Environnement de programmation

Aujourd'hui

Environnement de programmation :

- programme réparti dans plusieurs fichiers et compilation séparée ;
- préprocesseur ;
- Makefile pour la compilation automatique.

Programme dans plusieurs fichiers (1)

Intérêts :

- comme les fonctions, cela permet de structurer le programme ;
- permet de se rapprocher du comportement des classes en java ;
- compilation séparée des fichiers.

Programme dans plusieurs fichiers (2)

Premier fichier :

```
/* Fichier autre.c */

int triple(int nombre)
{
    return nombre * 3;
}
```

Deuxième fichier :

```
/* Fichier main.c */

int main(void)
{
    int nombre = triple(3);
    return 0;
}
```

Programme dans plusieurs fichiers (3)

Problème :

implicit declaration of function 'triple'

Programme dans plusieurs fichiers (3)

Problème :

implicit declaration of function 'triple'

Solution : déclarer la fonction `triple` dans le fichier `main.c`

```
/* Fichier main.c */

extern int triple(int nombre);

int main(void)
{
    int nombre = triple(3);
    return 0;
}
```

Le mot-clef `extern` est facultatif, mais ajoute de la lisibilité au code.

Variables

Comme pour les fonctions, les variables définies en dehors de toute fonction sont accessibles depuis tous les fichiers si elles y sont déclarées :

```
extern type variable;
```

Variables

Comme pour les fonctions, les variables définies en dehors de toute fonction sont accessibles depuis tous les fichiers si elles y sont déclarées :

```
extern type variable;
```

SAUF si la variable ou la fonction est déclarée *static* :

```
static int triple(int nombre);
static type variable;
```

auquel cas elle n'est accessible que depuis le fichier dans lequel elle est définie.

Fichiers d'en-tête

Plutôt que de déclarer les fonctions et variables externes dans chaque fichier, on utilise un fichier d'en-tête qui contient ces déclarations et on l'inclut au début des fichiers.

Fichiers d'en-tête

Plutôt que de déclarer les fonctions et variables externes dans chaque fichier, on utilise un fichier d'en-tête qui contient ces déclarations et on l'inclut au début des fichiers.

```
/* Fichier en-tete autre.h */
int triple(int nombre);
```

```
/* Fichier main.c */
#include "autre.h"

int main(void)
{
    int nombre = triple(3);
    return 0;
}
```

Ce fichier a le même nom que le fichier source, avec l'extension `.h`

Préprocesseurs : fonctionnalités

Le préprocesseur réalise des traitements sur le code source avant la compilation :

- inclusions de fichiers d'en-tête dans le code source ;
- définition de macros (bout de code) ;
- contrôle de la compilation conditionnelle.

Préprocesseurs : fonctionnalités

Le préprocesseur réalise des traitements sur le code source avant la compilation :

- inclusions de fichiers d'en-tête dans le code source ;
- définition de macros (bout de code) ;
- contrôle de la compilation conditionnelle.

Les directives du préprocesseur commencent par #
Effectuer uniquement la phase du préprocesseur :

```
gcc -E programme.c
```

Préprocesseur : inclusion de fichiers

3 possibilités avec la directive `#include` :

- `#include "include/autre.h"` : chemin relatif au fichier courant
- `#include "/home/toto/programme/include/autre.h"` : chemin absolu
- `#include <stdio.h>` : répertoire par défaut des librairies (`/usr/include/` sous unix).

Préprocesseur : macros

Une macro (directive `#define`) permet de substituer des morceaux de code :

```
#define TAILLE 100
#define BONJOUR puts("Bonjour\n")
```

Préprocesseur : macros avec paramètres

```
#define MACRO(paramètres) définition
#define CARRE(a) a*a
```

Préprocesseur : macros avec paramètres

```
#define MACRO(paramètres) définition
#define CARRE(a) a*a
```

Pièges :

- Priorité des opérateurs (ajouter des parenthèses aux paramètres) : `CARRE(2+3)`
- Effets de bord : `CARRE(x++)`

Préprocesseur : macros avec paramètres

```
#define MACRO(paramètres) définition
#define CARRE(a) a*a
```

Pièges :

- Priorité des opérateurs (ajouter des parenthèses aux paramètres) : `CARRE(2+3)`
- Effets de bord : `CARRE(x++)`

Comparatif macro vs fonctions :

- avantage : exécution plus rapide qu'une fonction (`fgets` est une macro par exemple) ;
- inconvénient : bugs inattendus et difficiles à déterminer.

Préprocesseur : directives de condition

Directives `#if`, `#elif`, `#else` (equivalent à `if`, `else if`, `else`) et `#endif`.
Cette dernière directive est obligatoire et conclut toute condition.

Préprocesseur : directives de condition

Directives `#if`, `#elif`, `#else` (equivalent à `if`, `else if`, `else`) et `#endif`.
Cette dernière directive est obligatoire et conclut toute condition.

```
#define A   2

#if (A) < 0
#define B puts("A < 0")
#elif (A) > 0
#define B puts("A > 0")
#else
#define B puts("A == 0")
#endif
```

Préprocesseur : directives de condition

Directives `#if`, `#elif`, `#else` (equivalent à `if`, `else if`, `else`) et `#endif`. Cette dernière directive est obligatoire et conclut toute condition.

```
#define A   2

#if (A) < 0
#define B puts("A < 0")
#elif (A) > 0
#define B puts("A > 0")
#else
#define B puts("A == 0")
#endif
```

`defined` teste si une constante a été définie :

```
#define DEBUG
#if defined DEBUG
Code1
#else
Code2
#endif
```

Préprocesseur : gérer les inclusions multiples

Le programme suivant génère une infinité d'inclusions :

```
/* Fichier toto.h */
#include "tata.h"
...

/* Fichier tata.h */
#include "toto.h"
...
```

Préprocesseur : gérer les inclusions multiples

Le programme suivant génère une infinité d'inclusions :

```
/* Fichier toto.h */
#include "tata.h"
...

/* Fichier tata.h */
#include "toto.h"
...
```

Solution :

```
/* Fichier toto.h */
#ifndef TATA_H
#define TATA_H
#include "tata.h"
#endif
...
```

Différence entre #define et typedef ?

```
#define BOOL int
typedef int bool;
```

- préprocesseur vs compilation

Différence entre #define et typedef ?

```
#define BOOL int
typedef int bool;
```

- préprocesseur vs compilation
- `typedef` permet une plus grande définition de types :

```
typedef int bool[10];
```

Différence entre #define et typedef ?

```
#define BOOL int
typedef int bool;
```

- préprocesseur vs compilation
- `typedef` permet une plus grande définition de types :

  ```
  typedef int bool[10];
  ```

- La portée du `typedef` suit celui des variables. La directive `#define` est valide jusqu'à la fin du fichier (ou si `#undef` est rencontré).

Compilation séparée

Les étapes pour générer un exécutable :

- Pour chaque fichier source :
  - préprocesseur ;
  - compilation en langage assembleur ;
  - transformation du code assembleur en fichier binaire (fichier objet `.o`) ;
- édition des liens entre les différents fichiers source et les librairies.

Compilation séparée

Les étapes pour générer un exécutable :

- Pour chaque fichier source :
  - préprocesseur ;
  - compilation en langage assembleur ;
  - transformation du code assembleur en fichier binaire (fichier objet `.o`) ;
- édition des liens entre les différents fichiers source et les librairies.

Pour générer un fichier objet

```
$ gcc -Wall -c programme.c
```

Compilation séparée

Les étapes pour générer un exécutable :

- Pour chaque fichier source :
  - préprocesseur ;
  - compilation en langage assembleur ;
  - transformation du code assembleur en fichier binaire (fichier objet `.o`) ;
- édition des liens entre les différents fichiers source et les librairies.

Pour générer un fichier objet

```
$ gcc -Wall -c programme.c
```

Pour réaliser l'édition de lien

```
$ gcc main.o autre.o
```

Logiciel make

```
$ man make
```

Exemple

Compilation d'un programme qui contient les fichiers

- `main.c`
- `autre.c` et `autre.h`

Exemple

Compilation d'un programme qui contient les fichiers

- `main.c`
- `autre.c` et `autre.h`

```
$ gcc -c main.c -o main.o
$ gcc -c autre.c -o autre.o
$ gcc autre.o main.o
```

Exemple

Compilation d'un programme qui contient les fichiers

- `main.c`
- `autre.c` et `autre.h`

```
$ gcc -c main.c -o main.o
$ gcc -c autre.c -o autre.o
$ gcc autre.o main.o
```

La commande *make* permet d'automatiser ces tâches, et de ne recompiler que ce qui a été modifié.

fichier Makefile

La commande `make` exécute les règles qui sont décrites dans le fichier **Makefile**.

fichier Makefile

La commande `make` exécute les règles qui sont décrites dans le fichier **Makefile**.

Une règle s'écrit sous la forme

```
cible: dépendances
	commandes
#commentaires éventuels
```

Il faut une tabulation devant les commandes.

fichier Makefile

La commande `make` exécute les règles qui sont décrites dans le fichier **Makefile**.

Une règle s'écrit sous la forme

```
cible: dépendances
	commandes
#commentaires éventuels
```

Il faut une tabulation devant les commandes.

Une règle particulière peut être évaluée dans le terminal avec

```
$ make cible
```

Sinon c'est la première règle qui est évaluée.

Makefile : dépendances

```
cible: dépendances
	commandes
#commentaires éventuels
```

Les dépendances sont une liste de noms de cibles et de fichiers.

- *make* évalue récursivement chaque cible dans les dépendances ;
- si la date du fichier portant le nom de la cible est inférieure à la date de modification d'un fichier dans les dépendances, les commandes sont exécutées.

Makefile : règles particulières

Habituellement un *Makefile* contient au moins ces 2 règles :

- la règle *all* : cette règle est placée en première position, et son évaluation génère l'ensemble des fichiers du programme ;
- la règle *clean* : supprime tous les fichiers objets et l'exécutable.

Makefile : variables et règles génériques

Il est possible de définir des variables qui contiennent des chaînes de caractères :

```
VAR1=hello
VAR2=$(VAR1) world #VAR2 vaudra "hello world"
```

L'opérateur `$()` permet d'accéder à leur contenu.

Makefile : variables et règles génériques

Il est possible de définir des variables qui contiennent des chaînes de caractères :

```
VAR1=hello
VAR2=$(VAR1) world #VAR2 vaudra "hello world"
```

L'opérateur `$()` permet d'accéder à leur contenu.

Des variables sont pré-définies :

- `@` le nom de la cible ;
- `^` la liste des dépendances de la règle.

Makefile : variables et règles génériques

Il est possible de définir des variables qui contiennent des chaînes de caractères :

```
VAR1=hello
VAR2=$(VAR1) world #VAR2 vaudra "hello world"
```

L'opérateur `$()` permet d'accéder à leur contenu.

Des variables sont pré-définies :

- `@` le nom de la cible ;
- `^` la liste des dépendances de la règle.

Le symbole % remplace n'importe quel préfixe et est utilisé pour les règles génériques :

```
%.o: %.c
	gcc -c $(^) -o $(@)
```