CARNUS Enseignement supérieur RODEZ

Compilation et Construction de bibliothèque

Création d'une bibliothèque (classe) pour Arduino

Prof : **K.B.**

18 mars 2025

2024–2025

BTS CIEL

Carnus Enseignement Supérieur
Avenue de Saint Pierre - Bourran - 12000 RODEZ, France

IR

Table des matières

Liste des figures

Liste des tableaux

1. Découper le programme en plusieurs fichiers

Le C (C++) permet de découper le programme en plusieurs fichiers source. Chaque fichier contient une ou plusieurs fonctions. On peut ensuite inclure les fichiers dont on a besoin dans différents projets.
Les fichiers d'en-tête contiennent les déclarations des types et fonctions que l'on souhaite créer.
Un programme écrit en C se compose généralement de plusieurs fichiers-sources. Il y a deux sortes de fichiers-sources :

— ceux qui contiennent effectivement des instructions ; leur nom possède l'extension `.c`,

— ceux qui ne contiennent que des déclarations ; leur nom possède l'extension `.h` (header ou en-tête).

Un fichier `.h` sert à regrouper des déclarations qui sont communes à plusieurs fichiers `.c`, et permet une compilation correcte de ceux-ci. Dans un fichier `.c` on prévoit l'inclusion automatique des fichiers `.h` qui lui sont nécessaires, grâce aux directives de compilation `#include`.
En supposant que le fichier à inclure s'appelle **entete.h**, on écrira `#include <entete.h>` s'il s'agit d'un fichier de la bibliothèque standard du C, ou `#include "entete.h"` s'il s'agit d'un fichier écrit par nous-mêmes.

Exemple 1.1

```c
/****** main.c ******/
#include <stdio.h>
#include "test.h"

int main()
{
    int const x(5);
    printf("Voici un calcul : %d", x+fonction());

    return 0;
}
```

```c
/****** test.c ******/
#include "test.h"

int fonction()
{
    return 12;
}
```

```c
/****** test.hpp ******/
#ifndef TEST_H
#define TEST_H

int fonction();

#endif
```

```
Générer les fichiers objet : gcc -c test.c main.c
Lier les fichiers :          gcc test.o main.o -o mon_programme.out
Exécuter :                   ./mon_programme.out

Ou
En une seule étape : gcc test.c main.c -o programme.out
Exécuter :           ./programme.out
```

Remarque 1.1 :

Pour l'exécution des programmes C++, il suffit de remplacer `gcc` par `g++`

2. Compilation

2.1 Les options du compilateur

- `gcc source.c` : Crée un fichier binaire exécutable `a.out`
- `-o` (`gcc source.c -o prog`) : Crée un fichier binaire exécutable nommé `Prog` (Spécifie le nom de l'exécutable à générer)
- `-v` : Visualise toutes les actions effectuées lors d'une compilation
- `-c` : Supprime l'édition de lien. Seul(s) le(s) fichier(s) `.o` est (sont) généré(s) pour réaliser des bibliothèques pré-compilées par exemple
- `-S` : Ne génère pas le fichier exécutable et crée un fichier `.s` contenant le programme en assembleur
- `-L` : Indique un chemin de recherche supplémentaire à l'éditeur de liens pour d'autres librairies
- `-l` : Indique une nouvelle librairie à charger

2.2 La compilation séparée

Lorsqu'on réalise de gros programmes (projets), on doit découper ceux-ci en plusieurs programmes sources. L'avantage est un gain de temps au moment de la compilation. Si une application est constituée des 3 fichiers sources `source1.c`, `source2.c` et `source3.c` la compilation d'un exécutable nommé `exec` est effectuée par la ligne suivante :

```
gcc -o exec source1.c source2.c source3.c
```

L'option `-c` du compilateur permet de générer des fichiers objets sans effectuer une édition de lien. Si on modifie un des fichiers sources (par exemple, `source2.c`), on utilise cette option pour ne compiler que le fichier modifié :

```
gcc -c source2.c
gcc -o exec source2.o source1.o source3.o
```

Remarque 2.1 : Compilation séparée

Cette technique devient indispensable lors de la création de gros logiciels qui peuvent nécessiter des temps de compilation de plusieurs minutes, ou dans le cas d'un travail collaboratif (projet par exemple).

Exemple

On réalise un programme qui demande de saisir le rayon d'un cercle, calcul et affiche le périmètre et la surface de ce cercle. Le programme source est découpé en trois fichiers sources :

— `main.c` : Programme principal

— `perimetre.c` , `surface.c` : Programmes contenant les fonctions `peri` et `surf` pour le calcul du périmètre et de la surface du cercle respectivement

— `cercle.h` : Le fichier d'en-tête (header) contenant le prototype des fonctions `peri` et `surf`

- Créer les trois programmes sources
- Créer le fichier d'en-tête `cercle.h`
- Compiler les fichiers `main.c perimetre.c surface.c` de manière à obtenir l'exécutable nommé `calcul`
  `$ gcc -o calcul main.c perimetre.c surface.c`
- Compiler `perimetre.c` et `surface.c` de manière à créer les fichiers objets `perimetre.o` et `surface.o`
  `$ gcc -c perimetre.c surface.c`
- Re-créer `calcul` en compilant et liant `main.c perimetre.o` et `surface.o`
  `$ gcc main.c perimetre.o surface.o` Exécutable généré ⟶ `a.out`
  `$ gcc -o calcul main.c perimetre.o surface.o` Exécutable généré ⟶ `calcul`

On souhaite modifier le programme `surface.c` de manière à retourner toujours la surface en cm^2 ($1\ m^2 = 10^4\ cm^2$)

- Modifier uniquement `surface.c`
- Créer uniquement un nouveau programme objet `surface.o` et re-compiler l'ensemble comme précédemment. Conclure.

```
$ gcc -c surface.c
$ gcc -o calcul surface.o main.c perimetre.o
$ ./calcul
```

Fichier `main.c`

```c
#include <stdio.h>
#include <math.h>
#include "cercle.h"

int main(void)
{
double r, p, s;
      printf("Entrer le rayon du cercle : ");
      scanf("%lf",&r);

      p = peri(r);
      s = surf(r);
      printf("\nLe périmètre du cercle de rayon %lf est : %lf",r,p);
      printf("\nLa surface du cercle de rayon %lf est : %lf\n",r,s);
      return(0);
}
```

Fichier `perimetre.c`

```c
#include <math.h>

double peri(double x)
{
      return 2*M_PI*x;
}
```

Fichier `surface.c`

```c
#include <math.h>

double surf(double x)
{
      return M_PI*pow(x,2)
      //return M_PI*pow(x,2)*10000;
}
```

Fichier `cercle.h`

```c
#ifndef CERCLE_H
#define CERCLE_H

double peri(double r);
double surf(double r);

#endif
```

3. Construction de librairie

Il est intéressant de se construire des bibliothèques (librairies) contenant les fonctions les plus fréquemment utilisées plutôt que de les réécrire à chaque projet ou programme. Il suffit ensuite d'indiquer vos librairies au moment de la compilation. Pour cela, les options `-L` et `-l` permettent respectivement d'inclure un nouveau chemin de recherche pour l'éditeur de lien et d'indiquer le nom de librairie.

3.1 Bibliothèque statique (.lib, .a) et bibliothèque partagée (.dll, .so)

Il existe deux types de bibliothèques externes : la bibliothèque statique et la bibliothèque partagée.

— Une bibliothèque statique a l'extension de fichier `.a` (fichier d'archive) sous Unix ou `.lib` (bibliothèque) sous Windows. Lorsque votre programme est lié à une bibliothèque statique, le code machine des fonctions externes utilisées dans votre programme est copié dans l'exécutable.

— Une bibliothèque partagée a l'extension de fichier `.so` (shared objects : objets partagés) sous Unix ou `.dll` (dynamic link library : bibliothèque de liens dynamiques) sous Windows. Lorsque votre programme est lié à une bibliothèque partagée, seule une petite table est créée dans l'exécutable.

Vous pouvez lister le contenu d'une bibliothèque via `nm filename`.

3.2 Exemple

On reprend les fichiers de l'exemple précédent :

- Créer un nouveau dossier (répertoire) nommé `TestLib`
- Copier et coller dans ce dossier uniquement les programmes `perimetre.c` et `surface.c`
- On compile les fichiers `perimetre.c` , `surface.c` pour générer les fichiers `objet` : `perimetre.o` , `surface.o`
  `$ gcc -c perimetre.c surface.c`
  Des avertissements (warning) peuvent apparaitre mais les fichiers `objet` seront générés normalement.
  Il est maintenant possible de lier ces fichiers `objet` lors d'une compilation.

Pour cette fois, il suffira de les déclarer dans un fichier d'en-tête (header), la compilation ne sera plus nécessaire d'où un gain de temps.

Fichier `libcercle.h`

```
#ifndef CERCLE_H
#define CERCLE_H

double peri(double r);
double surf(double r);

#endif
```

- Copier et coller dans le dossier `TestLib`, le programme `main.c` de l'exemple précédent
- Renommer ce programme `main2.c`
- Modifier-le comme montré ci-dessous

Fichier `main2.c`

```
#include <stdio.h>
#include <math.h>
#include "libcercle.h"

int main(void)
{
double r, p, s;
      printf("Entrer le rayon du cercle : ");
      scanf("%lf",&r);
```

```
        p = peri(r);
        s = surf(r);
        printf("\nLe périmètre du cercle de rayon %lf est : %lf",r,p);
        printf("\nLa surface du cercle de rayon %lf est : %lf\n",r,s);
        return(0);
}
```

- La compilation peut se faire par la commande : `$ gcc main2.c perimetre.o surface.o`
  Seul `main2.c` est compilé pour générer `main2.o` qui sera lié à `perimetre.o` et `surface.o`
  Les fichiers `objet *.o` peuvent être regroupés dans une bibliothèque.
- Créer la librairie `libcercle.a` (les noms des bibliothèques commencent toujours par `lib`) à l'aide de la commande `ar` qui crée un fichier archive :
  `ar r libcercle.a perimetre.o surface.o`
- On peut vérifier le contenu de la librairie en tapant :
  `ar t libcercle.a`

3.3 Exemple (suite) : Utilisation de la librairie

A partir de cette étape, on peut déjà utiliser ces fonctions en respectant quelques conditions.

— La première condition est de placer `libcercle.a` dans le répertoire où on effectue la compilation.

— La seconde condition est de compiler le programme en spécifiant le nom de la librairie.

- Compiler le programme en exécutant la ligne suivante :
  `$ gcc -o test main2.c libcercle.a`

— On peut également placer toutes nos librairies dans un de nos répertoires (par exemple, `"librairies"` qu'on créera par la commande `$ mkdir librairies`).

- On lance alors la compilation de la manière suivante :
  `$ gcc -o test2 main2.c -L librairies -l cercle`
  - `-L` indique au linker un nouveau chemin pour les bibliothèques
  - `-l` indique le nom de la bibliothèque. On remarque que le nom de la bibliothèque `libcercle.a` devient `cercle` dans la ligne de commande.

4. Création d'une librairie (classe) pour Arduino

4.1 Ecriture de la classe

On va implémenter les programmes dans une classe nommée Diode.

— Nom de la classe : `Diode`
— Attribut : `_pin` (privé)
— Méthodes :
  — `Diode(int pin)` : Constructeur de la classe (public)
  — `oncourt()` : void (public)
  — `onlong()` : void (public)

L'attribut (privé) de cette classe est `_pin` qui représente la broche où sera connectée la LED.
Les méthodes (publiques) sont `Diode(int pin)` le constructeur de la classe, `oncourt()` la méthode qui permet d'allumer la LED pendant une durée courte et `onlong()`, la méthode qui permet d'allumer la LED pendant une durée longue.

Diode
- _pin : int
+ Diode(int pin) : void + oncourt() : void + onlong() : void

- Création du fichier d'en-tête (header) nommé `Diode.h`

```
#ifndef DIODE_H
#define DIODE_H
#include "Arduino.h"

class Diode
{
    public:
        Diode(int pin);
        void oncourt();
        void onlong();
    private:
        int _pin;
};

#endif
```

La ligne `#include "Arduino.h"` permet d'avoir accès aux variables et constantes spécifiques à Arduino.
Une classe doit posséder une méthode spécifique (une fonction) appelée constructeur qui est chargé d'initialiser une instance (objet) de la classe. Le constructeur est appelé systématiquement au moment de la création de l'objet, il porte le même nom que la classe et ne retourne aucun type.
Les méthodes `oncourt()` et `onlong()` sont définies comme public.
L'attribut `_pin` est déclaré comme privé, cette variable contiendra le numéro du pin où la LED est branchée. On utilise généralement le caractère `_` pour distinguer l'appellation de l'attribut de l'argument de la fonction.

- Création du fichier nommé `Diode.cpp`
  Ce fichier contiendra la description des méthodes.

```
#include "Arduino.h"
#include "Diode.h"

Diode::Diode(int pin)
{
    pinMode(pin, OUTPUT);
    _pin = pin;
}

void Diode::oncourt()
{
    digitalWrite(_pin, HIGH);
    delay(250);
    digitalWrite(_pin, LOW);
    delay(250);
}

void Diode::onlong()
{
    digitalWrite(_pin, HIGH);
    delay(750);
    digitalWrite(_pin, LOW);
    delay(250);
}
```

Le fichier contient sur les deux premières lignes : `#include "Arduino.h"` (accès au fonctions Arduino) et `#include "Diode.h"` (définition de l'en-tête associé).
Puis vient la définition des méthodes.
Toutes les méthodes commencent par `Diode::`
On retrouve le constructeur (qui effectue aussi l'initialisation de la broche en sortie) puis les deux méthodes (`oncourt` et `onlong`) qui correspondaient aux fonctions dans le cas d'une "implémentation normale". On note la différence entre `pin` et `_pin`. Les méthodes ne "manipulent pas" `pin` mais sa "réplique" privée `_pin`.
`_pin` prend la valeur de `pin` au moment de l'instanciation de l'objet.

4.2 Utilisation de la classe

- Placer les deux fichiers créés précédemment dans un répertoire `Diode` (à créer) puis déplacer ce répertoire dans le repertoire `libraries` d'Arduino.
- Relancer Arduino et vérifier que le répertoire est bien vu par Arduino en effectuant l'opération `Croquis -> Importer bibliothèque`.
- Saisir le programme `ProgTest.ino` permettant de tester la classe réalisée (voir code ci-dessous)

```
#include <Diode.h>

Diode diode(13);

void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:
  diode.oncourt(); diode.oncourt(); diode.oncourt();
  delay(500);
  diode.onlong(); diode.onlong(); diode.onlong();
  delay(500);
  diode.oncourt(); diode.oncourt(); diode.oncourt();
  delay(1500);
}
```

- Vérifier le bon fonctionnement du programme `ProgTest.ino`

4.3 Création d'un exemple d'utilisation de la classe

Il est souhaitable lorsque l'on récupère une bibliothèque (une classe) d'avoir des exemples de mise en œuvre de cette bibliothèque.

- Il faut placer le répertoire `ProgTest` contenant le fichier `ProgTest.ino` (programme Arduino) dans un répertoire `examples` au sein de notre répertoire `Diode`.
  - ☐ `Arduino/Libraries/Diode/examples/ProgTest/ProgTest.ino`
  - ☐ `Diode.h` et `Diode.cpp` sont placés dans `Arduino/Libraries/Diode/`
- Relancer Arduino.
- Vérifier que le fichier `ProgTest.ino` est bien accessible à partir du menu Arduino `Fichier->Exemples->Diode`

4.4 Création d'un fichier de propriétés de la librairie

- Pour réaliser notre librairie dans les règles de l'art, on doit spécifier dans un fichier les propriétés de la librairie créée (voir l'exemple ci-dessous nommé `library.properties`) qui doit être inclus dans le répertoire `Diode`

```
name=Diode
version=1.0.0
author=Kamal B., LTP Charles Carnus
maintainer=LTP <carnuslab@carnus.fr>
sentence=Allows Arduino boards to control a variety of LEDs.
paragraph=This library can control a great number of LEDs.<br />
category=Device Control
url=https://www.carnus.fr/
architectures=avr,megaavr,sam,samd,nrf52,stm32f4,mbed,mbed_nano,mbed_portenta,mbed_rp2040
```

4.5 Coloration syntaxique de la classe et de ses méthodes

- Si on utilise les anciennes versions d'Arduino, il est possible que le nom de la classe et des méthodes soit coloré en noir. On peut obtenir la coloration syntaxique du programme réalisé en créant un fichier nommé `keywords.txt` (voir l'exemple ci-dessous).

```
#######################################
# Syntax Coloring Map LED
#######################################

#######################################
# Datatypes (KEYWORD1)
#######################################

Diode KEYWORD1 Diode

#######################################
# Methods and Functions (KEYWORD2)
#######################################
oncourt KEYWORD2
onlong KEYWORD2

#######################################
# Constants (LITERAL1)
#######################################
```

Chaque ligne porte le nom d'un mot clé suivi d'une tabulation (pas d'espaces) suivi de KEYWORD1 pour une classe ou de KEYWORD2 pour une méthode. La classe sera ainsi coloré en bleu-vert (ou orange), les méthodes en marron (la différence de couleur est faible). Il faut là aussi re-démarrer Arduino pour que la modification soit effective.

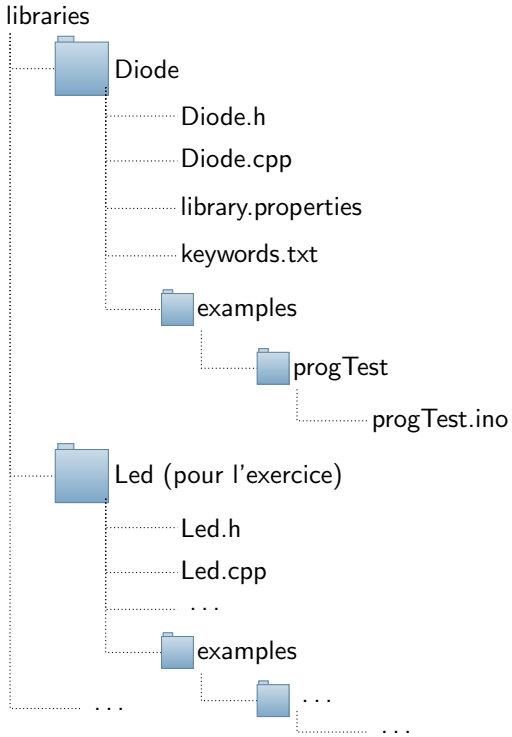

Figure 1. Contenu du répertoire

4.6 Exercice

Réaliser une classe permettant d'allumer une LED, de l'éteindre ou de la faire clignoter.

- Nom de la classe : `Led`
- Attributs :
  - `_pin` (privé)
  - `_etat` (privé)
- Méthodes :
  - `Led(int pin)` : Constructeur de la classe (public)
  - `ledOn()` : void (public)
  - `ledOff()` : void (public)
  - `ledClign(int T, int N)` : void (public)
  - `ledChange()` : void (public)
  - `ledEtat()` : void (public)

Led
- _pin : int - _etat : bool
+ Led(int pin) : void + ledOn() : void + ledOff() : void + ledClign(int T, int N) : void + ledChange() : void + ledEtat() : void

Définition des attributs

`_pin` : Correspond au numéro de la broche.
`_etat` : Correspond à l'état de la LED (false : LED éteinte ; true : LED allumée).

Définition des méthodes

`Led(int pin)` : Constructeur de la classe.
`ledOn()` : Allume la LED et renseigne son état.
`ledOff()` : Eteint la LED et renseigne son état.
`ledClign(int T, int N)` : Fait clignoter la LED N fois, la LED est allumée de 0 à $\frac{T}{2}$, éteinte de $\frac{T}{2}$ à T.
`ledChange()` : Allume la LED si elle est éteinte, éteint la LED si elle est allumée et renseigne l'état de la LED.
`ledEtat()` : Permet de récupérer l'état de la LED.

Remarque : `ledClign(int T, int N)` et `ledChange()` pourront utiliser `ledOn()` et `ledOff()`.

Ecrire tous les fichiers (`Led.h`, `Led.cpp`, fichier exemple, fichier keyword, fichier `library.properties`) permettant de faire fonctionner cette classe et de valider son bon fonctionnement.
Le fichier exemple devra tester toutes les méthodes et utilisera le moniteur série (en particulier) pour valider la méthode `ledEtat()`.

5. Résumé

5.1 Commande pour générer tous les fichiers

```
gcc -save-temps Prog.c -o progExe
```

5.2 Options

- `gcc -o exec source1.c source2.c source3.c` compilation et génération de l'exécutable `exec`
- Si on modifie un seul des fichiers :
  - `gcc -c source2.c`
  - puis `gcc -o exec source2.o source1.o source3.o`

5.3 Les librairies

- `-L` : Indique un chemin de recherche supplémentaire à l'éditeur de liens pour d'autres librairies
- `-l` : Indique une nouvelle librairie à charger
- `ar r libcercle.a perimetre.o surface.o` ensuite on vérifie le contenu de la librairie `ar t libcercle.a`
- `gcc -o test main2.c libcercle.a`
- `gcc -o test2 main2.c -L librairies -l cercle`

5.4 Autres options

- `$ g++ -Wall -g -o progExe source.cpp`
  - `-o` : spécifie le nom du fichier exécutable de sortie.
  - `-Wall` : imprime "tous" les messages d'avertissement.
  - `-g` : génère des informations de débogage symboliques supplémentaires à utiliser avec le débogueur gdb.