

Programmation III

Chaine de compilation & utilitaires

Basile Schaeli

Roger D. Hersch

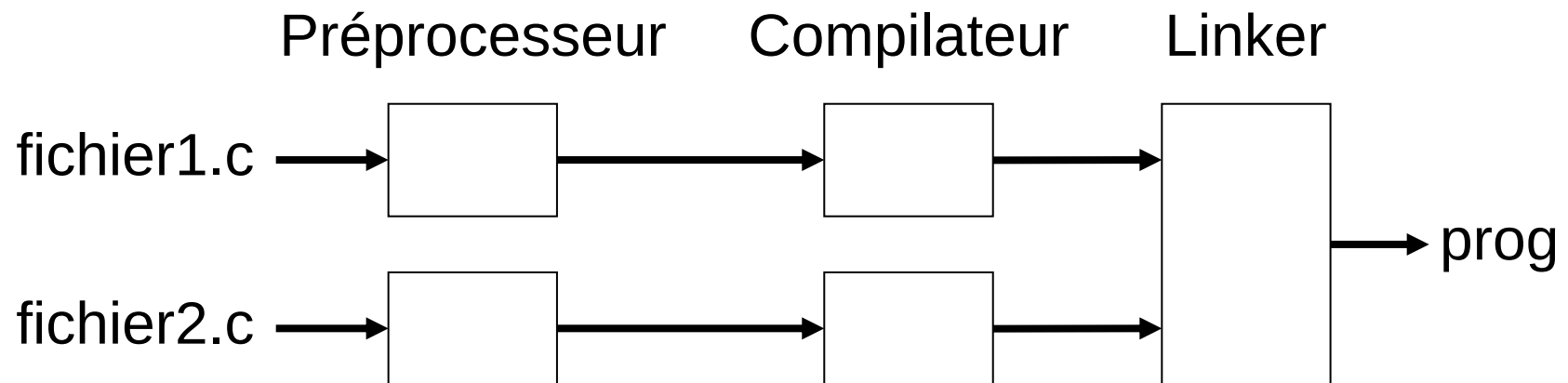
Laboratoire de Systèmes Périphériques (LSP)

EPFL

Contenu

- Chaîne de compilation
 - Préprocesseur
 - Compilateur
 - Editeur de lien
- Utilitaires
 - Make
 - Valgrind
 - gprof
 - CVS & Subversion

Chaine de compilation



- Le préprocesseur, le compilateur et le linker (ou éditeur de liens) sont généralement des programmes séparés
 - GCC appelle à tour de rôle les programmes *cpp*, *cc1/cc1plus* et *ld*
 - Microsoft utilise *cl* (préprocesseur et compilateur) et *link* (édition de lien)

Chaine de compilation

- Il est possible de n'exécuter qu'une partie de la chaine
- Avec GCC
 - Préprocesseur seulement: *gcc -E*
 - Préprocesseur et compilateur: *gcc -c*
 - Si aucune option n'est spécifiée, l'extension du nom de fichier détermine si les phases de preprocessing et de compilation doivent être exécutées

Préprocesseur

- Transforme un fichier texte en fichier texte
 - Le préprocesseur travaille uniquement avec des chaînes de caractères
- Le préprocesseur C/C++ traite les instructions commençant par #
- Le code vu par le compilateur n'est donc pas le même que le code vu par le programmeur
 - Une mauvaise utilisation d'instructions du préprocesseur sera donc généralement détectée uniquement lors de la compilation

#define

- Définit une macro, i.e. une règle de substitution de chaîne de caractères

#define PI 3.14

- Une macro peut aussi prendre des arguments

#define PRINT(integer) printf("%d\n",integer);
int main() { int a=2; PRINT(a); }

- Par convention, les macros sont en MAJUSCULES
- Problèmes potentiels
 - Paramètres invalides (aucune vérification de type)
 - Pas de vérification de l'existence d'une macro
 - Certains caractères ne sont pas traités par le préprocesseur (e.g. templates)

#undef

- Permet d'effacer la définition d'une macro
- Comme une macro ne peut être redéfinie, on peut donc utiliser *#undef* pour en changer la valeur

#define PI 3

#undef PI

#define PI 3.14

- Utile lorsque certains headers définissent des macros avec des noms trop communs
 - Ex: *#define max(a,b) (a<b) ? b : a*
- Problèmes potentiels:
 - Définitions de valeurs différentes par différents header

#ifdef, #ifndef, #else, #endif

- Conserve certaines parties de code selon qu'une macro soit définie ou non

```
#define TEST  
int main() {  
    #ifdef TEST  
        printf("TEST is defined\n");  
    #else  
        printf("TEST is not defined\n");  
    #endif  
}
```

#ifdef, #ifndef, #else, #endif

- C'est aussi utilisable avec des macros

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

```
#ifdef DEBUG
```

```
#define PRINT(integer) \
```

```
    printf("%s %d: %d\n",__FILE__,__LINE__,integer);
```

```
#else
```

```
#define PRINT(integer) printf("%d\n",integer);
```

```
#endif
```

```
int main() { PRINT(4); }
```

- Affichage

file.c 9: 4

#if, #else, #endif

- Evaluate une expression C simple
 - Opérateurs arithmétiques
 - Opérateurs logiques
- Conserve le code qui suit si l'expression est vraie (i.e. non nulle)

```
#if TEST==2 || TEST==3  
    /* Do something */  
#endif
```

#include

- Permet d'utiliser les fonctions, variables, et classes déclarées (ou prototypées) dans d'autre fichiers

#include <stdio.h>

#include "header.h"

- Le contenu du fichier à inclure est simplement copié à la place de l'instruction *#include*
 - Le compilateur peut trouver une erreur dans un fichier C alors qu'elle provient d'un fichier inclus
 - L'utilisation de l'extension .h (comme « header ») pour les fichiers contenant des déclarations n'est qu'une convention (qu'il est fortement recommandé de suivre)

<header.h> vs. "header.h"

- Les guillemets sont utilisés lorsque le fichier inclus est accessible depuis le répertoire courant

#include "path/to/header.h"

- Si le fichier se trouve à un emplacement connu du compilateur (i.e. dans son *path*), on peut utiliser *<>*

#include <header.h>

- On peut ajouter des répertoires au *path* du compilateur en utilisant l'option *-I*

gcc -Ipath/to -o prog prog.c

Inclusion en boucle

toto.h inclut tata.h qui inclut toto.h...

- Prototyper les classes et fonctions utilisées, ainsi que les méthodes qui les utilisent

- tata.h avant:

```
#include "toto.h"  
class Tata {  
    public: int f(Toto& t) { t.print(); }  
};
```

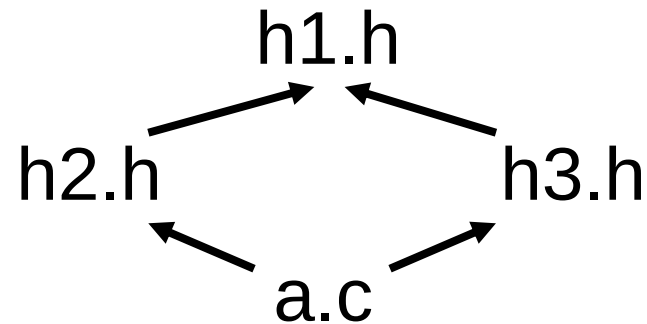
- tata.h après:

```
class Toto;  
class Tata { public: int f(Toto& t); };
```

- Cas extrême: utiliser pour chaque classe ou fonction un fichier en-tête et un fichier de définitions

Inclusions multiples

- Si un fichier est inclus plusieurs fois, les déclarations qu'il contient seront dupliquées



- On évite ceci en utilisant la construction suivante

```
#ifndef H1_H  
#define H1_H  
// Declare things  
#endif
```

Compilateur

- Le compilateur transforme un fichier texte en un fichier objet contenant deux éléments
 - Les instructions correspondantes compréhensibles par un processeur (ou par une famille de processeurs)
 - Une table de symboles décrivant les différents éléments (classes, variables, fonctions) définis dans le fichier compilé
- Sources d'erreurs
 - Erreurs de syntaxe
 - Utilisation incorrectes de types
 - Utilisation de fonctions/classes non prototypées

Editeur de liens

- Crée un fichier exécutable à partir d'un ou de plusieurs fichiers objets
 - Combine les tables de symboles
 - Remplace les références aux symboles utilisés par les pointeurs correspondants
- Sources d'erreurs
 - Définition multiple de symboles (e.g. deux fichiers incluent le même header définissant une variable)
 - Symbole manquant (e.g. une fonction est prototypée mais pas implémentée)

Edition de lien

- linkerror.h

```
void f() { /* Do something */ }
```

- linkerror1.cpp

```
#include "linkerror.h"
```

```
int main() { f(); }
```

- linkerror2.cpp

```
#include "linkerror.h"
```

```
/* Some other things */
```

Edition de lien (correct)

- linkerror.h

```
void f();
```

- linkerror1.cpp

```
#include "linkerror.h"
```

```
int main() { f(); }
```

- linkerror2.cpp

```
#include "linkerror.h"
```

```
void f() { /* Do something */ }
```

```
/* Some other things */
```

Contrôler la table de symbole

- Le mot-clé *static* empêche un symbole de figurer dans la table. Celui-ci n'est donc visible qu'à l'intérieur du fichier.
 - Valable pour les variables et les fonctions
 - ATTENTION! *static* n'a pas la même signification lorsqu'il qualifie un membre d'une classe!
- Le mot-clé *extern* indique qu'un symbole est défini dans un autre fichier, et qu'aucune mémoire ne doit donc être allouée pour ce symbole
 - Utile pour les variables
 - Les prototypes de fonctions et de classes sont implicitement définis comme *extern*

static

- static.h

```
int f();
```

- static1.cpp

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

```
#include "static.h"
```

```
static int counter=0;
```

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

- static2.cpp

```
#include "static.h"
```

```
static int counter=0;
```

```
void f() { ++counter; }
```

static II

- static_header.h

```
static int counter=0;  
void f();
```

- static_header1.cpp

```
#include <stdio.h>  
#include "static_header.h"  
int main() { f(); printf("%d\n",counter); }
```

- static_header2.cpp

```
#include "static_header.h"  
void f() { ++counter; }
```

extern

- extern.h

```
extern int counter;  
void f();
```

- extern1.cpp

```
#include <stdio.h>  
#include "extern.h"  
int main() { f(); printf("%d\n",counter); }
```

- extern2.cpp

```
#include "extern.h"  
int counter=0;  
void f() { ++counter; }
```

C vs. C++

- C
 - 1973: C
 - 1978: C K&R (Kernighan & Ritchie)
 - 1989: ANSI C
 - 1999: ISO/ANSI C99
- C++
 - 1983: C++ (Bjarne Stroustrup)
 - 1998: ISO C++
 - 200x: C++0x
- Des différences de syntaxe existent entre versions du langage
- Il est possible que le même code puisse être compilé ou non selon le compilateur (ou la version du compilateur) utilisé

C vs. C++

- On peut généralement compiler du code C avec un compilateur C++
 - La vérification de type est plus stricte en C++
 - Le C++ était une extension du C...
- En C, il est possible de lier des fichiers compilés avec différents compilateurs
 - Ce n'est pas le cas en C++
- On peut lier des fichiers C dans un programme C++
 - Utiliser des fonctions C dans un programme C++

Combiner C et C++

- Il est nécessaire d'informer le compilateur C++ qu'une fonction a été compilée avec un compilateur C
- `stdlib.h`

```
typedef double atof(const char *np);  
#ifdef __cplusplus /* starts with two _ */  
extern "C" {  
#endif  
double atof(const char *np);  
#ifdef __cplusplus  
}  
#endif
```

Références

- GCC

<http://gcc.gnu.org/onlinedocs/>

- Compilateur Microsoft

<http://msdn.microsoft.com/visualc/vctoolkit2003/>

- Historiques et incompatibilités entre C et C++

<http://www.jakeo.com/words/clanguage.php>

http://en.wikipedia.org/wiki/C_programming_language

http://en.wikipedia.org/wiki/C_Plus_Plus

<http://david.tribble.com/text/cdiffs.htm>

Makefiles

- Automatiser la compilation de programmes
 - Ou processus de traitement de fichiers
- Le processus à automatiser est décrit par un ensemble de *règles* contenant des *instructions*
- Chaque règle peut être soumise à un ensemble de *dépendances*

rule: dependencies

<tab>instruction1

<tab>instruction2

Makefiles

- Un makefile est interprété à l'aide de la commande *make*
- Make essaie d'interpréter un fichier nommé *makefile* ou *Makefile*
 - Le nom de ce fichier peut être spécifié avec *-f*
- On peut spécifier la règle à évaluer sur la ligne de commande, sinon la première règle du fichier est interprétée
make someRule
- Il y a *make* et *GNU make* (ou *gmake*)
 - On parle ici de la version GNU, mais les principes de base restent valables

Dépendances

- Les dépendances d'une règle peuvent être des fichiers, des répertoires ou d'autres règles
 - La plupart des règles ont pour but de générer un fichier portant le même nom que la règle
- Une règle est exécutée si
 - Aucun fichier ne porte le même nom que la règle
 - La règle et une dépendance correspondent à des noms de fichiers, et le fichier correspondant à la dépendance est plus récent que celui de la règle
 - Au moins une de ses dépendances est invalidée, i.e. il s'agit d'une règle qui vient d'être exécutée

Exemple

Makefiles can contain arbitrary instructions

someRule: someDir

@echo Create a dummy file

touch someDir/dummyfile

ls someDir

someDir:

mkdir someDir

Treedictionary

- Implémentation d'un dictionnaire en arbre
 - treedictionary.cpp et treedictionary.h
- Deux fonctions *main()* différentes
 - main.cpp et treedictmain.cpp
- Deux types de compilation
 - Avec et sans mesures de performances
- Quatre exécutables à générer
 - treedict et mydict
 - treedictM et mydictM

Makefile simple

Build all executables

all: treedict treedictM mydict mydictM

treedict: treedictionary.cpp treedictmain.cpp treedictionary.h

g++ -Wall -o treedict treedictionary.cpp treedictmain.cpp

mydict: treedictionary.cpp main.cpp treedictionary.h

g++ -Wall -o mydict treedictionary.cpp main.cpp

treedictM: treedictionary.cpp treedictmain.cpp treedictionary.h timer.h

g++ -Wall -DMEASURE_PERF -o treedictM treedictionary.cpp treedictmain.cpp

mydictM: treedictionary.cpp main.cpp treedictionary.h timer.h

g++ -Wall -DMEASURE_PERF -o mydictM treedictionary.cpp main.cpp

clean:

*rm -f *.o treedict treedictM mydict mydictM*

Variables

- Il est possible d'utiliser des variables
 - Affectation: `CXX=g++`
 - Utilisation: `$(CXX)`
- Noms de variables "réservés"
 - `CC`: compilateur C
 - `CXX`: compilateur C++
 - `CFLAGS/CXXFLAGS`: option de compilation
 - `LDFLAGS`: options de lien
- Make sait compiler des fichiers `.c` et `.cpp` en fichier `.o` en utilisant ces variables

Makefile v2

CXX=g++

CXXFLAGS=-Wall

EXECS=treedict treedictM mydict mydictM

all: \$(EXECS)

Specify dependency using object files built with implicit rule

treedict: treedictionary.o treedictmain.o treedictionary.h

\$(CXX) -o treedict treedictionary.o treedictmain.o

mydict: treedictionary.o main.o treedictionary.h

\$(CXX) -o mydict treedictionary.o main.o

treedictM: treedictionary.cpp treedictmain.cpp treedictionary.h timer.h

*\$(CXX) \$(CXXFLAGS) -DMEASURE_PERF -o treedictM treedictionary.cpp *
treedictmain.cpp

mydictM: treedictionary.cpp main.cpp treedictionary.h timer.h

*\$(CXX) \$(CXXFLAGS) -DMEASURE_PERF -o mydictM treedictionary.cpp *
main.cpp

*clean: rm -f *.o \$(EXECS)*

Variables automatiques et patterns

- Variables automatiques
 - $\$@$: Nom de la règle
 - $\$<$: Nom de la première dépendance
 - $\$^$: Nom de toutes les dépendances
- Pattern ' $\%$ '
 - Correspond à n'importe quelle chaîne de caractères
 - La substitution est faite de la même manière dans la règle et dans les dépendances
- $\%.o: \%.c$
 - fichier.o dépend de fichier.c
 - Le premier et le deuxième ' $\%$ ' prennent la même valeur

Makefile v3

CXX=g++

CXXFLAGS=-Wall

EXECS=treedict treedictM mydict mydictM

all: \$(EXECS)

treedict: treedictionary.o treedictmain.o

\$(CXX) -o \$@ \$^

mydict: treedictionary.o main.o

\$(CXX) -o \$@ \$^

treedictM: treedictionaryM.o treedictmainM.o

\$(CXX) -o \$@ \$^

mydictM: treedictionaryM.o mainM.o

\$(CXX) -o \$@ \$^

%.o: %.cpp treedictionary.h

\$(CXX) \$(CXXFLAGS) -c -o \$@ \$<

%M.o: %.cpp treedictionary.h timer.h

\$(CXX) \$(CXXFLAGS) -DMEASURE_PERF -c -o \$@ \$<

Génération de dépendances

- Il est possible de générer automatiquement les dépendances entre .o, .c/.cpp et .h
 - gcc -MM fichier.c
- Il suffit d'inclure le fichier contenant les dépendances, et d'ajouter une règle pour générer ce fichier

include depend

depend:

*\$(CXX) -MM *.cpp > depend*

Makefile générique

CXX=g++

CXXFLAGS=-Wall

EXEC=prog

OBJS=prog.o progmain.o

\$(EXEC): \$(OBJS)

\$(CXX) -o \$@ \$^

include depend

depend:

*\$(CXX) -MM *.cpp > depend*

clean:

rm -f depend \$(OBJS) \$(EXEC)

Référence

- <http://www.gnu.org/software/make/manual/>

Valgrind

- Débugueur et profiler d'utilisation mémoire
 - Utilisation de mémoire non initialisée
 - Lecture/écriture de blocs libérés (free)
 - Lecture/écriture après la fin de blocs alloués (malloc)
 - Fuites de mémoire (perte de pointeurs sur blocs alloués)
- Graphes de consommation mémoire
- Exécution beaucoup plus lente (10x-20x)
- Consommation de mémoire beaucoup plus importante
- Linux x86 seulement

val.c

```
1: #include <malloc.h>  
2:  
3: int main()  
4: {  
5:     int *a=(int*)malloc(2*sizeof(int));  
6:     a[0]=0;  
7:     a[1]=1;  
8:     a[2]=2;  
9:     return 0;  
10: }
```

Utiliser *valgrind*

```
% gcc -Wall -g val.c
```

```
% valgrind a.out
```

```
...
```

```
==20954==
```

```
==20954== Invalid write of size 4
```

```
==20954==    at 0x804837F: main (val.c:8)
```

```
==20954== Address 0x1BA50030 is 0 bytes after a block of size 8 alloc'd
```

```
==20954==    at 0x1B8FEA39: malloc (vg_replace_malloc.c:149)
```

```
==20954==    by 0x804835D: main (val.c:5)
```

```
==20954==
```

```
...
```

```
==20954==
```

```
==20954== LEAK SUMMARY:
```

```
==20954==    definitely lost: 8 bytes in 1 blocks.
```

```
==20954==    possibly lost: 0 bytes in 0 blocks.
```

```
==20954==    still reachable: 0 bytes in 0 blocks.
```

```
==20954==         suppressed: 0 bytes in 0 blocks.
```

```
==20954== Use --leak-check=full to see details of leaked memory.
```

Fuites mémoire

```
% valgrind --leak-check=full --show-reachable=yes a.out
```

```
...
```

```
==20922== Invalid write of size 4
```

```
==20922==    at 0x804837F: main (val.c:8)
```

```
==20922== Address 0x1BA50030 is 0 bytes after a block of size 8 alloc'd
```

```
==20922==    at 0x1B8FEA39: malloc (vg_replace_malloc.c:149)
```

```
==20922==    by 0x804835D: main (val.c:5)
```

```
==20922==
```

```
...
```

```
==20922==
```

```
==20922== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
```

```
==20922==    at 0x1B8FEA39: malloc (vg_replace_malloc.c:149)
```

```
==20922==    by 0x804835D: main (val.c:5)
```

```
==20922==
```

```
...
```

Utilisation mémoire

- Spécifier l'utilisation de l'outil *massif*
 - Génère un fichier texte et une image postscript

```
% valgrind --tool=massif lineardict list-full.txt 3 t.me
```

Heap allocation functions accounted for 99.9% of measured spacetime

Called from:

99.9% : 0x8048941: main (lineardict.cpp:65)

and 1 other insignificant place

```
% valgrind --tool=massif treedict list-full.txt 3 t.me
```

Heap allocation functions accounted for 93.2% of measured spacetime

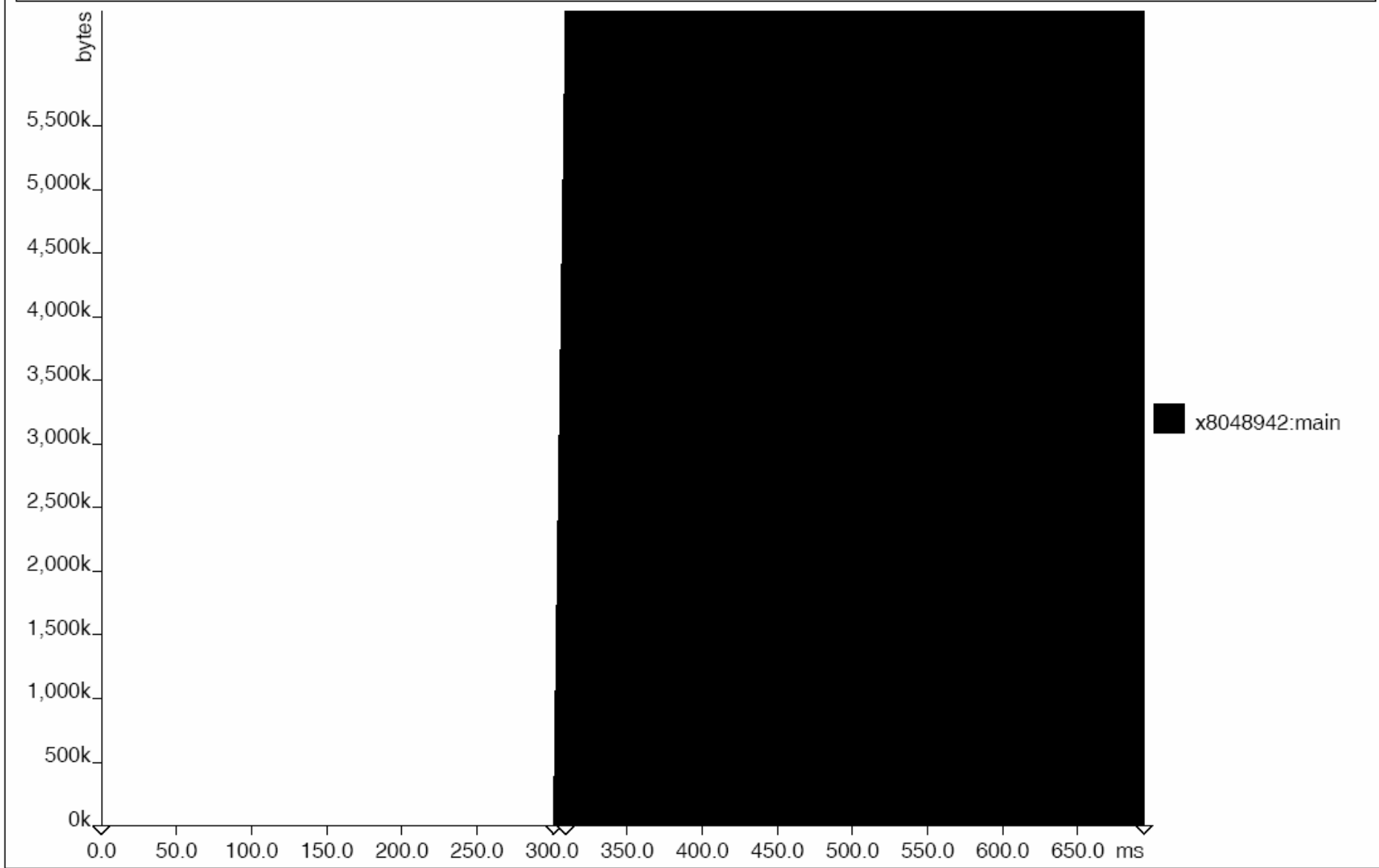
Called from:

93.2% : 0x8048C1E: TreeDictionary::addWord(char const*)
(treedictionary.cpp:110)

and 3 other insignificant places

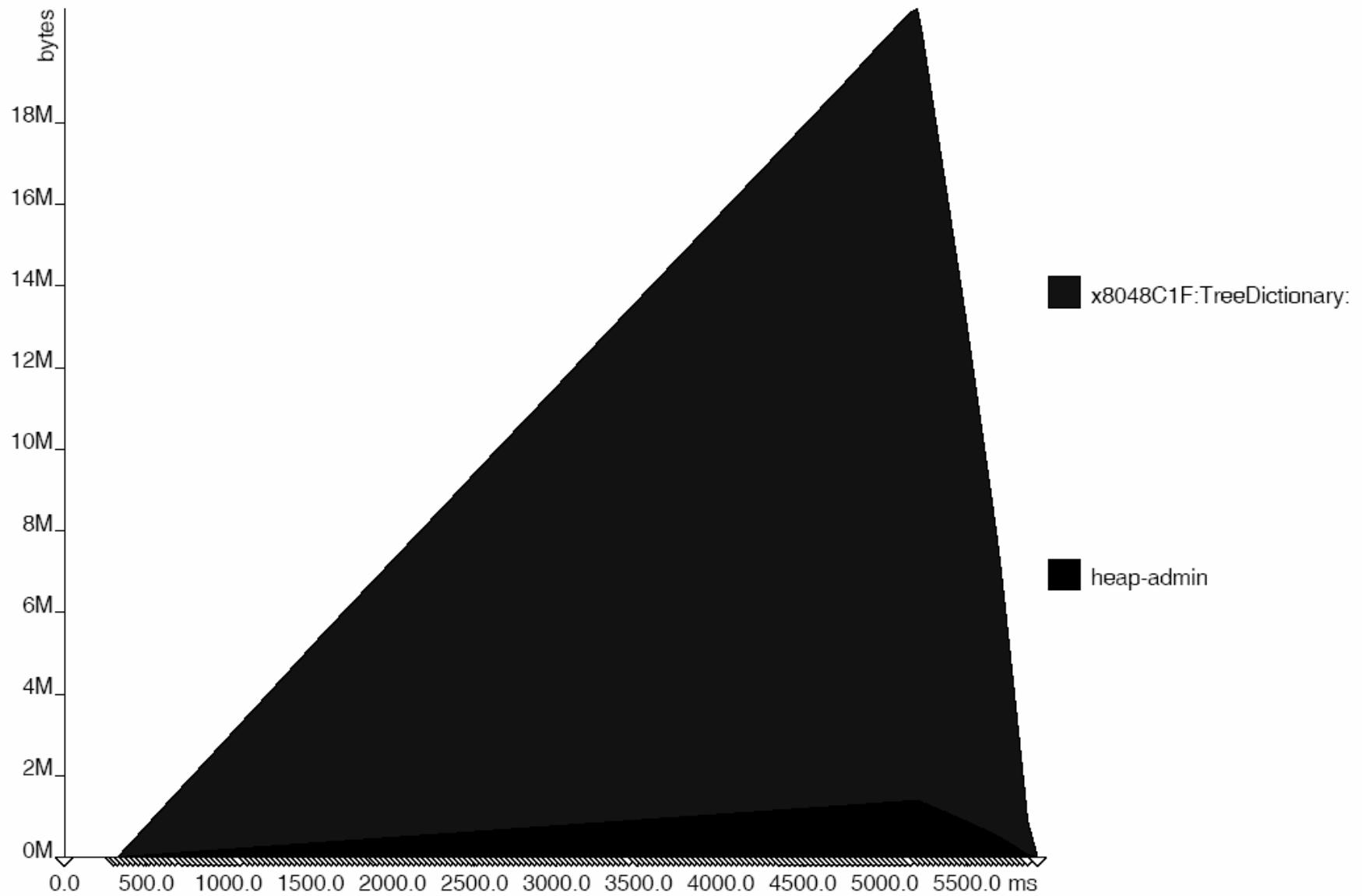
lineardict list-full.txt 3 t.me

2,489,600,000 bytes x ms



treedict list-full.txt 3 t.me

59,037,832,320 bytes x ms



Références

- <http://www.valgrind.org>
- Liste des outils disponibles
<http://www.valgrind.org/info/tools.html>

Optimisation de performance

- Votre programme compile et s'exécute correctement, mais il est trop lent
- Les options de compilation peuvent significativement améliorer le code généré
 - -O1, -O2, -O3
 - Différents niveaux d'optimisation
 - -march=*procname*
 - Génère du code spécifique au processeur spécifié
 - -funroll-loops
 - Essaie de dérouler les boucles (évite les sauts)

Profiling avec *gprof*

- Si les options de compilations ne suffisent pas, il est nécessaire d'analyser l'exécution du programme
 - D'où vient le problème?
 - Quelle partie faut-il optimiser?
- Un *profiler* exécute le programme et calcule le temps passé dans les différentes fonctions
- Compiler le programme avec *-pg*
 - Si vous utilisez des bibliothèques dynamiques, ajoutez *-static* lorsque vous liez le programme
- Exécutez le programme normalement
 - Un fichier *gmon.out* est généré
- Analysez le résultat
 - *gprof progName*

lineardict

```
void matchPattern(const char *pat, const char *word, size_t patLength) {
    if(patLength!=strlen(word))
        return;
    for(size_t i=0;i<patLength;++i)
        if(pat[i]!='.' && pat[i]!=word[i])
            return;
    ...
}

int main(int argc,char *argv[]) {
    ...
    double st=Timer::get();
    for(int j=0;j<1000;++j)
        for(int i=0;i<wordCount*MAX_LENGTH;i+=MAX_LENGTH)
            matchPattern(argv[3],words+i,strlen(argv[3]));
    double et=Timer::get();
    std::cout << "Pattern matching time: " << et-st << " s" << std::endl;
}
```

Sortie de *gprof*

% time	self seconds	calls	name
87.03	9.19		strlen
8.71	0.92	1	main
4.17	0.44	75273000	matchPattern(...)
0.09	0.01		fgets
0.00	0.00	2	Timer::get()
0.00	0.00	1	
			__GLOBAL__I__Z12matchPatternPKcS0_j
0.00	0.00	1	
			_static_initialization_and_destruction_0(...)
0.00	0.00	1	__tcf_0

lineardict, v2

```
void matchPattern(const char *pat, const char *word, size_t patLength) {
    if(patLength!=strlen(word))
        return;
    for(size_t i=0;i<patLength;++i)
        if(pat[i]!='.' && pat[i]!=word[i])
            return;
    ...
}
int main(int argc,char *argv[]) {
    ...
    size_t len= strlen(argv[3]);
    double st=Timer::get();
    for(int j=0;j<1000;++j)
        for(int i=0;i<wordCount*MAX_LENGTH;i+=MAX_LENGTH)
            matchPattern(argv[3],words+i,len);
    double et=Timer::get();
    std::cout << "Pattern matching time: " << et-st << " s" << std::endl;
}
```

Sortie de *gprof*

% time	self seconds	calls	name
82.99	4.88		strlen
9.61	0.56	75273000	matchPattern(...)
7.06	0.41	1	main
0.17	0.01		memcpy
0.17	0.01		strtok
0.00	0.00	2	Timer::get()
0.00	0.00	1	
			__GLOBAL__I__Z12matchPatternPKcS0_j
0.00	0.00	1	
			_static_initialization_and_destruction_0(...)
0.00	0.00	1	__tcf_0

Méthode d'échantillonnage

- Toutes les 10 ms, *gprof* regarde quelle fonction est en cours d'exécution et construit un histogramme
 - Deux exécutions peuvent donner des résultats différents pour des fonctions très courtes
 - Les profils obtenus sur un programme dont l'exécution est très courte ne sont pas fiables
- Fondamentalement, est-il utile d'optimiser un programme lorsque son exécution est très courte?

Références

- Pense-bête

<http://www.cs.duke.edu/~ola/courses/programming/gprof.html>

- Exemple d'utilisation

<http://www.tldp.org/linuxfocus/Francais/March2005/article371.shtml>

- Manuel de référence

<http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html>

CVS et Subversion

- Permettent à plusieurs utilisateurs de travailler simultanément sur les mêmes fichiers
 - Peut aussi être utile pour une personne travaillant avec plusieurs ordinateurs
- Contrôle de révisions
- Architecture clients-serveur (connectionless)
 - On crée un dépôt (repository) sur le serveur dans lequel on place les fichiers du projet
 - Chaque utilisateur récupère une copie des fichiers (checkout)
 - Après modifications, on stocke les changements sur le serveur (commit)
 - Les modifications sont stockées sous forme de différences
 - On met à jour sa copie locale avec les modifications des autres utilisateurs (update)

Références

- CVS (le standard)
 - <http://www.nongnu.org/cvs/>
 - <http://ximbiot.com/cvs/manual/>
 - <http://www.tortoisecvs.org/> (Client Windows)
 - <http://www.wincvs.org/> (Liste de clients et utilitaires)
- Subversion (le challenger)
 - <http://subversion.tigris.org/>
 - <http://svnbook.red-bean.com/>
 - <http://tortoisesvn.tigris.org/> (Client Windows)
 - <http://esvn.umputun.com/> (Client multiplateformes)