# חריגות ובדיקות

## טיפול בשגיאות

השגיאות העלולות לקרות בתוכנית מתחלקות לשני סוגים עיקריים:

1. שגיאות זמן ריצה המתגלות במהלך ריצה תקינה של התוכנית, למשל כתוצאה מקלט לא תקין של המשתמש. שגיאות אלו נקראות "חריגות". כדי לטפל בחריגות משתמשים בשלושה מילים שמורות: throw, try, catch.
2. שגיאות קומפילציה הנובעות מטעות של המתכנת. נקראות גם "באגים".

## זריקת חריגה

כאשר אנו כותבים קוד שבו אנו מזהים כי יכולה להיות חריגה כלשהי, נשתמש במילה השמורה **throw** כדי "לזרוק" את החריגה. בפונקציה בה יכולים להיות מספר סוגים של חריגות ננסח זריקה שונה לכל סוג של חריגה. בשפת c++ יכולים לזרוק מה שרוצים, כמו טקסט שיסביר את הבעיה למשתמש, המשתנה או אובייקט שיצר את החריגה. אך מקובל לזרוק אובייקט מתאים מתוך הספרייה הסטנדרטית לחריגות כמו שנלמד בהמשך.

double division(int a, int b) {

if( b == 0 )

throw "Division by zero condition!";

return (a/b);

}

## טיפול בחריגה

כאשר ישנו קטע קוד קריטי היכול לזרוק חריגה, נשתמש במילים השמורות try ו-catch כדי לתפוס את החריגה ולטפל בה. הטיפול בחריגות מאפשר העברת שליטה מחלק אחד של התוכנית אל חלק אחר, בו בנינו תוכנית שמטרתה להתמודד עם שגיאות אלו. בC++ כל סוגי החריגות הן unchecked, כלומר הקומפיילר לא מוודא בזמן קומפילציה שחריגות אלו מטופלות (בניגוד ל-java שם חלק מהחריגות בדוקות).

### try

זוהי מילה שמורה שלאחריה פותחים בלוק שבתוכו נמקם את שורות הקוד שיכולות לעורר ולזרוק חריגה. החריגה יכולה להיזרק או מתוך פונקציה שהקריאה להפעלתה ממוקמת בתוך אותו בלוק, או על ידי הפקודה throw שמופעלת בתוך אותו בלוק. המשמעות של פעולה זו היא שהקוד בתוך בלוק זה הוא מוגן. מערכת ההפעלה תפעיל קוד זה, וכאשר תזהה כי יש חריגה, מיד תעבור לבלוק של catch המתאים התופס את החריגה שנזרקה ויתמודד איתה. לאחר שנזרקת חריגה וטיפלנו בה לא ניתן לחזור למקום שממנו נזרקה החריגה בבלוק ה-try.

### catch

לאחר בלוק ה-try נכתוב בלוק catch שתופס חריגה מסוימת שנזרקה מבלוק ה-try. ניתן לכתוב כמה בלוקים של catch עבור כל סוג חריגה שייזרק מבלוק ה-try, כך שכל בלוק תופס חריגה שונה. מיד לאחר המילה catch, בסוגריים נרשום מהו טיפוס החריגה שאנו תופסים בהתאם לטיפוס שזרקנו וניתן שם לחריגה שנתפסה, באופן דומה לסוגריים בהצהרה של פונקציה. לאחר כתיבת הפרמטר נפתח בלוק ובו נכתוב מה אנחנו רוצים שהתוכנית תעשה כדי להתמודד עם החריגה שנזרקה מבלוק ה-try. ניתן לבצע פעולות על אובייקט החריגה באמצעות השם שנתנו לפרמטר.

int main () {

int x = 50, y=0;

double z = 0;

try {

z = division(x, y);

} catch (const char\* msg) {

cerr << msg << endl;

}

}

אחרי שבלוק catch מתבצע, התוכנית ממשיכה להתבצע מהנקודה שאחרי בלוק ה-catch, לא ניתן לחזור אל המקום שבו נוצרה החריגה בבלוק ה-try ולהמשיך משם.

אם חריגה שנזרקה לא מטופלת באף בלוק catch, אזי בדרך כלל התוכנית תסתיים עם הודעת שגיאה. ניתן גם לכתוב בלוק catch שמתמודד עם כל סוג של חריגה, לשם כך נכתוב בסוגריים שלוש נקודות.

catch(...) {

// code to handle any exception

}

### עקרון הכרז או טפל

כאשר תופסים חריגה ניתן לטפל בה בשני דרכים:

1. **להכריז** - לעצור את התוכנית ולזרוק את אובייקט החריגה המציין מה הייתה הבעיה, באיזה קובץ, ובאיזה שורה. דרך זו בעצם אומרת - זיהיתי שיש כאן בעיה אך זה לא האחריות שלי לטפל בה, אלא אני רק מודיע עליה, ומי שמקבל את ההודעה יחליט מה לעשות עם מידע זה.
2. **לטפל** - לא לעצור את התוכנית, אלא לטפל בחריגה. הטיפול בחריגה יכול להיות לנסות לבצע אותה פעולה שוב, או לבצע פעולות אחרות שמונעות את הבעיה. דרך זו אומרת - זיהיתי את הבעיה ואני אחראי לטפל בה כך שלא תפגע בתקינות של התוכנית.

## הספרייה הסטנדרטית לחריגות

הדרך המקובלת לזרוק חריגות היא זריקת אובייקטים מתוך הספריות הסטנדרטית <exception>, שצריך לכלול אותה בתחילת הקוד. כל אובייקט שמוגדר בספרייה זו מייצג סוג חריגה שונה. ניתן לקרוא על הסוגים כאן <http://www.cplusplus.com/reference/exception/exception/>. בזריקת החריגה נוכל להוסיף בסוגריים טקסט משלנו המסביר באופן מפורט יותר מדוע נזרקה החריגה.

throw std::ex\_type{"exception message"};

בתפיסת החריגה נרשום את השורה למטה, כאשר ex\_type הוא סוג החריגה, ו-ex\_name הוא שם החריגה שבחרנו ודרכו נוכל לגשת לאובייקט.

catch (const std::ex\_type& ex\_name) {}

כדי לתפוס כל סוג חריגה מספרייה exception פשוט נתפוס את החריגה: std::exception& ex.

בספרייה זו לכל אובייקט מוגדרת המתודה what() שמחזירה הסבר על החריגה. בדרך כלל נשתמש בה כאשר נרצה להודיע למשתמש על סיבת החריגה. דוגמא לזריקה ותפיסת חריגה מהספרייה הסטנדרטית:

include <exception>>#

double division(int a, int b) {

if( b == 0 )

throw std::out\_of\_range{"x must be positive"};

return (a/b);

}

int main () {

int x = 50, y = 0, z = 0;

try {

z = division(x, y);

} catch (const std::out\_of\_range& ex) {

cout << "caught exception: " << ex.what() << endl;

}

}

### יצירת מחלקת שגיאות

ניתן גם ליצור חריגה משלנו על ידי ירושה ודריסה ממחלקת exception. לדוגמא:

#include <exception>

struct MyException : public exception {

const char \* what () const throw () {

return "C++ Exception";

}

};

int main() {

try {

throw MyException();

} catch(MyException& ex) {

std::cout << "MyException caught" << ex.what() << std::endl;

}

}

## באגים

לא אמורות להתקיים בתוכנה תקינה, אבל תוך כדי פיתוח הן עלולות להופיע. כדי לתפוס אותן בצורה נוחה, משתמשים ב-assert. למשל, נניח שיש לנו פונקציה שאמורה להחזיר ערך חיובי, אבל משום מה היא מחזירה לפעמים ערך שלילי. כדי לתפוס את השגיאה ברגע שהיא מתרחשת, אפשר לשים פקודת assert בסוף הפונקציה, למשל:

assert (result>=0, "should return positive value");

נוכל להשתמש בפונקציית assert כדי לוודא שתנאי זה מתקיים, ובמקרה שאינו מתקיים הפונקציה תחסל את המשכיות התוכנית ותחזיר הודעה מתאימה ומיקום הטעות. לא שולחים לתוך assert פונקציה אלא רק משתנים. כדי להשתמש ב-assert יש להוסיף את הספרייה <cassert>.

המוטיבציה להשתמש ב-assert ולא בתנאי רגיל היא מכמה תנאים:

1. פונקציה זו מחזירה הודעה מתאימה כך שאנו יודעים בדיוק היכן הייתה הטעות.
2. מסיימת את התהליך וחוסכת מאתנו בדיקה בכל פעם שנפעיל פעולה זו האם היא הצליחה.
3. קוד קצר וברור יותר.

בניגוד לחריגות, באגים לא אמורים להתקיים בתוכנה הסופית, ולכן אנחנו לא רוצים לבזבז זמן ריצה על הבדיקות הללו לאחר שסיימנו לתקן את הבאגים. אפשר בבת אחת לבטל את כל הבדיקות מסוג assert על ידי הגדרת משתנה קומפילציה בשם NDEBUG. הדרך הנוחה ביותר להגדיר אותו היא על ידי פרמטר לקומפיילר, אמנם ניתן לעשות זאת גם על ידי #define :

clang++-9 --std=c++17 -DNDEBUG assert.cpp

#define NDEBUG

### static\_assert

זהה ל-assert רגיל, אלא שמיועד לבדוק האם טיפוס שהוכנס לתבנית (template) פונקציה או מחלקה הוא חוקי. הבדיקה מתבצעת בזמן קומפילציה שכן התבנית מוגדרת בזמן קומפילציה.

template<typename T> T add(T a, T b) {

static\_assert(is\_numeric<T>::value, "Can only be numeric type");

return a + b;

}

## שגיאות נפוצות

### קומפילציה

בשגיאות קומפילציה מופיע שם הקובץ ולאחר מכן מספר השורה והאות שבה התרחשה השגיאה.

* ambiguous - כאשר יש שני פונקציות עם אותו שם, והקומפיילר לא יודע לאיזה מהן להפנות. לדוגמה, כאשר הצהרנו על פונקציה, שקיימת במרחב שם, שעליו עשינו using namespace.
* non const static data member must be initialized out of line - כאשר מנסים לאתחל משתנה סטטי, שאינו const, בתוך הצהרת המחלקה. משתנה סטטי תמיד מאותחל מחוץ למחלקה.
* lvalue/rvalue - שגיאות שונות שבהם יכללו מילים אלו, הן שגיאות שמודיעות על כך שמנסים להכניס לתוך מצביע או רפרנס, שאינם const, משתנה זמני שאין להם מקום קבוע בזיכרון.
* no viable overloaded - כאשר אנו משתמשים באופרטור שאינו מוגדר במחלקה מסוג האובייקט עליו הפעלנו את האופרטור. יש לעשות העמסה עבור האופרטור שמצוין בשגיאה.
* no viable conversion - כאשר מבצעים השמה בין אובייקטים מטיפוסים שונים שלא מוגדר ביניהם המרה חוקית, כמו בנאי המרה או אופרטורי המרה.
* instantiation of function template specialization - כאשר יש שגיאה בתוך תבנית שהגדרנו. ההפניה למיקום השגיאה יהיה כאשר השתמשנו בתבנית לראשונה ולא במיקום התבנית עצמה.
* Couldn't infer template argument - יש פרמטר-סוג בתבנית שלא נתנו לו ערך, ועל כן הקומפיילר לא יכול להסיק מה הסוג שלו.
* member function … not viable – כאשר מנסים להפעיל על אובייקט const שיטה שאינה מוגדרת const.

### חיבור

בדרך כלל שגיאות אלו מופיע שם הקובץ ולאחר מכן מספר הבית שבו התרחשה השגיאה (text+0x..). בשלב החיבור כבר אין שורות, ולכן השגיאה היא לפי מספר הבית בקובץ בו התרחשה השגיאה. בדרך כלל גם יודיע "error: linker".

* undefined reference to .. - כאשר אנו קוראים לפונקציה שהצהרנו עליה אך לא מימשנו.
* multiple definition of .. - כאשר מימשנו את אותה הפונקציה ביותר מקובץ אחד.

### ריצה

בשגיאות אלו כבר אין הפניות למיקום השגיאה בקובץ, אלא הודעה כללית מה השגיאה. אם הרצנו דרך ה-makefile יראה לנו זאת, שכן השגיאה הייתה באחת הפקודות שלו. לרוב שגיאות זמן ריצה יסתיימו ב-terminated (core dump), המשמעות היא שמערכת ההפעלה זורקת את כל מצב המשתנים אל קובץ שנקרא core כדי שנוכל לדבג זאת.

* terminate called after throwing an instance of … - נזרקה חריגה שלא טיפלנו בה, כלומר לא הקפנו ב-try ו-catch. מודיע לנו איזה אובייקט חריגה נזרק.
* assert - כאשר חייבנו ב-assert תנאי כלשהו שלא מתקיים. תיזרק שגיאת זמן ריצה, שמראה באיזה שורה ועבור איזה תנאי נזרקה השגיאה.
* segmentation fault - גישה למיקום בזיכרון שלא הוקצה לנו. בדרך כלל קורה כאשר מקדמים חורגים מגבולות המערך, מקדמים איברים בצורה שגויה במיכל לא רציף בזיכרון, ניגשים לאובייקט שכבר נמחק, וכו'.
* double free - מחיקת אובייקט פעמיים. לא ניתן לשחרר שוב זיכרון שכבר שוחרר.
* Illegal instruction – מספר אפשרויות. אחת מהם היא כתיבה לכתובת לא חוקית.

### שגיאה לוגית

אלו הן שגיאות שלא נקבל עליהן הודעת שגיאה כלל, אלא נקבל פלט של התוכנית, אך הוא יהיה שגוי. כלומר התוכנית לא תעבוד כמו שהיא אמורה לעבוד. לדוגמה: אם נסכום איברים אל משתנה בשם sum בלי שאתחלנו את sum, נוכל לקבל שגיאה לוגית. שכן sum יכול להיות שקיבל ערך זבל שאנו מחברים אליו ערכים.

# בדיקות - Doctest

בקורס זה נעבוד עם פורמט בדיקות יחידה הנקרא Doctest. ישנו קובץ doctest.h שבו מוגדרות כל הפונקציות שהוא משתמש בהם.

### יצירת בדיקות

כאשר נרצה לבצע בדיקות יש לייבא את doctest.h ואת הקובץ שאותו רוצים לבדוק. נגדיר בלוק בשם TEST\_CASE(), שבסוגריים נרשום מה אנחנו רוצים לבדוק בבלוק זה. בתוך הבלוק נשתמש בפונקציות שונות, כמו CHECK ו-CHECK\_THROWS, כדי לבצע בדיקות מסוגים שונים.

#include "doctest.h"

#include "FileToCheck.hpp"

TEST\_CASE("What to check") {

CHECK(factorial(0) == 1);

CHECK(factorial(1) == 1);

…

}

### דיווח הבדיקות

בכל פרויקט יהיה קובץ בדיקות אחד מרכזי, בדרך כלל בשם TestMain.cpp, שיפעיל את כל הקבצים שבהם מוגדר בלוק TEST\_CASE, ויספק דיווח על הבדיקות לפי הפילטר שהוכנס בו. יש דרך סטנדרטית להגדיר את הקובץ בדיקות המרכזי. הפילטר יכול להיות מסוגים שונים כמו:

* Counter - שסופר את מספר הבדיקות שנמצאו.
* Grader - שבודק כמה בדיקות הצליחו וכמה נכשלו מתוך כל הבדיקות.