

Chapitre 6: Construction

INF3135

Construction et maintenance de logiciels

Alexandre Blondin Massé

Université du Québec à Montréal

v251



Construction

Définition (extraite de Wikipedia)

« Software construction is a software engineering discipline. It is the detailed creation of working meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging. It is linked to all the other software engineering disciplines, most strongly to software design and software testing. »

Mots-clé

- Combinaison
- Programmation (*coding*)
- Vérification
- Tests (unitaires et intégration)
- Débogage
- Design

Table des matières

- ① Modularité
- ② Macros
- ③ Modules en C
- ④ Bibliothèques

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
- **Réutilisation**: un module est souvent réutilisable

Terminologie

Varie selon le langage

- **Assemblage** (*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 colis (*package*)
- **C**: une paire de fichiers `.h/.c` (ou un seul)
- **C++**: une paire de fichiers `.hpp/.cpp` (ou un seul)
- **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** »
- Souvent présenté sous forme **déclarative**
- Documentation décrivant l'**utilisation**

Implémentation

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

Mesure de la modularité: couplage et cohésion

Objectifs

- **Minimiser** le couplage
- **Maximiser** la 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. »

Macros

Rappel

Précompilation

- **Interprétées** par le préprocesseur **avant** la compilation
- Symbole # au **début** de la ligne
- On peut insérer des **espaces** entre # et la directive

Directives permises

- #include: **inclusion** d'un fichier externe
- #define: définition d'un **symbole** ou d'une **macro**
- #undef: **annulation** d'un symbole ou d'une macro
- #ifdef/#ifndef: **vérifie** si une macro est définie ou non définie
- #if/#else/#elif/#endif: structure **conditionnelle**
- #error: indique une **erreur fatale**
- #pragma: pour des traitements plus **spécifiques**

Les macros

Macro

- **Fragment** de code
- Auquel on donne un **nom**
- Définie à l'aide de la directive `#define`

Deux types

- **Macro-objet**: ressemble à un objet ou une donnée
- **Macro-fonction**: ressemble à une fonction

Utilité

- Déclaration de **constantes** numériques ou textuelles
- Favorise la **réutilisation**
- Facilite la **méta-programmation**

Exemple

```
#include <stdio.h>
// Une macro définie par rapport à d'autres
#define AREA (WIDTH * HEIGHT)
// Même si elles sont définies après
#define WIDTH 100
#define HEIGHT 200
// On peut utiliser n'importe quelle expression
#define values {8, 3, 1, 4, 5};
// Une suite de plusieurs instructions
#define MULTIPLE_PRINTS printf("1-"); \
                        printf("2-"); \
                        printf("3")

int main(void) {
    printf("%d x %d = %d\n", WIDTH, HEIGHT, AREA);
    int t[] = values
    for (unsigned int i = 0; i < 5; ++i) printf("%d ", t[i]);
    MULTIPLE_PRINTS; printf("\n"); return 0;
}
```

Résultat:

```
100 x 200 = 20000
8 3 1 4 5 1-2-3
```

Mécanisme

- L'expression `#define symbole valeur`
- Remplace toutes les **occurrences** de symbole par valeur
- **Valeur**: reste de la ligne
- Valeur **multi-ligne**: à l'aide de \
- **Portée**: jusqu'à la fin du fichier
- Sauf si on trouve une commande `#undef`

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

```
int main(void) {  
# define X 10  
    printf("%d ", X);  
# undef X  
# define X 20  
    printf("%d ", X);  
    return 0;  
}
```

Résultat:

10 20

Visualiser l'expansion des macros

- Possible de visualiser l'**expansion** des macros
- En passant l'**option** -E à GCC:

```
$ gcc -E macro.c | tail -n 6
int main(void) {
    printf("%d x %d = %d\n", 100, 200, (100 * 200));
    int t[] = {8, 3, 1, 4, 5};
    for (unsigned int i = 0; i < 5; ++i) printf("%d ", t[i]);
    printf("1-"); printf("2-"); printf("3"); printf("\n"); return 0;
}
```

Options utiles:

- -E: n'effectue que la précompilation
- -C: conserver les commentaires
- -P: supprimer les marqueurs de ligne (de la forme #<i> <valeur>)

Symboles prédéfinis

Fichier predefined.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: predefined.c
Numéro de la ligne courante: 5
Date de compilation: Feb  3 2025
Heure de compilation: 13:04:59
Compilateur conforme à la norme ISO? oui
```

Définition de symboles à la compilation

- Il est possible de définir des symboles à la **compilation**
- Grâce à l'option `-D` de GCC
- **Exemples:**

Préciser la langue

```
gcc -DLANG=FR fichier.c
```

Récupérer une information dépendante du système

Rappel: `pwd` retourne le chemin absolu du répertoire courant

```
root_dir="$(pwd)"
```

```
gcc -DROOT_DIR="\ "$root_dir\" fichier.c
```

Récupérer le noyau du système

Rappel: `uname` retourne le nom du noyau

```
gcc -DKERNEL="\ "$(uname)\ " fichier.c
```

Désactiver les assertions

Remplacer `assert(...)` par `((void)0)`

On va y revenir

```
gcc -DNDEBUG fichier.c
```

Directives conditionnelles

```
// Si un symbole existe
```

```
#ifdef symbol
```

```
[...]
```

```
#else
```

```
[...]
```

```
#endif
```

```
// Si un symbole n'existe pas
```

```
#ifndef symbol
```

```
[...]
```

```
#else
```

```
[...]
```

```
#endif
```

```
// Structure conditionnelle générale
```

```
// condition doit être vérifiable à la précompilation
```

```
// On peut utiliser `defined` pour vérifier si un symbole est défini
```

```
#if condition
```

```
[...]
```

```
#elif condition
```

```
[...]
```

```
#else
```

```
[...]
```

```
#endif
```

Exemples

- Empêcher les inclusions **multiples**:

```
#ifndef MAP_H
#define MAP_H
[...]
#endif
```

- Corriger l'inclusion selon le système

```
#if defined(LINUX)
#   include <SDL2/SDL.h>
#   include <SDL2/SDL_image.h>
#elif defined(OSX) || defined(WINDOWS)
#   include <SDL.h>
#   include <SDL_image.h>
#endif
```

- Fournir une **valeur** par défaut à un symbole

```
#ifndef ROOT_DIR
#define ROOT_DIR "."
#endif
```

Macro-fonctions

- Ce sont des macros avec **paramètres**
- Le symbole doit être suivi d'une parenthèse **ouvrante**
- Les paramètres sont séparés par des **virgules**
- La **définition** commence après la parenthèse **fermante**

```
#include <stdio.h>

#define abs(X) ((X) >= 0 ? (X) : -(X))
#define min(X, Y) ((X) < (Y) ? (X) : (Y))

int main(void) {
    int a = 2, b = -5;
    printf("abs(%d) = %d", b, abs(b));
    printf("abs(%d) = %d\n", a - 9, abs(a - 9));
    printf("min(%d, %d) = %d", a, b, min(a, b));
    printf("min(%d, %d) = %d\n", -8, 5 * b, min(-8, 5 * b));
    return 0;
}
```

Résultat:

```
abs(-5) = 5      abs(-7) = 7
min(2, -5) = -5   min(-8, -25) = -25
```

Exemple: trace d'un programme

```
#include <stdio.h>
#include <math.h>
#ifdef NOTRACE
#   define trace(...) /**/
#else
#   define trace printf
#endif
int main(void) {
    trace("Program starts\n");
    for (unsigned int i = 0; i < 5; ++i) {
        trace("i = %d ", i);
        printf("sqrt(%d) = %lf\n", i, sqrt(i));
    }
    trace("Program ends\n");
}
```

Résultat:

```
$ gcc trace.c -lm -o trace
$ ./trace
Program starts
i = 0 sqrt(0) = 0.000000
i = 1 sqrt(1) = 1.000000
i = 2 sqrt(2) = 1.414214
i = 3 sqrt(3) = 1.732051
i = 4 sqrt(4) = 2.000000
Program ends
```

```
$ gcc -DNOTRACE trace.c -lm -o trace
$ ./trace
sqrt(0) = 0.000000
sqrt(1) = 1.000000
sqrt(2) = 1.414214
sqrt(3) = 1.732051
sqrt(4) = 2.000000
```

Substitution de chaîne (*stringizing*)

- Un **argument** ARG peut être utilisé comme chaîne littérale
- À l'aide de l'expression #ARG
- Noter que des chaînes **littérales** peuvent être **concaténées**
- Simplement en les séparant par des **espaces**

```
#include <stdio.h>
#include <stdbool.h>
#define warn_if(EXP) \
do { if (EXP) \
    printf("** " #EXP " ** "); } \
while (0)

bool divides_3(unsigned int i) { return i % 3 == 0; }

int main(void) {
    for (unsigned int i = 0; i < 8; ++i) {
        printf("%d ", i);
        warn_if(divides_3(i));
    }
    return 0;
}
```

Résultat:

```
0 ** divides_3(i) ** 1 2 3 ** divides_3(i) ** 4 5 6 ** divides_3(i) ** 7
```

Concaténation

- Un argument ARG peut être concaténé à un identifiant
- À l'aide de la notation ##
- **Exemple:**

```
#define function(NAME) draw_ ## NAME
```

- Alors `function(rectangle)` devient `draw_rectangle`
- Et `function(circle)` devient `draw_circle`

Fonction variadique

- Fonction ayant un nombre **arbitraire** d'arguments
- Commence avec k arguments **fixes**
- On doit en avoir **au moins un** ($k \geq 1$)
- Suivi de ...
- **Exemples**

```
int printf(const char *format, ...);  
int scanf(const char *format, ...);  
int fprintf(FILE *stream, const char *format, ...);
```

Combien d'arguments?

- Utiliser un des arguments **fixes**
- Utiliser un **format** ou un **masque**
- Utiliser une valeur spéciale qui **marque** la fin

Implémentation

- À l'aide de la bibliothèque standard `stdarg.h`
- Fournit un **type** `va_list`
- Et un ensemble de **fonctions** pour le manipuler:

```
// Initialise ap sur le premier argument  
void va_start(va_list ap, last);
```

```
// Récupère le prochain argument de ap  
type va_arg(va_list ap, type);
```

```
// Clôt la lecture d'arguments de ap  
void va_end(va_list ap);
```

- Les fonctions `va_start` et `va_end` doivent apparaître par **paires**

Exemple: `max` à plusieurs arguments

```
#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

Exemple: types variables (1/2)

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

void print_n_char(char c, int n) {
    for (unsigned int i = 0; i < n; ++i) printf("%c", c);
}

void print_n_chars(char c, int n, ...) {
    va_list list; va_start(list, n);
    print_n_char(c, n);
    while (true) {
        c = va_arg(list, int);
        if (c == '\0') break;
        int k = va_arg(list, int);
        print_n_char(c, k);
    }
}

int main (void) {
    print_n_chars('+', 1, '-', 10, '+', 1, '-', 10, '+', 1, '\0');
    printf("\n"); print_n_chars('a', 5, 'b', 8, 'c', 13, '\0');
    return 0;
}
```

Exemple: types variables (2/2)

Résultat:

```
+-----+-----+  
aaaaabbbbbbbcccccccccccc
```

Macros variadiques

- Macro-fonction ayant un nombre **arbitraire** d'arguments
- Commence avec k arguments **fixes**
- On peut avoir $k = 0$
- Suivi de ...
- **Exemples**

```
// Affichage formaté sur stderr
#define eprintf(...) fprintf(stderr, __VA_ARGS__)

// Débogage plus sophistiqué
#ifdef NDEBUG
#  define debug(format, ...) ((void)0)
#else
#  define debug(format, ...) \
    fprintf(stderr, "%s:%d:%s(): " FORMAT "\n", \
              __FILE__, __LINE__, __func__, __VA_ARGS__)
#endif
```

\end{lstlisting}

Exemple: Libtap

Extrait du fichier `tap.h`:

```
#define NO_PLAN                -1
#define SKIP_ALL               -2
#define ok(...)               ok_at_loc(__FILE__, __LINE__, __VA_ARGS__, NULL)
#define is(...)               is_at_loc(__FILE__, __LINE__, __VA_ARGS__, NULL)
#define isnt(...)            isnt_at_loc(__FILE__, __LINE__, __VA_ARGS__, NULL)
#define cmp_ok(...)          cmp_ok_at_loc(__FILE__, __LINE__, __VA_ARGS__, NULL)
#define cmp_mem(...)         cmp_mem_at_loc(__FILE__, __LINE__, __VA_ARGS__, NULL)
#define plan(...)            tap_plan(__VA_ARGS__, NULL)
#define done_testing()       return exit_status()
#define BAIL_OUT(...)        bail_out(0, "" __VA_ARGS__, NULL)
#define pass(...)            ok(1, "" __VA_ARGS__)
#define fail(...)            ok(0, "" __VA_ARGS__)

#define skip(test, ...)      do {if (test) {tap_skip(__VA_ARGS__, NULL); break;}} while (0)
#define end_skip

#define todo(...)            tap_todo(0, "" __VA_ARGS__, NULL)
#define end_todo             tap_end_todo()

#define dies_ok(...)         dies_ok_common(1, __VA_ARGS__)
#define lives_ok(...)        dies_ok_common(0, __VA_ARGS__)
```

Dangers associés aux macro-fonctions

```
#include <stdio.h>

#define abs(X) ((X) >= 0 ? (X) : -(X))
#define min(X, Y) ((X) < (Y) ? (X) : (Y))

int main(void) {
    int a = 2, b = -5;
    printf("abs(%d) = %d", b, abs(b));
    printf("abs(%d) = %d\n", a - 9, abs(a - 9));
    printf("min(%d, %d) = %d", a, b, min(a, b));
    printf("min(%d, %d) = %d\n", -8, 5 * b, min(-8, 5 * b));
    return 0;
}
```

- **Priorité** des opérateurs → bien parenthéser
- Inefficacité lors d'évaluations **multiples** → éviter les expressions non évaluées
- **Duplication** des effets de bord → éviter les expressions avec effets de bord
- **Instruction multiples** mal formatées → protéger avec des accolades

Modules en C

Modules en C

Deux types de fichiers

- `fichier.h`: contient l'**interface** ou l'en-tête (*header*)
- `fichier.c`: contient l'implémentation de cette interface

Trois types de modules

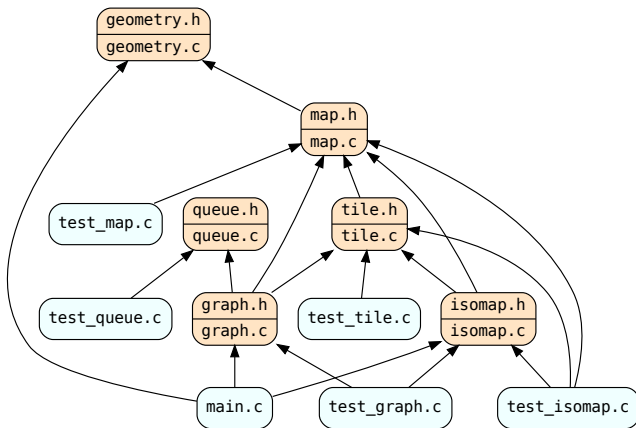
- **Régulier**: paire de fichier `.h` et `.c`
- **Déclaratifs**: seulement `.h`, avec déclarations et fonctions courtes
- **Point d'entrée**: seulement `.c` avec fonction `main`

Exemple

- `treemap.h`: interface d'un tableau associatif
- `treemap.c`: implémentation du tableau associatif
- `test_treemap.c`: teste le module `treemap`

Graphe de dépendances entre modules

- On trace une **flèche** du module m_1 vers le module m_2
- si m_1 **inclut** m_2 (« inclure » = utiliser la directive `#include`)
- Idéalement, le graphe est **acyclique**



Contenu d'une interface (fichier .h) (1/2)

- Types: enum, struct, union, typedef, ...
- Inclusions **minimales**
- Constantes, variables externes
- Macros, macros-fonctions
- Fonctions usuelles

Qualité de l'interface

- Documenter l'**utilisation**
- Protéger contre les **inclusions multiples** (`#ifndef MODULE_H`)
- **Uniformiser** les signatures des fonctions
- En particulier l'**ordre** des arguments
- Placer le **type principal** comme premier argument
- **Nommer** les paramètres dans les signatures
- **Préfixer** (ou **suffixer**) les noms avec celui du **module**
- Uniformiser l'**indirection** (pointeur ou pas pointeur)

Contenu d'une interface (fichier .h) (2/2)

Organisation suggérée:

```
/**
 * Docstring d'en-tête
 */
#ifndef MODULE_H
#define MODULE_H

// Inclusions avec < >
// Inclusions avec " "

// Déclaration des macros et des macros-fonctions

// Déclaration des types

// Déclaration des fonctions « publiques »
// Avec leur docstring respective

#endif
```

- Remplacer MODULE par le nom du module

Contenu de l'implémentation (fichier .c) (1/2)

Déclarations

- Inclure le fichier .h associé à l'implémentation
- Toute autre constante, macro, macro-fonction auxiliaire
- Fonctions auxiliaires

Qualité de l'implémentation

- Documenter le **développement**
- Mêmes remarques générales que pour l'**interface**
- Ne pas répéter les **inclusions** déjà dans .h
- Ne pas répéter les *docstrings* des fonctions « publiques »
- Idéalement, **documenter** les fonctions « privées »
- **Encapsulation**: cacher le plus possible l'implémentation

Contenu de l'implémentation (fichier .c) (2/2)

Organisation suggérée:

```
#include "MODULE.h"

// Inclusions avec < >
// Autres inclusions avec " "

// Déclaration de macros et de macros-fonctions auxiliaires
// Déclaration de types auxiliaires

// Déclaration et implémentation des fonctions auxiliaires
// Idéalement avec leur docstring respective

// Implémentation des fonctions « publiques »
// Sans leur docstring
```

- Remplacer MODULE par le nom du module

Compilation séparée

Étape 1: compilation des fichiers sources

- Avec `-Wall`, `-Wextra`, `-g`, ...
- Lien vers fichiers d'en-tête: `-I` (on va y revenir)

```
$ gcc -c [...] geometry.c
$ gcc -c [...] map.c
$ gcc -c [...] queue.c
[...]
```

- **Étape 2:** édition des liens
- Avec **options** pertinentes: `-lm`, `-ltap`, ...
- Lien vers bibliothèques: `-L` (on va y revenir)

```
$ gcc -o main main.o geometry.o map.o queue.o [...]
$ gcc -o test_queue queue.o test_queue.o [...]
$ gcc -o test_map map.o test_map.o [...]
[...]
```

Bibliothèques

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

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

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 **binaires**
- **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 **binaires** (`-L` et `-l`):

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

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

Exemple: Vec3D (1/2)

- Concevons notre propre bibliothèque: `vec3d`
- 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** avec `ar`:

```
$ 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

```
/home/blondin_al/clib/include  
/home/blondin_al/clib/lib
```

- Alors il suffit de compiler avec la commande

```
$ gcc -I/home/blondin_al/clib/include -c test_vec3d.c
```

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

```
$ gcc -L/home/blondin_al/clib/lib -o test_vec3d test_vec3d.o -lvec3d
```

Compilation de modules

- Souvent, un projet est divisé en plusieurs **modules**
- Il devient pénible de tout compiler **manuellement**
- En particulier en raison des **options** multiples
- **Compilation**: -Wall, -Wextra, -g, -I, ...
- **Édition des liens**: -l, -L, ...
- **Solution**: enrichir le contenu du Makefile

GNU Make

- Offre plusieurs **fonctions** (wildcard, shell, ...)
- Et **variables** spéciales (MAKE, MAKEFILE_LIST, ...)

Quelques mécanismes et fonctions

- 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 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** et **longues**
- **Courtes**: un tiret, suivi d'une lettre:

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

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

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

- Lorsque possible, préférer `getopts.h` à `unistd.h`
- Car gère **simultanément** les options courtes et longues

La bibliothèque Jansson (1/2)

- Permet de manipuler le format **JSON**
- Site officiel: <https://digip.org/jansson/>
- Dépôt Github: <https://github.com/akheron/jansson>
- Licence: **MIT**

```
{  
  "firstname": "Petri",  
  "lastname": "Lehtinen",  
  "num-followers": 295,  
  "projects": [  
    "jansson",  
    "optics-ts",  
    "sala"  
  ]  
}
```

La bibliothèque Jansson (2/2)

```
#include <stdio.h>
#include <string.h>
#include <jansson.h>

int main(void) {
    json_t* root = json_object();
    json_t* json_arr = json_array();
    json_object_set_new(root, "firstname", json_string("Petri"));
    json_object_set_new(root, "lastname", json_string("Lehtinen"));
    json_object_set_new(root, "num-followers", json_integer(295));
    json_object_set_new(root, "projects", json_arr);
    json_array_append(json_arr, json_string("jansson"));
    json_array_append(json_arr, json_string("optics-ts"));
    json_array_append(json_arr, json_string("sala"));
    char* s = json_dumps(root, 0);
    puts(s);
    json_decref(root);
    return 0;
}
```

Résultat:

```
$ gcc jansson.c -o jansson -ljansson && ./jansson
{"firstname": "Petri", "lastname": "Lehtinen", "num-followers": 295,
"projects": ["jansson", "optics-ts", "sala"]}
```

La bibliothèque Cairo (1/2)

- Permet de dessiner des images **vectérielles**
- Supporte différents **formats**: PNG, PDF, SVG, etc.
- Site officiel: <https://www.cairographics.org/>
- Compilation/édition des liens: plus facile avec 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

- Site officiel: <https://www.libsdl.org/>
- Permet de concevoir des applications **graphiques**
- À la **base** de plusieurs autres bibliothèques: Pygame, Kivy, ...
- Versions majeures: SDL**1.2** et SDL**2.0**
- Interaction de bas niveau avec les périphériques **graphiques** et **audio**
- Interface entre autres avec **OpenGL**
- Supporte seulement les formats BMP et WAV par défaut
- Voir l'application **Maze**

Bibliothèques compagnonnes

- `SDL_Image`: supporte le format PNG
- `SDL_mixer`: supporte le format MP3
- `SDL_ttf`: supporte le rendu de fontes (polices de caractères)

La bibliothèque Libtap (1/4)

- **Dépôt:** <https://github.com/zorgnax/libtap>
- Assez **élémentaire**
- Respecte le protocole **TAP**

Utilisation

- Installation **manuelle:**

```
$ make  
$ sudo make install  
$ sudo ldconfig
```

- Utilisation à l'aide de `#include <tap.h>`
- Fournit plusieurs **macros-fonctions** pratiques

La bibliothèque Libtap (2/4)

- Déclarer une **série** de tests:

```
plan(num_tests)
plan(NO_PLAN)
plan(SKIP_ALL)
plan(SKIP_ALL, format, ...)
done_testing()
```

- Vérifier une **assertion**:

```
ok(test)
ok(test, format, ...)
```

- Comparer des **chaînes**:

```
is(got, expected)
is(got, expected, format, ...)
isnt(got, unexpected)
isnt(got, unexpected, format, ...)
```

La bibliothèque Libtap (3/4)

- Comparer des valeurs **numériques**:

```
cmp_ok(a, op, b)
cmp_ok(a, op, b, format, ...)
```

- Comparer des **octets**:

```
cmp_mem(got, expected, n)
cmp_mem(got, expected, n, format, ...)
```

- Chercher des **expressions régulières**:

```
like(got, expected)
like(got, expected, format, ...)
unlike(got, unexpected)
unlike(got, unexpected, format, ...)
```

- Écrire des messages de **diagnostic**:

```
diag(fmt, ...)
```

La bibliothèque Libtap (4/4)

- Vérifier si un programme **plante**:

```
dies_ok(code)
dies_ok(code, format, ...)
lives_ok(code)
lives_ok(code, format, ...)
```

- **Sauter** une série de tests:

```
skip(test, n)
skip(test, n, format, ...)
end_skip
```

- Identifier des fonctionnalités **non implémentées**:

```
todo()
todo(fmt, ...)
end_todo
```