Programmation C : Méta & rappels

Aloÿs Dufour

ATER, LIPN équipe LoCal
Université Paris-Nord XIII

9 janvier 2026

Au programme

1. Bogues & bonnes pratiques
2. Testes & Types
3. Analyse statique
4. Analyse dynamique : débogage, profilage
5. Librairies et gestionnaires
6. Git et mails ?
7. Rappels

Types de bogues classiques

- *use-after-free*

```
int *p1 = malloc(sizeof (int));
int *p2 = p1;
free(p2);
*p1 = 8;
```

Types de bogues classiques

- *use-after-free*

```
int *p1 = malloc(sizeof (int));
int *p2 = p1;
free(p2);
*p1 = 8;
```

- *memory leak*

```
int *a = malloc(sizeof(int));
a = 0;
```

Types de bogues classiques

- *use-after-free*

```
int *p1 = malloc(sizeof (int));
int *p2 = p1;
free(p2);
*p1 = 8;
```

- *memory leak*

```
int *a = malloc(sizeof(int));
a = 0;
```

- *buffer overflow*

```
char buf[64];
gets(buf);
```

Types de bogues classiques

- *use-after-free*

```
int *p1 = malloc(sizeof (int));
int *p2 = p1;
free(p2);
*p1 = 8;
```

- *memory leak*

```
int *a = malloc(sizeof(int));
a = 0;
```

- *buffer overflow*

```
char buf[64];
gets(buf);
```

Types de bogues classiques

- *use-after-free*

```
int *p1 = malloc(sizeof (int));
int *p2 = p1;
free(p2);
*p1 = 8;
```

- *memory leak*

```
int *a = malloc(sizeof(int));
a = 0;
```

- *buffer overflow*

```
char buf[64];
gets(buf);
```

Tout bug est un problème de sécurité en programmation système : *Smashing the stack for fun and profit.*

Mauvaises pratiques

(Voir dangereuses)

- Ne pas initialiser les variables
- Ne pas tester les codes de retours des fonctions (systèmes)
- Utilisation de fonctions réputées dangereuses (pas de testes de place : overflow, core dumped)
  `sprintf, strcpy, strcat, vsprintf, gets`
- Oublier de désallouer/fermer ce qui ne sert plus.

Mauvaises pratiques

```
_,x,y,o       ,N;char       b[1920]       ;p(n,c)
{for(;n       --;x++)       c==10?y       +=80,x=
o-1:x>=       0?80>x?             c       !='-'?b
[y+x]=c       :0:                         0:0;}c(
       q,l,r)       char*l,*r;{while              (q>=0)q
       =("E"    "?yYrIxC{e^}KhE>[|LXbj}"         "dOVsJ"
       "@"    "idOV{Yab[bW}[bW}\\qFywyv{D"       "ma\\A"
       ""   "Ztq?Lyw>e{|Zq>Y\\gq\\qI[tYBe{w"     "yvDZE\
vBA[`_"    "Lo>}KcqdYrWqKxzKtW]|DXRwsfcUaT\\\
KXw{Y"    "RsFwsFwsFw{zaqyaz|FmMpyaoyI\\]cuUw{J"
[_/6]   -62>>_++%6&1?r[q]:l[ql)-99;return q;}E(a
){for   (o=x=a,y=0,_=0;_         <1006;)a=" /\\\n"
       "~|_."[c(6,"b"                "cd\\a[g","^~"   _e"
       "]fh")+8], p(   "#$%"         "&'()*+,-.1"[   c(11
      ,"_ac[]\\YZi"   "jkm",          "`bd^efghXWlV" )+13
      ]-34,a);}main   (k,Z )char**Z;{float z[1920]  ,A=0
,B=0  ,i,j; puts(""          "\x1b"   "[2J");for(;;
)  {  float e=sin(A          ),  n=   sin(B),g=cos(
A),m   =cos(B);for(k         =0;k<   1840;k++)y=-k
/80-   10,o=41+(k%80-              40)*1.3/y+n,N=
        A-100.0/y, b[k]=         ".#"[o+N&1],z[k]   =0;E(
        80-(int)(9*B)%250);for(j=0;6.28>j;j+=0.07  )for(
         i=0;6.28>i;i+=0.02) { float c=sin(i),d   =cos(j
           ),f=sin(j),h=d+2,D=15/(c*h*e+f*g+5    ),l=cos
(i),t=c      *h*g-f*e;x=40+2*D*(l*h*m-t*n),y=   12
+D*(l*h       *n+t*m),o=x+80*y,N=8*((f*e-c    *d*
g)*m-c*         d*e-f*g-l*d*n);if(D>z[o]    )z[o]
=D,b[o]              =" ..,,-++=#$@"[      N>0?N:0
       ]  ;  }                                    printf(
       "\x1b["       "H"           );for(k       =1;1841
       >k;k++)       putchar       (k%80?b       [k]:10)
       ;   A +=      0.053;B       +=0.037       ;  }  }
```

Bonnes pratiques

Localement, le contraire de l'IOCCC

Bonnes pratiques

Localement, le contraire de l'IOCCC

- style (indentation, format) de code propre
- commenter l'utilité des fonctions et les parties compliquées
- utiliser des noms de variables/fonctions courtes et explicites
- garder le code simple et minimal (principe KISS)
- préférer des algo simples avant de passer à des compliqués
- éviter la redondance

Bonnes pratiques

Localement, le contraire de l'IOCCC

- style (indentation, format) de code propre
- commenter l'utilité des fonctions et les parties compliquées
- utiliser des noms de variables/fonctions courtes et explicites
- garder le code simple et minimal (principe KISS)
- préférer des algo simples avant de passer à des compliqués
- éviter la redondance

Globalement,

- code "modulaire"
- documentation
- utilitaire de versionnage (`git`)
- Makefiles, ./configure
- analyse statique

Testes

Pour la détection d'erreurs de syntaxe, d'erreurs sémantiques : compiler.

Testes

Pour la détection d'erreurs de syntaxe, d'erreurs sémantiques : compiler.

Pour vérifier que le code fait ce que l'on veut : avoir quelques cas/exemples scriptés, mieux : "tests unitaires" (vérification du bon fonctionnement pour chaque partie/unité/module du code).

Testes

Pour la détection d'erreurs de syntaxe, d'erreurs sémantiques : compiler.

Pour vérifier que le code fait ce que l'on veut : avoir quelques cas/exemples scriptés, mieux : "tests unitaires" (vérification du bon fonctionnement pour chaque partie/unité/module du code).

Les bons tests explorent une large partie des possibilités et cas dégénérés.

Testes

Pour la détection d'erreurs de syntaxe, d'erreurs sémantiques : compiler.

Pour vérifier que le code fait ce que l'on veut : avoir quelques cas/exemples scriptés, mieux : "tests unitaires" (vérification du bon fonctionnement pour chaque partie/unité/module du code).

Les bons tests explorent une large partie des possibilités et cas dégénérés.

Séparer les étapes de développement : ajout de fonctionnalités d'un côté, et "refactoring" de l'autre (réorganisation, clarification, simplification, optimisation).

Au-delà des testes

Programmation “par contrats” : avec des assertions qui évaluent la véracité d’expression avec pré-condition, post-conditions et invariants (parallèle avec la logique de Hoare).
En C : `assert(expr)`, dans `assert.h`, se désactive avec l’option de compilation `-DNDEBUG`.

Au-delà des testes

Programmation “par contrats” : avec des assertions qui évaluent la véracité d’expression avec pré-condition, post-conditions et invariants (parallèle avec la logique de Hoare).
En C : `assert(expr)`, dans `assert.h`, se désactive avec l’option de compilation `-DNDEBUG`.

Problème profond sous-jacent : la correction du programme. Notion sémantique, un système de type suffisamment avancé peut remplacer beaucoup de testes à la compilation (langages statiques).
Programmation dans un langage “fortement typé” (langages fonctionnels tels que Haskell, OCamL, ou bien rust).

Analyse statique (*linter*)

Analyse du code sans l'exécuter.

Analyse statique (*linter*)

Analyse du code sans l'exécuter.
De nos jour, une bonne partie de l'analyse statique est passée dans les compilo,

- analyse lexicale (*lexer*)

Analyse statique (*linter*)

Analyse du code sans l'exécuter.
De nos jour, une bonne partie de l'analyse statique est passée dans les compilo,

- analyse lexicale (*lexer*)
- pré-traitement (préprocesseur)

Analyse statique (*linter*)

Analyse du code sans l'exécuter.
De nos jour, une bonne partie de l'analyse statique est passée dans les compilo,

- analyse lexicale (*lexer*)
- pré-traitement (préprocesseur)
- analyse syntaxique (*parsing*)

Analyse statique (*linter*)

Analyse du code sans l'exécuter.
De nos jour, une bonne partie de l'analyse statique est passée dans les compilo,

- analyse lexicale (*lexer*)
- pré-traitement (préprocesseur)
- analyse syntaxique (*parsing*)
- analyse sémantique (résolution de noms, vérification de type, affectations),

Analyse statique (*linter*)

Analyse du code sans l'exécuter.
De nos jour, une bonne partie de l'analyse statique est passée dans les compilo,

- analyse lexicale (*lexer*)
- pré-traitement (préprocesseur)
- analyse syntaxique (*parsing*)
- analyse sémantique (résolution de noms, vérification de type, affectations),
- optimisations

Analyse statique (*linter*)

Analyse du code sans l'exécuter.
De nos jour, une bonne partie de l'analyse statique est passée dans les compilo,

- analyse lexicale (*lexer*)
- pré-traitement (préprocesseur)
- analyse syntaxique (*parsing*)
- analyse sémantique (résolution de noms, vérification de type, affectations),
- optimisations
- édition de liens (*linker*)

Analyse statique (*linter*)

Analyse du code sans l'exécuter.
De nos jour, une bonne partie de l'analyse statique est passée dans les compilo,

- analyse lexicale (*lexer*)
- pré-traitement (préprocesseur)
- analyse syntaxique (*parsing*)
- analyse sémantique (résolution de noms, vérification de type, affectations),
- optimisations
- édition de liens (*linker*)

Analyse statique (*linter*)

Analyse du code sans l'exécuter.
De nos jour, une bonne partie de l'analyse statique est passée dans les compilo,

- analyse lexicale (*lexer*)
- pré-traitement (préprocesseur)
- analyse syntaxique (*parsing*)
- analyse sémantique (résolution de noms, vérification de type, affectations),
- optimisations
- édition de liens (*linker*)

Le tout dans le respect du format ELF de fichiers exécutables.

Analyse statique (*linter*)

Analyse du code sans l'exécuter.
De nos jour, une bonne partie de l'analyse statique est passée dans les compilo,

- analyse lexicale (*lexer*)
- pré-traitement (préprocesseur)
- analyse syntaxique (*parsing*)
- analyse sémantique (résolution de noms, vérification de type, affectations),
- optimisations
- édition de liens (*linker*)

Le tout dans le respect du format ELF de fichiers exécutables.
Spécifications formelles, *model checking*, interprétation abstraite.

Limite théorique

Détection d'inter-blocage, de boucles infinies... ?
Programme correct qui s'arrête sur toute entrée ?

Limite théorique

Détection d'inter-blocage, de boucles infinies… ?
Programme correct qui s'arrête sur toute entrée ?

Problème de l'arrêt : indécidable.

Theorem (de Rice)

Toute propriété sémantique non-triviale d'un programme est indécidable.

Analyse dynamique

Analyse du programme en cours d'exécution : vérification de types à la volée (python et beaucoup de langages interprétés), outils de visualisation de l'état du programmes, débogueurs.
Pour déboguer un programme C, le compiler avec l'option `-g`.

Valgrind détection des variables non-initialisées, de quelques types de fuites mémoires, dépassement de tableaux.
Divers outils disponibles via `valgrind --tool=<toolname>` :

Analyse dynamique

Analyse du programme en cours d'exécution : vérification de types à la volée (python et beaucoup de langages interprétés), outils de visualisation de l'état du programmes, débogueurs.
Pour déboguer un programme C, le compiler avec l'option `-g`.

Valgrind détection des variables non-initialisées, de quelques types de fuites mémoires, dépassement de tableaux.
Divers outils disponibles via `valgrind --tool=<toolname>` :

- par défaut, `memcheck` (non-allocation, non-initialisation, inaccessibilité), double-free...,

Analyse dynamique

Analyse du programme en cours d'exécution : vérification de types à la volée (python et beaucoup de langages interprétés), outils de visualisation de l'état du programmes, débogueurs. Pour déboguer un programme C, le compiler avec l'option `-g`.

Valgrind détection des variables non-initialisées, de quelques types de fuites mémoires, dépassement de tableaux.
Divers outils disponibles via `valgrind --tool=<toolname>` :

- par défaut, `memcheck` (non-allocation, non-initialisation, inaccessibilité), double-free...,
- `massif` : profilage de tas,

Analyse dynamique

Analyse du programme en cours d'exécution : vérification de types à la volée (python et beaucoup de langages interprétés), outils de visualisation de l'état du programmes, débogueurs. Pour déboguer un programme C, le compiler avec l'option `-g`.

Valgrind détection des variables non-initialisées, de quelques types de fuites mémoires, dépassement de tableaux.
Divers outils disponibles via `valgrind --tool=<toolname>` :

- par défaut, `memcheck` (non-allocation, non-initialisation, inaccessibilité), double-free...,
- `massif` : profilage de tas,
- `cachegrind, callgrind` : profilage de cache,

Analyse dynamique

Analyse du programme en cours d'exécution : vérification de types à la volée (python et beaucoup de langages interprétés), outils de visualisation de l'état du programmes, débogueurs. Pour déboguer un programme C, le compiler avec l'option `-g`.

Valgrind détection des variables non-initialisées, de quelques types de fuites mémoires, dépassement de tableaux.
Divers outils disponibles via `valgrind --tool=<toolname>` :

- par défaut, `memcheck` (non-allocation, non-initialisation, inaccessibilité), double-free...,
- `massif` : profilage de tas,
- `cachegrind, callgrind` : profilage de cache,
- `helgrind, DRD` : pour les programmes multithreadés.

Analyse dynamique

Analyse du programme en cours d'exécution : vérification de types à la volée (python et beaucoup de langages interprétés), outils de visualisation de l'état du programmes, débogueurs. Pour déboguer un programme C, le compiler avec l'option `-g`.

Valgrind détection des variables non-initialisées, de quelques types de fuites mémoires, dépassement de tableaux.
Divers outils disponibles via `valgrind --tool=<toolname>` :

- par défaut, `memcheck` (non-allocation, non-initialisation, inaccessibilité), double-free...,
- `massif` : profilage de tas,
- `cachegrind, callgrind` : profilage de cache,
- `helgrind, DRD` : pour les programmes multithreadés.

GDB Débogueur par défaut de la suite GNU :

Analyse dynamique

Analyse du programme en cours d'exécution : vérification de types à la volée (python et beaucoup de langages interprétés), outils de visualisation de l'état du programmes, débogueurs. Pour déboguer un programme C, le compiler avec l'option `-g`.

Valgrind détection des variables non-initialisées, de quelques types de fuites mémoires, dépassement de tableaux.
Divers outils disponibles via `valgrind --tool=<toolname>` :

- par défaut, `memcheck` (non-allocation, non-initialisation, inaccessibilité), double-free...,
- `massif` : profilage de tas,
- `cachegrind, callgrind` : profilage de cache,
- `helgrind, DRD` : pour les programmes multithreadés.

GDB Débogueur par défaut de la suite GNU :

- exécution jusqu'à des points d'arrêts,

Analyse dynamique

Analyse du programme en cours d'exécution : vérification de types à la volée (python et beaucoup de langages interprétés), outils de visualisation de l'état du programmes, débogueurs. Pour déboguer un programme C, le compiler avec l'option `-g`.

Valgrind détection des variables non-initialisées, de quelques types de fuites mémoires, dépassement de tableaux.
Divers outils disponibles via `valgrind --tool=<toolname>` :

- par défaut, `memcheck` (non-allocation, non-initialisation, inaccessibilité), double-free...,
- `massif` : profilage de tas,
- `cachegrind, callgrind` : profilage de cache,
- `helgrind, DRD` : pour les programmes multithreadés.

GDB Débogueur par défaut de la suite GNU :

- exécution jusqu'à des points d'arrêts,
- exécution pas à pas,

Analyse dynamique

Analyse du programme en cours d'exécution : vérification de types à la volée (python et beaucoup de langages interprétés), outils de visualisation de l'état du programmes, débogueurs. Pour déboguer un programme C, le compiler avec l'option `-g`.

Valgrind détection des variables non-initialisées, de quelques types de fuites mémoires, dépassement de tableaux.
Divers outils disponibles via `valgrind --tool=<toolname>` :

- par défaut, `memcheck` (non-allocation, non-initialisation, inaccessibilité), double-free...,
- `massif` : profilage de tas,
- `cachegrind, callgrind` : profilage de cache,
- `helgrind, DRD` : pour les programmes multithreadés.

GDB Débogueur par défaut de la suite GNU :

- exécution jusqu'à des points d'arrêts,
- exécution pas à pas,
- visualisation de l'état des variables, de la pile, des registres...

Analyse dynamique

Analyse du programme en cours d'exécution : vérification de types à la volée (python et beaucoup de langages interprétés), outils de visualisation de l'état du programmes, débogueurs. Pour déboguer un programme C, le compiler avec l'option `-g`.

Valgrind détection des variables non-initialisées, de quelques types de fuites mémoires, dépassement de tableaux.
Divers outils disponibles via `valgrind --tool=<toolname>` :

- par défaut, `memcheck` (non-allocation, non-initialisation, inaccessibilité), double-free...,
- `massif` : profilage de tas,
- `cachegrind, callgrind` : profilage de cache,
- `helgrind, DRD` : pour les programmes multithreadés.

GDB Débogueur par défaut de la suite GNU :

- exécution jusqu'à des points d'arrêts,
- exécution pas à pas,
- visualisation de l'état des variables, de la pile, des registres...

Analyse dynamique

Analyse du programme en cours d'exécution : vérification de types à la volée (python et beaucoup de langages interprétés), outils de visualisation de l'état du programmes, débogueurs. Pour déboguer un programme C, le compiler avec l'option `-g`.

Valgrind détection des variables non-initialisées, de quelques types de fuites mémoires, dépassement de tableaux.

Divers outils disponibles via `valgrind --tool=<toolname>` :

- par défaut, `memcheck` (non-allocation, non-initialisation, inaccessibilité), double-free...,
- `massif` : profilage de tas,
- `cachegrind, callgrind` : profilage de cache,
- `helgrind, DRD` : pour les programmes multithreadés.

GDB Débogueur par défaut de la suite GNU :

- exécution jusqu'à des points d'arrêts,
- exécution pas à pas,
- visualisation de l'état des variables, de la pile, des registres...

`gdb` a accès à l'espace mémoire du processus débogué, les "symboles de débogages" sont des instruction d'interruption (`INT 3`) qui lui rendent la main.

Rab de gdb

Interface semi-graphique : `-tui`

Commandes utiles pour lister le code :

- `run` : lance l'exécution
- `ctrl+c` : stoppe l'exécution
- `cont` : continue l'exécution
- `list` : lister le code (à la position courante)
- `list <fct>` : liste le code depuis le début de la fonction `fct`,
- `list <file>:<fct>` : " dans le fichier `fct`
- `list +, list -` : avancer réculer dans le fichier
- `step` : avancer d'un pas
- `next` : avance d'un pas (sans entrer dans les fonctions)
- `finish` : avance jusqu'à la fin de la fonction courante

Rab de gdb

Interface semi-graphique : `-tui`

Commandes utiles pour lister le code :

- `run` : lance l'exécution
- `ctrl+c` : stoppe l'exécution
- `cont` : continue l'exécution
- `list` : lister le code (à la position courante)
- `list <fct>` : liste le code depuis le début de la fonction `fct`,
- `list <file>:<fct>` : " dans le fichier `fct`
- `list +, list -` : avancer réculer dans le fichier
- `step` : avancer d'un pas
- `next` : avance d'un pas (sans entrer dans les fonctions)
- `finish` : avance jusqu'à la fin de la fonction courante

Points d'arrêts (*breakpoint*) :

- `break <fct>` : rajoute un point d'arrêt au début de la fonction `fct`
- `break <n>` : rajoute un point d'arrête à la ligne `n`
- `break <file>:<line>` : possibilités de points d'arrêts conditionnels
- `info breakpoints` : lister les points d'arrêts
- `clear <fct>` : retirer un point d'arrêt
- `delete <nb>` : retirer un point d'arrêt

Rab de gdb

Interface semi-graphique : `-tui`

Commandes utiles pour lister le code :

- `run` : lance l'exécution
- `ctrl+c` : stoppe l'exécution
- `cont` : continue l'exécution
- `list` : lister le code (à la position courante)
- `list <fct>` : liste le code depuis le début de la fonction `fct`,
- `list <file>:<fct>` : " dans le fichier `fct`
- `list +, list -` : avancer réculer dans le fichier
- `step` : avancer d'un pas
- `next` : avance d'un pas (sans entrer dans les fonctions)
- `finish` : avance jusqu'à la fin de la fonction courante

Points d'arrêts (*breakpoint*) :

- `break <fct>` : rajoute un point d'arrêt au début de la fonction `fct`
- `break <n>` : rajoute un point d'arrête à la ligne `n`
- `break <file>:<line>` : possibilités de points d'arrêts conditionnels
- `info breakpoints` : lister les points d'arrêts
- `clear <fct>` : retirer un point d'arrêt
- `delete <nb>` : retirer un point d'arrêt

Variables et *"watchpoints"*

- `print <var>` : affiche la valeur d'une variable ou expression
- `display <var>` : affiche à chaque pas
- `set <var> = <x>` : modifier une variable
- `watch <var>` : arrête le programme quand une variable est modifiée.

Rab de gdb

Interface semi-graphique : `-tui`

Commandes utiles pour lister le code :

- `run` : lance l'exécution
- `ctrl+c` : stoppe l'exécution
- `cont` : continue l'exécution
- `list` : lister le code (à la position courante)
- `list <fct>` : liste le code depuis le début de la fonction `fct`,
- `list <file>:<fct>` : " dans le fichier `fct`
- `list +, list -` : avancer réculer dans le fichier
- `step` : avancer d'un pas
- `next` : avance d'un pas (sans entrer dans les fonctions)
- `finish` : avance jusqu'à la fin de la fonction courante

Points d'arrêts (*breakpoint*) :

- `break <fct>` : rajoute un point d'arrêt au début de la fonction `fct`
- `break <n>` : rajoute un point d'arrête à la ligne `n`
- `break <file>:<line>` : possibilités de points d'arrêts conditionnels
- `info breakpoints` : lister les points d'arrêts
- `clear <fct>` : retirer un point d'arrêt
- `delete <nb>` : retirer un point d'arrêt

Variables et *"watchpoints"*

- `print <var>` : affiche la valeur d'une variable ou expression
- `display <var>` : affiche à chaque pas
- `set <var> = <x>` : modifier une variable
- `watch <var>` : arrête le programme quand une variable est modifiée.

Pile et threads

- `backtrace` : affiche la pile d'appels
- `up, down` : monter ou descendre dans les *frames*
- `frame <num>` : changer de *frame*
- `info threads` : liste les threads
- `thread <num>` : change le thread courant

Rab de gdb

Interface semi-graphique : `-tui`

Commandes utiles pour lister le code :

- `run` : lance l'exécution
- `ctrl+c` : stoppe l'exécution
- `cont` : continue l'exécution
- `list` : lister le code (à la position courante)
- `list <fct>` : liste le code depuis le début de la fonction `fct`,
- `list <file>:<fct>` : " dans le fichier `fct`
- `list +, list -` : avancer réculer dans le fichier
- `step` : avancer d'un pas
- `next` : avance d'un pas (sans entrer dans les fonctions)
- `finish` : avance jusqu'à la fin de la fonction courante

Points d'arrêts (*breakpoint*) :

- `break <fct>` : rajoute un point d'arrêt au début de la fonction `fct`
- `break <n>` : rajoute un point d'arrête à la ligne `n`
- `break <file>:<line>` : possibilités de points d'arrêts conditionnels
- `info breakpoints` : lister les points d'arrêts
- `clear <fct>` : retirer un point d'arrêt
- `delete <nb>` : retirer un point d'arrêt

Variables et *"watchpoints"*

- `print <var>` : affiche la valeur d'une variable ou expression
- `display <var>` : affiche à chaque pas
- `set <var> = <x>` : modifier une variable
- `watch <var>` : arrête le programme quand une variable est modifiée.

Pile et threads

- `backtrace` : affiche la pile d'appels
- `up, down` : monter ou descendre dans les *frames*
- `frame <num>` : changer de *frame*
- `info threads` : liste les threads
- `thread <num>` : change le thread courant

Registres et assembleur

- `info registers` : affiche les registres
- `layout asm` : affiche le code assembleur
- `layout src` : affiche le code source

Rab de gdb

Raccourcis

- `r` : run
- `l` : list
- `c` : continue
- `s` : step
- `n` : next
- `bt` : backtrace
- `i` : info
- `b` : breakpoints
- `i b` : info breakpoints
- …

Interlude récréatif sur les pointeurs en C

```c
/* Pointer Stew
 * Alan Feuer "The C Puzzle Book", 1998 (Addison-Wesley)
 */

#include <stdio.h>

char *c[] = { "ENTER", "NEW", "POINT", "FIRST" };
char **cp[] = { c + 3, c + 2, c + 1, c };
char ***cpp = cp;

int main(void)
{
    printf("%s", ***++cpp);
    printf("%s ", *--*++cpp + 3);
    printf("%s", *cpp[-2] + 3);
    printf("%s\n", cpp[-1][-1] + 1);
    return 0;
}
```

Débogage avancé : fonctions et préprocesseurs

```
{
#pragma GCC diagnostic ignored "-Wsign-compare"
...
#pragma GCC diagnostic warning "-Wsign-compare"
    void *bt[DEBUGMEM_MAXBT];
    int sizebt = backtrace(bt, DEBUGMEM_MAXBT);
    char **strings = backtrace_symbols(bt, sizebt);
    for (int i = 0; i < sizebt; i++)
        fprintf(stderr, "  %s\n", strings[i]);
    exit(1);
}
...
signal(SIGSEGV, (sighandler_t) sigsegv);
signal(SIGBUS, (sighandler_t) sigsegv);
```

Optimisations : options de compilation

Avec `gcc` :

- `-O0` : pas d'optimisation
- `-O1` : optimisations modérées
- `-O2` : pleines optimisations
- `-O3` : optimisations agressives
- `-Os` : optimisation en mémoire (taille de l'exécutable)
- `-march=native` : compile pour le processeur de la machine
- `-ffastmath` : active certaines optimisations sur les flottants (perte du respect de la norme IEEE 754)

Possibilités à explorer : passer des variables en registres, rendre des fonctions *inline*, dérécursiver, déboucler, ...

Les `malloc/free/realloc` et les appels systèmes sont lent, à utiliser avec modération (ie intelligemment).

Optimisations : outils de profilage

Analyse dynamique, encore.

GPROF calcule le temps passé dans chaque fonction, et le graphe d'appel (inconvénient : le code ne doit pas être optimisé, reste une bonne approximation).
Compilation avec l'option `-pg`, exécution normale (plus lente, forcément), regarder le fichier `gmon.out` généré, puis `gprof <exec>`.

GCOV Teste de la "couverture" : pour chaque ligne, nombre de fois qu'elle a été exécutée, compilation avec `-fprofile-arcs -ftest-coverage`, exécuter le code, exécuter `gcov <source>`.

Bonus : analyse et profilage sans code source

Analyse I/O :

- `dstat` : affiche toutes les secondes le traffic I/O (système & réseau)
- `opensnoop` : affiche en temps réels les fichiers ouverts par quel processus
- `strace` : (linux only) affiche tous les appels systèmes effectué par un programme (causant un gros ralentissement de l'exécution du programme), possibilité de filtrage par type d'appel système, d'inspecter les processus fils, d'analyser un programme *en cours d'exécution*, enregistrer pour analyse ultérieure, afficher les noms de fichiers plutôt que les descripteurs ouverts...

Bonus : analyse et profilage sans code source

Analyse I/O :

- `dstat` : affiche toutes les secondes le traffic I/O (système & réseau)
- `opensnoop` : affiche en temps réels les fichiers ouverts par quel processus
- `strace` : (linux only) affiche tous les appels systèmes effectué par un programme (causant un gros ralentissement de l'exécution du programme), possibilité de filtrage par type d'appel système, d'inspecter les processus fils, d'analyser un programme *en cours d'exécution*, enregistrer pour analyse ultérieure, afficher les noms de fichiers plutôt que les descripteurs ouverts...

Analyse rézo :

- `netstat` : (linux only) affiche les connections, tables de routage, interfaces, ...
- `tcpdump` : analyse du traffic TCP, inspection des paquets
- `wireshark` : analyse du traffic tout protocoles, couteau suisse de l'épillage du réseau.

Bonus : analyse et profilage sans code source

Analyse I/O :

- `dstat` : affiche toutes les secondes le traffic I/O (système & réseau)
- `opensnoop` : affiche en temps réels les fichiers ouverts par quel processus
- `strace` : (linux only) affiche tous les appels systèmes effectué par un programme (causant un gros ralentissement de l'exécution du programme), possibilité de filtrage par type d'appel système, d'inspecter les processus fils, d'analyser un programme *en cours d'exécution*, enregistrer pour analyse ultérieure, afficher les noms de fichiers plutôt que les descripteurs ouverts...

Analyse rézo :

- `netstat` : (linux only) affiche les connections, tables de routage, interfaces, ...
- `tcpdump` : analyse du traffic TCP, inspection des paquets
- `wireshark` : analyse du traffic tout protocoles, couteau suisse de l'épillage du réseau.

Profilage :

- `perf` : analyse de performances pour linux, permet de connaître l'utilisation CPU de chaque fonction d'un programme, de visualiser sous forme de *flamegraph*, enregistrer l'exécution d'un programme pour l'analyser plus tard, récupérer la trace d'appels des fonctions (pile), compter le nombre de paquets envoyés sur le réseau...

Librairies habituelles en C

Dans les librairies standards : la `stdlib` (GNU, musl), math, string, time, io, network...
Librairies répandues :

Crypto	: OpenSSL, LibreSSL, libsodium, NaCl
Graphique	: Gtk, nuklear, ncurses (TUI)
Scientifique	: GSL, GMP
HPC	: MPI, OpenMP

Gestionnaire de librairies & dépendances

Inexistant, `makefile`, auto*hell, `clib`

Interlude récratif, le boutisme

Qu'affiche le bout de code suivant ?

```
uint16_t n = 42;
uint8_t *o = (uint8_t *) &n;
size_t i;
for (i = 0; i < sizeof(n); ++i)
printf("%02x ", *(o++));
```

Manipulatios de fichiers

Les appels systèmes utiles :

`OPEN` pour ouvrir un fichier (création d'une entrée dans la table des fichiers ouverts du processus),

`WRITE` pour écrire (des octets) dans le fichier,

`READ` pour lire (des octets) dans le fichier,

`CLOSE` pour le fermer.

Manipulatios de fichiers

Les appels systèmes utiles :

`OPEN` pour ouvrir un fichier (création d'une entrée dans la table des fichiers ouverts du processus),

`WRITE` pour écrire (des octets) dans le fichier,

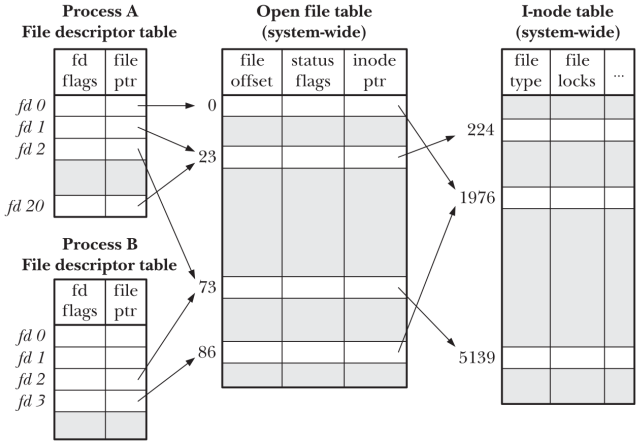
`READ` pour lire (des octets) dans le fichier,

`CLOSE` pour le fermer.

De plus :

- Le système maintient pour chaque processus une *table des fichiers ouverts*.
- À chaque fichier ouvert par le processus est associé un petit entier (son indice dans cette table) appelé *descripteur de fichier*.
- Ce descripteur de fichier est ensuite donné en argument à tous les appels systèmes qui permettent au processus d'interagir avec lui.
- Le système maintient également la position de *la tête de lecture ou écriture* dans le fichier.

Table des fichiers



OPEN

```c
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

`PATHNAME` chemin (relatif ou absolu) dans l'arborescence des fichiers ;

`FLAGS`

- `O_RDONLY` pour lecture seule
- `O_WRONLY` pour écriture seule
- `O_RDWR` pour lecture et écriture (plus rare)

Ces `flags` peuvent être « ou-bit-à-bit-és » avec des attributs de création ou d'états (voir plus loin).

`MODE` dans le cas où il faut créer le fichier, donne les permissions (avant application du masque utilisateur `umask`) associées au fichier à créer.

Exemple : ouverture d'un fichier en lecture

```c
#include <fcntl.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int fd;
    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        perror("open");
        return 1;
    } else {
        printf("fichier %s ouvert avec succès, descripteur %d\n",
                argv[1], fd);
    }
    return 0;
}
```

Exemple : Ouverture d'un fichier en lecture

Toujours tester la valeur de retour d'`open`.

Exemple : Ouverture d'un fichier en lecture

Toujours tester la valeur de retour d'open.

```
$ ./a.out open_exemple1.c
fichier open_exemple1.c ouvert avec succès, descripteur 3
$ ./a.out open_exemple1.c
open: No such file or directory
$ chmod u-r open_exemple1.c
$ ./a.out open_exemple1.c
open: Permission denied
$ ./a.out
open: Bad address
```

Exemple : Ouverture d'un fichier en écriture

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

int main(int argc, char *argv[])
{
    int fd;
    /* O_CREAT : créer le fichier s'il n'existe pas
     * O_TRUNC : si le fichier existe, le vider
     * 0666 : rw-rw-rw- mais enlever les bits de permissions de l'umask */
    fd = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (fd < 0) {
        perror("open");
        return 1;
    } else {
        printf("fichier %s ouvert avec succès, descripteur %d\n",
                argv[1], fd);
    }
    return 0;
}
```

Quelques états pour une ouverture O_WRONLY ou O_RDWR

- `O_TRUNC` : si le fichier existe, le vider
- `O_APPEND` : toutes les écritures se font à la fin du fichier
- `O_CREAT` : si le fichier n'existe pas, le créer (et utiliser le 3ème argument pour les permissions)
- `O_EXCL` : avec `O_CREAT`, si le fichier existe, échouer

READ

```c
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

`SSIZE_T` *signed size type*, un type entier assez grand, signé pour retourner des valeurs négatives (erreurs)

`FD` le descripteur de fichier du fichier dans lequel on lit des octets

`BUF` adresse dans la mémoire du processus où l'on veut copier ces octets

`COUNT` nombre maximal d'octets que l'on veut lire

- Valeur de retour : nombre d'octets effectivement lus ($\leqslant$ `count`)

READ : EXEMPLE

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    int fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        perror("open");
        return 1;
    }
    ssize_t n;
    char buf[4];
    n = read(fd, buf, 4);
    printf("n = %ld\n", n);
    n = read(fd, buf, 4);
    printf("n = %ld\n", n);
    n = read(fd, buf, 4);
    printf("n = %ld\n", n);
    return 0;
}
```

WRITE

```c
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

- `FD` descripteur de fichier du fichier dans lequel on écrit des octets
- `BUF` adresse dans la mémoire du processus du premier octet à écrire dans le fichier
- `COUNT` nombre maximal d'octets à écrire dans le fichier
- Valeur de retour : nombre d'octets effectivement écrits dans le fichier ($\leqslant$ `count`)

WRITE, EXEMPLE PÉDAGOGIQUE

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int fd = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (fd < 0) {
        perror("open");
        return 1;
    }
    ssize_t n;
    char buf[] = "Lol !!\n";
    /* On ne veut pas écrire l'octet nul final dans le fichier. */
    n = write(fd, buf, strlen(buf));
    printf("n = %ld\n", n);
    close(fd);

    return 0;
}
```

READ BLOQUANT

- Dans certaines circonstances, `read` est bloquant : il n'y a pas encore d'octets à lire dans le fichier
- le processus est endormi
- réveillé par le système lorsque des octets sont arrivés.
- C'est le cas pour la lecture sur le terminal ou dans un socket.

Un éditeur de texte

```c
/* super_text_editor.c */
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#define BUFSZ 256
int main(int argc, char *argv[])
{
    int fd = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (fd < 0) {
        perror("open");
        return 1;
    }
    ssize_t n;
    char buf[BUFSZ];
    while ((n = read(0, buf, 256)) > 0) {
        printf("%ld octets écrits dans %s\n", n, argv[1]);
        write(fd, buf, n);
    }
    close(fd);
    return 0;
}
```