Programmation C avancée

Concepts & Outils
pour le développement

maj 01/2023

Programmation C avancée

Quizz compilation

Bibliothèque...

#QDLE#Q#AB*#70#

- J’ai produit mon programme ./a.out en utilisant libtoto.a, si je supprime le fichier libtoto.a alors
  - A) ./a.out ne s’exécute pas
  - B) ./a.out s’exécute

Erreur de compilation...

#QDLE#Q#ABC*#70#

- Lors de la compilation, l'erreur :

undefined reference to ...

- est une erreur :
  - A) de pré-compilation
  - B) de compilation
  - C) d'édition de lien

Compilation...

#QDLE#Q#ABCD*EFG#70#

- Quelles sont les étapes utilisées pour la création d'un objet à partir d'un fichier source:
  - A) La pré-compilation
  - B) La compilation
  - C) L’édition de lien
  - D) A & B
  - E) A & C
  - F) B & C
  - G) A & B & C

dépendances...

- mon projet se compose de
  - list.h et list.c, qui implémente un module de listes chaînées
  - hash.h et hash.c, qui implémente une table de hachage. L'implémentation **interne** de la table repose sur les listes chaînées.
  - Ces 2 modules sont compilés dans un bibliothèque libalgo.so
  - demo.c est un programme de démonstration des tables de hachage.

dépendances...

#QDLE#Q#ABCDEF*GH#70#

- mon projet se compose de
  - list.h et list.c, qui implémente un module de listes chaînées
  - hash.h et hash.c, qui implémente une table de hachage. L'implémentation interne de la table repose sur les listes chaînées.
  - Ces 2 modules sont compilés dans un bibliothèque libalgo.so
  - demo.c est un programme de démonstration des tables de hachage.

- je modifie « list.c », je dois regénérer :
  - A) list.o
  - B) hash.o
  - C) libalgo.so
  - D) demo
  - E) A & B
  - F) A & C
  - G) A & B & C
  - H) tout

dépendances...

#QDLE#Q#ABCDEFG*H#70#

- mon projet se compose de
  - list.h et list.c, qui implémente un module de listes chaînées
  - hash.h et hash.c, qui implémente une table de hachage. L'implémentation interne de la table repose sur les listes chaînées.
  - Ces 2 modules sont compilés dans un bibliothèque libalgo.so
  - demo.c est un programme de démonstration des tables de hachage.

- je modifie « list.h », je dois regénérer :
  - A) list.o
  - B) hash.o
  - C) libalgo.so
  - D) demo
  - E) A & B
  - F) A & C
  - G) A & B & C
  - H) tout

dépendances...

#QDLE#Q#ABCDE*FGH#70#

- mon projet se compose de
  - list.h et list.c, qui implémente un module de listes chaînées
  - hash.h et hash.c, qui implémente une table de hachage. L'implémentation interne de la table repose sur les listes chaînées.
  - Ces 2 modules sont compilés dans un bibliothèque libalgo.so
  - demo.c est un programme de démonstration des tables de hachage.

- je modifie « hash.c », je dois regénérer :
  - A) list.o
  - B) hash.o
  - C) libalgo.so
  - D) demo
  - E) B & C
  - F) B & C & D
  - G) A & B & C
  - H) tout

dépendances...

#QDLE#Q#ABCDEF*GH#70#

- mon projet se compose de
  - list.h et list.c, qui implémente un module de listes chaînées
  - hash.h et hash.c, qui implémente une table de hachage. L'implémentation interne de la table repose sur les listes chaînées.
  - Ces 2 modules sont compilés dans un bibliothèque libalgo.so
  - demo.c est un programme de démonstration des tables de hachage.

- je modifie « hash.h », je dois regénérer :
  - A) list.o
  - B) hash.o
  - C) libalgo.so
  - D) demo
  - E) B & C
  - F) B & C & D
  - G) A & B & C
  - H) tout

Programmation C avancée

Fonctions et macros variadiques

Fonctions à arguments variables

- Au niveau de la fonction appelée, il faut disposer d'un mécanisme permettant de récupérer les paramètres passés, du sommet de la pile vers le fond
- Ces paramètres peuvent être de types variables, mais il faut que le premier paramètre soit de type fixé, connu par l'appelant et l'appelé, afin que ses valeurs renseignent sur la présence et le type des paramètres suivants éventuels

Fonctions à arguments variables

exemple de fonction à arguments variables:

- printf / fprintf / sprintf / snprintf ...
- scanf / ...
- execl / execlp / execle
- fcntl
- ...

Fonctions à arguments variables

- La récupération des paramètres dans la fonction appelée peut se faire « à la main »
  - Difficulté à prendre en compte les spécificités de chaque architecture
    - Tailles des différents types
    - Sens de la pile
    - Passage des paramètres comme registres
- Utilisation de macros normalisées : `va_list`
  - Fichier d'entête `<stdarg.h>`
  - Norme C89 (ajout de la macro va_copy C99)

Fonctions à arguments variables

- Pour déclarer le prototype de la fonction ou bien la définir, on utilise le mot-clé « . . . » pour indiquer à partir de quel point les paramètres peuvent être de type quelconque ou bien absents

```
int mon_fprintf (FILE *, const char *, ...);
```

```
#include <stdarg.h>

int
mon_fprintf (
FILE *              flot,
const char *        format,
...)
{
```

Fonctions à arguments variables

- Pour accéder aux arguments de la pile, on utilise une structure de type `va_list`, servant à parcourir la pile, et manipulée au moyen de plusieurs macros :
  - `va_start (`liste`,` derparam`)` : initialise le parcours de pile, en se basant sur le dernier paramètre connu situé dans la pile, dont le nom lui est passé
  - `va_arg (`liste`,` type`)` : récupère la valeur du paramètre suivant contenu dans la pile
  - `va_end (`liste`)` : termine le parcours courant et permet un nouveau `va_start` sur la liste
  - va_copy(liste,liste) : permet de faire une copie de la liste d'arguments

Fonctions à arguments variables

```
int
mon_fprintf (
FILE *            flot,
const char *      format,
...)
{
  va_list           liste;
  char *            forptr;

  va_start (liste, format);
  for (forptr = format; *forptr != '\0'; forptr ++) {
    switch (*forptr) {
      case 'd' :
        fprintf (flot, "%d", va_arg (liste, int));
        break;
      case ...       /* On continue de la sorte */
    }
  }
  va_end (liste);   /* Nécessaire pour désallouer les structures */
  return (0);
}
```

Fonctions à arguments variables

- Il existe une ancienne version du mécanisme de gestion des fonctions à nombre variable d'arguments : `<varargs.h>`
  - Non compatible avec `<stdarg.h>`
  - Permet les fonctions sans aucun argument fixe
  - Ne plus utiliser (dans POSIX, nécessite l'ancienne style de définition des fonctions).

```
int
mon_fprintf (flot,format,va_alist)
FILE *            flot;
const char *      format;
va_dcl
{
  ...         /* Le corps de la fonction est identique au précédent
*/
}
```

Quelques règles

- L'utilisation des variadiques est « dangereux » sur le plan de la sécurité/mémoire
- Il existe des « exploits » basé sur cela
- Si vous écrivez des fonctions variadiques, penser à en faire deux versions :

```
#include <stdarg.h>

int vprintf(const char *format, va_list ap);

#include <stdio.h>
int printf(const char *format, ...){
        va_list args ;
        va_start(args , format) ;
        vprintf(format,args) ;
        va_end(args) ;
}
```

- Cela permet la ré-exploitation de la fonction :

```
void debug(char *fmt,...){
        va_list ap ;
        puts("\x1b[31m") ;
        va_start(ap,fmt) ;
        vfprintf(stderr,fmt,ap) ;
        va_end(ap) ;
        puts("\x1b[0m") ;
}
```

MACRO VARIADIC

- Disponible depuis le C99
- Utilisation de … dans les paramètres de la macro
- La variable __VA_ARGS__ permet de réutiliser

```
#define DEBUG(...) fprintf(stderr,«debug:»__VA_ARGS__)

int main(){
   DEBUG(« %d »,10); // Affiche: debug: 10
    return EXIT_SUCCESS;
}
```

MACRO VARIADIC

- Il n'est pas possible de traiter les arguments un par un car il n'y a pas de récursivité au niveau des macros

```
#define PRINT_INT()
#define PRINT_INT(arg,...) printf(« %d »,arg);
PRINT_INT(__VA_ARGS__)
```

- Néamoins, il existe une astuce pour « compter » les arguments (crédit: Laurent Deniau):

```
#define VARIADIC_NARG(...) VARIADIC_NARG_(__VA_ARGS__,VARIADIC_RSEQ_N())
#define VARIADIC_NARG_(...) VARIADIC_ARG_N(__VA_ARGS__)
#define VARIADIC_ARG_N( _1, _2, _3, _4, N,...) N
#define VARIADIC_RSEQ_N() 4,3,2,1,0
```

- exemple: VARIADIC_NARG(a,b,c)
- => VARIADIC_NARG_(a,b,c,4,3,2,1,0)
- => VARIADIC_ARG_N(a,b,c,4,3,2,1,0) => 3

MACRO VARIADIC

- Exemple d'utilisation:

```
#define VARIADIC_NARG(...) VARIADIC_NARG_(__VA_ARGS__,VARIADIC_RSEQ_N())
#define VARIADIC_NARG_(...) VARIADIC_ARG_N(__VA_ARGS__)
#define VARIADIC_ARG_N( _1, _2, _3, _4, N,...) N
#define VARIADIC_RSEQ_N() 4,3,2,1,0

#define CONCAT1(a,b) a##b
#define CONCAT(a,b) CONCAT1(a,b)

#define PRINT_SINGLE_INT(i) printf(« %d »,i);

#define PRINT_INT_1(i)     PRINT_SINGLE_INT(i)
#define PRINT_INT_2(i,...) PRINT_SINGLE_INT(i) PRINT_INT_1(__VA_ARGS__)
#define PRINT_INT_3(i,...) PRINT_SINGLE_INT(i) PRINT_INT_2(__VA_ARGS__)
#define PRINT_INT_4(i,...) PRINT_SINGLE_INT(i) PRINT_INT_3(__VA_ARGS__)

#define PRINT_INT(...) CONCAT(PRINT_INT_,VARIADIC_NARG(__VA_ARGS__))(__VA_ARGS__)

int main(){
    PRINT_INT(1,2,3);
    return EXIT_SUCCESS;
}
```

Programmation C avancée

Quizz : mémoire et debug

segfault

#QDLE#Q#ABCDEFGH*#70#

- Quelles affirmations sont vraies :
  - A) Lors d'un segfault, le système crée systématiquement un fichier core
  - B) Le segfault correspond obligatoirement à une erreur mémoire
  - C) Tout dépassement de tableau crée un segfault
  - D) Un fichier core est l'image mémoire d'un processus à un instant donné
  - E) A & B
  - F) A & C
  - G) B & C
  - H) B & D

gdb

#QDLE#Q#A*BCDEFGH#70#

- gdb ne peut pas être utilisé sur:
  - A) une bibliothèque dynamique
  - B) un processus en cours d'exécution
  - C) un exécutable
  - D) un fichier core
  - E) A & B
  - F) A & D
  - G) B & C
  - H) B & D

gdb

#QDLE#Q#ABCD*EFGH#70#

- gdb ne permet pas de:
  - A) connaître la pile d'appels de fonction à un instant donné
  - B) afficher la valeur des paramètres d'une fonction lors d'un appel à celle-ci.
  - C) suspendre l'exécution d'un processus
  - D) connaître le temps passé dans une fonction
  - E) A & C
  - F) C & D
  - G) B & C
  - H) A & B

gdb

#QDLE#Q#ABCDEF*GH#70#

- Dans gdb, pour avoir une correspondance entre les instructions machines exécutées et le code source à l'origine de ces instructions je dois:
  - A) avoir compilé avec l'option -g
  - B) avoir compilé avec l'option -fPIC
  - C) avoir compilé avec l'option -c
  - D) disposer des fichiers sources
  - E) A & C
  - F) A & D
  - G) B & C
  - H) B & D

valgrind

#QDLE#Q#ABC*D#70#

- valgrind ne permet pas de
  - A) détecter l'utilisation de variables non initialisées
  - B) détecter des fuites mémoires
  - C) détecter tout les débordements de tableaux
  - D) détecter les doubles libérations

valgrind

#QDLE#Q#ABCD*#70#

```
int main(){
    int *p=malloc(4) ;
    return 0 ;
}
```

- Quel est le type de la fuite mémoire :
  - A) indireclty lost
  - B) possibly lost
  - C) still reachable
  - D) definitely lost

Programmation C avancée

readline, variables d’environnement, getopt

readline

- readline est une bibliothèque qui facilite la gestion d'un prompt/saisie sur terminal
  - char * readline (const char *prompt);
- gestion de l'historique
  - void using_history (void)
  - HISTORY_STATE * history_get_history_state (void)
  - void history_set_history_state (HISTORY_STATE *state)
  - void add_history (const char *string)
  - void clear_history (void)
  - …..
- C'est readline qui est utilisé par **bash.** Entre autre, c'est readline qui gère la complétion (liste des fichiers si TAB)

readline

- lors de l'appel à *readline*, l'argument (le prompt) est affiché et l'on est en attente de la commande utilisateur
- la valeur de retour est la ligne entrée par l'utilisateur
- il faut libérer cette ligne avec **free**

readline

- C’est readline qui permet :
  - de se déplacer sur la ligne Ctrl-a Ctrl-e
  - de rappeler les commandes passées (flèches)
  - de rechercher dans l’historique (Ctrl-r)
  - de recopier le dernier argument de la ligne précédente Ctrl-.
  - de copier/coller : Ctrl-k Ctrl-y
- Il est possible de « programmer » la complétion (pour lister les commandes de vos programmes ou les options d’une commande).

readline : inputrc

- le comportement de readline est paramétrable via le fichier « .inputrc » (plusieurs dizaines d'options) :
  - tous les raccourcis clavier
  - création de nouveau raccourci
  - divers :
    - set completion-ignore-case On
    - set expand-tilde On
    - set match-hidden-files off
    - …
- Le fichier doit être à la racine du compte
- On peut spécifier le fichier à travers la variable d'environnement INPUTRC

history

- La fonction *add_history(char *)* permet de sauvegarder une ligne dans l'historique
- *read_history* permet de charger un fichier d'historique
- *write_history* permet de sauvegarder l'historique dans un fichier
- Il existe de nombreuses fonctions pour manipuler l'historique (recherche etc.)

  => voir : GNU History Library

Les variables d'environnement

- Le système maintient un ensemble de variable dites « d'environnement)
- On peut déclarer une variable dans un terminal avec les commandes :

  set NAME=VALUE

  export NAME=VALUE

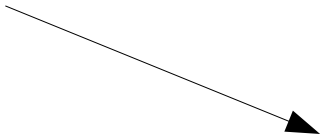
- Lors de l'exécution d'un processus, les variables sont transmises aux programmes (disponibles « comme » des variables statiques :

  int main(int argc, char **argv, char **envp)

- **envp** est un tableau de chaînes de la forme A=B (ou juste A=)

Les variables d’environnement

```
int main(int argc, char **argv, char **envp){
  int i=0;
  while(envp[i]!=NULL){
    printf("%s\n",envp[i]);
    ++i;
  }
}
```



```
.....
LESSCLOSE=/usr/bin/lesspipe %s %s
LC_PAPER=fr_FR.UTF-8
LC_MONETARY=fr_FR.UTF-8
TERMINATOR_UUID=urn:uuid:96d75675-e932-4457-8373-859145acd0e4
XDG_MENU_PREFIX=gnome-
ONSHAPE_SECRET_KEY=ululwuAOUYsEMLQToOWKywBttDsYHp9dja5JQAV
O1IXvgz4c
LANG=en_US.UTF-8
DISPLAY=:0
OLDPWD=/home/allali
GNOME_SHELL_SESSION_MODE=ubuntu
COLORTERM=truecolor
DESKTOP_AUTOSTART_ID=10d5a18baa5e367a8f161471507433378400000067840007
QT_SCREEN_SCALE_FACTORS=2
USERNAME=allali
XDG_VTNR=2
GIO_LAUNCHED_DESKTOP_FILE_PID=7084
SSH_AUTH_SOCK=/run/user/1001/keyring/ssh
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
LC_NAME=fr_FR.UTF-8
XDG_SESSION_ID=7
USER=allali
DESKTOP_SESSION=ubuntu
……..
```

getenv/setenv

- int main(int argc, char **argv, char**arge) ;

  OU :

- char *getenv(const char *name);

- int setenv(const char *name, const char *value, int overwrite);

- int putenv(char *string);

utilisation

- Voici quelques variables d'environnement « standards » :
  - PWD : current working directory
  - HOME : home directory
  - USER : user login
  - LANG et LC_… : ensemble de variable de régionalisation
- Exemple d'utilisation :
  - Indiquer un répertoire pour des plugins
  - Activer/Désactiver des fonctionnalités (cf. mallopt)

Programmation C avancée

Quizz : les outils

gestionnaire de source

#QDLE#Q#ABC*D#70#

- laquelle de ces affirmations est fausse :

A) svn est un gestionnaire centralisé

B) git intègre nativement le concept de branche

C) svn intègre nativement le concept de feuille

D) le numéro de révision de dépôt svn augmente après chaque commit

TDD

#QDLE#Q#ABC*DE#70#

- tdd s’est :

  A) une drogue des années 80’

  B) une technique de debug

  C) un environnement de développement

  D) une méthode de développement

  E) un travail très dirigé

tests

#QDLE#Q#A*BCD#70#

- Les tests d’intégrations :

  A) testent l’interopérabilité de deux ou plusieurs modules

  B) testent le fonctionnement global d’une application

  C) testent la conformité d’une fonction à une spécification

  D) valident un livrable pour le client

couverture

#QDLE#Q#ABC*DEF#70#

- Pour établir la couverture de mes tests je vais :
  - A) compiler avec les options -g -O0
  - B) compiler avec l'option –gcoverage
  - C) utiliser le programme gcov
  - D) compiler avec l'option -pg
  - E) compiler avec l'option –blanket
  - F) utiliser le programme gprof

l’intégration continue

#QDLE#Q#ABCDEF*G#80#

- l’intégration continue :
  - A) est une méthode d’écriture de tests
  - B) est un type de tests
  - C) permet de mieux gérer un ensemble hétérogène de configurations cibles
  - D) permet la validation d’un ensemble de points au cours du développement
  - E) A & B
  - F) C & D
  - G) E & F

Programmation C avancée

Résumé des points importants

La compilation

- Trois étapes :
  - Pré-compilation : regroupement des sources en un seul fichier (include) et traitement des macros
  - Compilation : depuis un seul fichier source, production d'un fichier objet binaire.
  - Édition de liens : produit un exécutable (la fonction main sert de point de départ) ou d'une bibliothèque dynamique

La pré-compilation

- Le résultat de la pré-compilation peut être obtenu avec l'option « -E »

  gcc -E fichier.c -o precompilation.c

- On peut définir des macros depuis la ligne de compilation avec l'option -DNOM ou -DNOM=VALEUR

  gcc -DNDEBUG -DSIZE=100 fichier.c

- L'option -Ichemin permet d'indiquer un répertoire où chercheur des fichiers inclus

Compilation

- Produit du code machine regroupé par fonctions :

  gcc -c fichier.c -o fichier.o

- On peut obtenir un version lisible en assembleur avec l'option -s

  gcc -S fichier.c -o fichier.s

- On peut lister les symboles avec *nm* ou *objdump*

  nm fichier.o

- Les symboles utilisés mais non implémentés sont indiqués comme « manquants »

Edition de liens

- On regroupe les .o en un executable ou une bibliothèque
- L'option -lnom permet d'ajouter la bibliothèque libnom.so pour la résolution des symboles manquants. L'option -Lchemin permet d'indiquer un répertoire où chercher les bibliothèques

```
gcc fichier.o -lsdl -o exec
```

- L'outil « ldd » permet de lister les dépendances en bibliothèques dynamiques :

```
ldd exec
```

bibliothèques

- Les bibliothèques sont des fichiers qui regroupent du code objet
- Les bibliothèques statiques sont une archive de .o créée avec l'outil « ar »
- Les bibliothèques dynamiques regroupe des .o compilés avec l'option -fPIC, elles sont créées avec l'éditeur de lien :

```
gcc -fPIC -c fichier.c -o fichier.o
gcc -shared fichier.o -o libfichier.so
```

Processus

- L’exécution d’un programme crée un processus dans le système
- La mémoire du processus est segmenté en page mémoire
- Le contenu du programme est chargé en mémoire ainsi que les bibliothèques dynamiques dont il dépend
- La mémoire de la pile est réservée dès le lancement
- La mémoire dynamique n’est pas réservée : il faut utiliser malloc pour qu’il demande au système de nouvelles pages

gdb

- Gdb est un debuger permettant de contrôler l'exécution d'un processus
- Pour faire le lien entre les instructions machines et le source il faut compiler avec l'option -g (étape de compilation). Il est souhaitable d'ajouter l'option -O0
- On peut lancer gdb sur un processus en cours d'exécution ou sur un fichier core (gcore permet de demander au système de produire un core)

valgrind

- Valgrind est une plateforme proposant différents outils pour l'analyse d'une execution
- Par défaut memcheck est l'outil d'analyse utilisé
- memcheck permet de détecter des accès mémoires problématiques (utilisation d'un variable non initialisée, dépassement de tableau,…)
- memcheck permet de détecter les fuites mémoires
- Il est possible de suspendre l'exécution lors d'une erreur avec l'option –vgdb-error=1 ce qui permet de faire une analyse avec gdb

Convention de nommage

- La convention de nommage (ou codage) est un document texte qui regroupe les règles d’écriture à suivre pour produire du code dans un projet.

doxygen

- Doxygen permet de générer de la documentation à partir de commentaires formatés dans les fichiers sources

Gestionnaire de sources

- Un gestionnaire de sources permet de garder l'historique des modifications d'un projet
- Deux types : centralisés (svn par ex) ou dé-centralisés (git par ex)
- Les gestionnaires sont basés sur diff et patch
- diff permet de comparer des fichiers pour mettre en avant les modifications. Les options « -rupN » permet de produire un fichier « patch »
- patch analyse un fichier patch et tente d'appliquer les modifications décrite sur les fichiers du répertoire courant. L'option -pX permet de dire à patch d'ignorer les X premiers répertoires

Gestion de sources

- Quel que soit le gestionnaire le développement doit se faire dans des branches
- La méthodologie consiste à régulièrement récupérer les modifications de trunk dans la branche : merge
- Une fois le développement achevé, on intègre celui-ci dans trunk/master avec merge

Automatisation

- make est un outil pour la production automatique de fichier à jour à partir de fichier présents sur le système
- make repose sur des règles de production décrites dans un fichier Makefile
- Les règles ont la forme :

  production : fichier1

      commande fichier1 > production

- Pour la compilation, c'est à vous de décrire correctement les dépendances

cmake

- cmake est un outil permettant de produire des règles de compilation en fonction de ce que l'on veut créer : executable ou bibliothèque
- cmake se base sur des fichiers CmakeLists.txt
- cmake est conçu pour faire de l'out-source building : on compile dans un répertoire séparé des sources
- cmake analyse les sources et génère les dépendances

IDE

- Un IDE est un environnement de développement intégrant plusieurs outils :
  - Un éditeur avec analyse des sources à la volée
  - Un debuger
  - L'interaction avec le gestionnaire de sources
- Quelques IDE :
  - Qtcreator / CLion/ VSCode / ...

Intégration continue

- Plateforme permettant d'exécuter des taches lors de la mise à jour d'un dépôt : compilation et exécution de tests
- La qualité de l'intégration continue dépend de deux choses :
  - L'exhaustivité des tests
  - La diversité des environnements sur lesquels les taches sont lancées
- Exemple de plateforme : jenkins

TDD

- TDD est une méthodologie de développement basée sur l'écriture de tests
  - On commence par écrire un test
  - On vérifie qu'il échoue
  - On écrit le code minimal permettant de valider le test
  - On recommence
- Permet d'avoir un code intégralement testé en permanence

Les tests :

- Les tests ont pour objectifs de valider votre code :
  - au niveau d'une fonction : tests unitaires
  - au niveau d'un module : tests fonctionnels
  - entre plusieurs modules : tests d'intégration
  - au niveau général, applicatif : tests de recette
- Les tests unitaires doivent être le plus indépendants possible (recours aux faussaires/mock).
- Les tests doivent être basés sur la spécification d’une fonction/module et non l’implémentation

Couverture

- La couverture d'une exécution consiste à connaître les lignes de code effectivement utilisées lors d'une exécution
- On compile avec l'option « --coverage »
- On lance une ou plusieurs executions
- On analyse le résultat avec « gcov » ou « lcov »

Performance

- L’organisation de la mémoire joue un rôle sur la performance (temps d’exécution/ressources utilisées)
- Le processeur utilise des caches mémoire internes avec une politique Least Recently Used
- On connaître le temps processeur utilisé par une exécution avec « time »

gprof

- L’option de compilation « -pg » permet de produire un exécutable qui va générer des traces intégrant des informations de temps passé dans chaque fonction
- « gprof » permet l’analyse de ces traces : il génère le graphe d’appel, le temps passé dans chaque fonction et dans les sous-fonctions…

Métriques

- Il est intéressant de suivre un ensemble de métriques sur un projet : cela permet de mettre en avant des problèmes
  - Taux de commentaires
  - Code dupliqué
  - Nombre de ligne de code / de fichiers
- Exemple d'outils : SonarQube