



Programmation Impérative Renforcée

Compilation séparée

L2 – Double Licence Math/Info

Antoine Spicher

`antoine.spicher@u-pec.fr`

Compilation avec gcc

■ Compilation par étapes

□ Préprocesseur ou précompilation

- Langage cible : C sans `#include`, `#define`, ...

- Ligne de commande : `> gcc -E helloworld.c -o helloworld-nopp.c`

□ Compilation vers un assembleur

- Langage cible : Assembleur (dépend de votre architecture)

- Ligne de commande : `> gcc -S helloworld-nopp.c -o helloworld.S`

□ Assemblage/édition des liens

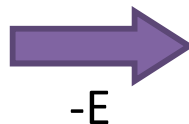
- Langage cible : langage machine (code binaire, édition des liens)

- Ligne de commande : `> gcc helloworld-nopp.S -o helloworld.exe`

helloworld.c

```
#include <stdi
#define RET 0

int main() {
    printf("hell
    return RET;
}
```



-E

helloworld-nopp.c

```
int printf(con

int main() {
    printf("hell
    return 0;
}
```



-S

helloworld.S

```
_main:
    leal    4(%esp
    and     $-16,
    pushl   -4(%ec
    pushl   %ebp
    movl    %esp,
    pushl   %
```



helloworld.exe

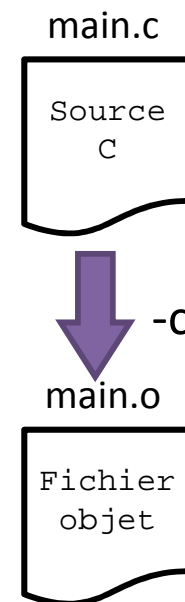
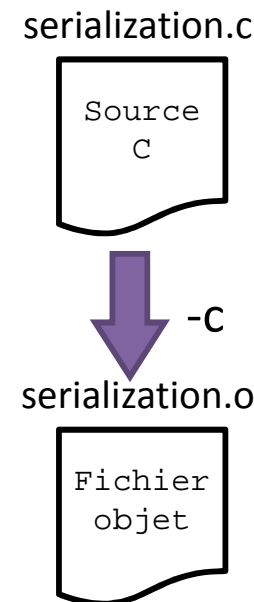
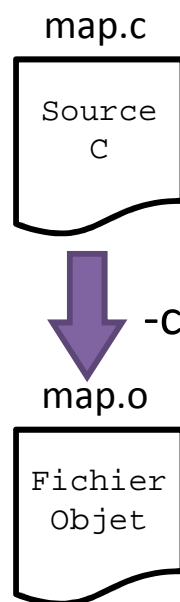
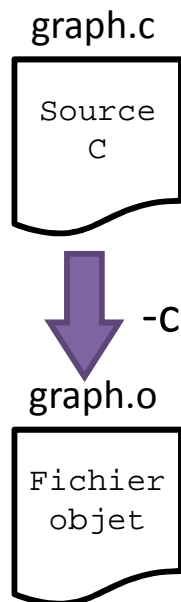
```
00010100100
10101010001
01010101010
11010010101
01001010101
00100
```

Compilation avec gcc

■ Compilation de plusieurs fichiers

- Création d'un *fichier objet* (i.e., binaire partiel) pour chaque fichier source
Précompilation + compilation + assemblage (**pas d'édition des liens**)

```
> gcc -c graph.c -o graph.o
```

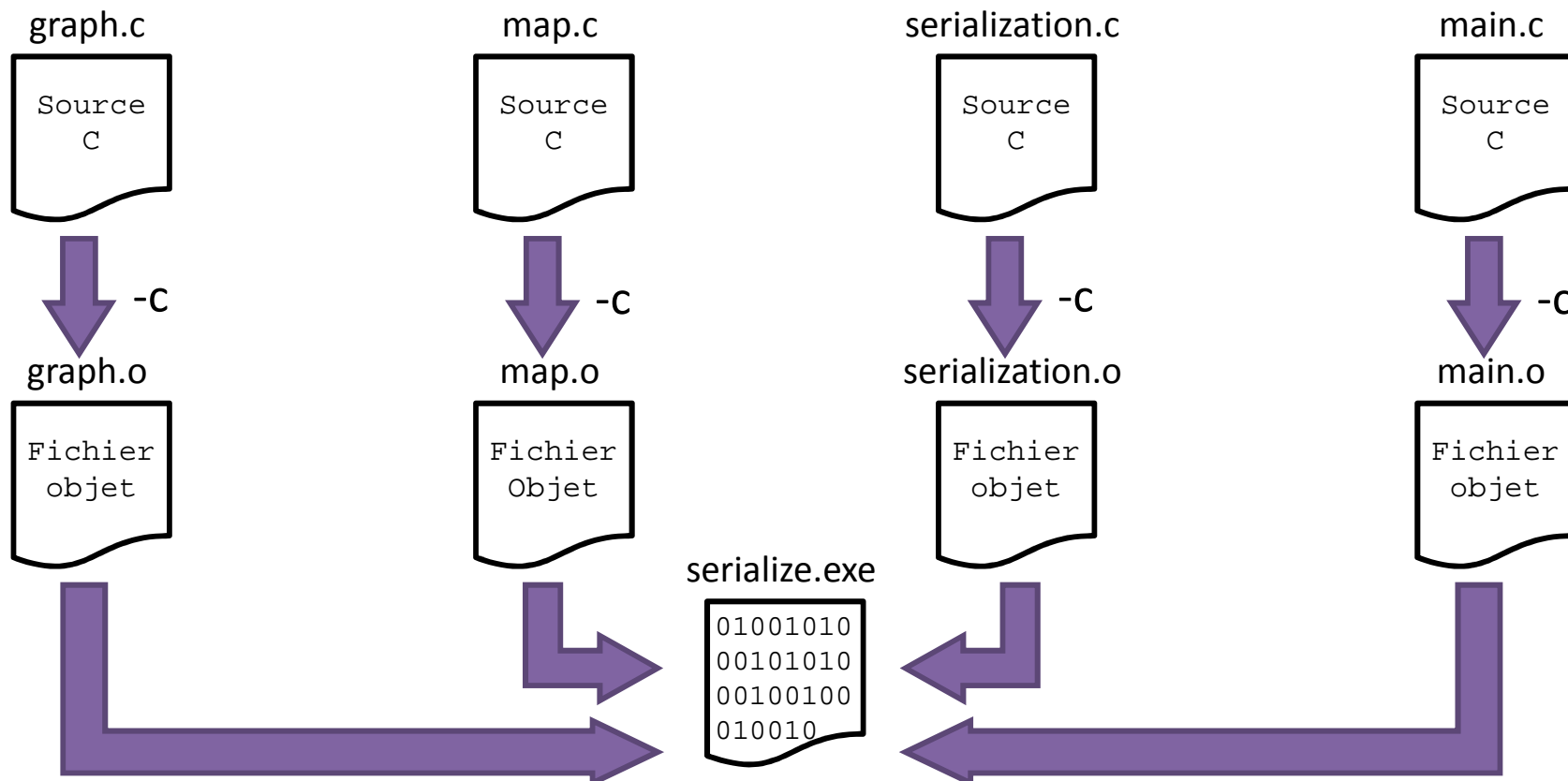


Compilation avec gcc

■ Compilation de plusieurs fichiers

- ❑ Création d'un *fichier objet* (i.e., binaire partiel) pour chaque fichier source
Précompilation + compilation + assemblage (**pas d'édition des liens**)
- ❑ *Édition des liens* entre les fichiers objets

```
> gcc -o serialize.exe graph.o map.o serialization.o main.o
```



Plan du cours : compilation séparée



- Précompilation en C
- Makefile
- Programmation modulaire

Précompilation en C

■ Précompilation

□ Définition

« Étape de *transformation d'un programme* avant la compilation (ou l'interprétation) proprement dite »

□ Préprocesseur *lexical*

« Type de précompilateur qui ne requiert qu'un mécanisme d'*analyse lexicale* et qui procède par *substitution* de sous-partie d'un programme à la syntaxe bien déterminée »

■ CPP (**C PreProcessor**)

□ Préprocesseur de C

□ *Substitution jusqu'à un point fixe* reposant sur 3 constructions syntaxiques

■ L'*inclusion* de fichier : `#include ...`

■ La définition de *macros* : `#define ...`

■ La *précompilation conditionnelle* : `#if(n)def ... #else ... #endif`

Précompilation en C

■ Inclusion de fichier

□ Syntaxe

```
#include "graph.h"

ou

#include <graph.h>
```

□ Effet

■ Cherche le fichier spécifié

- dans le répertoire courant si les guillemets sont utilisés
- dans les *répertoires standards* si les « <...> » (e.g., /usr/include)

option gcc d'ajout d'un répertoire d'inclusion : -Irep

```
> gcc -I/tmp/ -c test.c -o test.o
```

■ Remplace la ligne par le contenu du fichier spécifié

■ Attention à la récursivité

```
#include "test.h"      test.c
int main() { return 0; }
```

```
#include "test.h"      test.h
```

Précompilation en C

■ Définition de macros

□ Syntaxe

```
#define TOTO  
ou  
#define M_PI 3.1415  
ou  
#define max(x,y) (x>y?x:y)
```

□ Effet

■ Cherche toutes les occurrences dans la suite du programme

- 1^{er} cas : les remplace par rien (utilisation liée au `#ifdef`)
- 2^{ème} cas : les remplace par la valeur associée
- 3^{ème} cas : les remplace par l'expression associée avec les arguments

```
int i = max(0,x);
```



```
int i = (0>x?0:x);
```

■ Attention aux effets non désirés

```
#define error(mess,code) fprintf(stderr,"err: %s\n",mess); exit(code)
```

```
if ((ptr = malloc(sizeof(int)))==NULL) error("not enough memory",1);
```


Précompilation en C

■ Précompilation conditionnelle

□ Syntaxe

```
#ifdef DEBUG
... /* then */
#else
... /* else */
#endif
```

et

```
#ifndef DEBUG
... /* then */
#else
... /* else */
#endif
```

□ Effet

- Si la macro `DEBUG` est définie (resp. n'est pas définie), conserver uniquement les lignes de la branche « *then* », sinon conserver uniquement la branche « *else* »
- La branche « *else* » est optionnelle
- Exemple d'utilisation

```
#ifdef DEBUG
#define test(cond) { if (cond) error("assertion fails",2); }
#else
#define test(cond)
#endif
```

```
> gcc -DDEBUG -c test.c -o test.o
```

Plan du cours : compilation séparée



- Préprocessing en C
- Makefile
- Programmation modulaire

Makefile

■ Objectif du programme make

- ❑ Automatiser la compilation/construction d'un projet
- ❑ Ne recompiler que les parties modifiées d'un projet

■ Principe de fonctionnement

- ❑ Décrire les règles de construction dans un fichier Makefile
- ❑ Spécifier une règle revient à
 - Spécifier un fichier cible, i.e., qui doit être produit
 - Spécifier les fichiers dont dépend la construction de la cible
 - Spécifier la commande de construction

```
target: source1 ... sourceN  
    build -o target source1 ... sourceN
```

- ❑ Déterminer les dépendances entre les règles et en déduire une procédure de calcul

Makefile

■ Ecriture d'un Makefile

- Ecriture des règles de production les unes à la suite des autres

```
graph.o: graph.c graph.h
    gcc -o graph.o -c graph.c

map.o: map.c map.h
    gcc -o map.o -c map.c

serialization.o: serialization.c serialization.h map.h graph.h
    gcc -o serialization.o -c serialization.c

main.o: main.c serialization.h graph.h
    gcc -o main.o -c main.c

serialize.exe: graph.o map.o serialization.o main.o
    gcc -o serialize.exe graph.o map.o serialization.o main.o
```

Makefile

- Commande pour lancer la compilation

```
> make serialize.exe
```

Makefile

■ Ecriture d'un Makefile

- Ajout de règles ne correspondant pas à la génération de fichiers

```
.PHONY: default clean

default: serialize.exe

graph.o: graph.c graph.h
    gcc -o graph.o -c graph.c

...

serialize.exe: graph.o map.o serialization.o main.o
    gcc -o serialize.exe graph.o map.o serialization.o main.o

clean:
    rm serialize.exe graph.o map.o serialization.o main.o
```

Makefile

- La première règle est la règle appelée par défaut

- Pour compiler le projet : `> make`
- Pour nettoyer le projet : `> make clean`

Makefile

■ Ecriture d'un Makefile

- Utilisation de variables pour rendre le fichier plus « générique »

```
CC=gcc
CFLAGS=-W -Wall
EXEC=serialize.exe
OBJECTS=graph.o map.o serialization.o main.o
.PHONY: default clean

default: $(EXEC)

graph.o: graph.c graph.h
    $(CC) -o graph.o -c graph.c $(CFLAGS)

...

$(EXEC): $(OBJECTS)
    $(CC) -o $(EXEC) $(OBJECTS)

clean:
    rm $(EXEC) $(OBJECTS)
```

Makefile

Makefile

■ Ecriture d'un Makefile

- Référence à la cible ($\$@$) et aux dépendances ($\$^$, $\$<$) dans les règles

```
CC=gcc
CFLAGS=-W -Wall
EXEC=serialize.exe
OBJECTS=graph.o map.o serialization.o main.o
.PHONY: default clean

default: $(EXEC)

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

...

$(EXEC): $(OBJECTS)
    $(CC) -o $@ $^

clean:
    rm $(EXEC) $(OBJECTS)
```

Makefile

Makefile

■ Ecriture d'un Makefile

- Factorisation des règles en utilisant les extensions de fichiers

```
...
OBJECTS=graph.o map.o serialization.o main.o
.PHONY: default clean

default: $(EXEC)

graph.o: graph.c graph.h
map.o: map.c map.h
serialization.o: serialization.c serialization.h map.h graph.h
main.o: main.c serialization.h graph.h

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

$(EXEC): $(OBJECTS)
    $(CC) -o $@ $^

...
```

Makefile

Makefile

■ Ecriture d'un Makefile

- Récupération des fichiers objets en utilisant les *wildcards*

```
...
SOURCES=graph.c map.c serialization.c main.c
OBJECTS=$(SOURCES:.c=.o)
.PHONY: default clean

default: $(EXEC)

graph.o: graph.c graph.h
map.o: map.c map.h
serialization.o: serialization.c serialization.h map.h graph.h
main.o: main.c serialization.h graph.h

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

$(EXEC): $(OBJECTS)
    $(CC) -o $@ $^

...
```

Makefile

Makefile

Makefile

```
EXEC=serialize.exe
SOURCES=graph.c map.c serialization.c main.c
OBJECTS=$(SOURCES:.c=.o)
CC=gcc
CFLAGS=-W -Wall
.PHONY: default clean

default: $(EXEC)

graph.o: graph.c graph.h
map.o: map.c map.h
serialization.o: serialization.c serialization.h map.h graph.h
main.o: main.c serialization.h graph.h

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

$(EXEC): $(OBJECTS)
    $(CC) -o $@ $^

clean:
    rm $(EXEC) $(OBJECTS) $(SOURCES:.c=.c~) $(SOURCES:.c=.h~)
```

Plan du cours : compilation séparée



- Préprocessing en C
- Makefile
- Programmation modulaire

Programmation modulaire



■ Module

- Définition
- Génie logiciel (cf. cours de Dév. Prog. en L3)
- Dépendance entre modules


Programmation modulaire

■ Programmation modulaire en C

- Module = 2 fichiers
- Fichier *header* (*nom_du_module.h*)
 - *Dépendances* à d'autres modules
 - Spécification des *types de données*
 - *Prototypes* des fonctions
- Fichier *source* (*nom_du_module.c*)
 - Dépendances internes à d'autres modules
Modules utilisés uniquement dans le corps des fonctions
 - Définitions des fonctions
 - *Création* des valeurs des types définis par le module
 - *Destruction* des valeurs des types définis par le module
 - *Affichage* des valeurs des types définis par le module
 - *Algorithmes* propres aux valeurs des types définis par le module

Programmation modulaire

■ Structure d'un fichier *header*

```
#ifndef MODULE_H
#define MODULE_H  to avoid recursive inclusion module.h

/* module dependences */
#include "module_dep1.h"
#include "module_dep2.h"
...

/* type(s) declaration */
typedef ... module_type;
...

/* function prototypes */
module_type function1();
void          function2(module_type* t, int i);
char*         function3(module_type t);
...

#endif // MODULE_H
```

Programmation modulaire

■ Structure d'un fichier *source*

```
#include "module.h"

/* internal dependencies */
#include "dep1.h"
#include "dep2.h"
...

/* function definitions */
module_type function1() {
    ...
}

void function2(module_type* t, int i) {
    ...
}

char* function3(module_type t) {
    ...
}
```

module.c

Programmation modulaire

■ Fichier *header* du module `map`

```
#ifndef MAP_H
#define MAP_H

/* module dependences */
#include <stdio.h> // for type size_t

/* type(s) declaration */
struct couple { int id; void* ptr; struct couple* next; };

typedef struct map_ { struct couple* list; int size; }* map;

/* function prototypes */
map    new_map();
void   delete_map(map m);

int     get_id(map m, void* ptr);
void*   get_ptr(map m, int id, size_t sz);

#endif // MAP_H
```

map.h

Programmation modulaire

■ Fichier *source* du module `map`

```
#include "map.h"
#include <stdlib.h>

struct couple* create_couple(int id, void* ptr, struct couple* next){
    struct couple* cpl = (struct couple*)malloc(sizeof(struct couple));
    if (cpl == NULL) {
        fprintf(stderr, "map: create_couple: memory allocation failed");
        exit(1);
    }
    cpl->id = id; cpl->ptr = ptr; cpl->next = next;
    return cpl;
}

map new_map() {
    map d = (map)malloc(sizeof(struct map_));
    d->list = create_couple(0, NULL, NULL); d->size = 1;
    return d;
}
...
```

map.c