Mise à niveau en C

Préprocesseur, Makefile et compilation optimisée

Enseignant: P. Bertin-Johannet

Le préprocesseur

Le préprocesseur

- Le **préprocesseur** est la première étape de la compilation.
- On peut l'utiliser pour manipuler le texte de nos fichiers.
- Pour cela il faut utiliser des **directives** (commençant par un dièze).
- Toutes ces directives manipulent du texte sans se soucier de son interprétation (types etc…).
- Si on souhaite afficher le texte créé par le préprocesseur, on utilise l'option `-E`.

```
gcc main.c -E
```

La directive define

- La directive `#define` permet de créer une variable préprocesseur.
- Une variable préprocesseur contient uniquement du texte.
- Le nom de la variable sera remplacé par le texte dans la suite du programme.

```
#define N 1
#define PLUS +
#define A_EQ int a =
int main(){
  A_EQ N PLUS N;
}
```

Les conditions

- Il est possible d'inclure du code conditionellement en utilisant `#if`, `#else`, `#elif` et `#endif`.

```
#define N 1
int main(){
  #if N == 5
    printf("n = 5\n");
  #elif N == 6
    printf("n = 6\n");
  #endif
}
```

La directive ifdef

- La directive `#ifdef` permet d'include du code uniquement si une variable préprocesseur est définie.

```c
#define N 1
#ifdef N
int f(){return 1;}
#else
int f(){return 4;}
#endif

int main(){
  printf("%d\n", f());
} // affiche 1
```

La directive ifndef

- La directive `#ifndef` permet d'include du code uniquement si une variable préprocesseur n'est pas définie

```
#define N 1
#ifndef N
int f(){return 1;}
#else
int f(){return 4;}
#endif

int main(){
  printf("%d\n", f());
} // affiche 4
```

La directive undef

- La directive `#undef` permet de supprimer la définition d'une variable.

```c
#define N 1
//...
#undef N
#ifndef N
int f(){return 1;}
#else
int f(){return 4;}
#endif

int main(){
  printf("%d\n", f());
} // affiche 1
```

La directive include

- La directive préprocésseur `#include` permet de copier le texte écrit dans un fichier

```
// copie colle le texte des fichiers bonjour.txt, main.c et run.sh
#include "bonjour.txt"
#include "main.c"
#include "run.sh"
int a;
// copie colle une deuxième fois le texte du fichier bonjour.txt
#include "bonjour.txt"
```

Problème de l'utilisation d'include

- Si on inclut deux fichiers qui incluent un même fichier, ce dernier sera collé deux fois.

```
#include "user.h" // user.h copie le contenu de string.h
#include "file.h" // file.h copie le contenu de string.h
// string.h est copié deux fois
```

- Toutes les déclarations de fonctions ou de structs dans le fichier inclus seront donc dupliquées et gcc affichera une erreur.

Solution

- Quand on crée un fichier qui sera, on crée une variable préprocesseur unique.
- On inclut le contenu du fichier uniquement si la variable n'existe pas.
- Ainsi, si le fichier a déja été inclus, le code ne sera pas inséré.

```
// fichier user.h
// le code est inclus seulement si la variable USER_H n'existe pas
#ifndef USER_H
#define USER_H // on défini la variable au premier passage
...  // on insère le code du fichier ici
#endif
```

Optimisation de la compilation

Unité de compilation

- Lorsqu'on modifie un fichier C, le compilateur recompile tout le code du fichier.
- Si un fichier **A** inclus un fichier **B**, alors le compilateur devra recompiler les deux à chaque changement de l'un ou l'autre.
- Cela peut causer des lenteurs de compilation dans une grande base de code.
- Pour éviter cela, on peut demander à gcc de compiler des fichiers séparéments.

Unité de compilation

- On peut passer l'option `-c` à gcc pour qu'il crée un fichier **objet**.
- Cela permet de compiler le code des fonctions dans le fichier sans faire le lien avec les fonctions des autres fichiers.
- Une fois que l'on a compilé tous les fichiers séparéments, on passe les fichiers objets en argument pour créer un executable.

```
  # si j'ai modifié uniquement le fichier bdd.c, je ne recompile pas les
autres
  gcc -c bdd.c
  gcc bank.o bdd.o main.o
```

Fichiers headers

- Pour que le code d'un fichier puisse utiliser les fonctions définies dans un autre, il faut qu'elles soient déclarées.
- On sépare donc souvent:
  - Les fichiers contenant uniquement des **déclarations** (extension: .h) qui pourront être inclus dans les autres fichiers.
  - Les fichiers contenant les **implémentations** (extension: .cpp) qui devront être compilés séparéments.
- Le lien entre les fonctions sera effectué par une autre partie du compilateur appelée le **linker**.

Fichiers headers

```c
// main.c
// on inclu la définition de la fonction bonjour
#include "bonjour.h"
int main(){
  bonjour();
}
```

```c
// bonjour.h
#ifndef BONJOUR_H
#define BONJOUR_H
void bonjour();
#endif

// bonjour.c
void bonjour(){
  printf("bonjour\n");
}
```

Make

Sujet du cours

- Il peut être fastidieux de suivre manuellement les fichiers que nous avons modifiés pour les recompiler.
- Pour automatiser le processus de détection et de recompilation des fichiers modifiés on peut utiliser plusieurs outils.
- Ce cours présente brièvement l'utilisation de **GNU Make** mais il en existe d'autres (notament **CMAKE**).

Cibles

- Make est un langage qui permet d'executer des **commandes** afin de créer des **cibles**.
- Pour cela on définit dans un fichier une liste de **cibles** suivies de la commande qui permet de la réaliser.
- On peut ensuite écrire `make cible` pour créer la cible désirée.

```
ex1.out:
  gcc ex1.c -o ex1.out
data.csv:
  python extract_data.py
```

Dépendances

- On peut spécifier des **dépendances** pour une cible.
- Make réalisera alors les commandes correspondant aux dépendances avant d'executer la commande

```
# ici make executera les target gcc et data.csv avant d'executer ex1.out
    ex1.out: gcc, data.csv
      gcc ex1.c -o ex1.out
    data.csv:
      python extract_data.py
    gcc:
      sudo apt install build-essentials
```

Détection de modifications

- Make réalise une cible uniquement si une de ses dépendances a été modifiée plus récement.

```
# ici main.o ne sera recréé que si main.c a été modifié
main.o: main.c
  gcc -c main.c
# bank.o ne sera recréé que si bank.c a été modifié
bank.o: bank.c
  gcc -c bank.c
# ex1.ou ne sera recompilé que si une des deux dépendances a été recompilée
ex1.out: main.o bank.o
  gcc main.o bank.o -o ex1.out
```

Patterns

- On peut créer des cibles sous la forme de **pattern** en utilisant les symboles % et <$.

```
  # Ici on crée une cible pour tout fichier finissant par .o
  # on definit une dépendance vers un fichier du même nom mais finissant par .c
    # $< est automatiquement remplacé par la première dépendance spécifiée (ici le
fichier .c)
  %.o: %.c
      gcc -c $<

  ex1.out: main.o
      gcc main.o -o ex1.out
```

Variables

- On peut créer des **variables** en leur donnant un nom et une valeur.
- On peut ensuite les appeler en la plaçant entre parenthèses après un **$**.

```
# On crée la variable SOURCE
SOURCE = main.c

compile: $(SOURCE)
  gcc $(SOURCE) -o ex1.out
```

Substitution de variables

- On peut automatiquement créer des variables en **substituant** un texte par un autre.
- On précise donc:
  - La variable d'origine.
  - Le texte à remplacer.
  - Par quoi le remplacer.

```
SOURCES = main.c bank.c list.c
# On remplace .c par .o dans la variable SOURCES
OBJECTS = $(SOURCE:.c=.o)
# La variable OBJECTS contiendra alors:
# main.o bank.o list.o
ex1.out: OBJECTS
  gcc $(OBJECTS) -o ex1.out
```

Wildcard

- On peut aussi générer une variable qui contient tous les fichiers d'un répertoire correspondant à un pattern.
- On écrit alors: `$(wildcard pattern)`.

```
  # La variable SOURCES contient alors tous les noms de fichiers se terminant
par .c
  SOURCES = $(wildcard *.c)
  # On peut par exemple obtenir la liste des fichiers commençant par table et
terminant par .csv
  TABLES = $(wildcard table*.csv)
```

Makefile complet pour un projet C

```
# On obtient tous les fichiers .c
SRC = $(wildcard *.c)

# On génère automatiquement la liste des fichiers objets
OBJ = $(SRC:.c=.o)

# On spécifie comment créer un fichier objet
%.o: %.c
  gcc -c $<

# On spécifie comment créer l'executable final
final: $(OBJ)
  gcc $(OBJ) -o ex1.out
```

Mise en pratique

```
printf("

          ________
        < Mooooooo >
          --------
                \   ^__^
                 \  (oo)\_______
                    (__)\       )\/\
                        ||----w |
                        ||     ||

        ")
```