Le bien programmer

Michael Mrissa

Université de Pau
et des Pays de l'Adour

Merci à Nicolas Belloir pour le support

Introduction

Un programmeur n'a réellement besoin que :

- d'un éditeur en ligne (saisie du code),
- un compilateur (génération exécutable),
- un terminal (exécution d'un programme).

Suffisant pour un petit programme. Mais pour un programme industriel ?

Bien programmer c'est écrire un programme :

- rapidement et efficacement,
- qui s'exécute sans erreur,
- performant, fiable et évolutif.

Pour écrire un programme rapide et efficace :

- ne coder qu'une fois le problème analysé et l'algorithme écrit,
- utiliser des éditeurs adaptés,

Pour écrire un programme qui s'exécute sans erreur :

- tester et valider votre programme,
- déboguez le.

Pour écrire un programme performant, fiable et évolutif :

- utilisez de bonnes options de compilation,
- organisez le code de manière modulable,
- pensez aux évolutions éventuelles.

Méthodes de programmation

Factoriser

Comment faire ?

Rappel : $ca + cd - 2ec = c(a + d - 2e)$

- Isoler les redondances de code,
- Identifier les motifs qui se répètent,
- En faire des fonction.

Remarque

- La maintenance ou le débogage sont sources de redondances ! Soyez prudents !
- La factorisation réduit le nombre de ligne de code à écrire !

Quelques recettes

- Porter un soin particulier au nom de la fonction,
- Utiliser des structures de données si le nombre de paramètre est trop important.

Découper

Utilisation de Module / Paquetage

- Halte aux fichiers de 10000 lignes de code,
- Organiser le code par thème ou fonctionnalités (erreur, communication . . .),
- Découper les fonctions trop importantes.

Intérêt

- facilité de classement et d'organisation du code,
- facilité d'évolution,
- découpage logique salutaire pour le programmeur.

Qu'est ce donc ?

- Critère de jugement des qualités professionnels d'un programmeur (comme un artisan),
- Jugement subjectif,
- Attention :
  - non propreté $\Longrightarrow$ incompétence ou manque de rigueur,
  - propreté $\not\Rightarrow$ compétence.

Faire attention à (1) :

- Temps de compilation anormalement long,
  - Technique d'accélération de compilation ignorées,
  - En-têtes trop "grosses" et trop imbriquées,
  - Non utilisation de librairies.
- Présence de *warnings* de compilation,
  - Peut cacher un bogue,
  - Trop de warnings cachent les warnings **abusifs**,
- Mémoire dynamique non libérée,

Faire attention à (2) :

- Fichiers sources inclus et non utilisés,
- Includes inutiles,
- Includes non protégés contre l'inclusion multiple,
- Commentaires erronées,
- Lignes de code inutilisées,
- librairies incluses dans l'édition de lien mais non utilisées,
- code compilé en mode *debug* et mélangé avec du code *release*,
- ...

Une histoire de choix

- Choisir la technologie adaptée (pourquoi monter une usine à gaz quand on peut faire les choses simplement ?),
- Limiter les coûts annexes quand ils ne sont pas nécessaires (formation, achat de librairie).
- La simplicité est gage de fiabilité !

Pourquoi se défendre ?

Où il est question de Murphy

- "Si quelque chose peut mal tourner, alors ça tournera mal."
- "S'il existe deux ou plusieurs manières de faire quelque chose et que l'une de ces manières est susceptible de se solder par une catastrophe, on peut être certain que quelqu'un se débrouillera pour la choisir."

Règle d'or

La programmation défensive doit devenir un VRAI état d'esprit.

Quelques règles

Lors de l'écriture de code :

- Prévoir tous les cas d'erreur possibles,
- Traiter chaque cas d'erreur,
- Prévoir des lignes de *trace* pour le débogage.

Erreurs fréquentes

- Valeurs invalides (valeur de durée négative, débordement,
- Valeurs extrêmes (min, max, index négatifs, . . .),
- Boucles infinies,
- Pointeurs invalides,
- E/S,
- Écrasement mémoire, mémoire non initialisée,
- pas de valeur par défaut,
- cas non traités (switch...).

Traitement des erreurs

- Bas niveau,
  - Enregistrer les erreurs (fichier log avec trace),
  - Etat du programme à ce moment.
- Niveau utilisateur.
  - Propager l'erreur au niveau utilisateur,
  - Interrompre l'opération en cours,
  - Communiquer à l'utilisateur l'information.

L'exemple d'Ariane 5

- le 4 juin 1996 explosion d'Ariane 5 après 40 secondes de vol,
- 500 Millions de $ de perte financière estimée,
- Cause : erreur de logiciel : Système de Référence Inertielle.
  - Importé d'Ariane 4,
  - Non adapté à la plus grande vitesse horizontale d'Ariane 5,
  - Conversion d'un nombre flottant sur 64 bits en un entier sur 16 bits,
  - Débordement et exception non traitée...
  - $\Longrightarrow$ segmentation fault et terminaison du programme.

Quelques automatismes

- A chaque ouverture de fichier, écrire simultanément sa fermeture,
- A chaque *new*, écrire immédiatement le *delete* correspondant,
- Utilisez toujours le même nom de variable pour représenter la même chose,
- A chaque accolade ouvrante, écrire la fermante,
- Tester systématiquement la nullité des pointeurs,
- ...

Environnement de programmation

Les éditeurs

- Plusieurs éditeurs de texte sont disponibles sur Unix :
  - L'éditeur Unix standard **vi** : très puissant mais peu convivial,
  - L'autre éditeur parmi les plus populaires est **emacs** : très puissant aussi mais plus "gourmand" en ressources.
- Des clients XWindow permettent aussi de faire de l'édition de texte :
  - **xemacs** est une version d'emacs avec menus, encore plus "gourmand",
  - **xedit**, **kate** et **gedit** sont les éditeurs standards de gnome, KDE et XWindow, respectivement.
- Les environnements de programmation intégrés :
  - Intègre sous un même environnement toutes les fonctionnalités propre au développement, mais nécessitent des machines très performantes,
  - **Eclipse**, **Visual C++**, **Kdevelop** ...

Existe-t'il un éditeur idéal ?

Non, mais le choix de l'éditeur doit s'adapter à la situation du programmeur :

- Windows like Versus Unix like,
- Seul ou en équipe,
- Petits projets ou grands projets,
- Au bureau ou en mission,

Fonctionnalités obligatoires pour un éditeur

- colorisation ou mise en relief des mots clés du code,
- indentation automatique,
- mécanismes de recherche rapides,
- accès rapide à la compilation.

Fonctionnalités intéressantes pour un éditeur

- complétion,
- aide avancée,
- mécanismes complémentaires divers.
- permet de se passer de souris (rapidité)

Un éditeur nécessite

- un investissement pour connaître son fonctionnement,
- d'apprendre sa syntaxe (ex : Amadeus Vs Internet),
- parfois de savoir s'adapter.

Compilation

Définition

Compiler, c'est passer du code source d'un programme à un exécutable binaire à l'aide du logiciel approprié (le compilateur).

Déroulement de la compilation

1. traitement par le **préprocesseur**,
2. **compilation**,
3. **assemblage**,
4. **édition de lien**.

Le préprocesseur

- Analyse le fichier source (*.c*),
- Exécute des transformation purement textuelles.

La compilation

- Analyse le fichier (*.i*) généré par le préprocesseur ,
- Le traduit en assembleur (suite d'instruction propres au microprocesseur utilisant des mnémoniques lisibles).

Exemple

```
#include <iostream.h>
#define MAX 2
int main (){
        int i;
        for (i=0; i<MAX; i++)
                cout << "HelloWorld" << endl;
}
```

Exemple

```
int main (){
        int i;
        for (i=0; i<2; i++)
                cout << "HelloWorld" << endl;
}
```

L'assemblage

- Analyse le fichier assembleur (*.s*),
- Le transforme en fichier binaire (directement compréhensible par le processeur).

L'édition de lien

- Lie les différents fichiers binaires, appelés aussi objets (*.o*),
- Produit un fichier exécutable (*a.out*).

Exemple

```
.LC0:
        .string "HelloWorld"
        .text
        .align 2
.globl main
        .type   main,@function
main:   .LFB1479:
        pushl   %ebp
.LCFI0:
        movl    %esp, %ebp
.LCFI1:
        subl    $8, %esp
.LCFI2:
```

Paramétrage du compilateur

- Les compilateurs sont hautement paramétrables,
- Le programmeur doit être capable de les paramétrer en fonction des besoins,

Exemple

```
g++ -c Helloword.cpp
g++ -S Helloword.cpp
g++ -Wall Helloword.cpp
g++  Helloword.cpp -o HelloWorld
```

L'outil *make*

L'outil *make* permet

- d'automatiser le processus de compilation,
- d'optimiser la compilation,
- de faciliter l'organisation d'un projet,
- d'apporter plus de portabilité.

Principes élémentaires du *make*

- Effectue uniquement les étapes de compilation nécessaires à la création d'un exécutable,
- Recherche par défaut dans le répertoire courant un fichier *makefile*,
- Est paramétrable.

Exemple

```
make -f <nom_fichier>
make -C <path>
```

Création d'un *makefile*

- Un fichier *makefile* est composé d'une liste de règles de dépendances entre fichiers :

```
cible: liste_de_dependances
<TAB> commandes_a_effectuer
```

- La 1re ligne indique :
  - le nom du fichier cible (généré par la commande *commandes_a_effectuer*),
  - la liste des *fichiers* requis pour la génération de *cible*.
- Les lignes suivantes :
  - **doivent** commencer par *<TAB>*,
  - indiquent les commandes à exécuter pour générer le fichier *cible*.

Compilation manuelle

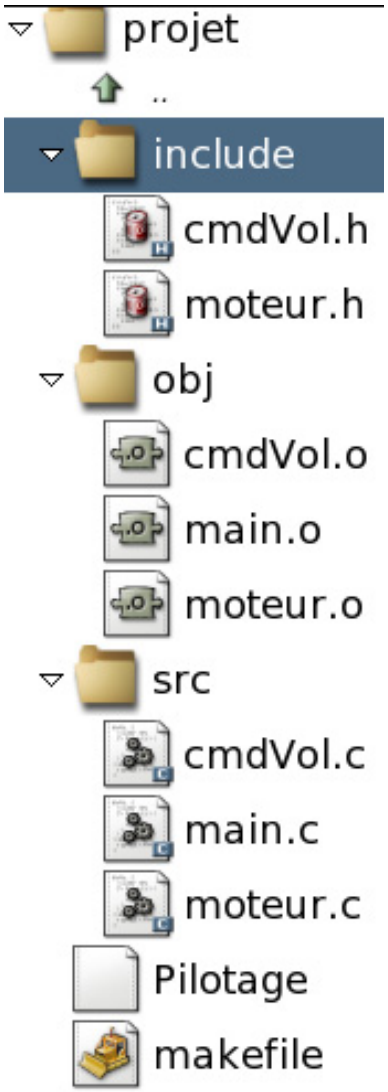
Exemple

```
gcc -g3 -Wall -c moteur.c
gcc -g3 -Wall -c cmdVol.c
gcc -g3 -Wall -c main.c
gcc -g3 -Wall -c Pilotage main.o moteur.o cmdVol.o
```

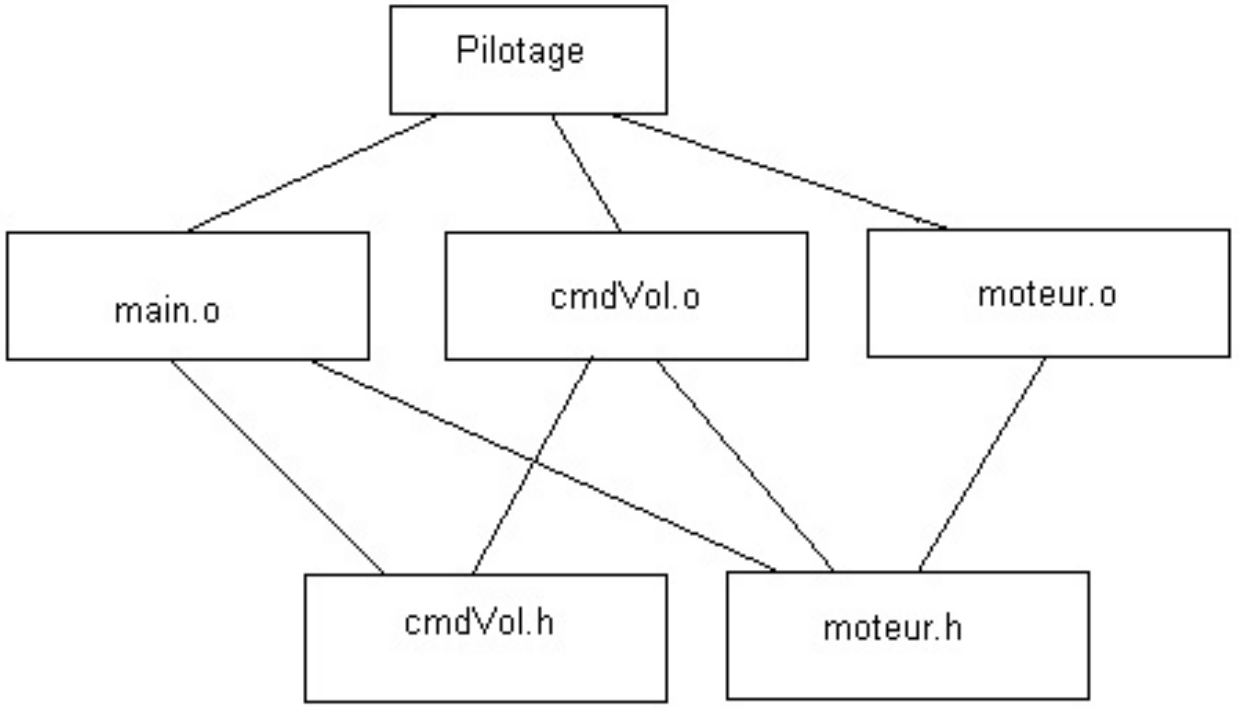
- Long et fastidieux,
- À répéter à chaque fois.

```
#generation de l exécutable Pilotage
all: Pilotage

#le fichier moteur.o dépend de moteur.c et moteur.h
moteur.o: moteur.c moteur.h
	gcc -g3 -Wall -c moteur.c

#le fichier cmdVol.o dépend de cmdVol.c et cmdVol.h
cmdVol.o: cmdVol.c cmdVol.h
	gcc -g3 -Wall -c cmdVol.c

#le fichier main.o dépend de main.c, moteur.h et cmdVol.h
main.o: main.c moteur.h cmdVol.h
	gcc -g3 -Wall -c main.c

#generation de l exécutable
Pilotage: main.o moteur.o cmdVol.o
	gcc -g3 -Wall -o Pilotage main.o moteur.o cmdVol.o
```

compilation dans une arborescence complexe



Graphe de dépendance



```
#generation de l exécutable Pilotage
all: Pilotage

#le fichier moteur.o dépend de moteur.c et moteur.h
obj/moteur.o: src/moteur.c include/moteur.h
	gcc -g3 -Wall -Iinclude -c src/moteur.c -o obj/moteur.o

#le fichier cmdVol.o dépend de cmdVol.c et cmdVol.h
obj/cmdVol.o: src/cmdVol.c include/cmdVol.h
	gcc -g3 -Wall -Iinclude -c src/cmdVol.c -o obj/cmdVol.o

#le fichier main.o dépend de main.c, moteur.h et cmdVol.h
obj/main.o: src/main.c include/moteur.h include/cmdVol.h
	gcc -g3 -Wall -Iinclude -c src/main.c -o obj/main.o

#generation de l exécutable
Pilotage: obj/main.o obj/moteur.o obj/cmdVol.o
	gcc -g3 -Wall -o Pilotage obj/main.o obj/moteur.o
	                          obj/cmdVol.o
```

Utilisation de variables

```
EXE = Pilotage
OBJ = obj
INC = include
SRC = src
OPT = -g3 -Wall

all: $(EXE)

$(OBJ)/moteur.o: $(SRC)/moteur.c $(INC)/moteur.h
	gcc $(OPT) -I$(INC) -c $(SRC)/moteur.c -o $(OBJ)/moteur.o

$(OBJ)/cmdVol.o: $(SRC)/cmdVol.c $(INC)/cmdVol.h
	gcc $(OPT) -I$(INC) -c src/cmdVol.c -o $(OBJ)/cmdVol.o

$(OBJ)/main.o: $(SRC)/main.c $(INC)/moteur.h $(INC)/cmdVol.h
	gcc $(OPT) -I$(INC) -c src/main.c -o $(OBJ)/main.o

$(EXE): $(OBJ)/main.o $(OBJ)/moteur.o $(OBJ)/cmdVol.o
	gcc -g3 -Wall -o $(EXE) $(OBJ)/main.o $(OBJ)/moteur.o $(OBJ)/c
```

Utilisation de variables prédéfinies

$@	le fichier cible (ex : obj/main.o)
$*	le fichier cible sans suffixe (ex : obj/main)
$<	le premier fichier de la liste des dépendances (ex : main.c)
$?	l'ensemble des fichiers de dépendances

```
EXE = Pilotage
OBJ = obj
INC = include
SRC = src
OPT = -g3 -Wall

all: $(EXE)

$(OBJ)/moteur.o: $(SRC)/moteur.c $(INC)/moteur.h
	gcc $(OPT) -I$(INC) -c $< -o $@

$(OBJ)/cmdVol.o: $(SRC)/cmdVol.c $(INC)/cmdVol.h
	gcc $(OPT) -I$(INC) -c -c $< -o $@

$(OBJ)/main.o: $(SRC)/main.c $(INC)/moteur.h $(INC)/cmdVol.h
	gcc $(OPT) -I$(INC) -c $< -o $@

$(EXE): $(OBJ)/main.o $(OBJ)/moteur.o $(OBJ)/cmdVol.o
	gcc -g3 -Wall -o $(EXE) $?
```

Finalisation du makefile

```
clean:
	rm $(OBJ)/*.o $(EXE)
```

Les outils évolués

- Make
  - L'ancêtre
- Autoconf/automake
  - GNU
- cmake
  - Kitware (VTK)
- Ant
  - Apache

Make

Avantages

- Outil standard
- Toutes les implantations ont une base commune
- Principes de fonctionnement simples
- Bien documenté

Inconvénients

- Syntaxe des commandes (shell)
- Portabilité des commandes

Autoconf/Automake

Principe

- *Automake* génère un/des squelettes de *Makefile* (*Makefile.in*)
- *Autoconf* génère un script (appelé *configure*)
- L'exécution de *configure* génère un/des *Makefile*

Autoconf/Automake

Inconvénients

- Apprentissage difficile et pénible
- Une syntaxe pour *automake*, une autre pour *autoconf*
- Unix. Windows avec cygwin

Avantages

- Macros de tests des caractéristiques/capacités de la plateforme
- Support (automake) pour l'exécution de suites de test
- Bien documenté

Cmake

Principe

- Génère un *makefile* (Unix) ou un *projet Visual C++* (Windows) à partir d'un fichier de description *CMakeLists.txt*
- Utilisable en ligne de commande (*cmake*) ou avec GUI (*ccmake*)
- Adapte la compilation à l'utilisation de librairies externes

Cmake

Inconvénients

- Chemins absolus dans le/les *Makefile*(s) générés : il faut réexécuter *cmake* si on déplace le répertoire

Avantages

- Une seule syntaxe simple à apprendre
- Portabilité du build

Ant

Principe

Outil pour la compilation, le test et le déploiement des programmes Java.

- Un fichier (par défaut *build.xml*) décrit le projet et les tâches correspondantes

Ant

Inconvénients

- Syntaxe XML

Avantages

- Presque un standard...
- Complet
- Extensible : une tâche est un objet java
- Bien documenté
- Utilisable en ligne de commandes et dans tous les bons IDE

Comparatif

	Langages	Plateformes	Apprentissage	Documentation
Make	Tous	Toutes	Moyen	Bonne
Automake	C/C++	UNIX/cygwin	Difficile	Bonne
Cmake	C/C++	UNIX/cygwin	Facile	Bonne
Ant	Java	Toutes	Facile	Bonne

Pour en savoir plus

http ://www-sop.inria.fr/dream/seminaires/buildtools-25-05-05.pdf

Le débogage

Élements de définition

Bogue [*Bug*, Angl.] : Erreur dans un programme entraînant des comportements étranges et rarement désirés du système.Entraîne généralement un juron du programmeur.

Débogage [*Debug*, Angl.] : Art de détecter, localiser puis corriger une erreur. Le mieux est de le faire sans rajouter une autre erreur.

Histoires de tests

Quelques tests

Tests essentiels : permettent de vérifier la validité des algorithmes implémentés.

Tests aléatoires : éprouvent le programme avec des données inhabituelles, dans le désordre. Permettent de vérifier la solidité globale et de détecter plus rapidement des bogues aléatoires.

Tests en charge : vérifient le bon fonctionnement du programme dans des situations critiques habituelles.

Tests en réel : permettent d'évaluer le programme dans son contexte normal d'utilisation.

Le débogage

- Se déroule durant 2 phases distinctes :
  - La phase de mise au point / vérification avant livraison (peut être très long),
  - La phase de maintenance lorsque des bogues apparaissent (parfois critique).
- Implique beaucoup de méthode, de finesse, et de rapidité,
- Peut-être intégré - ou non - à l'environnement de programmation,
- Ne se déroule pas de la même manière en fonction des différents types de programmes (temps réel, embarqué, gestion ...)

Remarque importante

Il convient de bien différencier l'erreur d'exécution de l'erreur de programmation.

L'erreur d'exécution est l'erreur qui fait se planter le programme. On peut parler d'erreur visible, ou de conséquence.

L'erreur de programmation est l'erreur qui induit un risque de plantage du programme. On peut parler d'erreur cachée ou de cause.

Exemple

```
...
int tab [3];
...
for (i=0; i<=3;i++){
    tab[i]=i;
}
```

Méthode

1. Localisation de l'erreur d'exécution,
2. Analyse de la cause de l'erreur d'exécution (cause directe ou indirect),
3. Localisation de l'erreur de programmation,
4. Analyse de la cause de l'erreur de programmation,
5. Correction de l'erreur de programmation,
6. Vérification de la bonne correction de l'erreur.

Les outils pour déboguer

- La trace manuelle,
- Le fichier log,
- La boîte noire,
- Le débogueur.

La trace manuelle

Principe

Intégration ponctuelle de messages de sortie erreur ou standard dans le code.

Avantages

- Méthode rapide pour des problèmes très simples,
- Visualisation d'une ou plusieurs variables précises.

Inconvénients

- Nécessite la recompilation du programme,
- Fastidieux pour les grands codes ou les erreurs complexes.

Exemple

```
...
int tab [3];
...
for (i=0; i<=3;i++){

#ifdef DEBUG
    cout << "Avant affectation [i= " << i << " ]";
#endif
    tab[i]=i;
#ifdef DEBUG
    cout << "Apres affectation [tab[i] = " << tab[i] << " ]";
#endif
}
```

Exemple

```
g++ -DDEGUG parcoursTab.c -o parcoursTab
```

Le fichier Log

Principe

Écriture en continue dans un fichier de traces de programmes.

Avantages

- Indispensable lorsque le programme devient complexe ou brasse beaucoup de données,
- Les traces peuvent être affinées au cours du débogage,
- Une version allégée peut aider l'utilisateur à comprendre un problème (maintenance).

Inconvénients

- Long à mettre en œuvre, il faut le prévoir dès le début (module Trace),

Exemple

```
14:25:43|NFO|main.c|0772|motor started
14:25:43|DEB|main.c|0775|normal rate
14:25:44|DEB|motor.c|0775|Receiving Msg
14:25:44|ERR|motor.c|9915|Anormal Communication ending
```

La boîte noire

Principe

Image de la mémoire stockée sur un fichier à intervalle régulier.

Avantages

- Complémentaire aux traces,
- Permet d'enregistrer toutes les valeurs de la mémoire sur une plage définie
- Permet de rejouer des scénarios d'erreurs.

Inconvénients

- Nécessite une mise en place lourde,

La débogueur

Principe

Permet de suivre le code se dérouler et d'accéder aux variables dynamiquement lors de l'exécution du programme.

Avantages

- Permet une observation complète du programme en exécution,
- Permet une interaction avec le programme.

Inconvénients

- Alourdi l'exécution du programme,
- Peut masquer des erreurs,
- Non permanent.

Types de débogueurs

- Indépendants (*gdb*, *ddd*, ...),
- Intégrés à des IDE (Visual, Eclipse, Sun One, ...).

Fonctionnalités des débogueurs

- Tracer pas à pas,
- Pose de points d'arrêts,
- Consultation de la pile des appels de fonctions,
- Consultation de la valeur des variables,
- inspection des registres et de la mémoire.

Exemple avec ddd