H.E.P.L. : Catégorie Technique

Bachelier en Informatique et Systèmes : finalité Informatique Industrielle

LABORATOIRE D'INFORMATIQUE INDUSTRIELLE

Informatique en milieu industriel

3ème année

Outils de développement

2012 - 2013

Van Gysegem Thomas
Delsaux Kevin

Groupe : 2322

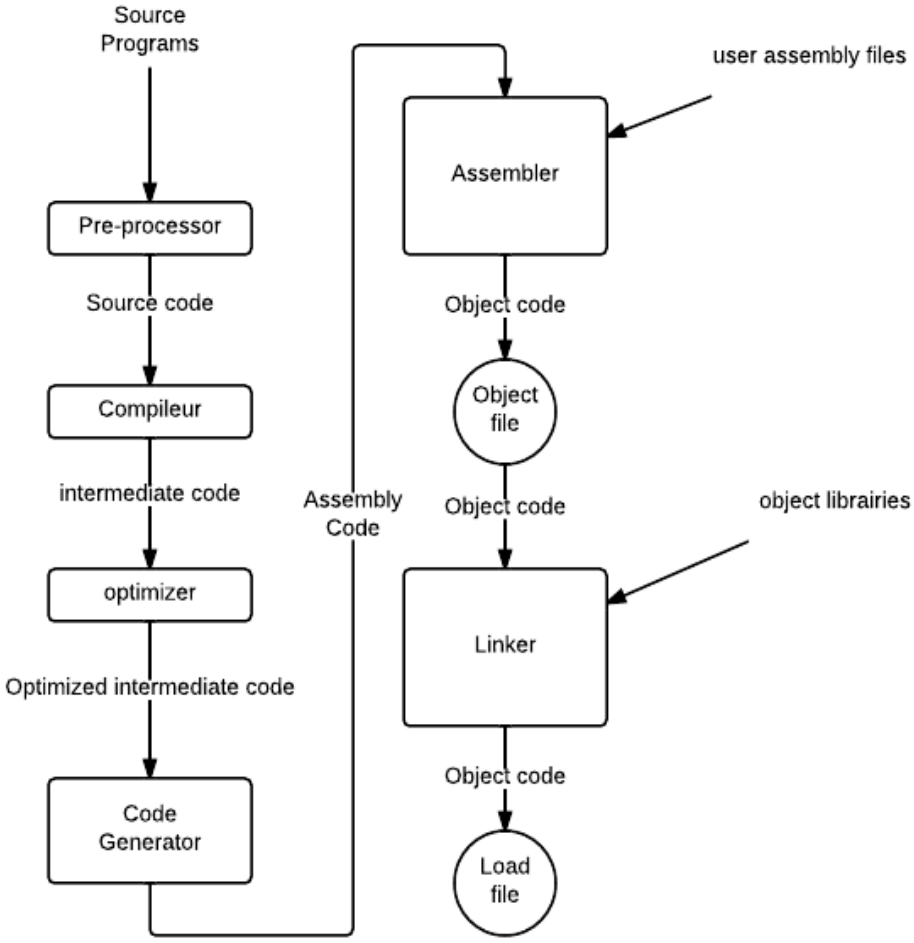
Table des matières

Les étapes de la compilation

Le dossier est divisé en 4 parties qui correspondent aux 4 étapes principales de la compilation. A ces 4 parties nous avons rajouté une partie sur l'optimisation qui se situe entre la compilation et le passage au code machine (étape 2 et étape 3)

Les différentes étapes sont donc :

1. Pré-processing
2. Compilation
3. Passage au code machine
4. Link (ou édition de lien)



Fichier original

Ce simple fichier sera utilisé pour illustrer les étapes de la compilation.

main.c

```c
int main(void)
{
  int i = 0;

  return 0;
}
```

Etape 1 : Pré-processing

Commande utilisée

gcc -E main.c > main.i

taille du fichier obtenu

119 octets

Le pré-processing permet de vérifier la syntaxe du programme comme le montre le screenshot ci-dessous :

```
delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$ cat main.c
#include <stdio.h>
#include "main.h"

int main(void)
{
  printf("Hello World\n");

  int tab[SIZE] = { 0 };
  int i;
  for(i=0; i>size; i++) {
    printf("tab[%d] = %d;\n, i, tab[i])

    i++;
    i--;
  }

  return 0;
}delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$ gcc -E main.c > main.i
main.c:11:12: attention : caractère " de terminaison manquant [enabled by default]
delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$
```

Le guillemet de fermeture de chaîne n'est pas présent et cela génère une erreur et interrompt la compilation.

Il se charge aussi de résoudre les lignes commençant par le caractère # (include, define, pragma, ...) et de retirer les commentaires (ils ne serviront à rien pour la compilation)

On se retrouve donc avec un fichier main.i ressemblant à celui-ci :

```
extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ ,
__leaf__));
# 940 "/usr/include/stdio.h" 3 4

# 2 "main.c" 2
# 1 "main.h" 1

const int size = 1000;
# 3 "main.c" 2
```

```c
int main(void)
{
  printf("Hello World\n");

  int tab[1000] = { 0 };
  int i;
  for(i=0; i>size; i++) {
    printf("tab[%d] = %d;\n", i, tab[i]);

    i++;
    i--;
  }

  return 0;
}
```

Nous n'avons ici que la fin du fichier (la version complète fait plus de 800 lignes simplement en incluant stdio.h)

Etape 2 : Compilation

commande utilisée

gcc -S main.i

taille du fichier obtenu

243 octets

code utilisé

```
int main(void)
{
  int i = 0;

  return 0;
}
```

code obtenu

```
	.file "main.c"
	.def	___main;	.scl	2;	.type 32;	.endef
	.text
.globl _main
	.def	_main;	.scl	2;	.type 32;	.endef
_main:
	pushl %ebp
	movl	%esp, %ebp
	andl	$-16, %esp
	subl	$16, %esp
	call	___main
	movl	$0, 12(%esp)
	movl	$0, %eax
	leave
	ret
```

Cette étape nous permet de passer du code *.c au code assembleur, elle se charge des erreurs non détectées lors du pré-processing comme les oubli de ;

Le screenshot ci-dessous le montre bien :

```
delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$ cat main.c
int main(void)
{
  int i=0

  return 0;
}
delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$ gcc -E main.c > main.i
delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$ gcc -S main.i
main.c: In function 'main':
main.c:5:3: erreur: expected ',' or ';' before 'return'
delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$
```

Etape 3 : Transformation en code machine

commade utilisée

gcc -c bonjour.s

taille du fichier obtenu

384 octets

Cette étape transforme le code assembleur lisible en code machine (illisible).

La commande « od -x main.o » permet d'afficher le code en hexadécimal. Le screenshot ci dessous montre ce qu'il se passe lorsqu'on essaye d'afficher le contenu du fichier :

```
delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$ cat main.o
ELF`4(.symtab.strtab.shstrtab.text.data.bss!444,xP      ``main.cdelskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$
delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$ od -x main.o
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000020 0001 0003 0001 0000 0000 0000 0000 0000
0000040 0060 0000 0000 0000 0034 0000 0000 0028
0000060 0007 0004 2e00 7973 746d 6261 2e00 7473
0000100 7472 6261 2e00 6873 7473 7472 6261 2e00
0000120 6574 7478 2e00 6164 6174 2e00 7362 0073
0000140 0000 0000 0000 0000 0000 0000 0000 0000
*
0000200 0000 0000 0000 0000 001b 0000 0001 0000
0000220 0006 0000 0000 0000 0034 0000 0000 0000
0000240 0000 0000 0000 0000 0004 0000 0000 0000
0000260 0021 0000 0001 0000 0003 0000 0000 0000
0000300 0034 0000 0000 0000 0000 0000 0000 0000
0000320 0004 0000 0000 0000 0027 0000 0008 0000
0000340 0003 0000 0000 0000 0034 0000 0000 0000
0000360 0000 0000 0000 0000 0004 0000 0000 0000
0000400 0011 0000 0003 0000 0000 0000 0000 0000
0000420 0034 0000 002c 0000 0000 0000 0000 0000
0000440 0001 0000 0000 0000 0001 0000 0002 0000
0000460 0000 0000 0000 0000 0178 0000 0050 0000
0000500 0006 0000 0005 0000 0004 0000 0010 0000
0000520 0009 0000 0003 0000 0000 0000 0000 0000
0000540 01c8 0000 0008 0000 0000 0000 0000 0000
0000560 0001 0000 0000 0000 0000 0000 0000 0000
0000600 0000 0000 0000 0000 0001 0000 0000 0000
0000620 0000 0000 0004 fff1 0000 0000 0000 0000
0000640 0000 0000 0003 0001 0000 0000 0000 0000
0000660 0000 0000 0003 0002 0000 0000 0000 0000
0000700 0000 0000 0003 0003 6d00 6961 2e6e 0063
0000720
delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$
```

Etape 4 : Link

commande utilisée

gcc main.o

taille du fichier obtenu

7,04 ko

Cette étape permet d'inclure le code des librairies utilisée dans le programme et produire l'exécutable désiré. Le code obtenu est toujours un code binaire lisible via un éditeur héxadécimal ou par la commande « od »

On voit à la taille du fichier que des ajouts on été fait depuis les étapes précédentes. Voici un aperçu de ce dernier :

```
delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$ gcc main.o
delskev@delskev-Laptop:~/Documents/Ecole/informatiqueIndustrielle/sortie$ od -x a.out
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000020 0002 0003 0001 0000 8300 0804 0034 0000
0000040 1138 0000 0000 0000 0034 0020 0009 0028
0000060 001e 001b 0006 0000 0034 0000 8034 0804
0000100 8034 0804 0120 0000 0120 0000 0005 0000
0000120 0004 0000 0003 0000 0154 0000 8154 0804
0000140 8154 0804 0013 0000 0013 0000 0004 0000
0000160 0001 0000 0001 0000 0000 0000 8000 0804
0000200 8000 0804 0598 0000 0598 0000 0005 0000
0000220 1000 0000 0001 0000 0f14 0000 9f14 0804
0000240 9f14 0804 00fc 0000 0104 0000 0006 0000
0000260 1000 0000 0002 0000 0f28 0000 9f28 0804
0000300 9f28 0804 00c8 0000 00c8 0000 0006 0000
0000320 0004 0000 0004 0000 0168 0000 8168 0804
0000340 8168 0804 0044 0000 0044 0000 0004 0000
0000360 0004 0000 e550 6474 04a0 0000 84a0 0804
0000400 84a0 0804 0034 0000 0034 0000 0004 0000
0000420 0004 0000 e551 6474 0000 0000 0000 0000
0000440 0000 0000 0000 0000 0000 0000 0006 0000
0000460 0004 0000 e552 6474 0f14 0000 9f14 0804
0000500 9f14 0804 00ec 0000 00ec 0000 0004 0000
0000520 0001 0000 6c2f 6269 6c2f 2d64 696c 756e
0000540 2e78 6f73 322e 0000 0004 0000 0010 0000
0000560 0001 0000 4e47 0055 0000 0000 0002 0000
0000600 0006 0000 0018 0000 0004 0000 0014 0000
0000620 0003 0000 4e47 0055 bbf1 fda1 7d8d b1d2
```

Comportement natif

Cette section montre les optimisations « naïves » ajoutées de base par GCC

Commande utilisée

Gcc -S main.c -o <fichier de sortie>

Code utilisé

Code 1

```
int main(void)
{
  int i = 0;

  i = 2;

  return 0;
}
```

Code 2

```
int main(void)
{
  int i = 0;

  i++;
  i--;

  i = 2;

  return 0;
}
```

Code 3

```
int main(void)
{
  int i = 0;

  i++;

  i = 2;

  return 0;
}
```

Code 4

```
int main(void)
{
  int i = 0;

  i++;

  return 0;
}
```

Code obtenu

Code 1

```
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $16, %esp
        movl    $0, -4(%ebp)
        movl    $2, -4(%ebp)
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Code 2

```
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $16, %esp
        movl    $0, -4(%ebp)
        addl    $1, -4(%ebp)
        subl    $1, -4(%ebp)
        movl    $2, -4(%ebp)
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Code 3

```
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $16, %esp
        movl    $0, -4(%ebp)
        addl    $1, -4(%ebp)
        movl    $2, -4(%ebp)
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Code 4

```
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $16, %esp
        movl    $0, -4(%ebp)
        addl    $1, -4(%ebp)
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Commentaires

Nous avons pu remarquer que les quatre codes sont plus ou moins similaire, mais se distinguent par leurs incrémentation de la variable i , en effet nous possédons plusieurs manières de faire mais au final on force la variable i à 2 ce qui va avoir de conséquence pour l'optimisation de ne pas tenir compte de l'incrémentation précédentes étant donné que celle-ci n'ont aucun impact sur le code.

Comparatif des optimisation disponibles

Commandes utilisées

Gcc -S main.c -01 -o 01

Gcc -S main.c -02 -o 02

Gcc -S main.c -03 -o 03

Gcc -S main.c -0s -o 0S

Code utilisé

```
int main(void)
{
        printf("Hello World\n");

        int tab[SIZE] = { 0 };
        int i;
        for(i=0; i>size; i++) {
        printf("tab[%d] = %d;\n", i, tab[i]);

        i++;
        i--;
  }

  return 0;
}
```

Code obtenus en -O1

```
.LFB22:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE22:
        .size   main, .-main
        .globl  size
        .section        .rodata
        .align 4
        .type   size, @object
        .size   size, 4
```

Temps d'exécution

real : 0m0,002s

user : 0m0,004s

sys : 0m,000s

Code obtenus en -O2

```
.LFB22:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        xorl    %eax, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE22:
        .size   main, .-main
        .globl  size
        .section        .rodata
        .align 4
        .type   size, @object
        .size   size, 4
```

Temps d'exécution

real : 0m0,002s

user : 0m0,000s

sys : 0m,000s

Code obtenus en -O3

```
.LFB22:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        xorl    %eax, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE22:
        .size   main, .-main
        .globl  size
        .section        .rodata
        .align 4
        .type   size, @object
        .size   size, 4
```

Temps d'exécution

real : 0m0,002s

user : 0m0,004s

sys : 0m,000s

Code obtenus en -Os

```
.LFB12:
        .cfi_startproc
        leal    4(%esp), %ecx
        .cfi_def_cfa 1, 0
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        .cfi_escape 0x10,0x5,0x2,0x75,0
        pushl   %ecx
        .cfi_escape 0xf,0x3,0x75,0x7c,0x6
        subl    $16, %esp
        .cfi_escape 0x2e,0xc
        pushl   $.LC0
        .cfi_escape 0x2e,0x10
        call    puts
        movl    -4(%ebp), %ecx
        .cfi_def_cfa 1, 0
        xorl    %eax, %eax
        leave
        leal    -4(%ecx), %esp
        .cfi_def_cfa 4, 4
        ret
        .cfi_def_cfa 4, -12
        .cfi_escape 0x2e,0
        .cfi_endproc
.LFE12:
        .size   main, .-main
        .globl  size
        .section        .rodata
        .align 4
        .type   size, @object
        .size   size, 4
```

Temps d'exécution

real : 0m0,002s

user : 0m0,000s

sys : 0m,000s

Explications des modes d'optimisations

L'outil Gcc propose plusieurs modes d'optimisations du code assembleur. Ces optimisation ce base sur plusieurs critères, on y retrouve par exemple l'ordonnancement des blocs, le renommage des registres ou bien encore optimisations des appels. Ci-dessous un tableau montrant les optimisations réalisées en fonctions du mode d'optimisations choisis :

	Included in Level			
Optimization	-O1	-O2	-Os	-O3
defer-pop	●	●	●	●
thread-jumps	●	●	●	●
branch-probabilities	●	●	●	●
cprop-registers	●	●	●	●
guess-branch-probability	●	●	●	●
omit-frame-pointer	●	●	●	●
align-loops	○	●	○	●
align-jumps	○	●	○	●
align-labels	○	●	○	●
align-functions	○	●	○	●
optimize-sibling-calls	○	●	●	●
cse-follow-jumps	○	●	●	●
cse-skip-blocks	○	●	●	●
gcse	○	●	●	●
expensive-optimizations	○	●	●	●
strength-reduce	○	●	●	●
rerun-cse-after-loop	○	●	●	●
rerun-loop-opt	○	●	●	●
caller-saves	○	●	●	●
force-mem	○	●	●	●
peephole2	○	●	●	●
regmove	○	●	●	●
strict-aliasing	○	●	●	●
delete-null-pointer-checks	○	●	●	●
reorder-blocks	○	●	●	●
schedule-insns	○	●	●	●
schedule-insns2	○	●	●	●
inline-functions	○	○	○	●
rename-registers	○	○	○	●

Niveau o1

Le but de ce niveau est d'optimiser le programme dans un délai assez court, peu d'optimisations sont donc effectuées lors de cette étape. On se charge principalement de réduire la taille du fichier compilé en augmentant sa vitesse d'exécution.

Niveau 02

Ici, la taille du fichier peut être augmentée si cela accélère l'exécution du code. Le temps nécessaire pour réaliser l'optimisation n'est plus pris en compte.

Niveau oS

Ce mode est identique au mode o2 sauf qu'il ne réalise pas les optimisation susceptible d'augmenter la taille du fichier exécutable.

Niveau o3

Ce mode inclut toutes les optimisations possible, cependant, le code n'en est pas forcément plus rapide pour autant car si la taille du code généré dépasse la taille du cache, le code s'en trouvera ralentit. Il faut donc utiliser ce mode en connaissance de cause !

Commentaires

Avec le code que nous avons utilisé nous constatons peu de différences entre les différentes optimisations proposé par le compilateur gcc.

Cependant, nous remarquons que pour "Gcc -S main.c -0s -o 0S" le code assembleur généré est totalement différent de ce qui à été généré dans les autres modes d'optimisations. Cela démontre bien que le compilateur joue un grand rôle dans les ressources utilisés par le programme ainsi que sur son temps d'exécution.

Les temps mentionnés sont des moyennes établies par nos soins, elles ont été faites sur base de 10 exécutions du programme dans un système conversationnel de type linux distribution ubuntu.

Ces optimisations n'ont pas un impact déterministes sur les performances de notre programme, cependant, il est fortement possible que dans un programme plus conséquent, ces optimisations soient intéressantes, si cela nous permet de gagner quelque centièmes de seconde il alors très intéressant de les utiliser dans la conception d'application temps réel.

Conclusion

Ce dossier nous a permis d'y voir beaucoup plus clair au sujet des étapes réellement effectués lors de la compilation d'un programme et contrairement à ce que la majorité des développeurs pensent, cela ne se résume pas à la compilation d'un fichier source qui donne un fichier objet et après l'édition des liens donne un fichier exécutables.

La compilation est une opération plus complexe qu'il n'y parait et l'étude que nous avons effectués grâce à ce dossier nous a permis de bien comprendre tout les mécanisme enfuis d'un compilateur.

Pour terminer, il est important de bien connaître les outils que nous utilisons pour le développement, cela pourrait nous permettre d'améliorer la rapidité ainsi que la rentabilités d'un programme et dans notre domaine il est judicieux d'être déterministe dans le temps.

Bibliographies

Site web

Environnement de programmation

http://www.cmi.univ-mrs.fr/~contensi/coursC/index.php?section=env&page=comp

Optimization in GCC

http://www.linuxjournal.com/article/7269