
Exception Semantics

- Error Handling
 - Function **return value**
 - C has the **convention** that a function that **executes properly should return a 0** if it is not using the function's return value for something else.
 - **Exceptions are thrown/caught**. This disrupts the normal flow of execution.
 - **Assertions**. Check the correctness of assumptions during run-time
 - **Signals**. A software interrupt delivered to a process. OS uses it to report exceptional situations to an executing program (e.g., references to invalid memory addresses).
- Exceptions
 - The formal semantics of exceptions is out of the scope of this course.
 - We will describe the flow of execution when an exception is encountered.

- C++ example: The exceptions can be any type (std::exception, int, etc.). The throw statement essentially calls the catch block. [Show [trycatch.cc](#)]

```
#include <cstdio> //stdio.h in C

int main (int argc, char *argv[]) {
    for (;;) {
        int q;
        try {
            printf("Enter a number (0..9): ");
            int k = scanf("%d", &q); // read formatted from stdin

            if (k == 0) {
                scanf("%*s"); //read the value but ignore
                throw "value is not a number";
            }

            //this part is unnecessary, just show different types of exceptions
            if (q < 0) {
                throw 3.0;
            }

            if (q < 0 || q > 9) {
                throw q;
            }
        }
        catch (const char *s) {
            printf("Error: %s\n", s);
            continue;
        }
        catch (int v) {
            printf("Error: number %d is out of range (0..9)\n", v);
            continue;
        }
        catch (...) {
            printf("Unsepcified error\n");
            continue;
        }

        if (q == 0)
            break;
    }
    return 0;
}
```

```
$ ./a.out
Enter a number (0..9): a
Error: value is not a number
Enter a number (0..9): 10
Error: number 10 is out of range (0..9)
Enter a number (0..9): -1
Unsepcified error
Enter a number (0..9): 0
$
```

- A throw within a function can be caught in the parent function. [Show trycatchPC.cpp]

```
#include <cstdio>

int childFunc () {
    int q;

    try {
        printf("Enter a number (0..9): ");
        int k = scanf("%d", &q);

        if (k == 0) {
            scanf("%*s");
            throw "value is not a number";
        }
        if (q < 0 || q > 9) {
            throw q;
        }
    }
    catch (int v) {
        printf("Error: number %d is out of range (0..9)\n", v);
        return v;
    }

    return q;
}

int main () {
    int val;

    for (;;) {
        try {
            val = childFunc();
        }
        catch (const char *s) {
            printf("Error: %s\n", s);
            continue;
        }

        if (val == 0)
            break;
    }

    return 0;
}
```

```
$ g++ trycatchPC.cc
$ ./a.out
Enter a number (0..9): -1
Error: number -1 is out of range (0..9)
Enter a number (0..9): 10
Error: number 10 is out of range (0..9)
Enter a number (0..9): 9
Enter a number (0..9): a
Error: value is not a number
Enter a number (0..9): 0
$
```

- Java example: **A function can throw an exception it does not catch.** To do this, it **must declare the property of the exception** to calling functions **by including a throw clause in its preamble.**
 - Like C++, **Java permits multiple catch blocks differentiated by their argument.**
 - The try/catch structure can also include a **block labeled with finally**, which has **no arguments. Code in the finally section is always executed** whether or not the code throws an exception. It is executed even if one of the try/catch blocks calls break or return. [Show trycatch.java, ask the outputs]

```
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class trycatch {

    public static void main (String[] args) throws IOException {
        int number;

        while (true) {
            try {
                BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
                System.out.print("Enter number: ");
                number = Integer.parseInt(in.readLine());

                if (number == 0) break;
            }
            catch (NumberFormatException e) {
                System.out.println("Illegal number");
            }
            finally {
                System.out.println("In the finally block");
            }
        }

        System.out.printf("number is %d\n", number);
    }
}
```

```
$ javac trycatch.java
$ java trycatch
Enter number: 1
In the finally block
Enter number: 2
In the finally block
Enter number: 3
In the finally block
Enter number: 4
In the finally block
Enter number: 0
In the finally block
number is 0
```

- Python example: If an exception occurs in the `try` statement, execution moves to the `except` block if the exception matches one of the exceptions listed.
 - The syntax **permits multiple except cases. An except block can also list multiple exceptions inside a tuple**, e.g. (`RuntimeError`, `TypeError`, `NameError`, `ValueError`).
 - An **`except` block with no arguments catches all exceptions not explicitly caught by other except cases.**
 - The equivalent of `throw` in Python is the **`raise`** keyword. The `raise` statement **takes the exception class as an argument**. [Show `trycatch.py`]
 - Python also includes a `finally` clause like Java.

```
def demo1():
    a = 0

    while a == 0:

        try:
            s = input('enter a number: ')
            val = int(s)

        except ValueError:
            print("not a valid number")
            continue

        if val == 0:
            a = 1

    print("terminating")

def demo2_helper():
    raise 1

def demo2():
    try:
        demo2_helper()
    except:
        print("demo 2 catching the error")

demo1()
#demo2()
```

Comment out `demo2()` and run the code

```
$ python3 trycatch.py
enter a number: a
not a valid number
enter a number: 1
enter a number: 0
terminating
```

Comment out `raise` inside the `except` of `demo2()` function, and run again

```
$ python3 trycatch.py
demo 2 catching the error
```

- Assertions
 - Statements used to [check the correctness of assumptions](#) made in a program [during run-time](#).
 - Debug tool
 - C example:

```
void assert (int expression);
```

- If the expression meaning is 0 (false), the expression, source code filename, and line number are sent to standard error, and then the abort function is invoked.

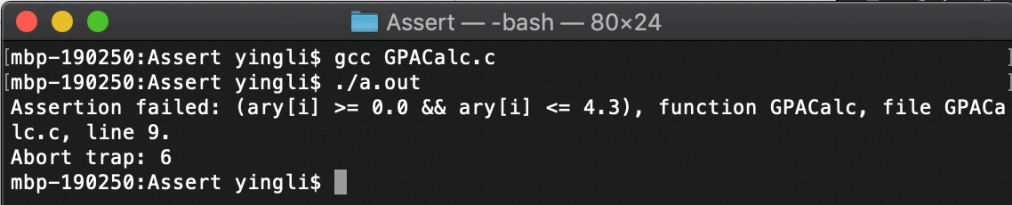
```
// GPACalc.c
#include <stdio.h>
#include <assert.h>

float GPACalc (int size, float *ary) {
    float sum = 0;
    for (int i = 0; i < size; i++) {
        // check invalid gpa
        assert (ary[i] >= 0.0 && ary[i] <= 4.3);
        sum += ary[i];
    }
    return sum/size;
}

int main () {
    float a[] = {3.5, 3.8, 4.0, 4.1, 4.5};

    printf("%f\n", GPACalc(5, a));

    return 0;
}
```



```
mbp-190250:Assert yingli$ gcc GPACalc.c
mbp-190250:Assert yingli$ ./a.out
Assertion failed: (ary[i] >= 0.0 && ary[i] <= 4.3), function GPACalc, file GPACalc.c, line 9.
Abort trap: 6
mbp-190250:Assert yingli$
```

- Java example:

```
assert expression;
assert expression1 : expression2;
```

- expression1: boolean expression; If false, JVM throws AssertionError.
- expression2: an expression that has a value.
- [java -ea](#) : enable assertions

```

public class GPACalc {
    public static double calc (double ary[]) {
        double sum = 0;
        for (double f : ary) {
            assert f >= 0.0 && f <= 4.3 : "Invalid value " + f;
            sum += f;
        }
        return sum/ary.length;
    }

    public static void main (String args[]) {
        double a[] = {3.5, 3.8, 4.0, 4.1, 4.5};

        System.out.println(calc(a));
    }
}

```

```

mbp-190250:Assert yingli$ javac GPACalc.java
mbp-190250:Assert yingli$ java GPACalc
3.9799999999999995
mbp-190250:Assert yingli$ java -ea GPACalc
Exception in thread "main" java.lang.AssertionError: Invalid value 4.5
    at GPACalc.calc(GPACalc.java:5)
    at GPACalc.main(GPACalc.java:14)
mbp-190250:Assert yingli$

```

- Python example

assert condition, error_message(optional)

- condition: boolean expression
- error_message: printed on terminal if AssertionError

```

# GPACalc.py

def GPACalc (ary):
    sum = 0.0
    for f in ary:
        assert f >= 0.0 and f <= 4.3, "Invalid value " + str(f)
        sum += f;
    return sum/len(ary)

def main ():
    a = [3.5, 3.8, 4.0, 4.1, 4.5]
    print(GPACalc(a))

if __name__ == "__main__":
    main()

```

```

mbp-190250:Assert yingli$ python3 GPACalc.py
Traceback (most recent call last):
  File "GPACalc.py", line 15, in <module>
    main()
  File "GPACalc.py", line 12, in main
    print(GPACalc(a))
  File "GPACalc.py", line 6, in GPACalc
    assert f >= 0.0 and f <= 4.3, "Invalid value " + str(f)
AssertionError: Invalid value 4.5
mbp-190250:Assert yingli$

```

- Signals

- In C, signals can be used to **handle exceptions**, and it has a **library to handle signals**. These are **not the same as exceptions** (though they could be caused by similar errors, such as dereferencing a null pointer).
- An example of a signal that is not an exception is a key-board interrupt. This is an external signal and it needs to be handled.
- Show signal-int.c, run code and press ctrl+c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
```

```
int quit = 0;
```

```
void handler (int signal) {
    printf("\nCaught signal %d\n", signal);
    quit = 1;
}
```

```
//return;  
exit(-1); //to terminate
```

}

```
int main () {
    // SIGINT "program interrupt" signal, the signal is sent when the user type
    // INTR character, usually Ctrl+c
    signal(SIGINT, *handler);
```

```
while (!quit) {
    printf("blah\n");
}
printf("\nCleaning up\n");
```

```
return 0;
```

}

```

blah
blah
^Cblah
blah
blah
blah
blah
blah
blah

```

Caught signal 2

- If the signal handler simply returns instead of exiting the program, the main program will continue where it left off. But it's not always possible. [comment out the `exit(-1)` in the above and `return` in the handler function, and run the code]

```

blah
^Cblah
blah
blah
blah
blah
blah
blah
blah

```

Caught signal 2

Cleaning up

- Show signal-bus.c [if use return in the handler, it will generate infinite loop]

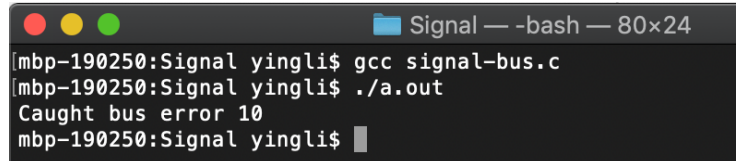
```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>

void bushandler( int signal ) {
    printf("Caught bus error %d\n", signal);
    // return; //infinite loop
    exit(-1); // gracefully leave program
}

int main(int argc, char *argv[]) {
    signal( SIGBUS, bushandler );

    // on Mac OS, the literal strings are stored in constant memory, so
    // we can't write to them.
    strcpy("hello", "goodbye");

    return 0;
}
```



```
Signal — -bash — 80x24
mbp-190250:Signal yingli$ gcc signal-bus.c
mbp-190250:Signal yingli$ ./a.out
Caught bus error 10
mbp-190250:Signal yingli$
```

- Show signal-segv.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

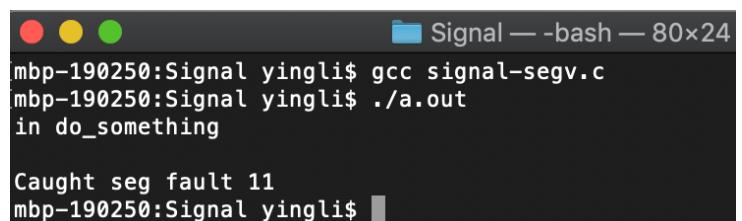
float *x = NULL;

// assume global variable x has been allocated. print its value.
void do_something() {
    printf( "in do_something\n" );
    // the error happens inside printf
    printf("%.2f\n", *x );
}

void seghandler(int signal) {
    printf("\nCaught seg fault %d\n", signal);
    exit(-1);
}

int main(int argc, char *argv[]) {
    signal(SIGSEGV, seghandler);
    do_something();

    return 0;
}
```



```
Signal — -bash — 80x24
mbp-190250:Signal yingli$ gcc signal-segv.c
mbp-190250:Signal yingli$ ./a.out
in do_something

Caught seg fault 11
mbp-190250:Signal yingli$
```

Notes on signals from Bruce Maxwell:

C includes a simple exception handling mechanism that relies on the operating system to pass control flow to an exception handling function if a run time error occurs. The signal package includes a function

```
signal( int sig, sig_t func )
```

that enables the programmer to attach a function pointer to either a standard OS signal or a user defined signal. After setting up the handler, if a signal occurs, the OS checks the signal handler table and routes execution to the user function. The user function has access to the program's memory space and global variables, but there is no guarantee they have not been corrupted by the event that caused the signal. In general, signal handlers shut down the program gracefully and exit, if possible.

The signal function also permits the user to specify that certain signals should be ignored, allowing program execution to continue where the signal left off. Some signals are not ignorable and cause program termination.

If the signal handler simply returns instead of exiting the program, then the main program will continue where it left off. Unfortunately, because of the parallel processes occurring in I/O situations, the exact state of the program upon return is not necessarily well defined. If the signal occurred during an I/O operation, the operation may be restarted. Therefore, it is possible to get in an infinite loop if restarting the operation causes another segmentation fault.

Semantically, the C signal handling is different from the prior 3 cases. In C++, Java, and Python the program does not attempt to restart execution where it left off. Instead, execution moves to the catch block, giving the program a well-defined behavior after an exception occurs. Therefore, exceptions are a reasonable method of handling IO errors, for example.

In C, execution order does not skip, but tries to pick up where the signal occurred. In the case of signals from outside (e.g. ctrl-c) ignoring the error and returning from the handler should have no effect since the signal did not occur because of the program's own behavior. In the case of signals generated by the program itself, since return from a signal handler can lead to re-execution of an I/O operation, the meaning of the program can be undefined.

Note that, because execution picks up at the machine instruction where the problem occurred, that it may not be possible to correct the problem. If the issue is the value of a variable, for example, that variable's memory location may have already been copied into a function, converted to a different type, or read into a register and put on the bus. Changing the value of a variable after any of these actions will have no impact on the program's ability to continue.

Notes on Error-Handling from Bruce (this really should be the intro, but Stephanie has spent enough time wrestling with image placement in Pages).

Early programming languages did not handle exceptions gracefully. Fortran and C, for example, have no built-in exception handling mechanisms. The programmer is responsible for handling all errors. The C standard libraries do provide a method of handling exceptions that interfaces with the operating system, but it is not part of the language syntax or semantics. C also has the convention that a function that executes properly should return a 0 if it is not using the function's return value for something else. C functions with a void return type tend to be rare because of this, while they are common in Java programming.

More recent programming languages provide extensive exception handling (Ada, C++, Java, Python). While exception handling is not the default action of a program in these languages, they all have syntactic and semantic elements that support exception handling. A formal description of exception handling semantics is beyond the scope of this course.

The issue with errors is that the operation or function in which the error occurs generally cannot handle the error. A division operator cannot handle a divide by zero error. A function that is supposed to open a file cannot handle the error that occurs when there is no file by the given name. A function that is supposed to execute an operation on an image cannot handle the error that occurs when there is insufficient memory for it to allocate temporary working space.

In all of these cases, information about the error has to propagate up to the next level of scope, if not further.

Error communication strategies:

- Have each function/method return a specific value when it encounters an error. The calling function/method must test the return value of the function and take appropriate action. Many library functions in C/C++ and Java use this method.
- Have each function/method set the value of a variable when it encounters an error. The calling function/method must be able to access this variable and test it after the function call to take appropriate action. Many library functions in C/C++ use this method, especially those interfacing with the operating system and file I/O. Sometimes functions will both return an error value and set an error variable, usually using the latter to communicate which error occurred in the function.
- A function/method that may potentially encounter an error should set up an exception handler that gets called if an exception occurs. In some cases, exception handlers allow the program to continue operation, in other cases program execution halts when the exception handler completes.
- A function/method that encounters an error should throw an exception. The calling function has the responsibility of catching the exception and either handling it or passing it on to a higher context.

Try/Catch Approach

The try/catch approach encapsulates code that could cause an exception in a try block. If an exception occurs, execution in the try block halts and restarts with the first statement in the catch block. The try/catch approach usually also incorporates a throw command. A throw command permits the programmer to throw an exception deliberately, usually with a programmer defined exception type.

C++

In the case of C++, the semantics are simple to describe in words. Execution begins with the code in the try block. If no exception occurs, execution skips the catch blocks. If an exception occurs, execution continues with the first statement in the appropriate catch block, if one exists. If there is no catch block for a given exception type, the program aborts.

The catch blocks are effectively overloaded functions called by the throw statement, except the catch blocks do not return to the next statement in the try block as a function would. The catch blocks must be differentiable by their parameter list. C++ includes a set of standard exceptions defined in `std::exception` that statements like `new` would throw. The list also includes a `catchall` exception. The benefit of using try/catch blocks is that the function in which the exception occurs can catch the problem.

Note that any function can throw an exception at any time. If the function call is within a try/catch block of a parent function, then the parent function will catch the exception if there is a matching catch block. You can even design a hierarchical system whereby the local function catches some exceptions and passes others on to the parent.

Java

Java uses a similar syntax to C++, but the semantics are somewhat different. One of the major differences is that exceptions in Java are Objects and are part of an Exception hierarchy. Therefore, when an exception occurs, Java is creating a new object and adding it to the symbol table.

A function that could throw an exception it does not catch must declare that property to calling functions by including a `throws` clause in its preamble. If a function `X` uses an IO call, for example, and wants the parent to handle exceptions, then `X` must declare that it throws an `IOException`, as shown in the example above. The method itself handles problems with the number format, but passes off other issues to the parent context (the JVM, in this case).

Like C++, Java permits multiple catch blocks differentiated by their argument. The try/catch structure can also include a block labeled with `finally`, which has no arguments. Code in the `finally` section is always executed whether or not the code throws an exception. It is executed even if one of the try/catch blocks calls `break` or `return`.

Python

As with the other cases, if an exception occurs in the try statement, execution moves to the except block if the exception matches one of the exceptions listed. The syntax permits multiple except cases. An except block can also list multiple exceptions inside a tuple, e.g. (RuntimeError, TypeError, NameError, ValueError). Python includes many standard error types generated by built-in functions. An except block with no arguments catches all exceptions not explicitly caught by other except cases.

Python also permits an else block at the end of the try/except sequence. Code in the else block executes after the code in the try block if no exception occurs.

The equivalent of a throw in Python is the raise keyword. The raise statement takes the exception class as an argument. A raise statement with no argument inside an except block re-raises the exception that caused the block to be executed and passes the buck to any encompassing try/except blocks.

Exceptions can be arbitrary user-defined classes. Usually they are simple.

Python, like Java, also includes a finally clause. Code in the finally block executes no matter what else occurs in the exception structure. The code in the finally block executes whether the try structure exits on no exception, a handled exception, an unhandled exception, a return, a break, or a continue. In the case of an unhandled exception, Python re-raises the exception once the finally block completes.

Python has one other convenience feature that falls in the category of “language features that avoid oopses”. Consider the following example.

```
with open("myfile.txt") as fp:
    for line in fp:
        print line
```

The with statement puts the result of the call to open, which is a file object, into the variable fp. The file class has a predefined cleanup action in its class definition. The with statement simply calls the cleanup method of the class when the block terminates.