Exception handling

מצבים חריגים של תוכנית

כאשר אנו כותבים קוד, אנחנו מתארים אלגוריתם לביצוע מטלה מסוימת. לפעמים, בזמן הריצה של האלגוריתם מתעוררות בעיות הדורשות טיפול מיוחד. לבעיות אלה ניקרא מיקרים או מצבים חריגים. בזמן כתיבת הקוד אין אפשרות לדעת אם המקרה החריג יתרחש או לא. אנו רוצים להתייחס לאפשרות שהוא יקרא ולטפל בו. הטיפול במקרים חריגים הוא דבר הכרחי על מנת לכתוב תוכניות **יציבות**.

דוגמאות למצבים חריגים:

- התוכנית דורשת הקצעת זיכרון דינמית שאיננה יכולה להיות מסופקת ע"י מערכת ההפעלה.
- התוכנית נדרשת לחלוקה ב – 0.
- התוכנית מנסה לפתוח קובץ שאיננו קיים במערכת הקבצים.
- שימוש אסור באובייקט (בדרך כלל במבנה נתונים, כמו פניה לאבר במערך החורגת מהתחום המותר)

לרוב, אין אפשרות לטפל בבעיות כאלה במקום בו הן התגלו ויש צורך לידע חלקים אחרים של התוכנית בבעיה .
לדוגמה:

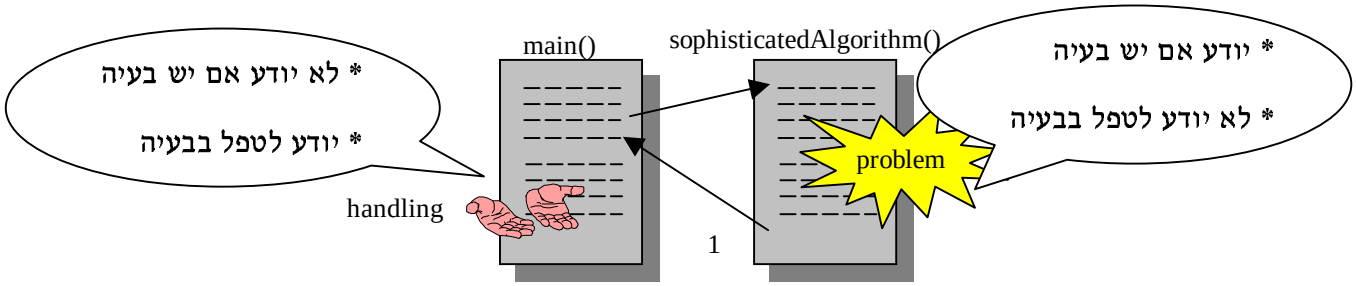
```
void sophisticatedAlgorithm (char* name) {
        std::ifstream in(name);

        // using the file for an algorithm
        // …
}

int main() {
        char name[100];
        std::cin>>name;
        sophisticatedAlgorithm (name);
        // …
}
```

בדוגמה זו יש שימוש באלגוריתם הדורש שימוש בקובץ. יכל להיווצר מצב בו אין אפשרות לפתוח את הקובץ. המקרה הזה יחשב מקרה חריג. נניח כי במקרה חריג זה, אנחנו רוצים לוותר על האלגוריתם ולבצע פעולות אחרות (כמו להפעיל אלגוריתם חלופי העובד עם זיכרון המחשב) . יש כאן שני חלקים של קוד, **האחד שבו מתגלה הבעיה** (sophisticatedAlgorithm) **וחלק אחר שאמור לטפל** בבעיה כאשר היא מתגלה (main).

אנחנו ניצבים בפני הבעיה הבאה:

הקוד של ה - main איננו יודע אם יקרה מצב חריג, אבל הוא יודע מה יש לעשות במקרה שהוא יקרה.

הקוד של ה – sophisticatedAlgorithm() יודע אם קורה מצב חריג אבל איננו יודע איך לטפל בו.



שיטות מוכרות לטיפול במצבים חריגים

ישנן מספר שיטות פשוטות לטפל בבעיות כאלה. האחת היא להחזיר מהפונקציה ערך המסמל כי ארע מקרה חריג:

```
int sophisticatedAlgorithm (char* name) {
        std::ifstream in(name);
        if(!in)
                return –1;
        // …
        return 0; // indicate a normal termination of the function
}
int main() {
        …
        if(sophisticatedAlgorithm(name) == -1) {
                // the exceptional case
        }
        else {
                // the normal case
        }
}
```

מה החסרונות של פתרון כזה?

- לעיתים קרובות הפונקציה שבה עלולה להיווצר הבעיה מחזירה ערך משמעותי אחר (במקרה זה, תוצאת חישוב של האלגוריתם). אם סט הערכים המשמעותיים שיכולים לחזור מפונקציה הוא מוגבל, אפשר ליחד ערכים מסוימים לקודים של שגיאה אבל במקרה של פונקציה כמו tan(), כל ערך ממשי עשוי לחזור ואי אפשר לייחד אף ערך לקוד שגיאה .

- לאחר כל קריאה לפונקציה בה עלולה להיווצר הבעיה, יש לבדוק אם הבעיה התעוררה. בדיקות אלה יכולות להגדיל מאד את הקוד ולגרום לו להיות פחות בהיר.

- אי אפשר להחזיר ערך המסמל קוד שגיאה מ – ctor ו – dtor .

- לעיתים הפונקציה אמורה להחזיר אובייקט, ואז צריך לבנות אובייקט מיוחד המסמל מצב חריג. אם מדובר באובייקט כללי (כאשר הפונקציה היא template) זה מחייב אותנו להניח הנחות מגבילות על הטיפוס.

פתרון אפשרי אחר הוא אחזקה של משתנה גלובאלי המחזיק את קוד השגיאה.

חסרונות הפתרון הזה:

- לא אסתטי מבחינת הנדסת תוכנה. לא משתלב יפה עם עקרונות כמו אנקפסולציה ותכנות מובנה.
- דורש מקום בזיכרון גם במקרים של הרצות תקינות.
- גם פתרון זה דורש בדיקה לאחר כל קריאה לפונקציה בעייתית.

מנגנון ה – exception handling ב – C++

ב – C++ יש מנגנון מיוחד המיועד לטיפול במצבים חריגים. המנגנון מבוסס על 'זריקה' ו 'תפיסה' של אובייקטים המייצגים חריגות. בכל מקום בתוכנית בו מתגלה בעיה אנו 'זורקים' אובייקט חריגה. בכל מקום בקוד בו מעונינים לטפל בבעיה אנחנו 'תופסים' אובייקט המייצג את הבעיה. הקוד הבא הוא דוגמה פשוטה לשימוש במנגנון:

```
class MyException {};

int main() {
        try {
                //…
                if (…) { // oops, we found a problem
                        MyException exp;
                        throw exp;
                }
                //…
        } catch (MyException) {
                // handling the problem
        }
}
```

ראשית הצהרנו על מחלקה של אובייקטים המייצגים מקרה חריג מסוים. הבלוק ב – main המתחיל במילה - try הוא קוד שאנו חוששים שבזמן ההרצה שלו, עלולה להתעורר בעיה. הקוד החשוד כבעייתי מתחיל במילה try כיוון שאנו **מנסים** לבצע אותו בצורה תקינה. אם באמת לא התעוררה שום בעיה, בלוק ה – try יסתיים, הקוד המתחיל ב – catch, לא יתבצע והתוכנית תמשיך משם. אם התעוררה בעיה, ונזרק אובייקט המצביע על מקרה חריג, בלוק ה – try לא יסתיים ותתבצע קפיצה לאזור ה – catch. כיוון שהאובייקט הניזרק הוא מאותו טיפוס אותו מנסה לתפוס ה – catch, יתבצע הקוד בבלוק של ה – catch. בקוד כזה, ברור היכן האזור הבעייתי והיכן מקום הטיפול בבעיה.

הבעיה עלולה להתעורר גם במהלך ביצוע אחת מהפונקציות הנקראות מתוך בלוק ה – try. לדוגמה:

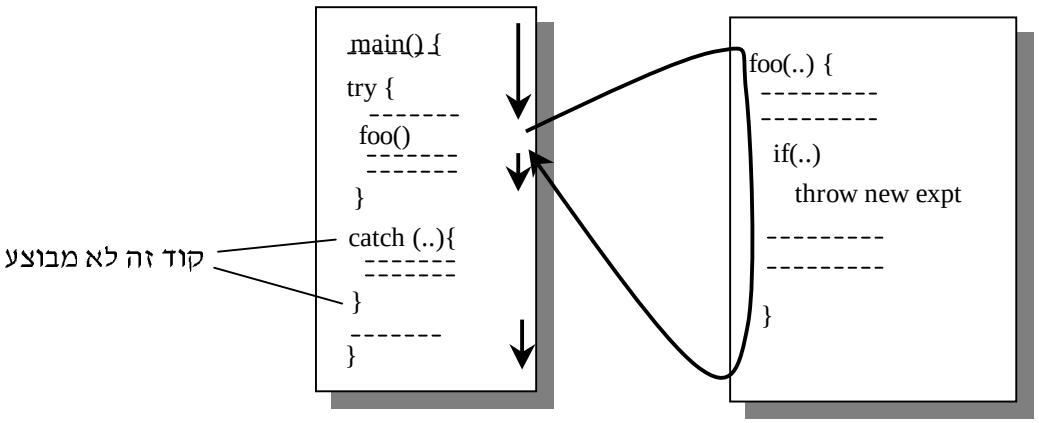
```
class MyException {};
void foo() {
        if(…) //  case of a problem
                throw MyException();
        // we won't get here if the exception was thrown
}

int main() {
        try {
                //…
                foo();
                //…
```

```
        } catch (MyException) {
                // handling the problem
        }
}
```

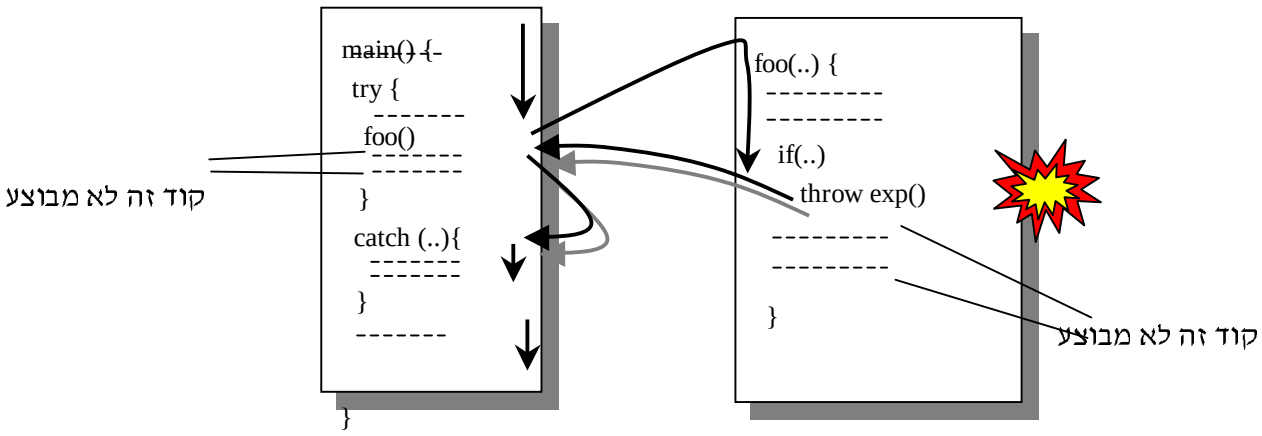
אובייקט החריגה שניזרק ב – foo() לא ניתפס בשום מקום בפונקציה. זריקת האובייקט גורמת לסיום פעולת הפונקציה (כמו – return). האובייקט שניזרק מגיע למקום בו נקראה foo() ומשם ניזרק כמו במקרה הקודם וניתפס ב – catch. גם כאן ברור המקום בו התעוררה הבעיה והיכן הקוד שאמור לטפל בה.
שימו לב שבזריקת האובייקט השתמשנו ב – syntax של אובייקט אנונימי.

הסכמה הבאה מתארת את מנגנון בקרת הזרימה של מנגנון ה – Exceptions:

<u>מהלך תקין של התוכנית</u>



<u>מיקרה של חריגה</u>



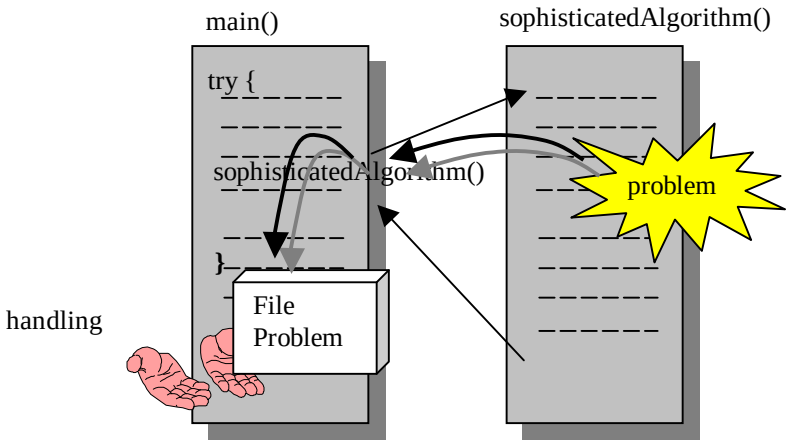
התהליך של היזרקות Exception מפונקציה וחזרה למקום הקריאה לפונקציה, ניקרא stack unwinding. בתהליך זה יש מעקב אחרי הקריאות לפונקציה במחסנית המחשב, כל זריקה של exception מפונקציה, גורמת להורדה של הקריאה לפונקציה מהמחסנית.

ניראה איך יראה הפתרון לבעיה הקודמת בעזרת exception handling:

file Main.cc:

```
class FileProblem {};  // declaration of a new Exception class


void sophisticatedAlgorithm (char* name) {
        std::ifstream in(name);…
        if(!in)
                throw FileProblem();
        // …
}

int main() {
        char name[100];
        try {
                std::cin>>name;
                sophisticatedAlgorithm (name);
                // no problem with the allocation
                // …
        } catch (FileProblem) {
                // deal with the allocation problem
        }
}
```

האיור הבא בא להמחיש את התהליך:



אם רוצים לנסות להשתקם מהבעיה ע"י ניסיונות חוזרים, ניתן להכניס את בלוק ה - try לתוך לולאה:

```
int main()
{
   char name[100];
   bool isOk = false;
   while(!isOk) {
      try {
         std::cout << "please enter a file name ";
         std::cin >> name;
         sophisticatedAlgorithm(name);
         isOk = true;
      }
      catch (FileProblem) {
         std::cout << "couldn't open the file \"" << name
                  << "\", please try again\n";
      }
   }
}
```

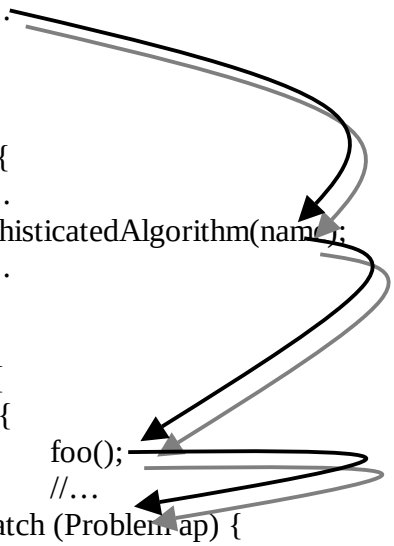
האובייקט אותו אנו זורקים הוא אובייקט C++ רגיל. אפשר לאכסן בתוכו אינפורמציה לגבי סוג הבעיה שהתעוררה. בדוגמה הבאה מאכסן האובייקט את תאור של הבעיה את מספר שורת הקוד בה היא התגלתה:

```
#include <iostream>
#include <fstream>
class Problem {
   char* _description;
   int _line;
 public:
   Problem(char* description, int line):_description(description),
                                       _line(line){}
   void show() {
      std::cout << _description << std::endl << "at line: " << _line
               << std::endl;
   }
};

void sophisticatedAlgorithm(char* name)
{
   std::ifstream in(name);
   if (!in)
      throw Problem("couldn't open file",18);
   // ...
}

int main()
{
   char name[100];
```

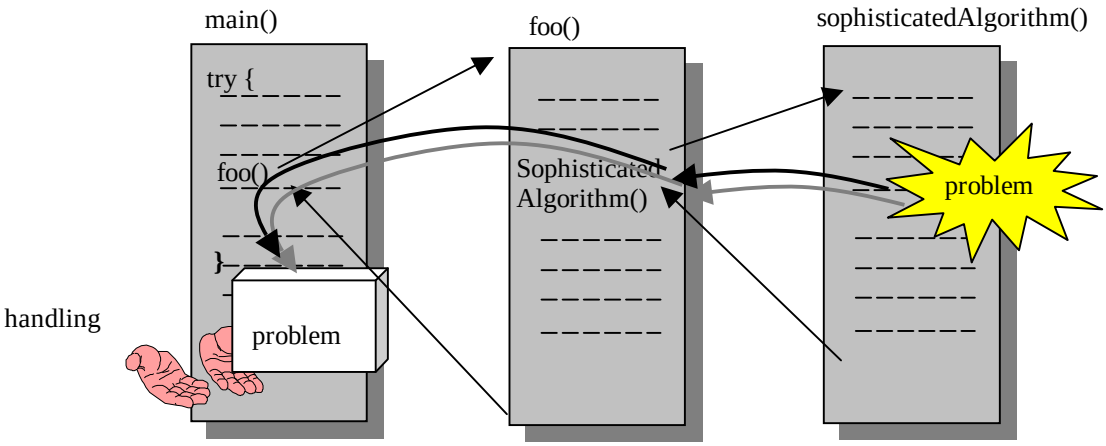
```
    try {
        std::cout << "please enter a file name ";
        std::cin >> name;
        sophisticatedAlgorithm(name);
    }
    catch (Problem p) {
        p.show();
    }
}
```

כמו שראינו קודם, אם אובייקט Exception שניזרק לא מטופל בפונקציה, הוא ממשיך להיזרק ממנה:

```
class Problem {…};

void sophisticatedAlgorithm (char* name) {
        //…
        if(!in)
                throw Problem (…);
        // …
}


void foo() {
        // …
        sophisticatedAlgorithm(name);
        // …
}

int main() {
        try {
                foo();
                //…
        } catch (Problem ap) {
                ap.show();
        }

}
```

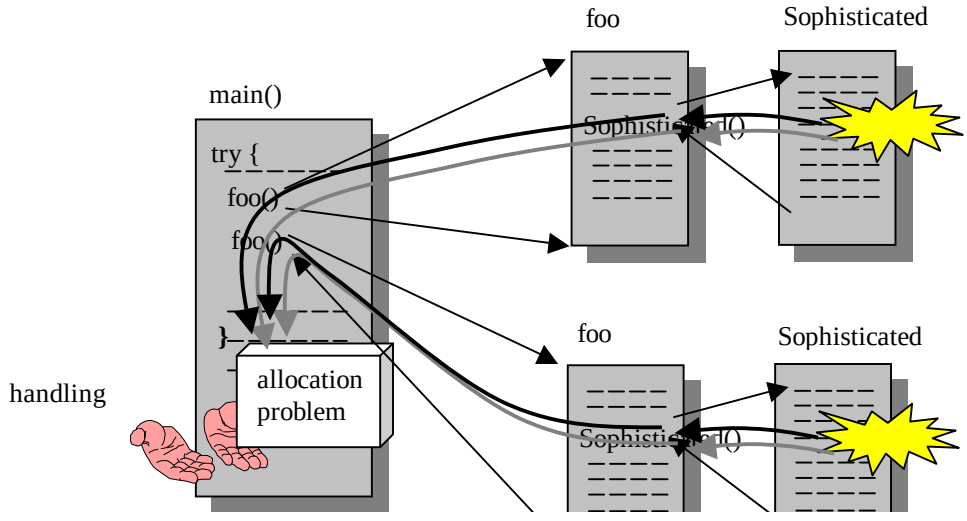


מכיוון שהפונקציה foo() לא דואגת לטיפול ב – Exception שניזרק מ-
sophisticatedAlgorithm ה – exption ניזרק מ – foo ומגיע ל – main.



דוגמה זו מראה יתרון חשוב של מנגנון ה – exception handling על השיטות האחרות לטיפול בשגיאות. המנגנון מאפשר לטפל בעיות גם ברמה עליונה בקוד מבלי להוסיף לקוד הביניים (foo במקרה שלנו) שום קוד המתייחס אליהם. המנגנון מאפשר לנו גם טיפול מרוכז, במקום אחד, בבעיות שמתעוררות במקומות שונים:

```
int main() {
        try {
                foo();
                foo();
                //…
        } catch (Problem p) {
                p.show();
        }

}
```



חשוב להבין שכל זריקה של אובייקט גורמת לסיום ה – scope שמתוכו הוא ניזרק:

```
#include <iostream>

class MyException {};

class A{
public:
	~A() {std::cout << "dtor\n";}
};

void foo() {
	A varOfFoo;
	throw MyException();
}

int main() {
	try {
		A varOfTry;
		foo();
		// we won't get here
	} catch (MyException) {
		std::cout << "caught !\n";
	}
}
```

הפלט יהיה:

```
dtor
dtor
caught !
```

ה – dtor הראשון מודפס כתוצאה מחסול varOfFoo והשני כתוצאה מחיסול varOfTry.

זריקת Exception מ – ctor

אם ניזרק exception מ – constructor, ה – destructor של האובייקט לא יופעל לעולם:

```
#include <iostream>

class MyException {};
```

```
class A{
public:
	A() {throw MyException();}
	~A() {std::cout << "dtor\n";}
};

int main() {
	try {
		A a;
	} catch (MyException) {
		std::cout << "caught !\n";
	}

}
```

פלט:

```
caught!
```

ה – constructor אחראי על בניית האובייקט. אם ניזרק ממנו Exception המונע את השלמת הבנייה שלו, אין בידנו אובייקט שלם ולכן אי אפשר להרוס אותו בצורה מסודרת. תהליך הבניה של אובייקט הוא קטע רגיש בקוד ולכן כדאי להשתדל שה – constructor יהיה פשוט וקצר ככל שניתן. לעיתים מוסיפים לאובייקט פונקציה נוספת האחראית על האתחול. כאשר יש הפרדה כזאת בין בניה לאתחול, אם יש בעיה באתחול האובייקט, האובייקט כבר קיים ואפשר לחסל אותו בצורה מסודרת.

זריקת exception מ – dtor

זריקת exception מ – dtor עלולה לגרום לבעיה הבאה:

```
#include <iostream>

class Exp {};

class A{
public:
	~A() { throw Exp();}
};

int main() {
	try {
		A a; // try to delete this line
		throw Exp(); // try to delete this line
	} catch (Exp) {}
}
```

תוכנית זו לא תסתיים בצורה סדירה.

זריקת ה – exception בבלוק ה – try גורמת לחיסול כל המשתנים בבלוק. חיסול a גורם לזריקת Exception נוסף. מצב של שני Exception פעילים הוא מצב לא סדיר. בגלל הבעיה הזאת, **יש להימנע מזריקת Exception ב – dtor**.

Exception הנזרק מ – main

גם מ – main , כמו מכל פונקציה יכול להיזרק exception. Exception הנזרק מ – main גורם לסיום מסודר של התוכנית או להפעלת פונקציה מיוחדת שאנו מגדירים עבור מקרה כזה. כל exception שנזרוק בתוכנית ולא נתפוס באף מקום אחר, ייזרק בסופו של דבר מ – main:

```
class Exp {};

void foo() {
        throw Exp();
}

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

כאשר אנחנו כותבים תוכנית וחושבים על אפשרות שתתעורר בעיה פתולוגית שאין ברצוננו לטפל בה, אפשר פשוט לזרוק Exception במקום בו מתגלה הבעיה ולא לתפוס אותו בשום מקום. במקרה והבעיה תתעורר, התוכנית תסתיים באופן מסודר.

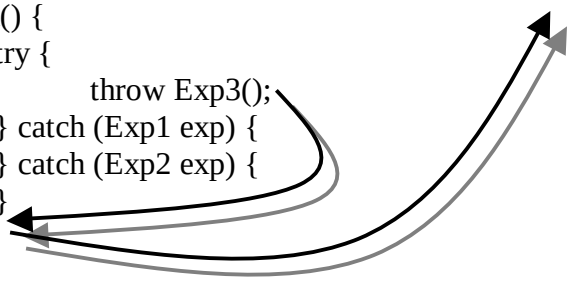
סידרה של 'תופסים ומטפלים'

עד כה ראינו דוגמאות בהם ניסינו לתפוס רק סוג אחד של exception בבלוק - try . אפשר לנסות לתפוס מספר סוגים של exception הנזרקים מבלוק try:

```
class Exp1{};
class Exp2{};
class Exp3{};

int main() {
        try {
                //…
        } catch (Exp1 exp) {
        } catch (Exp2 exp) {
        }
}
```

אם מבלוק ה – try ייזרק Exception מסוג Exp1 הוא ייתפס ע"י ה – catch הראשון. אם ייזרק Exception מסוג Exp2 הוא ייתפס ויטופל ע"י בלוק ה – catch השני. במקרה שייזרק Exception מסוג Exp3 , הוא ייזרק גם החוצה מ – main כיוון שאין טיפול במקרה כזה ע"י catch.

```
int main() {
        try {
                throw Exp3();
        } catch (Exp1 exp) {
        } catch (Exp2 exp) {
        }
}
```



שימוש בהיררכית ירושה של Exception

מנגנון ה – exception handling של C++ מאפשר שימוש בהיררכית ירושה של מחלקות Exception. שימוש זה בהיררכיה מאפשר טיפול בבעיות ברמת ההפשטה הרצויה:

```
class BaseException{};
class Derived1 : public BaseException {};
class Derived2 : public BaseException {};

int main() {
	try {
		// throw BaseException();
		// throw Derived1();
		// throw Derived2();
	} catch (Derived1 e) {
		std::cout << "derived1\n";
	} catch (BaseException e) {
		std::cout << "Base\n";
	}
}
```

Exception ניתפס ע"י catch אם הוא **סוג של** האובייקט שה - catch מנסה לתפוס.

אם ייזרק Derived1 הוא ייתפס ע"י ה – catch הראשון. אם ייזרקו Derived2 או BaseException הם יתפסו ע"י השני.

ניסיון התפיסה נעשה לפי סדר. אם היינו מחליפים את סדר ה – catch -ים בתוכנית היינו מקבלים תוכנית לא הגיונית כיוון ש – Exception מסוג Derived1 היה ניתפס תמיד ב – catch התופס BaseException וה – catch התופס את Derived1 היה מיותר.

האפשרות לתפוס Exception הנמצא במקום גבוהה בהיררכית הירושה של ה - Exception מאפשרת לטפל בהרבה סוגים של בעיות ע"י קוד יחיד. לדוגמה:

```
class AllocationProblem {…};
class MemoryAllocationProblem : public AllocationProblem {…};
class FileAllocationProblem : public AllocationProblem {…};
// …

class IoProblem {…};
class InputProblem : public IoProblem {…};
class OutputProblem : public IoProblem {…};
// …

int main() {
	try {
		// …
	}
	catch (AllocationProblem ap) {
		// deal with all kind of Allocation problems in the same way
	}
	catch  (IoProblem iop) {
```

```
            // deal with all kind of IO problems in the same way
        }
}
```

בהמשך ניראה איך אפשר להשתמש ב – polymorphism כדי לשכלל רעיון זה.

catch(…)

אפשר לתפוס כל Exception שניזרק בבלוק try ע"י catch(…) :

```
class Exp1 {}; class Exp2{};
int main() {
        try {
                throw Exp1();
        }
        catch (Exp2) {}
        catch (...) {
                std::cout << "catched!\n";
        }
        // no Exception will be thrown from this main
}
```

catch(…) מהווה רשת לא עבירה ל – Exception. כל exception שניזרק מה – try block לא ימשיך להיזרק הלאה מהפונקציה.

משך החיים של האובייקטים הנזרקים כ - Exceptions

נתבונן בדוגמה הבאה:

```
#include <iostream>
class E {
  public:
    E() {}
    E(const E& other) { std::cout << "copy ctor\n";}
};

int main(){
    try {
        E e;
        throw e;
    } catch (E e1) {}
}
```

האובייקט e הוא אובייקט לוקאלי ל-scope של ה - try וייהרס כאשר ה - scope ייסגר.
למעשה, האובייקט שייזרק יהיה **העתק** של e. מכיוון שהפרמטר e1 של ה - catch מתקבל by-value, האובייקט e1 יהיה **העתק** של אותו אובייקט שניזרק. מכיוון שמתבצעות שתי העתקות, פלט הקוד יהיה:

```
copy ctor
copy ctor
```

אם נשנה את אופן תפיסת e1 לתפיסה by-reference, תהיה רק העתקה אחת:

```
int main(){
```

```
    try {
        E e;
        throw e;
    } catch (E& e1) {}
}
```

פלט:

```
copy ctor
```

נשאלת השאלה, מי אחראי להריסת האובייקט שניזרק ובאיזה שלב היא מתבצעת ?

התשובה היא שאותו אובייקט שנזרק, נהרס באופן אוטומטי בסיום הבלוק של ה - catch והמתכנת לא אמור להרוס אותו בעצמו.
הקוד הבאה מדגים זאת:

```
#include <iostream>
class E {
  public:
    E() {}
    E(const E& other) { std::cout << "copy ctor\n";}
    ~E() { std::cout << "dtor\n";}
};

int main(){
    try {
        E e;
        throw e;
    } catch (E& e1) {
        std::cout << "in catch\n";
    }
    std::cout << "after all\n";
}
```

פלט:

```
copy ctor
dtor
in catch
dtor
after all
```

ה dtor הראשון הוא תוצאה של הריסת e וה-dtor השני כתוצאה מהריסת אותו העתק שניזרק.

בכדי להדגים ביתר פרוט מתי נבנה, משוכפל ונהרס כל אובייקט נשתמש בדוגמה הבאה:

```
#include <iostream>
class E {
    static int counter;
    int _serial;
  public:
    E() : _serial(++counter) {
        std::cout << "ctor of object #" << _serial << std::endl;
    }
    E(const E& other) :_serial(++counter) {
        std::cout << "copy ctor of object #" << _serial << std::endl;
    }
    ~E() {
        std::cout << "dtor of object #" << _serial << std::endl;
```

```
    }
};

int E::counter = 0;

int main(){
    try {
        throw E();
    } catch (E& e) {
        std::cout << "inside catch\n";
    }
}
```

כפי שניתן ליראות, לכל אובייקט מסוג E יש מספר סידורי המאפשר מעקב אחריו.
הפלט יהיה:

```
ctor of object #1
copy ctor of object #2
dtor of object #1
inside catch
dtor of object #2
```

אם משנים לתפיסה - by value :

```
int main(){
    try {
        throw E();
    } catch (E e) {
        std::cout << "inside catch\n";
    }
}
```

הפלט יהיה :

```
ctor of object #1
copy ctor of object #2
dtor of object #1
copy ctor of object #3
inside catch
dtor of object #3
dtor of object #2
```

שימוש ב – polymorphism של Exception.

היכולת לתפוס reference של exception מאפשר לנו לנצל את ה – polymorphism כדי לכתוב קוד גנרי לטיפול במצבים חריגים שונים:

```
class BaseException{
public:
        virtual void showDetails() = 0;
};

class Derived1 : public BaseException {
public:
        virtual void showDetails() {std::cout << "Derived1\n"; }

} ;
```

```
class Derived2 : public BaseException {
public:
	virtual void showDetails() {std::cout << "Derived2\n"; }
};

int main() {
	try {
		throw Derived1();
		//throw Derived2();
	} catch (BaseException& e) {
		e.showDetails();
	}
}
```

דוגמה : מערך בטוח

הדוגמה הבאה מראה שימוש ב – exception handling כדי לממש מערך המוגן מפני חריגת טווח האינדקסים:

```
class OutOfRangeException {
public:
	int _max,_index;
	OutOfRangeException(int max, int index)
		: _max(max),_index(index) {}
};

class SafeArray {
	double* _array;
	int _size;
public:
	SafeArray(int size) : _size(size) {
		_array = new double[_size];
	}
	double& operator[](int i) {
		if(i < 0 || i >= _size)
			throw OutOfRangeException(_size,i);
		return _array[i];
	}
};

int main() {
	try {
		SafeArray array(10);
		for(int i=0; i < 10; i++)
			array[i] = i*i;
		for(i=0; i <= 10; i++)
			std::cout << array[i] << " ";
	} catch (OutOfRangeException exp) {
		std::cout << "\nlimit: " << exp._max << " accessed: " << exp._index
<< std::endl;
	}
}
```

הקוד של המערך זורק Exception כאשר השימוש במערך לא תקין. המשתמש במערך יכול עקרונית לדעת אם תתעורר בעיה ולכן זה לא בדיוק המקרה הקלאסי של מיקרה חריג עליו דיברנו בתחילת הדיון. המשתמש במערך יכל לתפוס את ה – exception במקום שנח לו ולטפל בו.

Exception הנזרק כאשר הקצאה דינמית נכשלת

כאשר הקצאה דינמית נכשלת, נזרק exception מטיפוס bad_alloc. ההצהרה של bad_alloc נמצאת ב - header בשם new. הקוד הבא מדגים תפיסה של exception כזה:

```
#include <iostream>
#include <new>

int main() {
   try {
      while(true)
         new int[10000000];
   } catch (std::bad_alloc e) {
         std::cout << "allocation failure\n";
   }
}
```

re-throw

לאחר טיפול בבעיה בבלוק catch, ניתן לזרוק את האובייקט שניתפס הלאה, להמשך טיפול ע"י הפונקציות הקוראות. בכדי לעשות זאת יש פשוט לכתוב throw ללא פרמטרים בתוך בלוק ה - catch. לדוגמה:

```
#include <iostream>

void foo1() {
    throw "hmmm…";
}

void foo2() {
   try {
      foo1();
   } catch (const char* str) {
      std::cout << "foo2 caught: " << str << std::endl;
      throw;
   }
}

int main() {
   try {
      foo2();
   } catch (const char* s) {
      std::cout << "main caught: " << s << std::endl;
   }
}
```

פלט:

```
foo2 caught: hmmm...
main caught: hmmm...
```

Exception Specifications

ניתן להוסיף להצהרה על פונקציה גם את רשימת ה - exception שהיא עלולה לזרוק. זאת בעצם התחייבות שמהפונקציה לא יזרקו Exception מסוג אחר:

```
class Exp1{}; class Exp2{}; class Exp3{};
void foo() throw (Exp1,Exp2){
        // …
}
```

אם מ foo() ייזרק exception מסוג Exp3 אז **בזמן ריצה** נדווח על זה. Visual c++ מתעלם מ – exception specifications ולא מודיע בזמן ריצה אם ניזרק Exception שהתחייבנו לא לזרוק.

אופן השימוש ב – exception handling , סיכום

מנגנון ה – exception handling מאפשר התייחסות נוחה בקוד לבעיות שעלולות להתעורר בזמן ריצה. כאשר אנחנו כותבים קוד ומגיעים למקום בו עלולה להתעורר בעיה, כדאי לזרוק Exception. ה – Exception שנזרוק יהיה אובייקט שייצג את הבעיה הספציפית שהתעוררה. אם נרצה, נוכל להכניס לאובייקט זה פרטים נוספים על הבעיה שהתעוררה. המחלקה שתייצג את הבעיה עשויה להיגזר ממחלקות אחרות המתארות בעיות כלליות יותר.

אם הבעיה שנזרקה פתולוגית במיוחד, אפשר לא לטפל בה בשום מקום בתוכנית ואז כאשר היא תתעורר, היא תגרום לסיום מסודר של התוכנית. אפשר להחליט לטפל בבעיה שזרקנו או בבעיה כללית יותר, ככל שנטפל בבעיה כללית יותר הטיפול יהיה רלוונטי ליותר סוגים של בעיות שעלולות להיווצר . יש לנו חופש לבחור באיזה רמה בקוד לבצע את הטיפול. ככל שניבחר ברמה גבוהה יותר בקוד, יטפל הקוד שנכתוב בבעיות המתגלות במקומות רבים יותר בקוד שלנו.

אורי מוסנזון GNU FDL FREE DOC LICENSE