compilation+ édition de liens :

```
$ cc -g bonjour2.c util.c -o bonjour
```

appelle l'éditeur de lien, sans laisser les objets intermédiaires (fichiers " **.o** ") et nomme l'exécutable " **bonjour** " (option « -o »)

- idem, mais différent :

```
$ cc -g bonjour2.c util.o -o bonjour
```

(même résultat, « **util.o** » reste)

- et plus, si affinités :

```
$ man cc
```


Programmation en C
Esisar - CS210

Outils de programmation en C, make, débugger.

Développement logiciel et outils

Pour simplifier, un développement logiciel peut être décomposé en 3 phases :

- Analyse du problème
- Programmation (codage)
- Tests

Phase analyse :

C'est la partie la plus importante du développement logiciel. Dans le cadre de petits programmes (par exemple : mini-projets, TP…), cette phase consiste essentiellement en découpe fonctionnelle ou modulaire du programme, et réalisation d'algorithmes. Les outils pour la phase 1 sont :

- Crayon, papier, gomme ! ☺ ☺ ☺
- Outils de modélisation :
  - UML ☺
  - Merise, etc…

Phase codage :

L'analyse étant faite, il faut la réaliser (« l'implémenter ») dans le langage et sur le système choisi en phase d'analyse. Cette phase (rélativement mécanique) se décompose en 3 phases:

- Φ C1 : édition
- Φ C2 : compilation et édition de liens
- Φ C3 : tests unitaires, tests élémentaires

Phase tests :

Afin de garantir un bon fonctionnement du programme au « client » (par exemple, le prof…), il est indispensable de réaliser un certain nombre de tests : tests unitaires, tests d 'intégration, tests en charge, etc…

- outils pour cette phase :
  - batterie de tests
  - générateurs de traces
  - simulateurs

outils Φ C1 : éditeurs de texte

- fonctionnalités nécessaires : celles d 'un éditeur de texte « classique » + connaissance syntaxique du langage :
  - édition multi-fichiers,
  - coloration syntaxique,
  - gestion de l 'indentation, etc…
- exemples :
  - vi/vim : disponibles sur tous les OS, petite taille
  - nedit, gedit, kwrite : ☺

outils Φ C2 : make

- Problématique :
  Les diverses opérations à enchaîner pour reconstituer l 'exécutable peuvent être nombreuses (et donc pénibles) si l 'exécutable final est constitué de beaucoup de fichiers.
- Solution :
  - utilitaire unix « make »

Biblio « *make* »

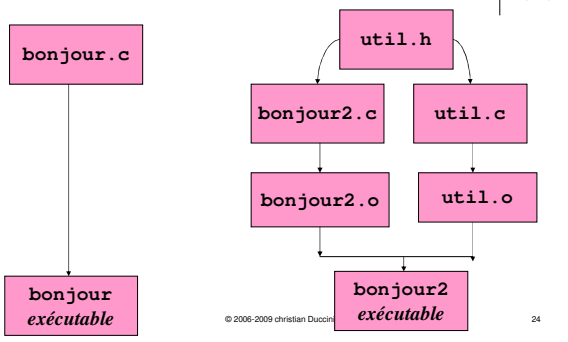
- « *managing projects with make* » Oram/Talbott
- « *running linux* » Welsh/Dalheimer/Kaufman
- « *linux in a nutshell* » Siever
- site « gnu »
- manuel « *on line* » :

```
$ man make
```


make

- Il faut décrire une liste de dépendances temporelles dans un fichier ("**`makefile`**" *ou* "**`Makefile`**") du répertoire courant.
- Le but essentiel est de faire reconstruire un projet/un fichier par étapes (par exemple pour éviter de tout recompiler…) .
- " **`make`** " est un utilitaire très général qui construit des " cibles " (*targets*) à partir de " préalables " (*prerequisites*).
- Les " cibles " peuvent aussi bien être des exécutables que des pages Webs,….

Exemple pour TP «Esisar Bonjour !»



Un « makefile » (un peu simple) pour TP « IUT Bonjour !»

```
bonjour : bonjour.c
     gcc -g -Wall -pedantic bonjour.c -o bonjour
bonjour2 : bonjour2.o util.o
     gcc bonjour2.o util.o -o bonjour2
util.o : util.c util.h
     gcc -c -g -Wall -pedantic util.c
bonjour2.o : bonjour2.c util.h
     gcc -c -g -Wall -pedantic bonjour2.c
clean :
     rm *.o bonjour bonjour2 bonjour*.exe
```


Exemples d 'utilisation

`$ make bonjour2`

- « bonjour2 » dépend de bonjour2.o et util.o (dépendance 4) : Il doit être plus récent qu 'eux! (utiliser « **`ls -ltr`** ») : si ce n 'est pas le cas, « make » va provoquer la ligne indiquée après la dépendance.
- Néanmoins la vérification temporelle est récursive : « make » va donc avant s 'assurer que « bonjour2.o » et « util.o » sont eux-mêmes postérieurs aux fichiers dont ils dépendent.
- La cible « clean » est factice : un « make clean » va provoquer l 'effacement des fichiers indiqués.

Principes d 'élaboration

- Lors de la réalisation d 'un « makefile », on partira du fichier terminal (en général l 'exécutable), pour remonter jusqu 'aux fichiers « sources C ».
- Les fichiers « headers » (« .h ») écrits par le programmeur doivent être intégrés aux dépendances temporelles (au même niveau que tous les fichiers « .c » dans lesquels ils sont inclus.
- Par contre on considère en général que tous les fichiers « header » standard sont installés une fois pour toute en même temps que le compilateur et donc qu' ils ne rentrent pas dans les définitions des dépendances.

syntaxe de « make »

- L'ordre des cibles n'a aucune importance, car " make " regarde dans toutes les dépendances définies, néanmoins la commande *"***`make`***"*est équivalente à la commande *"***`make target1`***"*.
- Les lignes de commandes doivent commencer par une tabulation.
- On peut commencer un commentaire par un signe dièse (**`#`**), tout le reste de la ligne est alors ignoré.
- Une ligne peut être étendue à l'aide du " **`\`** ".
- Pour éviter de répéter des chaînes de caractères, on utilise des « macros ».

Exemple avec macros (1)

- Avec les macros « **`CC`** » et « **`CFLAGS`** » :

```
CC = gcc -c
CFLAGS = -g -Wall -pedantic
```

- Les dépendances de **`bonjour2.o`** et **`util.o`** deviennent :

```
util.o : util.c util.h
  $(CC) $(CFLAGS) util.c
bonjour2.o : bonjour2.c util.h
  $(CC) $(CFLAGS) bonjour2.c
```


Exemple avec macros (2)

- les macros peuvent être étendues :

```
ifdef ANSI_WANTED
   CFLAGS := $(CFLAGS) -ansi
endif
```

- la macro **`ANSI_WANTED`** peut être définie dans le " **`makefile`** " lui-même :

  (**`ANSI_WANTED=oh, oui !`**)

- ou lors de l'invocation dans le shell :

```
$ make ANSI_WANTED=yes bonjour
```

Macros internes

- $@ : nom de la cible courante
- $< : nom du préalable qui a été changé plus récemment que la cible courante.
- $* : nom du préalable sans le suffixe qui a été changé plus récemment que la cible courante.
- etc.. (voir « **man** »)

Règles de suffixe, *Pattern Rules*

- Elles sont utilisées pour simplifier la définition des dépendances, ainsi :

```
.c.o :
    $(CC) $(CFLAGS) $<
```

indique qu'un fichier " `.o` " nécessite un fichier " `.c` " pour préalable.

- De même, le signe " `%` ", qui signifie " n'importe quelle chaîne " créé un *pattern* (schéma) :

```
%.o :%.c
    $(CC) -o $@ $(CFLAGS) $<
```

ou encore

```
%.o :%.c
    $(CC) -o $*.exe $(CFLAGS) $<
```


« makefile » (un peu moins simple) pour TP « IUT Bonjour »

```
CC = gcc -c
CFLAGS = -g -Wall -pedantic
LD = gcc
SOURCES=bonjour2.c util.c util.h
OBJ=bonjour2.o util.o
bonjour : $(OBJ)
      $(LD) $(OBJ) -o bonjour
.c.o :
      $(CC) $(CFLAGS) $<
bonjour.c : util.h
      touch $@
util.c : util.h
      touch $@
clean :
      rm *.o bonjour
print :
      cat $(SOURCES) > sources.txt
      lp sources.txt
      rm sources.txt
```


« make » : en guise de conclusion

- Rapidement, une fois les options de compilation choisies, il suffira de définir la liste des dépendances temporelles entre fichiers, et d'écrire le « makefile » permettant d'automatiser la phase de compilation.
- NB : il existe un outil « automake » …

Problématique Φ C3

- Le fait que la compilation/édition de lien ne délivre plus aucun message d 'erreur signifie uniquement que le compilateur/éditeur de lien est arrivé à traduire le texte-source « C » en langage machine pour faire un exécutable.
- Mais le traitement algorithmique peut encore être complètement ou partiellement erroné (ex : boucle infinie).

Problématique Φ C3

- Le problème va alors se manifester lors d 'un essai d 'exécution par :
  - plantage franc et massif (écran bleu *M$*, « core dump » Unix)
  - « résultats » fournis systématiquement incorrects
  - dysfonctionnements systématiques dans certains cas
  - dysfonctionnements aléatoires

Outils Φ C3

- Pour corriger les erreurs élémentaires :
  - re-lecture du code et/ou « eXtrême Programming »
  - utilisation d 'un débogueur (*debugger, dévermineur*)
- Principe d 'un debugger :
  - permet au programmeur d 'exécuter son programme en observant le flux des instructions, l 'état du contexte d 'exécution, etc…
  - équivalent pour l 'informaticien de l ' oscilloscope de l 'électronicien
- debuggers « C » :
  - gdb : en mode texte
  - kgdb, ddd : en mode graphique

Debugger : fonctionnalités de base

- contrôle de l 'exécution :
  - Pas à pas (*step*)
  - Par traitement (*next*)
  - Points d'arrêt (*breakpoint*)
  - Arrêt de l'exécution sur critères
- examen de l 'état du contexte d 'exécution :
  - Examen du contenu des variables
  - Examen de la mémoire, de la pile
  - Modification des variables / du code
- Mise en œuvre de *ddd*
  - Compilation/édition de liens avec l'option « –g »

outils Φ C1+C2+C3 : environnements intégrés

- Lors du codage d 'un programme, les phases 1,2 et 3 sont effectuées dans un délai court, d 'où l 'intégration dans une « chaîne de développement » des divers outils indépendants précédents. Ex :
  - Kdevelop (linux)
  - Visual XX (*M$*)
  - Eclipse, Jgrasp (logiciel libre compatible linux/M$), mais nécessitent l 'installation sous M$ de l 'environnement Java et de compilateurs C

