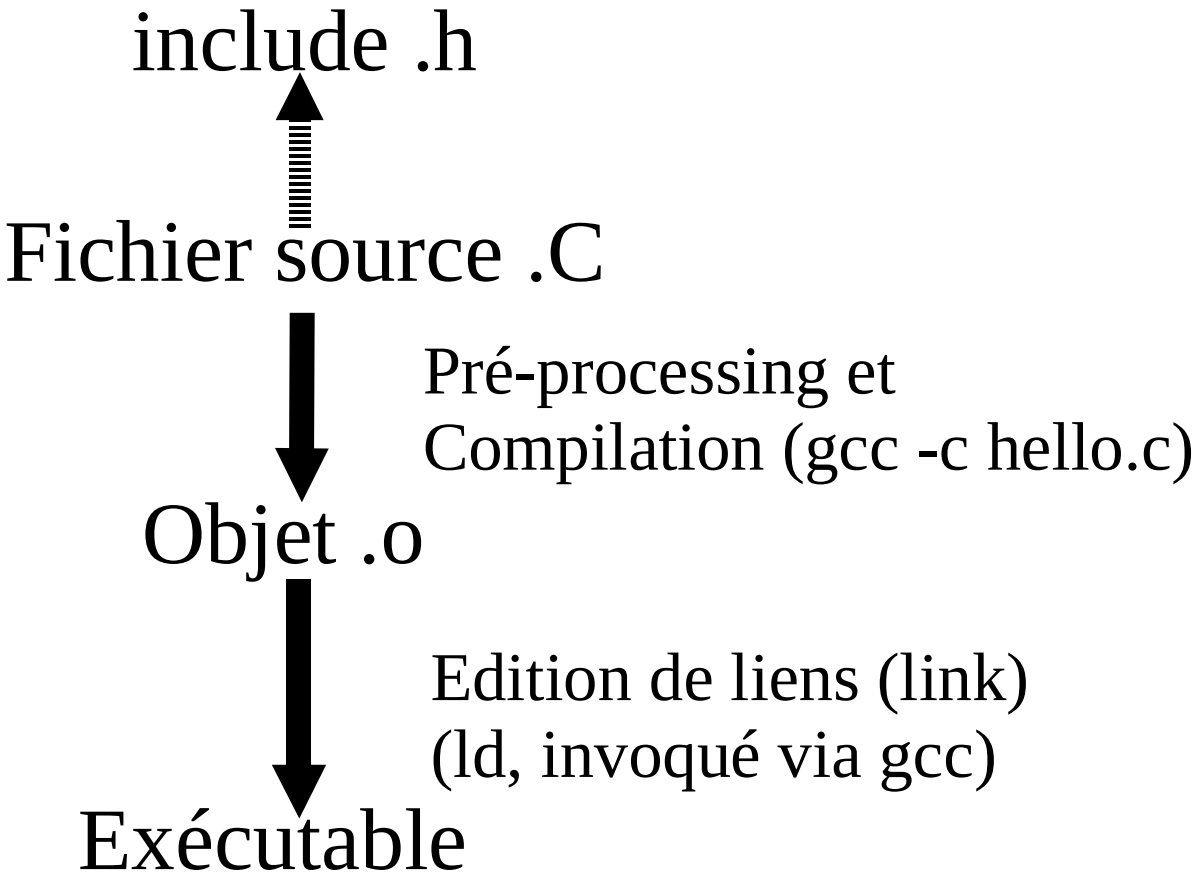
Notions de compilation, Compilation séparée, Architecture logicielle

...

Ou l'art de décomposer un projet en programmes et en fichiers

Du source à l'exécutable (Cas trivial, le "visible")



Organisation des fichiers (rappel ?)

- Dans les includes (.h) ou header
  - Protection contre les inclusions multiples (`#ifndef`)
  - Prototype des fonctions (`void fonction(int);`)
  - Définition des variables globales exportées (`extern int varGlobale`)
  - Types (`struct cplx { double x; double y;}`)
- Dans les sources (.c)
  - Corps des fonctions (**static** ou pas)
  - Déclarations des variables globales (`int varGlobale;`)

Ce que fait un compilateur

- Vérification de la **syntaxe** (Accolades, mots-clefs, respect de la grammaire)
- Vérification de la **correspondance des types**
- Donc besoin de
  - Prototypes des fonctions,
  - Déclarations de type,
  - Type des variables
- Mais pas du corps des fonctions appelées.
- Génération de code machine pour les fonctions définies localement

Ce qui se passe réellement

1) "Compilation" au sens large
    1) Pré-processeur
    2) Compilateur
    3) Assembleur
2) Edition de liens

Pré-processing

- Remplacement de chaînes de caractères ds un prog.
- En C, tout ce qui commence par "#"
  - #include: Inclusion de fichier
  - #define: Remplacement de chaînes de caractère (+macros)
  - #pragma: Directives aux compilateur
  - #ifdef: Test
- Effectué par gcc
  - On peut voir le résultat par "gcc -E monprog.C"

Compilation

- Traduit un langage de programmation "haut niveau" en assembleur.
- résultat visible par "gcc -c -s monProg.c

```
void toto() {
    int i=0;
    while(i<20)
        i++;
}
```

```
.globl toto
        .type    toto,@function
toto:
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp
        movl $0,-4(%ebp)
        .p2align 4,,7
.L3:
        cmpl $19,-4(%ebp)
        jle .L5
        jmp .L4
        .p2align 4,,7
.L5:
        incl -4(%ebp)
        jmp .L3
        .p2align 4,,7
.L4:
.L2:
        leave
        ret
```

Assemblage

- Traduction de l'assembleur en 'code machine'
  - instruction assembleur=remprésentation sous forme de chaîne de caractère
  - instruction machine=représentation sous forme d'un numéro
- Résolution des labels
- Fait par gcc
- Produit un "fichier objet", de suffixe .o

Symboles et fichier ".o"

```
int pasBeauGlobal;
void lapin();
void toto() {
        int i=0;
        while(i<20) i++;
        lapin();}
```

```
00000000 t   gcc2_compiled.
         U lapin
00000004 C pasBeauGlobal
00000000 T toto
```

- "nm" permet de lister les symboles d'un fichier objet.
  - usage: nm toto.o
  - U: "Undefined". Symbole utilisé dans le .o, mais non définit dans celui ci. Doit se trouver dans un autre .o
  - Autres lettres: CF manuel de nm

L'édition de lien

- Construit un exécutable à partir de fichiers objets
- Résolution des symboles
  - Remplacer les noms de symboles par des adresses
  - Vérifier que tous les symboles utilisés sont définis une fois et une seule
- Entrée du programme (main...)
- Progamme: "ld"
  - implicitement appelé par gcc dans (eg. "gcc *.o")

Qu'est ce qu'un .o?

- .o ou .obj, objets
- Ne peut être exécuté directement par la machine
- Les fonctions sont présentes sous forme de langage machine
- Le nom des symboles (fonctions, variables) est préservé

Conseils de programmation C

- Mettre dans les include le minimum de chose:
  - Uniquement ce qui est mis à disposition des autres programmeurs
  - Uniquement des prototypes et définitions
  - Inclure uniquement si nécessaire (dans le .C si possible).
  - Protection contre les inclusions multiples (#ifndef)

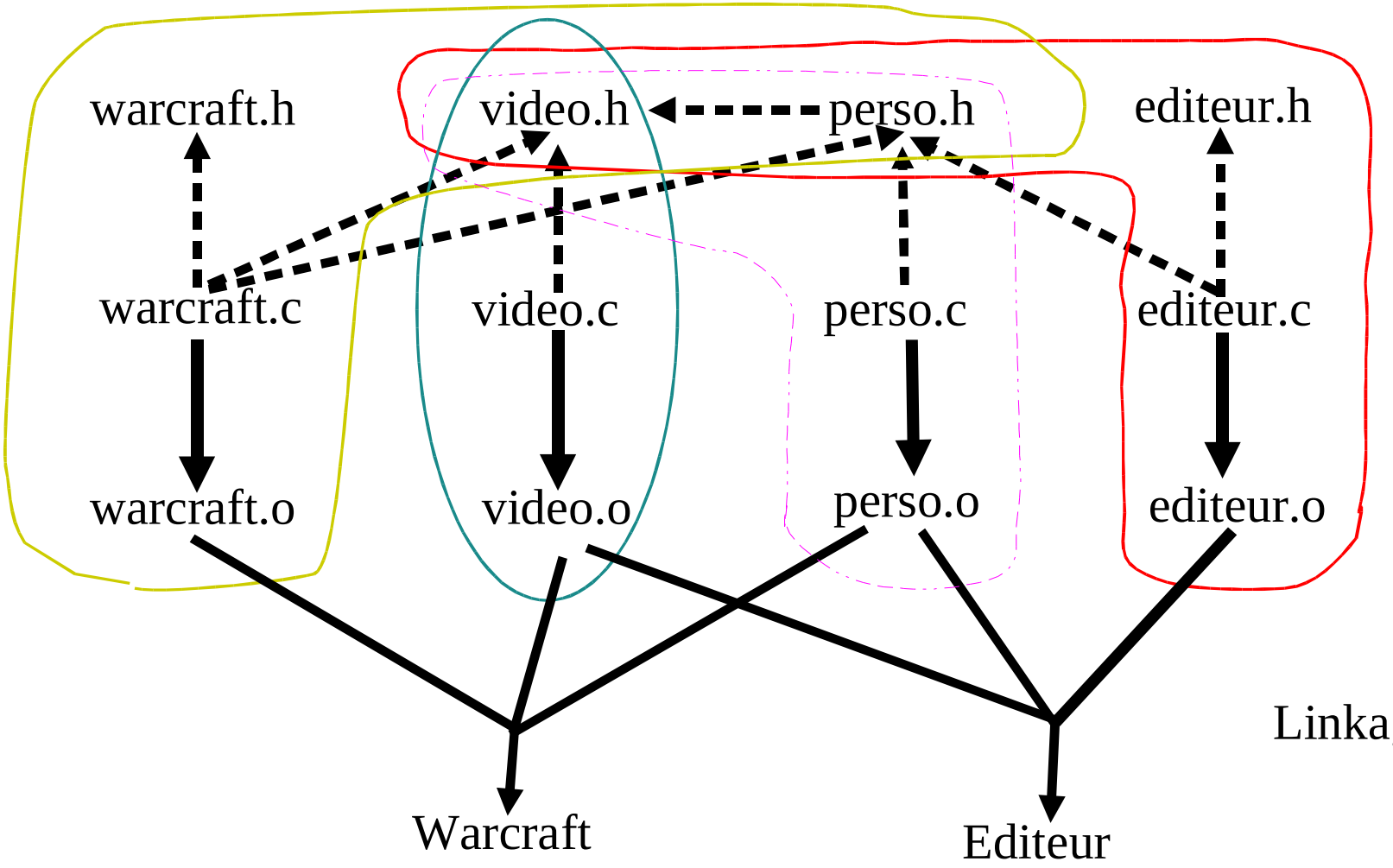
Problématique pratique de la compilation

- Gérer des projets de taille importante
- Exemple: un noyau unix (linux 2.4.2)
  - 3352 fichiers .c (2.5 millions lignes)
  - 4010 fichiers .h (650 000 lignes)
  - Produit 1 exécutable (714 ko) + modules (.o)
- Favoriser la réutilisabilité du code (plus lisible, plus modulaire)
- Limiter les temps de compilation

Exemple (peu) pratique

- Programmes principaux:
  - warcraft.c warcraft.h: corps du jeu (contient une fonction 'main()')
  - editeur.c editeur.h: corps de l'éditeur de niveau (contient une fonction 'main()')
- Utilitaires:
  - video.c video.h: ensemble de routines d'affichage
  - perso.c perso.h: gestion des personnages du jeu

Structure des programmes: Cas général



Notion de dépendances

- Liste des dépendances
- A dépend de B ⇔ Si A est modifié, alors il faut refaire (recalculer) B
- Version linéaire du graphe!
- Lourd à gérer
- Mais permet de minimiser les temps de compilation
- Que compiler, dans quel ordre?

Dépendances

- Que dois-je faire (recompiler, lier) si je modifie les fichiers suivants:
  - perso.h?
  - video.c?
  - warcraft.c?
  - video.h?
  - editeur.h?

Le Makefile

- But: Recompiler le strict nécessaire, uniquement quand c'est nécessaire!
- Dépendances
  - `warcraft.o: warcraft.c warcraft.h video.h perso.h`
- Règles
  - `gcc -c warcraft.c -o warcraft.o`
- Syntaxe générale
  - `cible: dépendances`

```
    règle 1
    règle 2
```

Un Makefile élémentaire

```
#Compilations
perso.o: perso.c perso.h video.h
    gcc -c perso.c -o perso.o
video.o: video.c video.h
    gcc -c video.c -o video.o
warcraft.o: warcraft.c warcraft.h video.h perso.h
    gcc -c warcraft.c -o warcraf.o
editeur.o: editeur.c editeur.h video.h perso.h
    gcc -c editeur.c -o editeur.o

#Editions de lien
warcraft: warcraft.o video.o perso.o
    gcc  warcraft.o video.o perso.o -o warcraft
editeur: editeur.o video.o perso.o
    gcc editeur.o video.o perso.o -o editeur
```

Makefile: Astuces

- Raccourcis:
  - `$@` : nom de la cible
  - `$<` : première dépendance
  - `$^` : toutes les dépendances
  - `$?` : les dépendances plus récentes que la cible
- Variables (presque comme en shell):
  - `COMPILATEUR = gcc` (affectation)
  - `$(COMPILATEUR) -c warcraft.c`
- Les cibles ne sont pas forcément des fichiers

Makefile: Exemple

```
CC    =   gcc
all: warcraft editeur
perso.o: perso.c perso.h video.h
    $(CC) -c $< -o $@
video.o: video.c video.h
    $(CC) -c $< -o $@
warcraft.o: warcraft.c warcraft.h video.h perso.h
    $(CC) -c $< -o $@
editeur.o: editeur.c editeur.h video.h perso.h
    $(CC) -c $< -o $@
warcraft: warcraft.o video.o perso.o
    $(CC) $^ -o $@
editeur: editeur.o video.o perso.o
    $(CC) $^ -o $@
clean:
    rm -f *.o editeur warcraft
```

Utilisation du Makefile

- Appel par l'utilitaire Make:
  - `make all:` construit tous les programmes
  - `make:` construit la première cible rencontrée
  - `make clean:` appelle la règle qui supprime les « .o » et les exécutables
  - `make warcraft.o:` construit l'objet warcraft.o
- Pour les curieux: « man makedepend »...
  - permet de générer automatiquement la liste des dépendances

Règles génériques (Patterns)

```
CC =    gcc
CFLAGS   =    -Wall

all: warcraft editeur

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

perso.o: perso.c perso.h video.h
video.o: video.c video.h
warcraft.o: warcraft.c warcraft.h video.h perso.h
editeur.o: editeur.c editeur.h video.h perso.h
warcraft: warcraft.o video.o perso.o
editeur: editeur.o video.o perso.o
```

Variables globales

- Usage **exceptionnel**!
- Si doit être exportée:
  - `extern int varGlobale` dans le header (n'alloue pas la variable mais la définit)
  - `int varGlobale` dans le .C
- Si elle est interne au programme:
  - Uniquement dans le .C
  - `static int varGlobale;`
  - Cache la variable au reste du monde

Makefile: usages surprenants

```
archive: Makefile *.c *.h
	tar cf sauvegarde.tar *.c *.h Makefile
	gzip sauvegarde.tar

depend:
	makedepend -- $(CFLAGS) -- $(SRCS)
```