Programmation Avancée

Cours 4 : Les erreurs

Simon Forest

18 février 2021

Différents types d'erreurs

On distingue différents types d'erreurs quand on programme :

- erreurs de compilation
- erreurs de logique
- erreurs d'exécution

Erreurs de compilation

Les erreurs de compilation, vous connaissez !

```
int main()
{
  return 0
}
```

```
erreur-compile.c: In function ‘main’:
erreur-compile.c:4:1: error: expected ‘;’ before ‘}’ token
}
^
```

`erreur-compile.c:4:1` indique que l'erreur se trouve à la ligne 4 et au premier caractère.

Erreurs de compilation

On peut avoir des informations intéressantes sur les erreurs dans les notes qui suivent :

```c
#include "point.h"
#include "structures.h"
int main()
{
  point p;
  p.x = 3;
  return 0;
}
```

```
In file included from main.c:2:0:
point.h:3:3: error: conflicting types for ‘point’
 } point;
   ^
```

Erreurs de compilation

On peut avoir des informations intéressantes sur les erreurs dans les notes qui suivent :

```c
#include "point.h"
#include "structures.h"
int main()
{
  point p;
  p.x = 3;
  return 0;
}
```

```
structures.h:3:3: note: previous declaration of ‘point’ was here
 } point;
```

Erreurs de compilation

... en effet, `point` était défini à la fois dans `point.h` et `structures.h` :

```c
// dans point.h
typedef struct{
  int x,y;
} point;
```

```c
// dans structures.h
typedef struct{
  int x,y;
} point;
```

d'où l'erreur.

Erreurs de logique

Les erreurs de logiques n'engendrent pas d'erreurs à la compilation.

À la place, le code ne fait simplement pas ce qui est attendu.

```
int prod(int x, int y)
{
  return x * y;
}

int somme(int x, int y) // erreur de logique par un copier-coller
{
  return x * y;
}
```

Erreurs de logique

Les erreurs de logiques n'engendrent pas d'erreurs à la compilation.

À la place, le code ne fait simplement pas ce qui est attendu.

```
void echange(int *x, int *y)
{
  int temp = *x;
  *y = temp;      // erreur de logique pour l'échange
  temp = *x;
}
```

Erreurs de logique

Les erreurs de logiques n'engendrent pas d'erreurs à la compilation.

À la place, le code ne fait simplement pas ce qui est attendu.

```
void echange(int *x, int *y)
{
  int temp = *x;
  *x = *y;      // bonne version
  *y = temp;
}
```

Tests

Pour repérer les erreurs de logique, il faut **tester** !

Pour cela, le plus propre est de faire des **fonctions de test**. Exemple pour `somme` :

```c
void test_somme()
{
  if(somme(3,4) != 7)
  {
    printf("Erreur dans le calcul de somme(3,4)\n");
  }
  if(somme(5,-5) != 0)
  {
    printf("Erreur dans le calcul de somme(5,-5)\n");
  }
  // ...
}
```

Tests

Pour repérer les erreurs de logique, il faut **tester** !

Une bonne idée est de tester avec de l'aléatoire. Exemple pour `somme` :

```
void test_somme()
{
  for(int i = 0; i < 100; i++)
  {
    int x = rand(), y = rand();
    int r;
    if((r = somme(x,y)) != x + y)
    {
      printf("Erreur dans le calcul de somme(%d,%d) -> %d\n",x,y,r);
    }
  }
}
```

Tests

Pour repérer les erreurs de logique, il faut **tester** !

Ne pas oublier de tester plus spéciquement certains cas quand cela est utile.

```
int inc_mod_1000(int x)
{
  if(x == 999)
  {
    return 0;
  }
  return x+1;
}
```

Tests

Pour repérer les erreurs de logique, il faut **tester** !

Ne pas oublier de tester plus spéciquement certains cas quand cela est utile.

```c
int test_inc_mod_1000()
{
  int r;
  if((r = inc_mod_1000(999)) != 0)
    printf("Erreur avec test_inc_mod(999) -> %d",r);
  // puis tests aléatoires pour les valeurs entre 0 et 998
  for(int i = 0; i < 100; i++)
  {
    int x = rand() % 998;
    // ...
  }
}
```

assert

Écrire des `printf` à la main pour chaque erreur peut être pénible.

À la place, on peut utiliser `assert` (`#include <assert.h>`) pour tester une condition et afficher un message en cas d'erreur.

```c
int main()
{
  int x = 0;
  assert(x == 0); // n'affiche rien car la condition est vraie
  assert(x == 1); // affiche une erreur et arrête le programme
}
```

```
prog: main.c:5: main: Assertion `x == 1' failed.
Abandon (core dumped)
```

assert

Écrire des `printf` à la main pour chaque erreur peut être pénible.

À la place, on peut utiliser `assert` (`#include <assert.h>`) pour tester une condition et afficher un message en cas d'erreur.

```
void test_somme()
{
  for(int i = 0; i < 100; i++)
  {
    int x = rand(), y = rand();
    assert(somme(x,y) == x+y);
    // par contre, on ne voit plus la valeur calculée...
  }
}
```

Programme de test

Une fois que l'on a écrit toutes les fonctions de test, on peut les rassembler dans un programme qui les appelle toutes, autre que le programme principal.

```c
// test.c
int main()
{
  test_somme();
  test_prod();
  test_truc();
  // ...
  return 0;
}
```

Programme de test

Une fois que l'on a écrit toutes les fonctions de test, on peut les rassembler dans un programme qui les appelle toutes, autre que le programme principal.

```
# Makefile
prog: fonctions.c prog.c
	gcc -o prog prog.c

run: prog
	./prog

test: fonctions.c prog.c
	gcc -o test test.c

run-test: test
	./test
```

Erreurs d'exécution

On peut aussi avoir des erreurs que l'on ne peut voir qu'à l'exécution.

L'apparition ou non de ces erreurs dépend de paramètres non connus à la compilation : l'état de la mémoire, le système de fichiers, les entrées utilisateur, *etc.*

Ces erreurs peuvent aussi souvent être vues comme des erreurs de logique.

Exemples

Elles peuvent être dues à des cas non traités jugés improbables.

```c
int main()
{
  int* tab = malloc(10 * sizeof(int));
  tab[3] = 5; // on suppose ici que tab est valide
  // ...
}
```

Si pas assez de mémoire, on risque `tab = NULL` et `Erreur de segmentation`.

Exemples

Elles peuvent être dues à des variables mal initialisées.

```
int somme_tab(int n, int* tab)
{
  int res; // oubli de l'initialisation ici
  for(int i = 0; i < n; i++)
  {
    res += tab[i];
  }
  return res; // le résultat est sûrement mauvais ici
}
```

Exemples

Elles peuvent être dues à des mauvais index de tableaux.

```
int max_tab(int n, int* tab)
{
  int res = INT_MIN;
  for(int i = 1; i <= n; i++)
  {
    res = max(tab[i],res); // mauvaise indexation de tab
  }
  return res;
}
```

Exemples

Elles peuvent être dues à des entrées utilisateur invalides...

```
int main()
{
  int x;
  printf("Veuillez entrer un nombre positif: ");
  scanf("%d",&x);
  // ...
}
```

```
Veuillez rentrer un nombre positif: cheval
```

Exemples

... en particulier, des entrées qui ne respectent pas les spécifications du programme.

```c
int main()
{
  char tab[100];
  printf("Veuillez entrer le nom d'un fichier existant : ");
  scanf("%s",tab);
  FILE* file = fopen(tab,"r");
  int c = fgetc(file);
  // ...
}
```

```
Veuillez entrer le nom d'un fichier existant : ExistePas.txt
Erreur de segmentation (core dumped)
```

assert

Pour gérer sans effort la plupart des erreurs d'exécution, on peut utiliser des `assert` :

```
int* tab = malloc(10 * sizeof(int));
assert(tab != NULL);

for(int i = ??; i < ??; i++)
{
  assert(0 <= i && i < 10);
  tab[i] = i;
}

FILE* file = fopen("MonFichier.txt","r");
assert(file != NULL);

// ...
```

`assert`

Pour gérer sans effort la plupart des erreurs d'exécution, on peut utiliser des `assert` :

On obtiendra alors des erreurs avec indication de ligne dans le code source :

```
prog: fichier.c:123: fonction: Assertion `...' failed.
Abandon (core dumped)
```

assert

Pour gérer sans effort la plupart des erreurs d'exécution, on peut utiliser des `assert` :

On obtiendra alors des erreurs avec indication de ligne dans le code source :

```
prog: fichier.c:123: fonction: Assertion `...' failed.
Abandon (core dumped)
```

C'est déjà bien, mais on souhaite souvent avoir plus d'informations :

```
FILE* file = fopen("MonFichier.txt","r");
assert(file != NULL);
```

```
prog: fichier.c:123: fonction: Assertion `file != NULL' failed.
Abandon (core dumped)
```

Mais pourquoi l'`assert` a échoué ici ?

`errno`

Pour les fonctions de la librairie standard, il existe un méchanisme commun pour rapporter les erreurs.

Lors d'une erreur, une valeur absurde est renvoyée :

- `NULL` pour un pointeur (`malloc` , `fopen` , *etc.*)
- une valeur négative pour un entier (`scanf` , `fgetc` , *etc.*)

On peut alors avoir une information plus précise sur l'erreur en utilisant la constante `errno` .

Valeur d'`errno`

```c
#include <errno.h>
// ...
fopen("Existe.txt","r");
printf("errno = %d\n", errno);

fopen("ExistePas.txt","r");
printf("errno = %d\n", errno);

fopen("PasAutorisé.txt","r");
printf("errno = %d\n", errno);
```

```
errno = 0
errno = 2
errno = 13
```

Une valeur d'`errno` est affectée pour chaque type d'erreur.

perror

Pour afficher un message d'erreur plus compréhensible, on peut utiliser `perror` :

```c
#include <errno.h>
// ...
fopen("Existe.txt","r");
perror(NULL);

fopen("ExistePas.txt","r");
perror(NULL);

fopen("PasAutorisé.txt","r");
perror(NULL);
```

```
Success
No such file or directory
Permission denied
```

perror

Pour afficher un message d'erreur plus compréhensible, on peut utiliser `perror` :

```
#include <errno.h>
// ...
fopen("Existe.txt","r");
perror("Ouverture du premier");

fopen("ExistePas.txt","r");
perror("Ouverture du deuxième");

fopen("PasAutorisé.txt","r");
perror("Ouverture du troisième");
```

```
Ouverture du premier: Success
Ouverture du deuxième: No such file or directory
Ouverture du troisième: Permission denied
```

`err` / `errx`

On peut avoir des comportements en cas d'erreur plus complexes.

Avec `err`, on quitte avec un code d'erreur en affichant l'erreur d' `errno` et un message formaté :

```c
#include <err.h>
FILE* file = fopen("ExistePas.txt","r");
if(file == NULL)
  err(42,"Problème avec l'ouverture de %s", "ExistePas.txt");
```

```
prog: Problème avec l'ouverture de ExistePas.txt:
No such file or directory
Le programme s'est terminé avec le code 42.
```

`err` / `errx`

On peut avoir des comportements en cas d'erreur plus complexes.

Avec `errx`, on quitte avec un code d'erreur et un message formaté **sans afficher** l'erreur d'`errno` :

```c
#include <err.h>
FILE* file = fopen("ExistePas.txt","r");
if(file == NULL)
  errx(42,"Problème avec l'ouverture de %s", "ExistePas.txt");
```

```
prog: Problème avec l'ouverture de ExistePas.txt
Le programme s'est terminé avec le code 42.
```

`warn` / `warnx`

On peut aussi simplement afficher un avertissement sans quitter le programme.

Avec `warn`, on affiche un message formaté avec l'erreur d'`errno` :

```c
#include <err.h>
FILE* file = fopen("ExistePas.txt","r");
if(file == NULL)
  warn("Avertissement pour le fichier %s", "ExistePas.txt");
```

```
prog: Avertissement pour le fichier ExistePas.txt:
No such file or directory
```

`warn` / `warnx`

On peut aussi simplement afficher un avertissement sans quitter le programme.

Avec `warnx`, on affiche un message formaté sans l'erreur d'`errno` :

```c
#include <err.h>
FILE* file = fopen("ExistePas.txt","r");
if(file == NULL)
  warnx("Avertissement pour le fichier %s", "ExistePas.txt");
```

```
prog: Avertissement pour le fichier ExistePas.txt
```

error_at_line

On peut aussi vouloir préciser où se trouve l'erreur dans les sources. On peut utiliser pour cela les macros spéciales :

- `__FILE__` : vaut le fichier courant là où elle est utilisée,
- `__LINE__` : vaut la ligne courante là où elle est utilisée.

On peut alors faire un message d'erreur avec `error_at_line` :

```
#include <errno.h>
#include <error.h>
FILE* file = fopen("ExistePas.txt","r");
if(file == NULL)
  error_at_line(42, errno, __FILE__, __LINE__,
                    "Erreur avec le fichier %s", "ExistePas.txt");
```

```
./prog:code.c:11: Erreur avec le fichier ExistePas.txt:
No such file or directory
Le programme s'est terminé avec le code 42.
```

Erreurs de segmentation

Malgré l'utilisation de `assert`, `err`, `warn`, *etc.*, on peut passer à côté d'erreurs d'exécution qui induisent des erreurs de segmentation.

Les erreurs de segmentation arrêtent immédiatement le programme sans donner plus d'informations.

Cela rend potentiellement difficile la localisation de ces erreurs.

La méthode du `printf`

Étant donné un programme qui `segfault` ...

```c
int main()
{
  f();
  g();
  for(int i = 0; i < 10; i++)
    h(i);
  return 0;
}
```

```
Erreur de segmentation (core dumped)
```

La méthode du `printf`

... on peut localiser l'erreur à coups de `printf` :

```
int main()
{
  f();
  printf("f() OK\n");
  g();
  printf("g() OK\n");
  for(int i = 0; i < 10; i++)
  {
    h(i);
    printf("h(%d) OK\n",i);
  }
  return 0;
}
```

La méthode du `printf`

... on peut localiser l'erreur à coups de `printf` :

```
f() OK
g() OK
h(0) OK
h(1) OK
h(2) OK
Erreur de segmentation (core dumped)
```

Ici, le problème est sûrement dans le calcul de `h(3)`.

Attention avec `printf`

`printf` affiche sur la sortie standard, appelée `stdout`.

L'affichage sur cette sortie ne se fait pas directement mais s'accumule sur un tampon. Le tampon est affiché quand il est plein ou qu'un caractère `'\n'` est rencontré.

Cela peut poser problème quand on débogue.

Exemple

```c
int main()
{
  fonctionQuiPlantePas();
  printf("fonctionQuiPlantePas OK");
  fonctionQuiPlante();
  return 0;
}
```

affiche directement

```
Erreur de segmentation (core dumped)
```

et s'arrête, sans afficher `fonctionQuiPlantePas OK` avec.

Solution

Solution 1 : toujours terminer les `printf` de débogage par `'\n'`.

Solution 2 : utiliser `fflush` après un `printf` pour forcer l'affichage du tampon.

```
f();
printf("f OK");
fflush(stdout); // affichage du message ici
g();
```

Solution 3 : utiliser la sortie spécifique au débogage et aux erreurs, c-à-d `stderr`.

stderr

Au lancement d'un programme, on dispose de 3 `FILE*` déjà ouverts :

- `stdin`, ouvert en lecture
- `stdout` et `stderr`, ouverts en écriture.

`stdin` correspond à l'entrée du terminal. C'est ce qui est utilisé quand on utilise `scanf`, `getchar`, *etc.*

```c
int x;
scanf("%d",&x);
// équivalent à:
fscanf(stdin,"%d",&x);

int y = getchar();
// équivalent à:
int y = fgetc(stdin);
```

stderr

Au lancement d'un programme, on dispose de 3 `FILE*` déjà ouverts :

- `stdin`, ouvert en lecture
- `stdout` et `stderr`, ouverts en écriture.

`stdout` correspond à la sortie normale du terminal. C'est ce qui est utilisé quand on utilise `printf`, `putchar`, *etc.*

```
printf("%s!!!","Bonjour");
// équivalent à:
fprintf(stdout,"%s!!!","Bonjour");

putchar(42);
// équivalent à:
fputc(42,stdin);
```

`stdout` est **bufferisée**, ce qui explique pourquoi `printf` l'est aussi.

stderr

Au lancement d'un programme, on dispose de 3 `FILE*` déjà ouverts :

- `stdin`, ouvert en lecture
- `stdout` et `stderr`, ouverts en écriture.

`stderr` correspond à la sortie pour le débogage, les avertissements ou les erreurs. On peut l'utiliser avec `fprintf(stderr,...)`.

```
fprintf(stderr,"Avertissememt: option --max manquante");
// ...
fprintf(stderr,"Oups, g(3) a renvoyé %d", 5);
```

Contrairement à `stdout`, `stderr` **n'est pas bufferisée**. Le contenu de `fprintf` s'affiche directement sans `'\n'` ni `fflush`.

À préférer pour le débogage.

Problème

Trouver une erreur d'exécution à coup de `fprintf(stderr,...)` fonctionne mais peut être fastidieux.

```
f();
fprintf(stderr,"f OK\n");
g();
fprintf(stderr,"g OK\n");
h();
fprintf(stderr,"h OK\n");
// ...
```

On préfère habituellement utiliser un débogueur pour trouver les `Erreurs de segmentation`.

Débogueur

Un **débogueur** est un programme qui permet d'inspecter l'exécution d'un programme. Il permet de voir en particulier la cause d'une `Erreur de segmentation`.

Pour les programmes en `C`, le débogueur standard est `gdb`.

Exemple

Un code en C qui produit une erreur :

```c
void f(int x)
{
  int r = rand();
  int* ptr = NULL;
  *ptr = 42;
}
void g()
{
  f(21);
}
int main()
{
  g();
}
```

Exemple

On le compile et on le lance avec `gdb` pour voir ce qu'il se passe :

```
gcc -g prog.c -o prog
gdb prog
```

Noter l'option `-g` à mettre quand on utilise `gdb`.

On obtient une nouvelle invite :

```
(gdb)
```

qui permet d'interagir avec `gdb`.

Exemple

On lance alors le programme avec la commande `run` :

```
(gdb) run
```

Et on obtient alors :

```
Starting program: /dossier/prog

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400545 in f (x=21) at prog.c:9
9         *ptr = 42;
```

On voit ainsi la position dans le code qui a déclenché l'erreur : `at prog.c:9` .

Exemple

On peut voir la pile des fonctions appelées avec `backtrace` (ou juste `bt`) :

```
(gdb) bt
#0  0x0000000000400545 in f (x=21) at prog.c:9
#1  0x000000000040055c in g () at prog.c:13
#2  0x000000000040056d in main () at prog.c:17
```

On voit de plus où ces fonctions ont été appelées.

Exemple

On peut afficher la valeur des variables locales avec `print` (ou juste `p`) :

```
(gdb) p x
$1 = 21
(gdb) p r
$2 = 1804289383
(gdb) p ptr
$3 = (int *) 0x0
```

Exemple

On peut quitter le débogueur avec `quit` (ou juste `q`) :

```
(gdb) q
```

D'autres fonctionnalités

- l'exécution pas à pas du programme
- l'arrêt de l'exécution à des endroits définis
- la surveillance de la modification de variables

On peut consulter ces fonctionnalités avec

```
(gdb) help
```

Se référer au TP6 pour les détails.