Algorithmique et programmation

Makefiles

R.Gosswiller

Sommaire

Modularité

Modularité du code

Problématique

Comment morceler du code entre modules ?

Problématique

Répartir son code entre plusieurs fichiers en utilisant des mécanismes d'inclusion

Les fichiers headers

Définition

Un fichier header est une interface de communication entre la source et le reste du code

Détails

Définition de tous les aspects publics du code : prototypes, structures, variables globales, inclusions...

Les fichiers headers

file.h

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <libext.h>
4
5  int f(int a, int b);
```

file.c

```
1  #include "file.h"
2
3  int f(int a, int b) {
4      return a+b;
5  }
```

Les fichiers headers

Compilation

```
1                #include <stdio.h>
2                #include <stdlib.h>
3                #include <libext.h>
4
5                int f(int a, int b);
6
7                int f(int a, int b) {
8                    return a+b;
9                }
```

Modularité et compilation

Syntaxe

Le lien est fait automatiquement du .c au .h par le compilateur

Syntaxe

```
1                gcc file.c
```

Les structures opaques

Principe

Types et structures peuvent être définis séparément
La définition du type peut être faite publiquement (.h) et la structure en privé (.c)

Syntaxe

```
1  struct toto_ {};
2
3  typedef struct toto_ toto;
```

Intérêt

Dissimuler la définition d'une structure
Différencier les types publics et protégés

Les fichiers headers

file.h

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct toto_ * toto;
```

file.c

```
1  struct toto_ {
2      int a;
3      int b;
4      element e;
5  };
```

Makefiles

Makefiles

Définition

Un makefile est un script de compilation automatique

Intérêt

Garantir la structure d'un projet
Ne recompiler que les sources mises à jour
Faciliter le travail du développeur

Pourquoi les Makefiles ?

Etape 1 : la compilation

```
1                    gcc file.c
```

Etape 2 : Les options

```
1                gcc file.c -Wall -ansi -o exec -j2
2                    -ffloat-store
3                    -fforward-propagate
4                    -ffunction-sections -ffast-math
```

Pourquoi les Makefiles ?

Etape 3 : Les fichiers

```
1  gcc file.c main.c toto.c prog.c lib.c functions.c
2      tests.c external.c source.c bob.c -Wall -ansi
3      -o exec -j2  -ffloat-store -fforward-propagate
4      -ffunction-sections -ffast-math
```

Etape 4 : Les chemins

```
1  gcc ../prog/file.c main.c lib/toto.c
2      /home/toto/lib/prog.c
3      /home/toto/lib/lib.c main/functions.c
4      main/tests.c ext/external.c
5      source.c bob.c -Wall -ansi -o exec -j2
6      -ffloat-store
7      -fforward-propagate -ffunction-sections
8      -ffast-math
```

Pourquoi les Makefiles ?

Conclusion

Synthétiser la compilation dans un script dédié !

Pourquoi les Makefiles ?

Problème

La compilation est une étape pouvant prendre un délai très long

Exemple

Compilation programme lourd avec dépendances (quelques minutes)
Compilation noyau/OS (quelques heures)
Compilation multi-projets (quelques jours)

Solution

Ne recompiler que le strict nécessaire : utiliser les Makefiles

Makefiles

Règles de compilation

Un Makefile est composé d'un ensemble de règles de compilation

Syntaxe

```
1  cible : dependances
2      commande de compilation
```

Exemple

```
1  prog : file.o
2      gcc -o prog file.o
3
4  file.o : file.c file.h
5      gcc -Wall -o file.o file.c
```

Makefiles

Makefile enrichi

Un makefile peut ensuite être enrichi
Définition de règles, de variables, d'options

Exemple

```
1   CC=gcc
2   CFLAGS= -Wall -ansi -W -pedantic
3
4   all : main.o
5       $(CC) -o prog file.o $(CFLAGS)
6
7   main.o : file.o
8       $(CC) -o main.o file.o main.c $(CFLAGS)
9
10  file.o : file.c file.h
11      $(CC) -o file.o file.c $(CFLAGS)
12
13  clean :
14      rm -rf *.o
```

Makefiles

Nécessite l'outil de compilation make

Commande de compilation

make cible

Exemples

make all
make clean

Compilation et gcc

Généralités sur la compilation

Etapes

"Compiler" avec un gcc est un processus découpé en 4 étapes :

- Le pre-processing
- La compilation
- L'assemblage
- Le linking (ou édition de liens)

La compilation

La compilation

Génération d'un fichier assembleur (.as) depuis la source C

Assemblage

Construction d'un fichier objet (.o) depuis le fichier assembleur
Assemblage en mémoire des instructions

Le linking

Principe

Vérification de l'implémentation des fonctions
Regroupement des fichiers
Production de l'exécutable (binaire) de sortie

Options utiles de GCC

- -Wall : Affiche tous les warnings de compilation
- -o : Détermination de l'exécutable de sortie
- -c : Générer les fichiers .o mais pas de link
- -pedantic : Détection d'avertissements liés aux standards C
- -Wextra : Règles supplémentaires
- -std : Standard de langage auquel être conforme (C89, C99, ...)
- -g : mode debug

GDB

Qu'est-ce que gdb ?

GNU DeBugger est un outil d'analyse de l'exécution d'un programme, à des fins de test ou de debug

Détails

GDB fonctionne sur un mécanisme de points d'arrêt pour exécuter le programme et l'arrêter à des instructions précises

Lancer GDB

Commandes

```
1          gcc -o prog -g prog.c
2
3          gdb prog
4          >> run
5          ...
6
7          >> quit
```

Breakpoints

Définition

Les breakpoints (ou points d'arrêt) sont des instructions auxquelles le programme suspend son exécution.
Il est alors possible d'analyser le contenu de la mémoire (variables, pointeurs) et l'état du programme à un instant donné.

Placer un breakpoint

```
1                break f // Breakpoint a l'appel de la fonction f
2
3                break 98 // Breakpoint ligne 98
```

Commandes GDB

- print : afficher la valeur d'une variable
- step, next : passer à l'instruction suivante (avec ou sans sous-fonction)
- continue : aller au breakpoint suivant
- watchpoint : breakpoint sur la valeur d'une variable
- backtrace : affichage de la pile d'exécution
- list : afficher le code autour du breakpoint
- clear : supprimer les breakpoints
- delete : supprimer un breakpoint

Pre-processing

Le pré-processeur

Définition

Le pré-processeur (ou pre-processing) est une étape de transformation du code source intervenant avant la compilation

Principe

Remplacer un tag ou un symbole par un bloc de texte
Conditionner et structurer la compilation
Basé sur un ensemble de macros

Les directives de pre-processing

Détails

Les directives sont des mots-clés permettant d'implémenter du pre-processing
Ce sont des instructions interprêtées avant la compilation

Syntaxe

```
1 #directive argumentA argumentB ...
```

Les macros

Exemples

```
1 #include "file.h" // Inclusion de fichiers
2 #define TOTO 100 // Constantes.
3 #pragma
4 #line
5 #error
```

La directive include

Principe

Inclusion de fichiers source répartis

Syntaxe

```
1 #include <file.h>
2 #include "file.h>
```

La directive include

Inclusion de bibliothèques

Fichiers dans le path

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

Inclusion de fichiers

Chemin relatif au fichier courant

```
1 #include "file.h"
```

La directive define

Principe

La directive definit une macro qui sera remplacée par une expression lors du pre-processing

Exemple

```
1 #define TOTO 100
2
3 int tab[TOTO]; // Sera remplace par int tab[100];
```

La directive define

Détails

Il est possible de définir des fonctions ou expressions complexes avec define

Syntaxe

```
1   #define f(arg1,arg2, ...,argn) fonction
```

Exemple

```
1   #define f(a,b) a+b
2
3   #define swap(a,b,type) type s = a;a=b;b=s;
4
5   swap(a,b,int); //Appel
```

La directive error

Détails

Permet de renvoyer des erreurs spécifiques à la compilation

Syntaxe

```
1  #include<stdio.h>
2   #ifndef __MATH_H
3   #error Please include math.h
4   #endif
```

La compilation conditionnelle

Principe

Certaines portions de code peuvent n'être **compilées** que si une condition est vraie

Utilisation

Dédier certaines portions de code à des architectures spécifiques
Moduler la compilation suivant la cible
Compilation variée selon l'architecture ou l'OS

La compilation conditionnelle

Syntaxe

```
1  #if condition
2
3  #ifdef // equivalent de if defined()
4
5  #ifndef // if not defined()
6  #else
7  #elif
8  #endif
```

La compilation conditionnelle

Exemple

```
1  #define VALUE 15
2  #ifndef VALUE
3      //Compile si value n'est pas defini
4  #elif VALUE == 15
5      // Compile si VALUE vaut 15
6  #endif
7
8  #if defined(__WIN32__)
9      // Si windows 32 bits
10 #elif defined(__linux__)
11     //Si linux 64 bits
12 #else
13     // Sinon
14 #endif
```

Conclusion

- Moduler son code en fichiers
- Utiliser et écrire des Makefiles
- Comprendre les étapes de la compilation
- Utiliser les directives et macros de pre-processing
- Debugger finement avec GDB ou autres outils d'environement