

INF3135

Construction et maintenance de logiciels

Cours 10: Modularité

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

- 1 Entente d'évaluation
- 2 Travail pratique 1
- 3 Travail pratique 2
- 4 Modularité
- 5 Retour sur la précompilation
- 6 Modules en C
- 7 Makefiles
- 8 Bibliothèques

Entente d'évaluation

Proposition de modifications

- Déplacer la date de remise du TP2 au **31 juillet**
- Déplacer le quiz 3 au **21 juillet**
- Déplacer la date de remise du TP3 au **23 août** (date limite du registrariat: 28 août)
- Rendre le TP3 optionnel, sur une base individuelle
 - TP1 = 30% et TP2 = 30%
 - TP1 = 20%, TP2 = 20%, TP3 = 20%

Travail pratique 1

Travail pratique 1

Style de programmation

- **Syntaxe**: indentation, aération, longueur de ligne, etc.
- Pas de variables **globales**
- **Factorisation**: identifier les redondances et les faire disparaître

Git

- **Granularité** des *commits*
- **Messages** courts et significatifs
- **Syntaxe** uniforme (majuscule pas de point, verbe)

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

Définition (extraite de [Wikipedia](#))

« Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality. »

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

Terminologie

Varie selon le langage

- **Montage** (*assembly*): spécifique à Microsoft
- **Module**: un seul fichier ou un ensemble de fichiers
- **Paquet** (*package*): ensemble de modules
- **Composante**: une partie d'un système complexe

Exemples

- **Java**: un paquet (*package*)
- **C**: une paire de fichiers `.h/.c` (ou juste `.c`)
- **C++**: une paire de fichiers `.hpp/.cpp` (ou juste `.cpp`)
- **Python**: module = fichier, paquet = ensemble de modules
- **Haskell**: un fichier

Contenu d'un module

Interface

- Ce qui est **fourni** et **requis**
- Généralement visible de façon « **publique** »
- Documentation décrivant **utilisation**

Implémentation

- **Mise en oeuvre** de ce qui est déclaré dans l'interface
- Généralement « **privé** »
- Documentation décrivant le **développement**

Couplage et cohésion

Couplage (inter-modules)

« In software engineering, coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules. »

Cohésion (intra-module)

« In computer programming, cohesion refers to the degree to which the elements inside a module belong together. »

Objectifs

- **Minimiser** le couplage
- **Maximiser** la cohésion

Retour sur la précompilation

Directives au préprocesseur

- **Préfixées** par le symbole #
- **Lues et interprétées** avant même de procéder à la compilation
- Remplacement **textuel**

Exemples

- `#include`
- `#define`
- `#if`
- `#endif`
- `#ifndef`

Symboles

- Pour définir un **symbole** ou une macro, on utilise la directive

```
#define <identifiant> <valeur>
```

- Remplace toutes les **occurrences** de <identifiant> (comme mot) par <valeur>
- La valeur est donnée par **le reste de la ligne**
- Pour affecter une valeur sur **plusieurs lignes**, il faut utiliser le caractère \
- La **portée** du symbole s'étend jusqu'à la **fin du fichier** dans lequel il est défini
- Sauf si on trouve une commande

```
#undef <identificateur>
```


Exemple

Fichier preproc.c:

```
#include <stdio.h>

#define i x

/*
 * Commentaire quelconque
 */
int main() {
    int i = 6, j;
    if (i) {
#undef i
        j = i * 2;
    }
    return 0;
}
```

Après gcc -E preproc.c:

```
# 1 "preproc.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 325 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "preproc.c" 2

# 1 "/usr/include/stdio.h" 1 3 4
# 64 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 506 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/sys/_symbol_aliasing.h" 1 3 4
# 507 "/usr/include/sys/cdefs.h" 2 3 4
# 572 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/sys/_posix_availability.h" 1 3 4
# 573 "/usr/include/sys/cdefs.h" 2 3 4
# 65 "/usr/include/stdio.h" 2 3 4
# 1 "/usr/include/Availability.h" 1 3 4
# 153 "/usr/include/Availability.h" 3 4
# 1 "/usr/include/AvailabilityInternal.h" 1 3 4
# 154 "/usr/include/Availability.h" 2 3 4
# 66 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/include/_types.h" 1 3 4

:
:
```

Précompilation

```
# Pour voir le résultat de la précompilation
$ gcc -E fichier.c
# Pour conserver les commentaires
$ gcc -E -C fichier.c
# Pour enlever les lignes de la forme #<i> <valeur>
$ gcc -E -P fichier.c
```

- Pour éviter toute substitution **inattendue**...
- ...définir les symboles en **majuscules** exclusivement;

Définition de symboles à la compilation

- Il est possible de définir des symboles à la compilation seulement:

```
$ gcc -DLINUX fichier.c
```

- C'est équivalent à mettre la directive suivante dans `fichier.c`:

```
#define LINUX
```

- On peut également donner une **valeur** au symbole:

```
$ gcc -DLANGUE=FR fichier.c
```

- C'est équivalent à

```
#define LANGUE FR
```

Symboles prédéfinis

Fichier predefini.c:

```
#include <stdio.h>

int main() {
    printf("Nom du fichier source courant: %s\n", __FILE__);
    printf("Numéro de la ligne courante: %d\n", __LINE__);
    printf("Date de compilation: %s\n", __DATE__);
    printf("Heure de compilation: %s\n", __TIME__);
    printf("Compilateur conforme à la norme ISO? %s\n",
           __STDC__ == 1 ? "oui" : "non");
    return 0;
}
```

Affiche:

```
Nom du fichier source courant: predefini.c
Numéro de la ligne courante: 5
Date de compilation: Nov  5 2019
Heure de compilation: 09:32:14
Compilateur conforme à la norme ISO? oui
```

Macro-fonctions

- Une **macro-fonction** est un symbole **paramétrable**
- **Syntaxe:**

```
#define f(x1,x2,...,xn) <corps>
```

- Le remplacement ne se fait que pour les **occurrences** de la forme

```
f(v1,v2,...,vn)
```

Exemple: la fonction ABS

Fichier abs.c :

```
#include <stdio.h>

#define ABS(x) ((x) > 0 ? (x) : -(x))

int main() {
    int i = -6, j, k;
    j = ABS(i);
    k = ABS;
    printf("ABS(%d) = %d\n", i, j);
    return 0;
}
```

Après gcc -E -P abs.c :

```
// Du code provenant de stdio.h

int main() {
    int i = -6, j, k;
    j = ((i) > 0 ? (i) : -(i));
    k = ABS;
    printf("ABS(%d) = %d\n", i, j);
    return 0;
}
```

Seule la **première** des trois occurrences de ABS est remplacée

Exemple: la fonction CARRE

Fichier carre.c :

```
#include <stdio.h>

#define CARRE(x) ((x) * (x))
#define CARRE1(x) x * x
#define CARRE2(x) (x * x)

int main() {
    int x = 6, j, k, m, n;
    j = -CARRE1(x+1);
    k = -CARRE2(x+1);
    m = -CARRE(x+1);
    n = -CARRE(x++);
}
```

Après gcc -E -P carre.c :

```
// Du code provenant de stdio.h

int main() {
    int x = 6, j, k, m, n;
    j = -x+1 * x+1;
    k = -(x+1 * x+1);
    m = -((x+1) * (x+1));
    n = -((x++) * (x++));
}
```

Seule la valeur de la variable `m` est celle attendue

Dangers associés aux macro-fonctions

- Mauvaise **substitution** si le corps et les paramètres ne sont pas correctement **parenthésés**
- Les paramètres peuvent être évalués **plusieurs fois**
- Erreurs lorsqu'il y a des **effets de bord**
- Inefficacité lors d'évaluations **multiples**

Conclusion

- Éviter d'utiliser les **macro-fonctions**
- Sauf dans de **rares** cas
- Et favoriser l'utilisation de fonctions de la façon habituelle.

Directives conditionnelles

Les directives

```
#if  
#elif  
#else  
#ifdef  
#ifndef  
#endif
```

permettent d'indiquer au précompilateur d'effectuer certains traitements **avant compilation**

Utilisations fréquentes

- Gestion du paramétrage de **différentes versions** d'un même programme:

```
#ifdef LINUX
#   include "linux.h"
#endif
#ifdef MAC_OS
#   include "mac_os.h"
#endif
```

- Blocage des **inclusions multiples** des en-tête:

```
#ifndef PILE_H
#define PILE_H

[...]

#endif
```

Modules en C

Modules en C

- Typiquement, un **module** en C est divisé en **deux** fichiers
- Un premier fichier `fichier.h`, qui contient l'**interface** ou l'en-tête (*header*)
- Et un second fichier `fichier.c` qui contient l'implémentation de cette interface

Exemple

- `stack.h`: interface d'une pile
- `stack.c`: implémentation
- `test_stack.c`: utilisation de l'interface

Cycle de compilation

- **Étape 1:** compilation des fichiers sources

```
$ gcc -c stack.c  
$ gcc -c test_stack.c
```

- **Étape 2:** édition des liens

```
$ gcc -o test_stack stack.o test_stack.o
```

- **Étape 3:** lancement de l'exécutable

```
$ ./test_stack
```

Qualité d'une interface

- Utilisation de synonymes adéquats (avec typedef) pour clarifier le rôle des arguments et de façon **uniforme**
- Placer le **type manipulé** comme premier argument
- **Uniformiser** l'ordre des autres arguments
- Inclure des **noms significatifs** de paramètres
- **Préfixer** ou **suffixer** le nom des fonctions et des types définis
- Uniformiser l'indirection: `struct foo` ou `struct foo *` partout
- Utiliser des valeurs d'**état** plutôt que des **nombres** arbitraires indiquant une erreur

Séparer l'interface de l'implémentation

- On peut alors utiliser les **services** offerts par l'interface sans se soucier de ce qui se passe réellement (approche en **boîte noire**)
- On **cache** à l'utilisateur les données/calculs privés pour que la communication avec les autres modules se fasse à l'aide de fonctions seulement (principe d'**encapsulation**)
- Si besoin est, on peut fournir **différentes implémentations** pour une même interface, afin d'améliorer les **performances**
- Plus facile à **maintenir** et à **tester**

Makefiles

Compilation de modules

- Lorsqu'un projet utilise plusieurs modules, il devient pénible de tout compiler **manuellement**
- **Solution:** utiliser un Makefile

Nous allons explorer les éléments suivants:

- Les règles implicites
- La fonction wildcard
- La fonction patsubst

Cas d'étude

Considérons un projet dans lequel on trouve **trois modules**:

- `game.h/game.c`
- `menu.h/menu.c`
- `credits.h/credits.c`

et **un** fichier principal:

- `main.c`

Dépendances explicites

```
game: game.o menu.o credits.o main.o  
    gcc -o game game.o menu.o credits.o main.o
```

```
main.o: main.c  
    gcc -c main.c
```

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

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

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

Règles implicites

- La première amélioration possible est d'utiliser des **règles implicites**
- La syntaxe est la suivante

```
%.o: %.c  
gcc -c $<
```

- Cette règle indique que pour générer un fichier avec l'extension `.o`, il suffit de prendre le même fichier avec l'extension `.c` puis de lui appliquer la commande `gcc -c`

La fonction wildcard

- La deuxième amélioration consiste à utiliser la fonction `wildcard`
- La syntaxe est la suivante

```
$(wildcard *.c)
```

- Cet appel de fonction remplace l'expression par tous les fichiers avec l'extension `.c` dans le répertoire courant, séparés par des espaces
- Dans notre exemple, on obtiendrait

```
credits.c game.c main.c menu.c
```

La fonction patsubst

- La troisième amélioration consiste à utiliser la fonction patsubst
- La syntaxe est la suivante

```
$(patsubst %.c,%.o,$(wildcard *.c))
```

- Cette expression remplace toutes les extensions .c avec l'extension .o
- Dans notre exemple, on obtiendrait

```
credits.o game.o main.o menu.o
```

Makefile complet

```
CC = gcc
CFLAGS = -Wall
LFLAGS =
OBJECTS = $(patsubst %.c,%.o,$(wildcard *.c))
EXEC = main

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

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

.PHONY: clean

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

Bibliothèques

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 binaires (fichiers `.o`) correspondants

Question

À quel **endroit** GCC cherche ces fichiers?

Emplacement des fichiers d'en-tête

- À la **compilation**, GCC tente de trouver les fichiers d'**en-tête** seulement (fichiers `.h`)
- Sur une installation **typique Unix**, GCC inspecte les répertoires suivants:

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

- Si le fichier `.h` se trouve ailleurs, il faut le **spécifier** avec 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 tente de trouver les **implémentations** correspondantes
- Il inspecte **plusieurs répertoires**, qu'on peut connaître via la commande (sur Linux):

```
$ gcc -v hello.c -Wl,--verbose
```

- Si votre 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
- Une solution à ce problème consiste à utiliser le programme **PKG-config**
- Généralement, il suffit d'entrer

```
$ pkg-config --cflags <nom-bibliotheque>
```

pour obtenir les chemins contenant les **en-tête**

- Puis

```
$ pkg-config --libs <nom-bibliotheque>
```

pour obtenir les chemins contenant les **implémentations**

Deux types de bibliothèques

Statique:

- Extension: .a en Unix, .lib sous Windows
- La bibliothèque est **incluse** dans l'exécutable
- **Avantage:** réduit les dépendances
- **Inconvénient:** exécutables plus volumineux

Dynamique:

- Extension: .so en Unix, .dll sous Windows
- La bibliothèque est **liée dynamiquement**
- **Avantage:** évite les **redondances**, exécutables moins volumineux
- **Inconvénient:** nécessite une installation, problèmes de version

Exemple: Vec3D (1/2)

- Supposons que nous avons conçu une bibliothèque supportant la manipulation de **vecteurs**
- Voir les fichiers `vec3d.h` et `vec3d.c`
- Tout d'abord, on **compile** le fichier `vec3d.c` en objet `vec3d.o`:

```
$ gcc -o vec3d.o -c vec3d.c
```

- Ensuite, on crée la bibliothèque statique:

```
$ ar -cvq libvec3d.a vec3d.o
```

- On peut ensuite l'inclure via l'instruction en autant que l'**en-tête** et l'**implémentation** soient disponibles

```
#include <vec3d.h>
```

Exemple: Vec3D (2/2)

- Par exemple, supposons que les fichiers `vec3d.h` et `libvec3d.a` se trouvent respectivement dans les répertoires

```
/Users/blondin_al/clib/include  
/Users/blondin_al/clib/lib
```

- Alors il suffit de compiler avec la commande

```
$ gcc -I/Users/blondin_al/clib/include \  
>      -c test_vec3d.c
```

- Puis de compléter l'**édition des liens** avec

```
$ gcc -L/Users/blondin_al/clib/lib -o \  
>      test_vec3d test_vec3d.o -lvec3d
```

Les bibliothèques `unistd.h` et `getopts.h`

- Facilite le **traitement** des arguments récupérés par la fonction `main`
- Autrement dit, simplifie le traitement de `argc` et `argv`

Deux types d'**options**:

- **Courtes:** (`unistd.h`) un tiret, suivi d'une lettre, par exemple

```
$ ls -als  
$ gcc -o tp1 tp1.c
```

- **Longues:** deux tirets, suivis d'un mot pouvant contenir des tirets, par exemple:

```
$ valgrind --leak-check=yes ./prog  
$ ./isomap --help
```

- La bibliothèque `getopts.h` permet de gérer les options courtes et longues simultanément

La bibliothèque Cairo (1/2)

- Cairo est une bibliothèque permettant de dessiner des images **vectérielles**
- Elle supporte différents **formats**: PNG, PDF, SVG, etc.
- Site officiel: <https://www.cairographics.org/>
- Pour compiler/faire l'édition des liens, mieux vaut utiliser PKG-config

```
EXEC = hello
CFLAGS = $(shell pkg-config --cflags cairo)
LFLAGS = $(shell pkg-config --libs cairo)
```

```
$(EXEC): $(EXEC).o
        gcc $< -o $@ $(LFLAGS)
```

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

```
.PHONY: clean
```

```
clean:
        rm -f *.o $(EXEC)
```

La bibliothèque Cairo (2/2)

```
#include <cairo.h>

int main (int argc, char *argv[]) {
    cairo_surface_t *surface
        = cairo_image_surface_create(CAIRO_FORMAT_ARGB32,
                                     240, 80);
    cairo_t *cr = cairo_create (surface);

    cairo_select_font_face(cr, "serif",
        CAIRO_FONT_SLANT_NORMAL, CAIRO_FONT_WEIGHT_BOLD);
    cairo_set_font_size (cr, 32.0);
    cairo_set_source_rgb (cr, 0.0, 0.0, 1.0);
    cairo_move_to (cr, 10.0, 50.0);
    cairo_show_text (cr, "Hello, world");

    cairo_destroy (cr);
    cairo_surface_write_to_png (surface, "hello.png");
    cairo_surface_destroy (surface);
    return 0;
}
```

La bibliothèque SDL

- **SDL** est une autre bibliothèque C permettant de concevoir des applications graphiques
- Plusieurs autres applications sont basées sur SDL (Pygame, Kivy, etc.)
- Versions majeures: **SDL1.2** et **SDL2.0**
- Interaction de bas niveau avec les périphériques **graphiques** et **audio**
- Supporte seulement les formats BMP et WAV par défaut
- Bibliothèques **compagnonne** pour support PNG et MP3
- Voir l'application **Maze**