חריגות (exceptions)

אוהד ברזילי
אוניברסיטת תל אביב

טיפול בשגיאות

חלופות "ידניות":

- סיום התוכנית
- החזרת ערך המייצג שגיאה
- החזרת ערך חוקי, אך יצירת מצב לא חוקי
- קריאה לפונקציה המטפלת בשגיאות

הצלחה וכשלון

- **הצלחה** – קריאה לפונקציה מצליחה אם ריצת הפונקציה מסתיימת במצב המקיים את החוזה של הפונקציה
- **כשלון** - קריאה לפונקציה נכשלת אם ריצת הפונקציה מסתיימת במצב שאינו מקיים את החוזה של הפונקציה
- **חריגה** (exception) – היא ארוע זמן ריצה, אשר גורם לפונקציה להיכשל

חריגות

חריגה מתרחשת במהלך ביצוע פונקציה f אם קורים אחד הבאים:

- מעקב (dereferencing) אחרי מצביע שאינו מוקצה
- ביצוע פקודה מסוימת זורק (throw) חריגה בצורה מפורשת
- פונקציה שנקראה ע"י f נכשלה

הערה: לא כל פעולה שאינה חוקית ב C++ יוצרת חריגה (למשל: נסיון לשחרר זכרון על המחסנית)

חריגות וחוזים

- כלים אשר אוכפים טענות וחוזים ב ++C בדר"כ עושים זאת ע"י זריקת חריגה עבור ההפרות הבאות:
  - הפרת תנאי קדם בכניסה לפונקציה
  - הפרת תנאי בתר ביציאה מפונקציה
  - הפרת שמורה של מחלקה בכניסה וביציאה של כל פונקציה מיוצאת של אותה מחלקה
  - שמורת הלולאה אינה מתקיימת בכניסה למחזור הראשון או בסופו של כל מחזור
  - משתנה הלולאה (loop variant) לא קטן מהמחזור הקודם
  - ביטוי assert נכשל

חריגה וכשלון

- קריאה לפונקציה נכשלת אם קרתה חריגה במהלך ריצת הפונקציה והפונקציה לא התאוששה מהחריגה
- כשלון של פונקציה יוצרת חריגה אצל מי שקרא לפונקציה
- מילים שמורות:
  - try : בלוק 'מועד לפורענות' (שעלול ליצר חריגות)
  - catch : בלוק התאוששות מה'פורענות'
  - throw : ייצור של חריגה

Div

```
/** returns a/b
 *  @pre: b>0 , "Div handles only positive b's"
 */
int Div(int a, int b)
{
  if (b == 0)
       throw "Dividing by Zero";
  else if (b < 0)
       throw 0;
  else
       return a/b;
}
```

```
int main()
{
    try
    {
        int r1 = Div(3,4);
        int r2 = Div(4,0);
        int r3 = Div(2,-5);
    }
    catch (char* msg)
    {
        cout<<"An error occurred!!!!!"<<endl;
        cout<<msg<<endl;
    }
    catch (int num)
    {
        cout<<"An error occurred!!!!!"<<endl;
        cout<<"the number "<<num<<"was caught."<<endl;
    }
    catch (...)
    {
        cout<<"An unknown error occurred!!!!!"<<endl;
    }

}
```

ניהול חריגות מתקדם

- ברור סוג החריגה שקרתה
- ניתן לספק טיפול שונה לכל אחת מהחריגות
- ניתן להתעלם מחריגות מסוימות
- ניתן ליזום חריגות בצורה מפורשת כחלק מתזרים התוכנית
- ניתן להשתמש בטיפוסי מחלקה סטנדרטים ובכאלה שהוגדרו ע"י המשתמש בתור טיפוסי העצם הנזרק

דוגמא: חריגה מתחום

```
struct Range_error {
   int i;
   Range_error(int ii) { i = ii; } // constructor
};

char to_char(int i)
{
   if (i < numeric_limits<char>::min() ||
                numeric_limits<char>::max() < i )
         throw Range_Error(i);
   return i;
}
```

הערכות לחריגה מתחום

```
void g(int i)
{
  try {
      char c = to_char(i);
      // ...
  }
  catch(Range_error) {
      cerr << "oops\n";
  }
}
```

שימוש בעצם שנזרק

```
void h(int i)
{
   try {
        char c = to_char(i);
        // ...
   }
   catch(Range_error x) {
        cerr << "oops : to _char(" << x.i << ")\n ";
   }
}
```

משפחות של עצמים נזרקים

- לעיתים די בהגדרת מחלקה ריקה כדי לציין את סוג השגיאה
- ניתן להגדיר 'משפחות' של מחלקות הנפרדות זו מזו רק בשמן (בהמשך נוכל לממש זאת בקלות ע"י ירושה)

```cpp
class Overflow {};
class Underflow {};
class Zerodivide {};

void f()
{
   try {
              // ...
   }
   catch (Overflow) {
       // handle Overflow or anything derived from
       // Overflow
   }
   catch (Underflow) {
       // handle Overflow or anything derived from
       // Underflow
   }
   catch (Zerodivide ) {
       // handle Overflow or anything derived from
       // Zerodivide
   }
}
```

זריקה חוזרת

- כאשר לבלוק ה catch (exception handler) אין מספיק מידע כדי לטפל בבעיה שנוצרה, הוא עשוי לזרוק אותה הלאה
- הדבר נעשה ע"י שימוש במילה throw ללא שם עצם אחריה

```
void h()
{
   try {
        // code that might throw Math errors
   }
   catch (Matherr) {
        if (can_handle_it_completely)
        {
              // handle the Matherr
              return ;
        }
        else
        {
              // do what can be done here
              throw; // rethrow the exception
        }
   }
}
```

עקרון הפשטות

- מטרתן של הפעולות המתבצעות בבלוק ה catch – היא להחזיר את העצם (הספק) למצב יציב:
  - שחרור משאבים (כגון זכרון) אם הוקצו טרם החריגה
  - קיום שמורת המחלקה
  - נסיון חוזר או ויתור במקרה הצורך
- על כל הפעולות בבלוק להיות ממוקדות בהשגת מטרות אלו

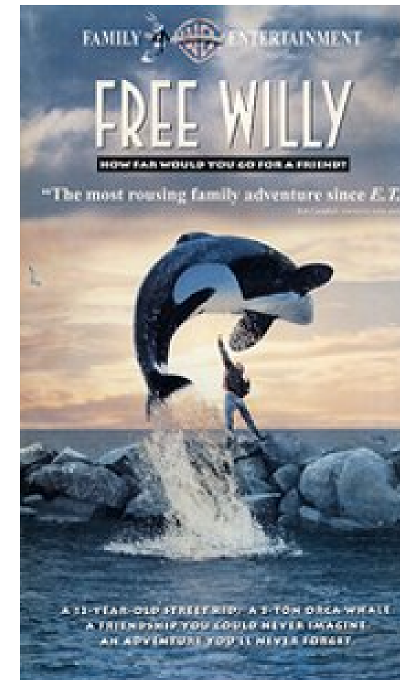
שחרור משאבים

```
void use_file(const char *fn)
{
  FILE* f = fopen(fn , "w");
  // use f
  fclose(f);
}
```

- מה עלול להשתבש?

שחרור משאבים

```
void use_file(const char *fn)
{
   FILE *f = fopen(fn ,"r");
   try {
        // use f
   }
   catch (...)
   {
        fclose(f);
        throw ;
   }
   fclose(f);
}
```

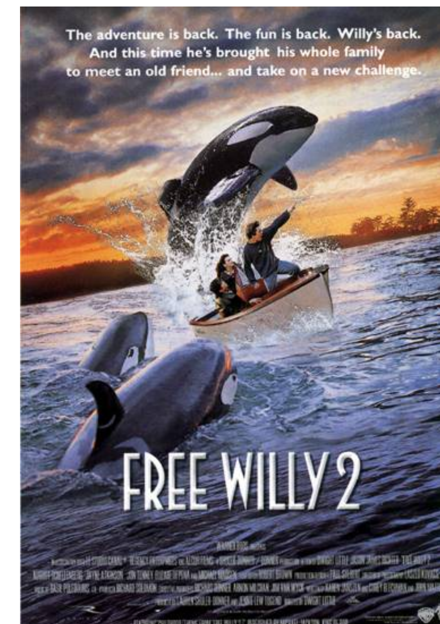


שחרור משאבים במקרה הכללי

```
void acquire()
{
  // acquire resource 1
  // ...
  // acquire resource n

  // use resources

  // release resource n
  // ...
  // release resource 1
}
```



שחרור משאבים במקרה הכללי

- נעטוף את המשאבים המוקצים בעצמים אשר ינהלו את הקצאת ושחרור המשאבים בבנאי ובמפרק
- השיטה מודגמת עבור קבצים אולם היא זהה גם עבור נעילות, הקצאות זכרון, קווי תקשורת ועוד...

```cpp
class  File_ptr {
  FILE *p;
public:
  File_ptr(const char *n , const char *a)
  { p = fopen(n,a); }

  File_ptr(FILE *pp) { p = pp;}

  ~File_ptr() { fclose(p);}

  operator FILE *() { return p ;}
};
```

ניהול משאב ע"י עצם עוטף

```
void use_file(const char *fn)
{
  File_ptr f(fn ,"r");
  // use f
}
```

- אם במהלך השימוש בקובץ נזרקת חריגה, בעת הריסת העצם המקומי f ישתחרר המשאב המוקצה

ריבוי משאבים

- שימוש בשיטת העצם העוטף אינו מסבך את המימוש גם עבור ריבוי משאבים

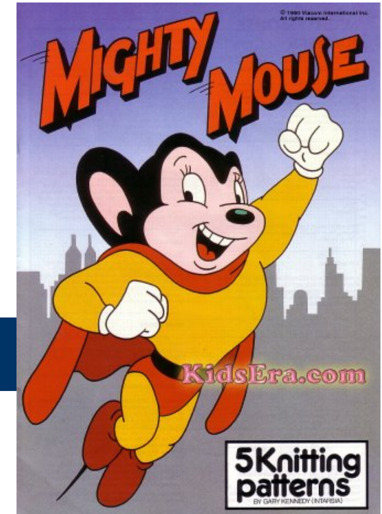
```
class X {
  File_ptr aa;
  Lock_ptr bb;
public:
  X(const char *x , const char *y)
      : aa(x ,"rw"), // acquire 'x'
        bb(y)        // acquire 'y'
  {}
  // ...
};
```

חריגות בנייה

- מה קורה אם init זורק exception ?

```
class Y {
  int *p;
  void init();
public:
  Y(int s) { p = new int[s]; init(); }
  ~Y() { delete [] p ; }
  // ...
};
```

וקטור בא לעזרה




```
class Z {
  vector<int> p;
  void init();
public:
  Z(int s) : p(s) { init(); }
  // ...
};
```

auto_ptr

- טכניקת שחרור המשאבים מומשה בספרייה הסטנדרטית (std) ע"י auto_ptr<T>
- זהו מצביע כללי המשחרר את הזכרון המוצבע על ידו בעת היציאה מהתחום שבו הוגדר (בין אם ע"י סיום תקין או עקב זריקה של חריגה)

עוד על חריגות בנייה

```cpp
class Vector {
public:
   class Size {};
   enum {max = 32000};
   Vector(int sz)
   {
       if (sz<0 || max<sz) throw Size();
       // ...
   }
   // ...
};
```

עוד על חריגות בנייה

```
Vector *f(int i)
{
   try {
        Vector *p = new Vector(i);
        // ...
        return p;
   }
   catch(Vector::Size) {
        // deal with size error
   }
}
```

```
class X {
   Vector v;
   // ...
public:
   X(int);
   // ...
};

X::X(int s)
try
   :v(s) // initialize v by s
{
   // ...
}
catch(Vector::Size) {    // exceptions thrown for
                         // v are caught here
   // ...
}
```

וכאשר Vector הוא אחד השדות

חריגות בהעתקה ובפירוק

- יש להימנע מיצירת חריגות בתוך בנאי העתקה ובתוך אופרטור ההשמה
  - שני מקרים אלו נקראים בצורה מרומזת אשר קשה מאוד לעטוף בבלוק try
- יש להימנע מיצירת חריגות בתוך מפרק
  - במקרה זה המפרק עשוי להיקרא במסגרת טיפול בחריגה קודמת

הצהרה על חריגות

- פונקציות ומתודות יכולות להצהיר על מגוון סוגי החריגות שהן זורקות
- לדוגמא:

```
void f(int a) throw(x2 , x3);
```

f רשאית לזרוק רק חריגות מטיפוס x2 או x3

```
int g(); // can throw any exception
int h() throw (); // no exception thrown
```

חריגה שאינה שגיאה
(אלא מבנה בקרה נוסף)

```
void f(Queue<X>& q)
{
  try {
    for (;;) {
      X m = q.get(); // throws ‘Empty’ if queue is empty
      // ...
    }
  }

  catch (Queue<X>::Empty) {
    return;
  }
}
```

חריגות סטנדרטיות

Standard Exceptions (thrown by the language)

Name	Thrown by	Reference	Header
bad_alloc	*new*	§6.2.6.2, §19.4.5	<new>
bad_cast	*dynamic_cast*	§15.4.1.1	<typeinfo>
bad_typeid	*typeid*	§15.4.4	<typeinfo>
bad_exception	*exception specification*	§14.6.3	<exception>

Standard Exceptions (thrown by the standard library)

Name	Thrown by	Reference	Header
out_of_range	*at()*	§3.7.2, §16.3.3, §20.3.3	<stdexcept>
	bitset<>::operator[]()	§17.5.3	<stdexcept>
invalid_argument	*bitset constructor*	§17.5.3.1	<stdexcept>
overflow_error	*bitset<>::to_ulong()*	§17.5.3.3	<stdexcept>
ios_base::failure	*ios_base::clear()*	§21.3.6	<ios>