Recitation - 13

Jahnavi - jp5867@nyu.edu

Exception Handling

Exception:

-> An exception is an event that disrupts the normal operation of the program's execution.

-> It occurs during the execution.

Exception handler: it is a process that handles those "exceptional situations."

-> Handler requires a special process to respond those exception during computation.

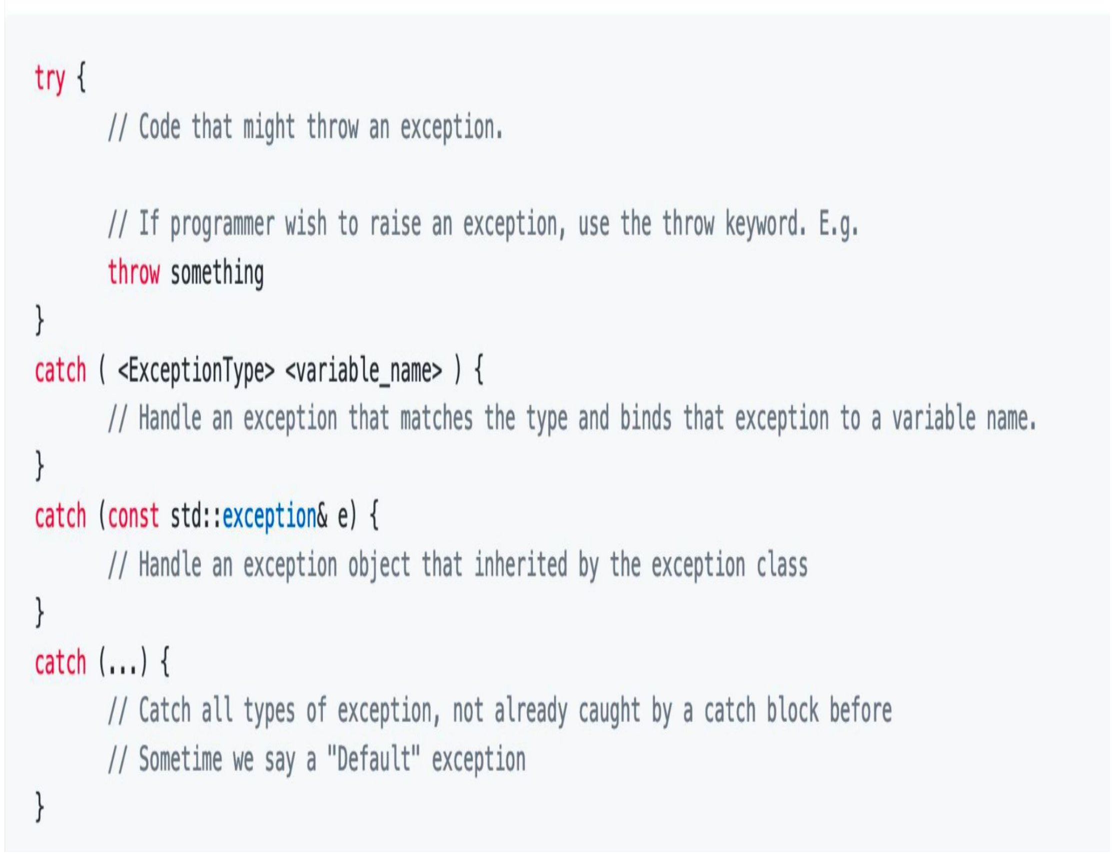
At programming language level:

- The exception is used for storing information about an exceptional condition.
- The throw is used to raise an exception.
- Once an exception is thrown, the execution now is transferred to a catch

C++

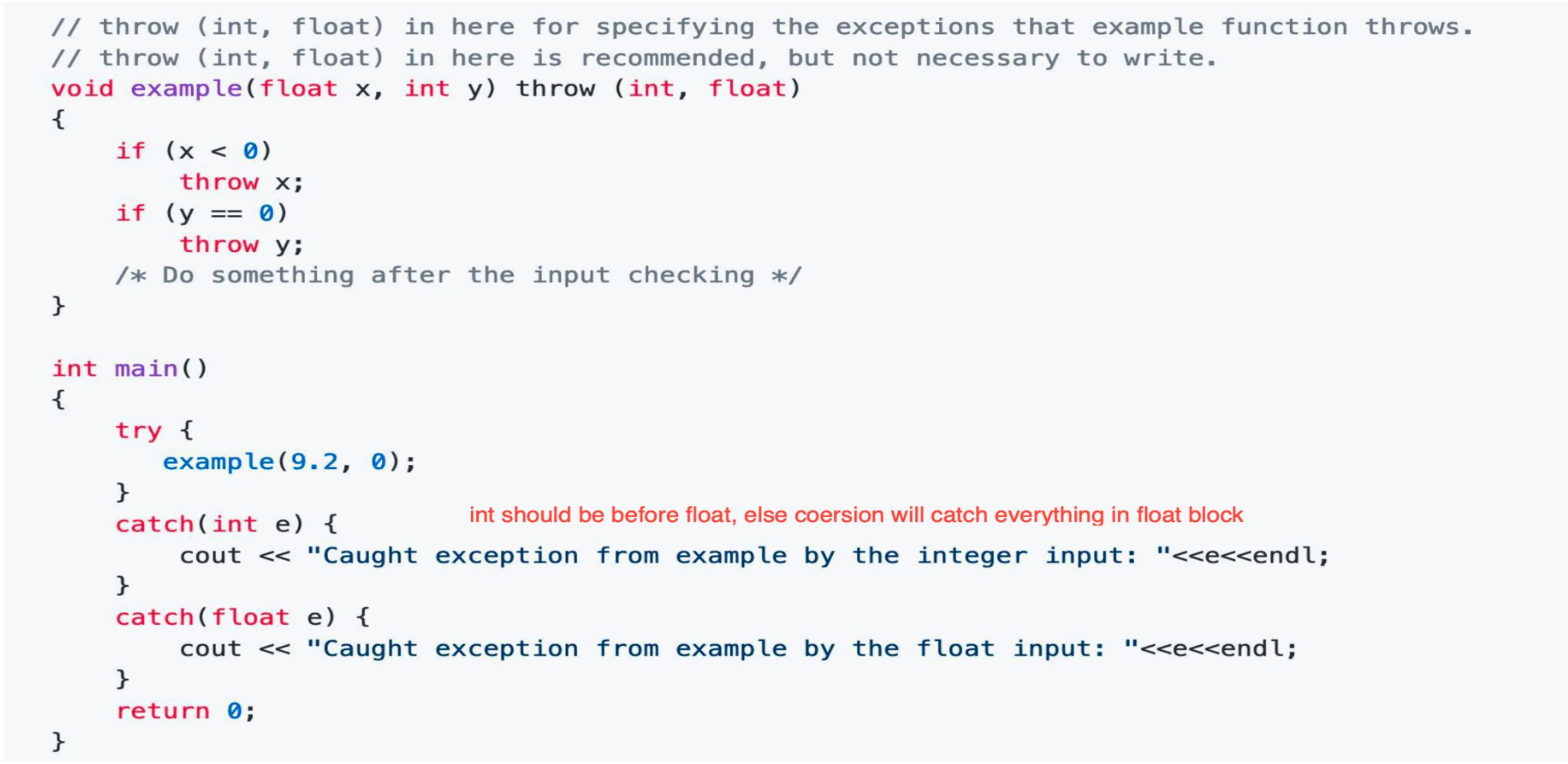
Exception in C++ is intuitively used to handle abnormal, unpredictable or erroneous conditions.

If an exception is thrown and not caught anywhere, the program terminates abnormally.



```
try {
        // Code that might throw an exception.

        // If programmer wish to raise an exception, use the throw keyword. E.g.
        throw something
}
catch ( <ExceptionType> <variable_name> ) {
        // Handle an exception that matches the type and binds that exception to a variable name.
}
catch (const std::exception& e) {
        // Handle an exception object that inherited by the exception class
}
catch (...) {
        // Catch all types of exception, not already caught by a catch block before
        // Sometime we say a "Default" exception
}
```



```
// throw (int, float) in here for specifying the exceptions that example function throws.
// throw (int, float) in here is recommended, but not necessary to write.
void example(float x, int y) throw (int, float)
{
    if (x < 0)
        throw x;
    if (y == 0)
        throw y;
    /* Do something after the input checking */
}

int main()
{
    try {
       example(9.2, 0);
    }
    catch(int e) {
        cout << "Caught exception from example by the integer input: "<<e<<endl;
    }
    catch(float e) {
        cout << "Caught exception from example by the float input: "<<e<<endl;
    }
    return 0;
}
```

int should be before float, else coersion will catch everything in float block

Java

Flow of control in try-catch-finally block is:

-> If an exception occurs in the try block, then the control is transferred to the catch block immediately.

-> Once the execution of the catch block is finished, then finally block is executed (if exists).

-> If no catch block handles the exception, then finally block is executed (if exists).

-> If an exception does not occur in the try block, then the control is transferred to either the finally block (if exists) or to the rest of the program.

```
try {
    // Normal execution path.
    // If programmer wish to raise an exception, use the throw keyword. E.g.
    throw new ExceptionType();
} catch (ExceptionType exception_name) {
    // Deal with the ExceptionType.
} finally {
    // Always executes this block when leaving the try block, regardless of whether any exceptions were thrown
    // Often used to clean up and close resources such a file handles.
}
```

Java

Handling rules in Java

-> finally clause is an option to present.

-> No code should be present between try, catch, and finally blocks.

Checked and Unchecked exceptions

-> Checked exceptions are checked at the compile time. If an exception is checked, the corresponded method must either handle it or specify the exception.

-> Unchecked exceptions are not checked at the compile time, so they are not forced by the compiler to handle or specify.

```java
public static void checkedExcept() throws IOException {
    // Checked exception must either handle this exception or throw it to the caller.
    FileReader file = new FileReader("somefile.txt");
}

public static void uncheckedExcept() {
    int a[] = new int[10];
    a[11] = 9;
}

public static void main(String []args)  {
    try {
        checkedExcept();
    }
    catch (IOException e) {
        System.out.println ("File Not Found");
    }
    try {
        uncheckedExcept();
    }
    catch(ArrayIndexOutOfBoundsException e){
        System.out.println ("Array Index is Out of Bounds");
    }
}
```

Java:

Try-with-resources (aka automatic resource management)

-> Resource: A resource in Java is an object that must be closed after we no longer use it. For instance, FileReader, BufferedWriter, etc.

-> Def. it is a try statement that declares some resources, and this statement ensures that each resource is closed at the end of the try block.

```java
try (BufferedReader br = new BufferedReader(new FileReader("file_name.txt"))) {
        System.out.println(br.readLine());
}
```

```
 is equivalent to:
```

```java
BufferedReader br = new BufferedReader(new FileReader("file_name.txt"));
try {
        System.out.println(br.readLine());
} finally {
        if (br != null) br.close();
}
```

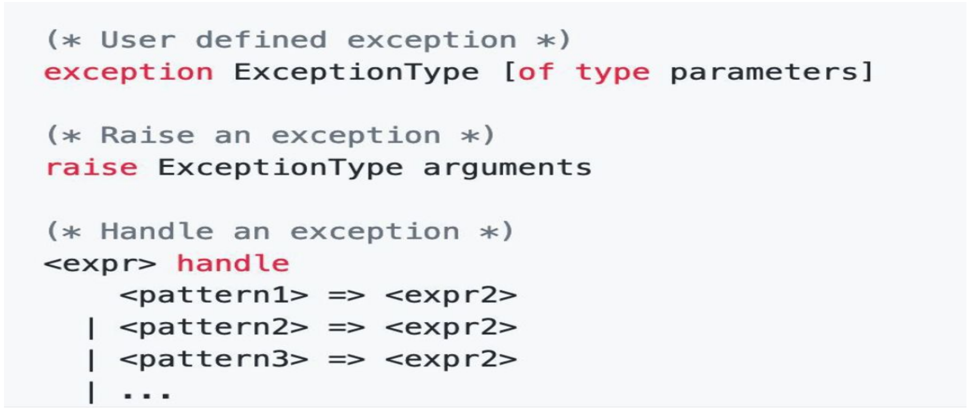
SML:

- In SML, we have two options to handle the failing computation.

○ option type: use NONE to signal the failure.

This has some limitations. What if the "exception" condition is more complex than option?

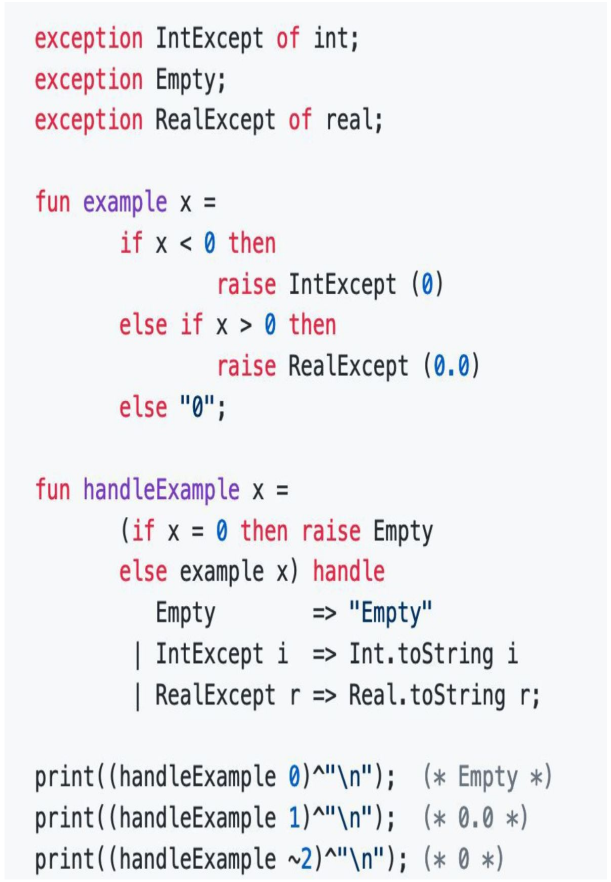
- exception: use exception handler instead.



```
(* User defined exception *)
exception ExceptionType [of type parameters]

(* Raise an exception *)
raise ExceptionType arguments

(* Handle an exception *)
<expr> handle
      <pattern1> => <expr2>
    | <pattern2> => <expr2>
    | <pattern3> => <expr2>
    | ...
```



```
exception IntExcept of int;
exception Empty;
exception RealExcept of real;

fun example x =
       if x < 0 then
              raise IntExcept (0)
       else if x > 0 then
              raise RealExcept (0.0)
       else "0";

fun handleExample x =
       (if x = 0 then raise Empty
       else example x) handle
         Empty        => "Empty"
        | IntExcept i  => Int.toString i
        | RealExcept r => Real.toString r;

print((handleExample 0)^"\n");  (* Empty *)
print((handleExample 1)^"\n");  (* 0.0 *)
print((handleExample ~2)^"\n"); (* 0 *)
```