

# INF3135

## Construction et maintenance de logiciels

### **Cours 11: Les branches en Git**

Alexandre Blondin Massé

Université du Québec à Montréal  
Département d'informatique

Été 2020

# Table des matières

**1** Dates et contenu à venir

**2** Quiz 3

**3** Travail pratique 2

**4** Modularité

**5** Les branches sous Git

Dates et contenu à venir

# Point sur le cours

## Dates

- **Remise du TP2:** 31 juillet
- **Quiz 3:** 21 juillet
- **Sujet du TP3:** avant le 31 juillet
- **Remise du TP3:** 23 août
- **TP3:** optionnel, sur une base individuelle

## Contenu

- **Chapitre 6:** modularité
- **Chapitre 7:** maintenance
- **Chapitre 8:** tests et intégration continue

## Quiz 3

# Quiz 3

## Matière évaluée

- Date: **21 juillet**
- **Chapitre 5**: structures de données
- **Chapitre 6**: modularité
- **Labo 8**: modules
- **Labo 9**: les branches sous Git

## Forme du quiz

- 1 question courte sur chapitre 5
- 1 question courte sur chapitre 6
- 1 question courte sur Git
- 1 question longue

## Travail pratique 2

## Travail pratique 2

- **20%** de la note finale
- Fait **seul**
- Date de remise: **31 juillet**
- Le travail doit être réparti sur des **branches**
- Attention à la rédaction des **requêtes d'intégration**
- Attention à la qualité des *commits* et de leur message
- Mettre à jour votre **.gitconfig**
- Ne pas mettre de *commits* sur la branche `master`: utilisée pour se **synchroniser** avec le dépôt parent
- Récupérer les versions les plus à jour



# Modularité

# Modularité

## Caractéristiques

- **Séparation**: les préoccupations sont divisées en composantes
- **Indépendance**: les dépendances entre modules sont minimales
- **Interchangeabilité**: facilité de remplacer une composante
- **Spécifique**: un module règle une préoccupation précise
- **Réutilisation**: un module est souvent réutilisable

## Remarques

- Varie selon le **langage**
- Notion d'**interface** et d'**implémentation**
- Séparer les **préoccupations**

# Fonctions variadiques

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

int max(int n, ...) {
    va_list list;
    va_start(list, n);
    if (n <= 0) return 0;
    int m = va_arg(list, int);
    for (unsigned int i = 1; i < n; ++i) {
        int m2 = va_arg(list, int);
        m = m2 > m ? m2 : m;
    }
    va_end(list);
    return m;
}

int main (void) {
    printf("%d ", max(3, 8, 2, -3));
    printf("%d\n", max(5, -4, -1, -8, 7, 6));
    return 0;
}
```

**Résultat:**

8 7

# Précompilation

## Directives #define/#undef

- Macros
- Macros-fonctions
- Attention aux pièges

## Directives conditionnelles

- #ifdef/#ifndef: vérifie si une macro est définie ou non définie
- #if/#else/#elif/#endif: structure conditionnelle

## Fonctions et macro-fonctions variadiques

Nombre quelconque d'**arguments**

# Bibliothèque

## Définition (extraite de Wikipedia)

« En informatique, une bibliothèque logicielle est une collection de **routines**, qui peuvent être déjà compilées et prêtes à être utilisées par des programmes. Les bibliothèques sont enregistrées dans des fichiers semblables, voire identiques aux fichiers de programmes, sous la forme d'une collection de fichiers de code objet rassemblés accompagnée d'un index permettant de **retrouver facilement** chaque routine. Le mot librairie est souvent utilisé à tort pour désigner une bibliothèque logicielle. Il s'agit d'un anglicisme fautif dû à un **faux-ami** (library). »

## En C

- Il suffit d'ajouter `#include <bibliotheque.h>`
- Puis, lors de la **compilation** et de l'**édition des liens**,
- on indique à GCC où trouver les **fichiers** nécessaires

# Compilation et édition des liens

Rappel sur les **étapes** de compilation:

- **Compilation**: `.c`  $\rightarrow$  `.o`
- **Édition des liens**: `.o`  $\rightarrow$  exécutable

Comment GCC gère-t-il ces deux étapes?

- **Compilation**: trouver les **en-tête** (fichiers `.h`)
- **Édition des liens**: trouver les fichiers **binares**
- **Plusieurs formats** possibles: `.o`, `.a`, `.so`, `.dll`, ...

## Deux syntaxes possibles

- À quel **endroit** GCC cherche ces fichiers?
- `#include "module.h"`: cherche dans le répertoire courant
- `#include <module.h>`: cherche dans le système

# Emplacement des fichiers d'en-tête

- À la **compilation**
- GCC cherche seulement les fichiers d'**en-tête** (.h)
- Sur une installation **typique Unix**:

```
/usr/local/include  
libdir/gcc/target/version/include  
/usr/target/include  
/usr/include
```

- Si le fichier .h est **ailleurs**, il faut le **spécifier**
- À l'aide de l'option **-I** (pour *include*):

```
$ gcc -I<chemin> ...
```

## Attention

- Éviter les chemins **absolus** en **dur**
- Sinon le code n'est pas **portable**

# Emplacement des binaires

- À l'**édition des liens**
- GCC cherche les implémentations **binaires** (.o, .so, .dll, ...)
- Il inspecte **plusieurs répertoires**
- Pour les **connaître** (sur Linux), utiliser `ldconfig -v`:

```
$ ldconfig -v 2>/dev/null | grep -v ^$'\t'  
/usr/lib/x86_64-linux-gnu/libfakeroot:  
    libfakeroot-0.so -> libfakeroot-tcp.so  
/lib/i386-linux-gnu:  
    libwrap.so.0 -> libwrap.so.0.7.6  
    libnss_dns.so.2 -> libnss_dns-2.27.so  
[...]
```

- Si la bibliothèque se trouve ailleurs, il faut le **spécifier**:

```
$ gcc -L<chemin> ...
```

## Attention

Ici aussi, éviter les chemins **absolus** en **dur**



# L'utilitaire pkg-config

- L'utilisation de **chemins absolus en dur** n'est pas acceptable
- Si on souhaite qu'une application soit **portable**
- **Solution**: utiliser le programme **PKG-config**
- Pour les **inclusions** (-I):

```
$ pkg-config --cflags bibliotheque
```

- Pour les **binares** (-L et -l):

```
$ pkg-config --libs bibliotheque
```

- Remplacer bibliotheque par la bibliothèque correspondante: cairo, tap, jansson, etc.

# Retour sur les Makefiles

- Les **règles à motifs** (*pattern rule*)
- **wildcard**: lister des fichiers (*glob*)
- **patsubst**: substituer des motifs
- **shell**: invoquer une commande shell
- **filter-out**: retire un motif d'une liste de mots
- **realpath**: résoudre un chemin (simplifier et liens symboliques)
- **dir**: semblable à la commande `dirname`
- **abspath**: chemin absolu d'un fichier
- **lastword**: récupère le dernier mot d'une liste de mots
- ...

Voir **la documentation officielle** pour plus d'informations

# Règles à motifs

- Permet de déclarer des règles **générales**
- Le caractère % est utilisé pour indiquer la **substitution**
- \$<: première dépendance
- \$@: cible

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

- On peut **restreindre** les cibles visées
- À l'aide d'une règle **statique**:

```
objects = geometry.o graph.o isomap.o map.o queue.o tile.o  
  
$(objects): %.o: %.c %.h  
    gcc -c $(CFLAGS) $< -o $@
```

# Fonctions utiles (1/2)

## – La fonction wildcard:

```
# Tous les fichiers avec extension .c
$(wildcard *.c)
# Tous les fichiers commençant avec test et finissant par .c
$(wildcard test*.c)
```

## – La fonction patsubst:

```
# Fichiers objets souhaités
$(patsubst %.o, %.c, $(wildcard *.c))
# Exécutables des tests souhaités
$(patsubst %, %.c, $(wildcard test*.c))
```

## – La fonction shell:

```
# Options de GCC pour la compilation
CFLAGS = "-std=c11 -Wall -Wextra $(shell pkg-config --cflags cairo)"
"
# Options de GCC pour l'édition des liens
LFLAGS = "$(shell pkg-config --libs cairo)"
```

## Fonctions utiles (2/2)

- La fonction filter-out:

```
# Pour retirer les tests de la liste
test_c_files = $(wildcard test*.c)
c_files = $(filter-out $(test_c_files), $(wildcard *.c))
objects=$(patsubst %.o, %.c, $(c_files))
```

- Les fonctions realpath, dir et abspath:

```
# Récupérer le chemin absolu du répertoire parent d'un Makefile
# $(lastword $(MAKEFILE_LIST)) récupère le nom du Makefile courant
# Ensuite on calcule avec $(abspath ...) le chemin absolu
# Puis $(dir ...) permet de récupérer le répertoire parent
# Et finalement on prend le chemin simplifié avec $(realpath ...)
current_make = $(lastword $(MAKEFILE_LIST))
root_dir := $(realpath $(dir $(abspath $(current_make))))/..
```

## Les branches sous Git

# Les branches sous Git

## Branches

- `git branch -a`: lister toutes les branches
- `git checkout name`: passer sur la branche `name`
- `git checkout -b name`: créer une branche `name` à partir de `HEAD`
- `git checkout -B name`: même si la branche existe déjà
- `git branch -d`: supprimer une branche fusionnée
- `git branch -D`: supprimer une branche
- `git rebase`: rebaser une branche
- `git rebase -i`: rebasement interactif

## Intégration

- `git cherry-pick`: sélectionner un *commit* et l'appliquer
- `git merge`: fusionner deux branches