Types Abstraits & Base de la POO - TD 1

‡ ‡ ‡

Environnement de développement pour le langage C, C++ & Ada

I - Environnement de Développement

Exercice I.1

IDE Visual Studio Code

VisualStudio Code est un IDE Microsoft, multi-plateformes, multi-langages. Il devrait être installé sur les machines de l'École. Suivez les étapes suivantes afin de configurer votre dossier de travail.

1. Créer un répertoire de travail sur votre disque nommé Workspace-tabpoo
2. Ouvrir VisualStudio Code et cliquer sur « Add workspace folder ... ». Choisir le répertoire précédemment créé.
3. Dans l'explorer de l'IDE, créer un répertoire `td01-environnement` à la racine de votre workspace.
4. Dans ce nouveau répertoire, créer un nouveau fichier `un_test.c` et installer le plug-in associé au langage C si nécessaire.
5. Recopier le code suivant.

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
  printf("Coucou\n");
  printf("les\n");
  printf("gens de TABPOO !!\n");
  printf("en... C !\n");
  return 0;
}
```

Listing 1 – Fichier un_test.c

II - Toolchain GCC

Durant les activités en langage C de ce cours, nous utiliserons la *toolchain* GCC.

GCC :

L'exécutable gcc est un programme permettant d'appeler le préprocesseur, le compilateur, l'assembleur et le linker pour les langages C et C++. Les opérations réalisées sont les suivantes :

- appel du préprocesseur C (nommé cpp) ;
- appel du compilateur C (nommé cc1), ou C++ (cc1plus) selon le type de fichier à compiler (les fichiers .c sont compilés en C, les fichiers .C, .cc et .cpp sont compilés en C++). Les compilateurs génèrent un fichier assembleur ;
- appel de l'assembleur (as) pour générer le fichier objet, si l'option -S n'est pas spécifiée ;
- appel de l'éditeur de liens (ld) pour générer l'exécutable ou la bibliothèque, si l'option -c n'est pas spécifiée. Dans ce cas, la librairie C standard est incluse par défaut dans la ligne de commande de l'édition de liens.

Le programme gcc accepte différentes options. Voici quelques-unes des options les plus fréquentes :

Option	Description
`--help`	Affiche l'aide de gcc.
`--version`	Donne la version de gcc.
`-E`	Appelle le préprocesseur. N'effectue pas la compilation.
`-S`	Appelle le préprocesseur et effectue la compilation.
`-c`	Appelle le préprocesseur, le compilateur et l'assembleur, mais pas l'éditeur de lien. Seuls les fichiers objets (.o) sont générés.
`-o nom`	Spécifie le nom du fichier objet généré.
`-w`	Supprime tous les warnings.
`-W`	Active les warnings supplémentaires.
`-Wall`	Active tous les warnings possibles.
`-g`	Construit une version exécutable par le debugger.

Exercice II.1

Compilation & Construction

1. Dans la vue « terminal » de l'IDE, saisir la commande suivante : `gcc -o un_test.exe un_test.c`
2. Lister le contenu du répertoire et remarquer la création du nouvel exécutable
3. Lancer l'exécution par la commande : `./un_test.exe`

Exercice II.2

Debuggage

Nous allons maintenant exécuter notre programme en mode « debug ».

GCC - Debuggage :

Pour pouvoir utiliser le mode de debuggage proposé par GCC, il est nécessaire de reconstruire l'exécutable en utilisant l'option -g (Cf. tableau précédent). La toolchain GCC permet alors d'exécuter ce programme en permettant de nombreuses opérations durant l'exécution. Voici un petit résumé des actions les plus classiques :

Action	Description
quit(q)	quitter gdb
run(r)	lancer l'exécution
start	lancer l'exécution et arrêt au début de la procédure principale
break,watch,clear,delete(b,w,cl,d)	introduire un point d'arrêt, ou « surveiller » une variable, effacer un point d'arrêt
info break (i b)	affiche tous les break et watch
step,next,until,continue(s,n,u,c)	avancer d'un pas en entrant ou pas dans les sous-fonctions ou fin de boucle, continuer
print,backtrace,list(p,bt,l)	afficher la valeur d'une variable, la pile d'exécution, afficher le code où on est arrêté.

4. Construire le programme en mode « debug » : `gcc -g -o un_test.exe un_test.c`
5. Lancer le debugger sur le programme : `gdb un_test.exe`
6. Réaliser une exécution complète.
7. Réaliser une exécution « pas-à-pas »

III - Makefile

Dans le but de se simplifier la vie lors de la compilation, de la construction et du nettoyage des différents projets que nous réaliserons, nous utiliserons l'outil `make`.
Cet outil est très général et son rôle est d'automatiser différentes tâches, et en particulier la compilation et la construction d'un projet.
Voyons, très rapidement, les bases du fonctionnement de cet outil.

Make & Makefile :

L'outil `make`, exécuté dans un répertoire, fonctionne en interprétant un fichier `Makefile`. Ce dernier contient différentes directives, ou règles, écrites sous la forme suivante :

```
production: source(s)
    commande(s)
```

La ou les source·s nécessaire·s à une production peuvent être soit des fichiers existants, soit des productions provenant d'autre règles. Il est ainsi possible d'enchaîner différentes règles pour réaliser le travail souhaité.
Par exemple, si nous souhaitons réaliser la compilation d'un fichier `prog.c`, un fichier `Makefile` très simple serait le suivant :

```
1 prog: prog.o
2     gcc -o prog prog.o
3 prog.o: prog.c
4     gcc -o prog.o -c prog.c
```

L'idée n'étant pas ici de recopier le manuel de l'outil `make`, nous allons voir, au travers d'exemples, différentes notions comme la déclaration et l'utilisation de variables ou encore l'utilisation de variables automatiques et de règles implicites.

Néanmoins, voici un petit pense-bête donnant la sémantique de diverses variables automatiques :

$@	produit de la règle
$<	nom de la première dépendance
$?	toutes les dépendances plus récentes que le produit
$^	toutes les dépendances
%	joker permettant la définition de pattern

Enfin, l'utilisation de `make` est très simple et se fait selon la syntaxe suivante :

```
make [-f nom_fichier] [regle]
```

Exercice III.1

Un premier `Makefile`...

Nous allons faire une première utilisation de l'outil `make` et, au passage organiser un peu mieux notre répertoire de projet.

1. Dans votre répertoire `td01-environnement`, créer un fichier `Makefile` et recopier le code ci-dessous :

```
CC=gcc
CFLAGS= -Wall -std=c90 -pedantic -g
EXEC=un_test
OBJ_DIR=.obj
SRC_DIR=src_c
BIN_DIR=exec

all: $(EXEC)

un_test: $(BIN_DIR)/un_test

$(BIN_DIR)/un_test: $(OBJ_DIR)/un_test.o
  $(CC) $(CFLAGS) -o $(BIN_DIR)/un_test $(OBJ_DIR)/un_test.o

$(OBJ_DIR)/un_test.o: $(SRC_DIR)/un_test.c
  $(CC) $(CFLAGS) -o $(OBJ_DIR)/un_test.o -c $(SRC_DIR)/un_test.c

clean:
  rm -rf $(OBJ_DIR)/*.o

mrproper : clean
  rm -rf $(BIN_DIR)/$(EXEC)
```

Listing 2 – Fichier Makefile

2. Analyser ce contenu et réaliser les modifications nécessaires à votre projet afin de compiler et construire l'exécutable `un_test`.
3. Nettoyer votre projet à l'aide de la commande `make`.

Exercice III.2

Un `Makefile` plus générique...

Comme vous avez pu le remarquer, le nom du fichier à traiter apparaît de nombreuses fois dans la version du fichier précédente. L'objectif est donc ici de créer un nouveau fichier plus générique qui pourra s'appliquer aux différents projets.

1. Supprimer les répertoires `.obj` et `exec`
2. Créer un fichier `Makefile_v2` dans votre répertoire de projet et recopier le code ci-dessous :

```
CC=gcc
MKDIR=mkdir -p
CFLAGS= -Wall -std=c90 -pedantic -g
OBJ_DIR=.obj
SRC_DIR=src_c
BIN_DIR=exec
SRC= $(wildcard $(SRC_DIR)/*.c)
OBJ=$(patsubst $(SRC_DIR)/%.c,$(OBJ_DIR)/%.o,$(SRC))
EXEC=$(patsubst $(SRC_DIR)/%.c,%,$(SRC))
OUT_DIRS = $(OBJ_DIR) $(BIN_DIR)
```

```
11
12 .SECONDARY: $(OBJ)
13
14 .PHONY: all clean mrproper
15
16 all: $(EXEC)
17
18 $(EXEC): %: $(BIN_DIR)/%
19
20 $(BIN_DIR)/%: $(OBJ_DIR)/%.o $(BIN_DIR)
21   @echo "[Link]"
22   $(CC) -o $@ $<
23
24 $(OBJ_DIR)/%.o: $(SRC_DIR)/%.c $(OBJ_DIR)
25   @echo "[Compile & Build]"
26   $(CC) $(CFLAGS) -o $@ -c $<
27
28 $(OUT_DIRS):
29   @echo "[Setup]"
30   $(MKDIR) $@
31
32 clean:
33   @echo "[Suppr. Obj files]"
34   rm -rf $(OBJ_DIR)/*.o
35
36 mrproper : clean
37   @echo "[Suppr. Exec files]"
38   rm -rf $(BIN_DIR)/$(EXEC)
```

Listing 3 – Fichier Makefile_v2

3. Construire l'exécutable à l'aide de ce nouveau fichier et de la commande `make`.
4. Nettoyer votre projet.
5. Renommer votre fichier source, reconstruire l'exécutable et nettoyer de nouveau votre projet.
6. Redonner le nom initial à votre fichier source.

IV - Devenons polyglotte... C++ & Ada

Parmi les multiples objectifs de ce cours, il y a celui de vous aguerrir dans différents langages...
Nous essaierons donc, autant que faire se peut, de réaliser les différents exercices à venir non seulement en C, mais aussi en C++ et en Ada !

Exercice IV.1

Du code... C++

1. Créer, dans votre répertoire de projet, un répertoire `src_c++`.
2. Créer le fichier `un_test.cpp` à l'intérieur de ce nouveau répertoire et recopier le code ci-dessous :

```cpp
#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    cout << "Coucou\n";
    cout << "les\n";
    cout << "gens de TABPOO !!!\n";
    cout << "en... C++ !\n";
    return 0;
}
```

Listing 4 – Fichier un_test.cpp

3. Compiler ce programme à l'aide de la commande `g++ -W -Wall -std=c++20 -pedantic -o un_test src_c++/un_test.cpp`
4. Exécuter puis supprimer le fichier exécutable créé.

Exercice IV.2

Du code... en Ada

1. Créer, dans votre répertoire de projet, un répertoire `src_ada`
2. Créer le fichier `un_test.adb` à l'intérieur de ce nouveau répertoire et recopier le code ci-dessous :

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure Un_Test is
begin
    Put_Line (Item => "Coucou");
    Put_Line (Item => "les");
    Put_Line (Item => "gens de TABPOO !!!");
    Put_Line (Item => "en... Ada !");
end Un_Test;
```

Listing 5 – Fichier un_test.adb

3. Compiler ce programme à l'aide de la commande `gnatmake src_ada/un_test.adb`
4. Exécuter puis supprimer le fichier exécutable créé, ainsi que les fichiers temporaires.

Exercice IV.3

Un manager... pour les compiler tous : Gpr !

Afin de pouvoir compiler et construire les exécutables issus des sources écrites dans ces trois langages, nous utiliserons le manager de projet Gpr (Gnat Project Manager).
C'est un outil assez puissant que vous avez déjà utilisé lors de votre première année à l'École. Il permet, entre autres de prendre en charge différents langages et différentes configurations. C'est ce que nous allons découvrir maintenant.

1. À la racine de votre répertoire de projet, créer un fichier `td01.gpr` et recopier le code suivant :

```
project Td01 is
    type T_Lang is ("c", "c++", "ada");
    type T_Ext is ("c", "cpp", "adb");
    Lang : T_Lang := external ("lang", "c");
    Ext : T_Ext := "c";
    case Lang is
        when "ada" => Ext := "adb";
        when "c++" => Ext := "cpp";
        when others => Ext := "c";
    end case;
    type Mode_Type is ("debug", "release");
    Mode : Mode_Type := external ("mode", "debug");

    for Create_Missing_Dirs use "True";
    for Languages use (external ("lang", "c"));
    for Source_Dirs use ("src_" & Lang);
    for Main use (
        "un_test." & Ext
        );
    for Object_Dir use ".obj/"  & Lang &"/" & Mode;
    for Exec_Dir use "exec/" & Lang &"/" & Mode;

    package Compiler is
        for Driver ("C") use "gcc";
        for Driver ("c++") use "g++";
        case Mode is
            when "debug" =>
            for Switches ("C") use ("-g","-W","-Wall","-Werror","-std=c90","-pedantic");
            for Switches ("C++") use ("-g","-W","-Wall","-Werror","-std=c++20","-pedantic");
            for Switches ("Ada") use ("-g","-gnatwae","-gnatyguxSI","-gnaty4");
            when "release" =>
            for Switches ("C") use ("-O2","-W","-Wall","-Werror","-std=c90","-pedantic");
            for Switches ("C++") use ("-O2","-W","-Wall","-Werror","-std=c++20","-pedantic");
            for Switches ("Ada") use ("-O2","-gnatwae","-gnatyguxSI","-gnaty4");
        end case;
    end Compiler;
end Td01;
```

N'ayez craintes, votre gentil encadrant va prendre le temps de vous expliquer tout-cela...

2. Nous allons maintenant lancer la construction du programme en langage C. Dans votre terminal, exécuter la commande `gprbuild`
   Analyser l'impact sur votre répertoire de projet et exécuter le fichier résultant.
   Faire de même avec la commande `gprbuild -Xmode=release`
3. Passons au langage C++. Toujours dans le terminal, exécuter la commande `gprbuild -Xlang=c++`
   Analyser l'impact sur votre répertoire de projet et exécuter le fichier résultant.
   Faire de même avec la commande `gprbuild -Xlang=c++ -Xmode=release`
4. Voyons maintenant le cas du langage Ada. Toujours dans le terminal, exécuter la commande `gprbuild -Xlang=ada`
   Analyser l'impact sur votre répertoire de projet et exécuter le fichier résultant. Faire de même avec la commande `gprbuild -Xlang=ada -Xmode=release`
5. Pour « nettoyer » votre répertoire de projet, exécuter la suite de commandes suivantes et noter l'impact de chacune d'elle :
   ```
   gprclean
   gprclean -Xmode=release
   gprclean -Xlang=c++
   gprclean -Xlang=c++ -Xmode=release
   gprclean -Xlang=ada
   gprclean -Xlang=ada -Xmode=release
   ```
6. En associant l'outil `make` au manager de projet `Gpr`, nous pouvons encore nous simplifier la vie !
   À la racine de votre répertoire de projet, créer le fichier `Makefile_gpr`.
   En vous basant sur les exemples précédents de ce type de fichier, ajouter le contenu de ce fichier afin de pouvoir construire/nettoyer toutes les versions d'exécutables issus des 3 langages en une seule commande.

‡ ‡ ‡