TP 2 : Compilation séparée
Makefile, fichiers .h et .c

Objectifs :

- Maîtriser l'outil make et les fichiers Makefile.
- Maîtriser la compilation séparée.
- Savoir utiliser les bibliothèques prédéfinies en C.

Compilation séparée

1. Introduction

1.1. Comment marche la compilation ?

La compilation avec gcc s'effectue en quatre étapes pour produire, à partir d'un programme C, un code exécutable :

1. passage au pré-processeur (*preprocessing*) ;
2. compilation en langage assembleur (*compiling*) ;
3. conversion du langage assembleur en code machine (*assembling*) ;
4. édition des liens (*linking*).

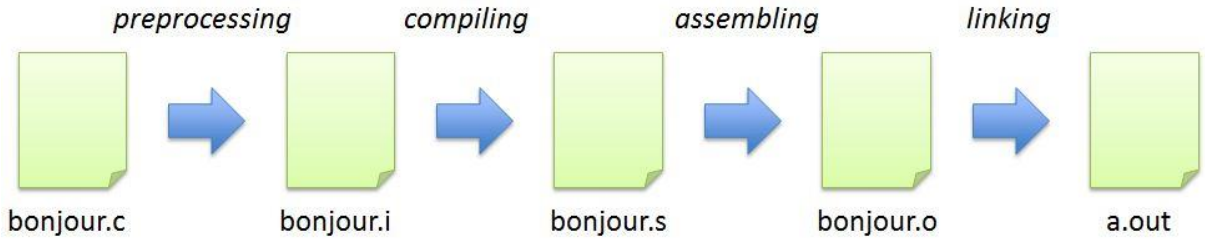


Figure 1 : Chaîne de compilation.

Le prétraitement (*preprocessing*) consiste à transformer le code source d'un programme C en un autre code source C débarrassé des directives de prétraitement et où les macros du processeur ont été développées. Les directives de prétraitement permettent d'inclure d'autres fichiers contenant des déclarations (ex : `#include <fichier.h>`) et de définir de nouvelles macros (ex : `#define PI 3.14`). Les directives du préprocesseur commencent toutes par un caractère dièse (#).

La compilation (*compiling*) du code C prétraité en assembleur est la phase la plus compliquée. Pendant cette phase, on passe d'un langage de haut niveau vers un langage de très bas niveau : l'assembleur.

L'assemblage (*assembling*) permet la traduction du langage assembleur (texte) en langage machine binaire (fichier "objet").

La commande gcc utilisée avec l'option –c permet de réaliser ces trois premières étapes. Ainsi, la commande `gcc –c bonjour.c` permet de produire le fichier `bonjour.o` (la commande `gcc –c bonjour.c –o bonjour.o` a le même résultat).

D'autres options de compilation peuvent être utilisées. Nous en utilisons systématiquement trois dans le cadre de cet enseignement :

```
gcc –c bonjour.c -Wall -ansi -pedantic –o bonjour.o
```

Elles permettent de garantir que la syntaxe utilisée est la plus standard et la plus correcte possible (ce qui est important en phase d'apprentissage).

Enfin, l'édition de liens (*linking*) permet de combiner éventuellement plusieurs fichiers "objet" produits par la phase d'assemblage ainsi que les éventuelles bibliothèques utilisées

(p.ex. pour les entrées et sorties) en un seul programme exécutable. Elle fait le lien entre des symboles définis dans les fichiers objets et ces mêmes symboles utilisés dans d'autres objets. La commande suivante effectue l'édition des liens entre les fichiers objet `bonjour.o` et `bonsoir.o` et produit le fichier exécutable `progexe` : `gcc bonjour.o bonsoir.o -o progexe`. Si le nom de fichier résultant de la compilation n'est pas spécifiée à l'aide de l'option `-o (gcc bonjour.o bonsoir.o)`, il prend par défaut la valeur "a.out".

On peut également écrire `gcc bonjour.c -o bonjour`. Dans ce cas, la compilation et l'édition des liens est faite est le programme exécutable `bonjour` est produit (ceci n'a évidemment pas de sens si notre application est composée de plusieurs fichiers .c)).

1.2. Intérêt de la compilation séparée

La compilation séparée est nécessaire à partir du moment où une application est écrite à l'aide de plusieurs fichiers sources (.c), ce qui est en général le cas, pour plusieurs raisons :

- la programmation est modulaire, donc plus compréhensible ;
- la séparation en plusieurs fichiers produit des listings plus lisibles ;
- la maintenance est plus facile car seuls les modules modifiés sont recompilés.

Sous UNIX, la commande `make`[1] permet d'automatiser l'exécution des étapes de compilation séparée. Cette commande :

- effectue la compilation séparée grâce à la commande gcc ;
- utilise des macro-commandes et des variables ;
- permet de ne recompiler que les fichiers sources modifiés ;
- permet d'utiliser des commandes shell.

1.3. Mise en œuvre

La commande make requiert pour son fonctionnement un fichier nommé (par défaut) Makefile (avec un M majuscule et le reste en minuscule). Il doit se trouver dans le répertoire courant lorsque l'on appelle la commande make à l'invite du shell. Les instructions contenues dans ce fichier doivent respecter une syntaxe particulière expliquée ci-après.

2. Règles

Les fichiers Makefile sont structurés grâce aux *règles* ; ce sont elles qui définissent ce qui doit être exécuté.

2.1. Définition

Une règle est une suite d'instructions qui seront exécutées pour construire *une cible*, mais uniquement si des *dépendances* ont été modifiées depuis la dernière construction de la cible. La syntaxe d'une règle est la suivante :

```
cible : dépendances
	commandes
```

L'espace avant commandes est obligatoire et doit être fait avec une tabulation.

[1] make est essentiel lorsque l'on veut effectuer un portage, car la plupart des logiciels libres UNIX (c'est-à-dire des logiciels qui sont fournis avec le code source) l'utilisent pour leur installation.

2.2. Cible

La cible est souvent (mais pas nécessairement) le nom d'un fichier qui va être généré par les commandes qui vont suivre.

2.3. Dépendances

Les dépendances sont les fichiers nécessaires à la création de la cible. Par exemple, pour la compilation C, les dépendances peuvent contenir un fichier d'en-tête ou un fichier source.

2.4. Commandes

Les commandes sont des commandes shell qui seront exécutées au moment de la construction de la cible. La tabulation avant les commandes est obligatoire et si la commande dépasse une ligne, il est nécessaire de signaler la fin de ligne avec un caractère antislash "\".

3. Un exemple de fichier Makefile

3.1. Mon premier Makefile

```
# Mon premier Makefile

all: foobar.o main.o
     gcc –o main foobar.o main.o

foobar.o: foobar.c foobar.h
     gcc –c foobar.c –Wall –ansi –pedantic –o foobar.o

main.o: main.c
     gcc –c main.c –Wall –ansi –pedantic –o main.o
```

Les lignes commençant par le caractère # sont des commentaires.
Le Makefile ci-dessus a trois règles destinées à construire les cibles `all`, `foobar.o` et `main.o`. La cible principale `all` nécessite les fichiers `foobar.o` et `main.o` pour être exécutée afin d'obtenir le fichier exécutable `main`. Pour obtenir les fichiers `foobar.o` et `main.o`, il faut utiliser les deux cibles suivantes.

3.2. Comment utiliser un Makefile

La commande `make` permet d'exécuter le fichier Makefile :

```
machine@esisar> make all
```

La commande `make` interprète le fichier Makefile et exécute les commandes contenues dans **la règle dont la cible est all**. Au préalable les dépendances `foobar.o` et `main.o` ont été vérifiées (i.e. les règles éventuelles dont la cible est `foobar.o` ou `main.o` ont été exécutées). Cela signifie, par exemple, que si `foobar.c` ou bien `foobar.h` ont été modifiés depuis la dernière compilation, alors `foobar.o` sera d'abord mis à jour avant que la commande de la règle all soit exécutée.

La commande `make` peut être exécutée sans argument. Dans ce cas, elle exécute la première règle rencontrée. Elle peut également utiliser un fichier appelé autrement que Makefile, auquel cas la syntaxe suivante est utilisée : `make -f monMakeFileAMoi`.

4. Variables

4.1. Définition et utilisation

Les variables doivent être vues comme des macro-commandes (comme #define en C). La déclaration se fait ainsi :

```
NOM = VALEUR
```

La valeur affectée à la variable peut comporter n'importe quels caractères. Elle peut être aussi une autre variable.
La syntaxe de l'appel de la variable est la suivante :

```
$(NOM)
```

4.2. Exemple de fichier Makefile avec des variables

```
# $(BIN) est le nom du fichier binaire généré
BIN = foo
# $(OBJECTS) sont les objets qui seront générés
# après la compilation
OBJECTS = main.o foo.o
# $(CC) est le compilateur utilisé
CC = gcc
# all est la première règle à être exécutée car elle est
# la première dans le fichier Makefile. Notons que les
# dépendances peuvent être remplacées par une variable,
# ainsi que n'importe quelle chaîne de caractères des
# commandes
all: $(OBJECTS)
     $(CC) $(OBJECTS) -o $(BIN)

main.o: main.c main.h
     $(CC) -c main.c
foo.o: foo.c foo.h main.h
     $(CC) -c foo.c
```

5. Caractères jockers et variables automatiques

5.1. Caractères jockers

Les caractères jockers s'utilisent comme en shell. Les caractères valides sont par exemple : *, ?. L'expression toto?.c représente tous les fichiers commençant par toto et finissant par .c avec une lettre entre ces deux chaînes (toto1.c, toto2.c…). Comme en shell, le caractère \ permet d'inhiber l'action des caractères jockers.

5.2. Variables automatiques

Les variables automatiques sont variables qui sont actualisées au moment de l'exécution de chaque règle en fonction de la cible et des dépendances.

$@	Nom de la cible
$<	Première dépendance de la liste des dépendances
$?	Les dépendances les plus récentes de la cible
$^	Toutes les dépendances

6. Conventions de nommage

6.1. Noms d'exécutables et arguments

CC	Compilateur C (cc ou gcc)
CXX	Compilateur C++ (c++ ou g++)
RM	Commande pour effacer un fichier (rm)
CFLAGS	Paramètres à passer au compilateur C
CXXFLAGS	Paramètres à passer au compilateur C++

6.2. Noms des cibles

Un utilisateur de make peut donner à ses cibles le nom qu'il désire, mais pour des raisons de lisibilité, on donne toujours un nom standard à certaines cibles :

all	Compile tous les fichiers source pour créer l'exécutable principal
install	Exécute all et copie l'exécutable, les bibliothèques et les fichiers d'en-tête s'il y en a dans le répertoire de destination
uninstall	Détruit les fichiers créés lors de l'installation, mais pas les fichiers du répertoire d'installation où se trouvent les fichiers sources et le Makefile
clean	Détruit tous les fichiers créés par all
dist	Crée un fichier tar de distribution

Exercice

Exercice 1 : Makefile

Ecrire un fichier Makefile simple qui permette de compiler le code de l'exercice du jeu de la multiplication du TP 1.

Organisation d'un programme

Jusqu'à présent, tout votre code était dans un unique fichier source .c. La structure d'un fichier *monProgramme.c* est en général la suivante :

```
// Inclusion de fichiers permettant d'utiliser des fonctions
// de bibliothèques
#include <stdio.h>
#include <stdlib.h>

// Déclarations des constantes
#define TAILLE 100

// Déclarations des variables globales
int entier1 ;
// Déclarations de toutes les fonctions
// Le détail des fonctions se trouve après la fonction main()
void fonction1(void) ;
void fonction2(void) ;

// Programme principal
int main(int argc, char * argv[]) {
...
...
}

// Implémentation des fonctions déclarées précédemment
void fonction1(void) {
...
}

void fonction2(void) {
...
}
```

Cette méthode fonctionne bien pour de petits programmes assez simples. Mais dans le cas de plus grosses applications, on organise le code en plusieurs fichiers qu'on appelle "modules". Chaque module contient des fonctions offrant un ensemble de fonctionnalités cohérent (on parle souvent de "séparation des préoccupations"). Par exemple, pour un logiciel de jeu, il est possible d'avoir des fonctions pour le graphisme et d'autres pour gérer les règles du jeu et les différents joueurs.

1. Découpage d'un programme en modules

En C, un module est réalisé à l'aide de deux sortes de fichiers reconnaissables à leur extension :

- les fichiers .c qui sont des *fichiers source* ;
- les fichiers .h (h pour header en anglais) qui sont les *fichiers d'en-tête*.

Le fichier .h contient les déclarations de type, de fonctions, de constantes du module ; en d'autres termes, tout ce qu'un autre module aurait besoin pour utiliser les fonctions proposées par celui-ci. La réalisation (ou implémentation) des fonctions du module est, quant à elle, contenue dans le fichier .c.

Le principe généralement appliqué consiste à écrire un fichier .h pour chaque fichier .c et à déclarer dans le fichier .h tout ce que le module souhaite "exporter" : lorsqu'un module A utilise une fonction déclarée dans un module B, alors A.h doit inclure (#include) le fichier B.h.

Exemple : découpage en modules de l'exemple précédent.

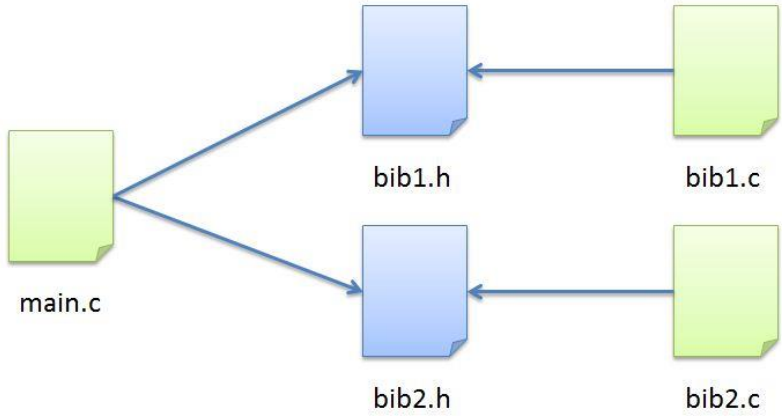


Figure 2 : Architecture du programme.

Les flèches → représentent les inclusions.

Les fichiers bib1.h et bib2.h contiennent les déclarations qui sont utiles aux modules bib1 et bib2 :

Le fichier bib1.h :

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

#define TAILLE=100

void fonction1(void) ;
```

Le fichier bib2.h :

```
#include <stdio.h>

int entier1 ;

void fonction2(void) ;
```

Les fichiers .c contiennent les implémentations des fonctions :

Le fichier bib1.c

```
#include "bib1.h"

void fonction1(void) {
...
}
```

Le fichier bib2.c

```
#include "bib2.h"

void fonction2(void) {
...
```

```
}
```

Le programme principal ne contient plus que les inclusions nécessaires ainsi que la fonction main().

```
// Inclusion de fichiers permettant d'utiliser des fonctions
// de bibliothèques

#include "bib1.h"
#include "bib2.h"

// Programme principal
int main(int argc, char * argv[]) {
...
...
}
```

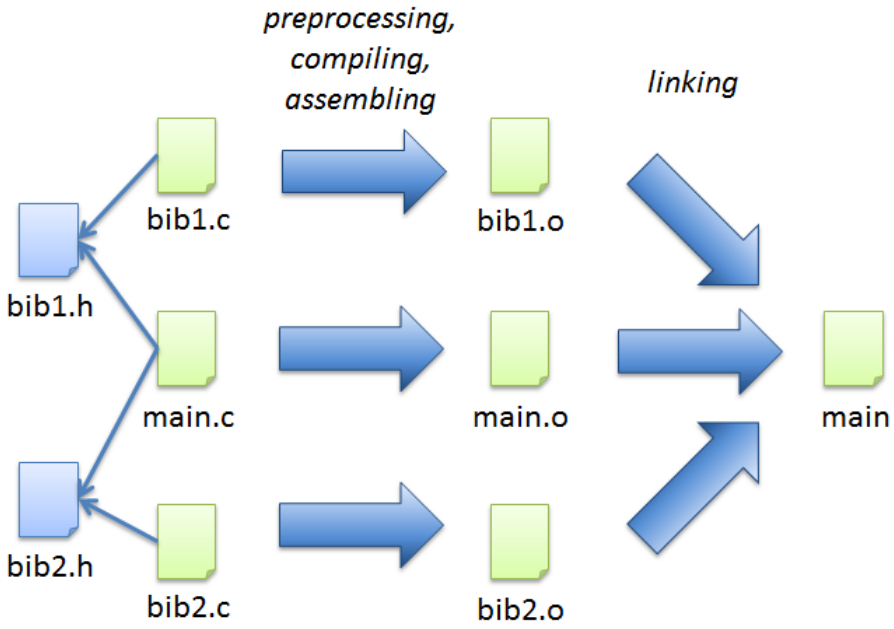


Figure 3 : Chaîne de compilation à plusieurs fichiers.

2. Compilation conditionnelle

Dans l'exemple précédent, on remarque que des bibliothèques identiques ont été incluses dans différents modules (par exemple, la bibliothèque stdio.h). Par conséquent, lors de la compilation du module principal, ces bibliothèques seront lues plusieurs fois. Ceci peut faire échouer la compilation (par exemple, des éventuelles déclarations de constante seront lues plusieurs fois, ce qui équivaut à une déclaration multiple, interdite en C). Pour éviter au compilateur de lire plusieurs fois les mêmes fichiers, on utilise le mécanisme de *compilation conditionnelle*. Le compilateur tient compte d'un ensemble de commandes pour savoir ce qui doit être compilé ou non.

Des *directives* permettent d'incorporer ou d'exclure des portions du fichier dans le texte qui sera analysé par le préprocesseur. Ces directives se classent en deux catégories selon la condition qui régit l'incorporation :

- Existence ou inexistence des symboles ;
- Valeur d'une expression.

Le détail des *directives* sont présentés dans la section 8.2 du document « *Introduction au langage C* » à la page 139.

Pour résumer, les fichiers .h doivent avoir cette forme (illustrée sur l'exemple du fichier bib1.h) :

```
#ifndef __BIB1_H__
#define __BIB1_H__

void fonction1(void) ;

#endif
```

Exercices

Exercice 2 : Exemple de compilation conditionnelle

Le but de cet exercice est d'écrire un programme qui a un comportement différent en fonction des options de compilation utilisées. On reprend, pour cela, le programme de calcul de pgcd du TP1.

- Ecrire une nouvelle version de ce programme telle que, quand il est compilé de la manière suivante: gcc –c –D MISEAUPOINT pgcd.c et après édition de liens, affiche, à chaque itération un message : "valeur courante de b = x" (x étant la valeur de b à l'entrée de l'itération). Ce message ne doit pas être affiché si l'option MISEAUPOINT n'est pas utilisée (gcc –c pgcd.c).

Exercice 3 : Bibliothèque mathématique

Le but de cet exercice est d'écrire une bibliothèque de fonctions mathématiques. Vous devrez structurer correctement votre code (fichiers .h et .c).

- Ecrire une fonction *quotient()* ayant pour paramètres deux entiers positifs a et b et qui retourne le quotient de a et b. Ce calcul doit se faire par soustractions successives (interdiction d'utiliser l'opérateur de division du langage C).
- Ecrire une fonction *reste()*, ayant pour paramètres deux entiers positifs a et b qui retourne le reste de la division de a par b. Cette fonction doit utiliser la fonction *quotient()* (et non pas l'opérateur %).
- Ecrire une fonction *valeurAbsolue*() qui calcule la valeur absolue d'un entier. Cette fonction peut utiliser la fonction abs() de la bibliothèque stdlib.h.

- Ecrire une fonction *ppcm()* qui calcule le plus petit commun multiple. Pour rappel, il est possible de le calculer ainsi :

$$ppcm(a,b) = \frac{|ab|}{pgcd(a,b)}$$

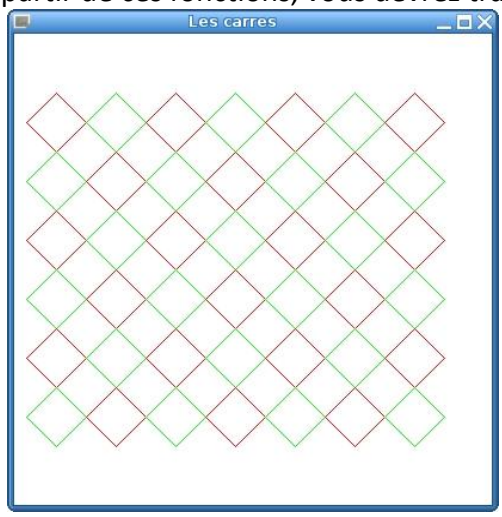
- Ecrire une fonction *testBibliothèque*() qui effectue des tests des fonctions ci-dessus. Pour chaque test effectué on précisera (sous la forme d'un commentaire) le choix des valeurs des entrées. *Il faudra notamment que les tests couvrent tous les cas particuliers : division par un nombre plus grand, division par 0…*

Exercice 4 : Mosaïque

Le but de cet exercice est d'utiliser une bibliothèque graphique afin de tracer un carré puis une mosaïque constituée de carrés.
La bibliothèque contient des fonctions qui permettent :

- d'ouvrir et d'initialiser une fenêtre pour dessiner : gr_inits_w () ;
- de changer la couleur du pinceau : set_blue(), set_red(), set_green(), set_yellow(), set_black() ;
- de tracer une ligne : line() ;
- de récupérer les coordonnées d'un clic souris (cette commande attend qu'un clic se produise) : cliquer_xy() ;
- de bloquer l'exécution en attendant un clic de souris : cliquer() – *utile notamment en fin de dessin pour donner le temps au système de tout afficher*.

A partir de ces fonctions, vous devrez tracer un carré « sur la pointe » :



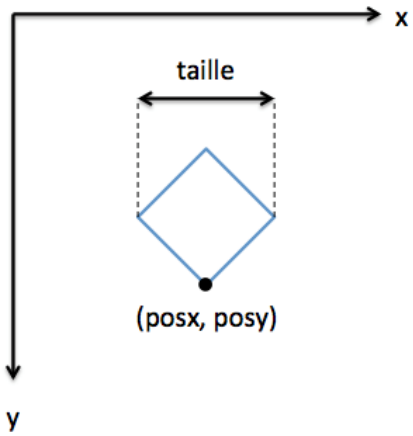


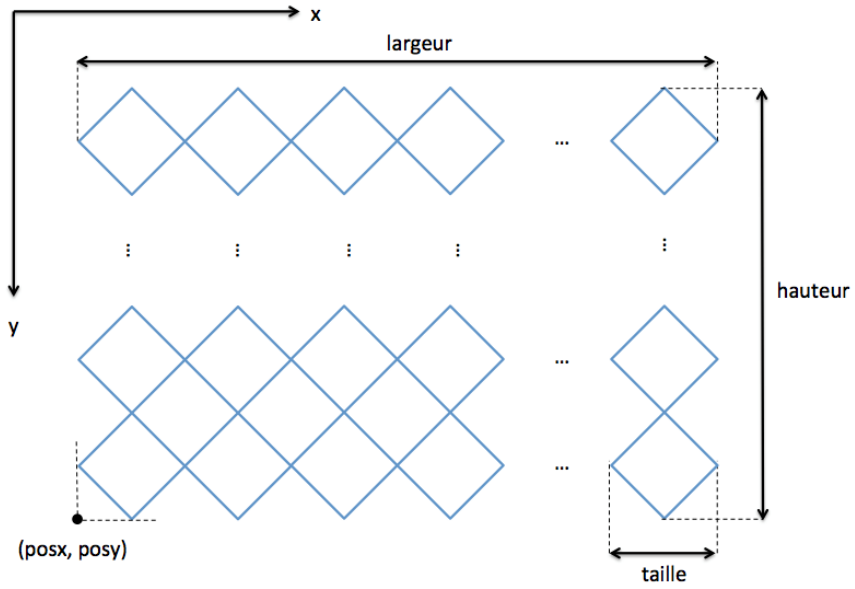
Figure 4 : Mosaïque.

1. Ecrire le fichier *Makefile* qui permet de compiler les fichiers sources qui vous sont fournis. L'utilisation de la bibliothèque graphique nécessite d'utiliser les options `-lX11` et `-L/usr/X11R6/lib` lors de l'édition des liens[2].

Pour la suite, vous devez créer deux fichiers (*dessine.h* et *dessine.c*) dans lesquels vous allez déclarer et implémenter les fonctions suivantes qui seront testées dans le fichier *main.c* :

[2] Ce sont des inclusions de librairies graphiques binaires nécessaires à graphlib.

2. Ecrire une fonction dessineCarre() qui dessine un carré « sur la pointe ». Cette fonction prend en paramètre les coordonnées d'un point ainsi que la taille du carré.
3. Ecrire une fonction dessineCarreDiagonale() qui dessine un carré ainsi que ses diagonales. Cette fonction prend en paramètre les coordonnées d'un point ainsi que la taille du carré.
4. Ecrire une fonction dessineMosaique() qui dessine une mosaïque comme l'illustre la Figure 4. Cette fonction prend en paramètre la taille des carrés, la position de la mosaïque, sa largeur et sa hauteur en nombre de carrés :



5. Ecrire une fonction dessineMosaiqueAvecSouris(), similaire à la fonction précédente mais qui ne prend en paramètre que la taille des carrés, sa largeur et sa hauteur en nombre de carrés. La position de la mosaïque sera précisée par un clic souris de l'utilisateur.