Errors and Exceptions

Traditional way:

```java
double MyDivide (double num, double den)
{
    if (den == 0.0)
        // Indicate that an error occurred
    else
        return num/den;
}
```

* Need to communicate to the calling code
* If the calling function doesn't know how to resolve this error, it should propagate it further up.

```java
double MyLazyDivide (double num, double den)
{
    return MyDivide (num, den);
}
```

* Dealing with Errors:

```java
double MyLazyDivide (double num, double den)
{
    MyDivide (num, den);
    if (MyDivide() indicated an error)
        // Propagate that error upwards??
    else
        // Return result of call to MyDivide()
}
```
Traditional ways

1. Check return values → doesn’t always work. What return value indicates “bad” result for `MyDivide()`?

2. Set a global variable
   → who cares to check the globals for every function call?

3. Pass a parameter (via pointer) into the function which indicates the error condition.
   → Awkward

4* - Drawback: Code used to identify errors is mixed with the code to resolve errors.

C++ uses Exceptions

- Exception is an object
- Process of transferring exception object from called function to the calling function is called throwing an exception.
- Exception are received by the calling function which is said to have caught the exception

```cpp
int main() {
    double x, y;
    cin >> x >> y;
    cout << MyDivide(x, y) << endl;
}
```
what happens when an exception is thrown?

Syntax

Three features for exception handling:

1. Create and throw (use throw keyword)
2. Catch and resolve (use catch keyword)
3. Separate the handling logic from the rest of the code (use try keyword)

Syntax:

throw <object>; // object could be a primitive
  e.g. throw 15;
  e.g. throw Exception();

  e.g. double MyDivide(double num, double den)
  
  if (den == 0.0)
  
  throw String("The denominator is zero");
  
  else
  
  return num/den;

Creating a Class for Exception:

class MyExceptionClass
  
  & public:
  
  MyExceptionClass();
  int getA(); int getB(); ~MyExceptionClass();

  & private:
  
  int a; int b;
In our code:

```cpp
int x, y;

if ("something exceptional happens")
    throw MyExceptionClass(x, y);
```

* C++ provides some standard exception classes.

---

**The try/catch block**

```cpp
try
    // code that may throw an exception
catch (Type & Exception)
    // code that resolves that exception
```

* Algorithm code is cleanly separated from error-handling code.
  - Algorithm code: try block
  - Error-handling code: catch block(s)

* You can catch multiple exceptions differing by types.

* A semi-colon follows the final catch block.
Example:

```cpp
int main()
{
    double x, y;  // double result;
    cin >> x >> y;
    try
    {
        result = MyDivide(x, y);
    }
    catch (string)
    {
        // do something here
    }
    cout << result;
}
```

What to do in the catch block?

```cpp
int main()
{
    double x, y, result;  // bool success;
    do
    {
        success = true;
        cin >> x >> y;
        try
        {
            result = MyDivide(x, y);
        }
        catch (string& s)
        {
            cout << s << endl;
            success = false;  // continue;
        }
    } while (success == false)
    cout << result;
}
```
Exception Matching

* Exception handling system examines the type(s) contained in the catch blocks in the order in which they are listed.

* There is no automatic conversion (float will not convert to an int or vice versa)

* However, an object or reference of base class will match a derived class object. (throwing a Grad will match a Student)

try {
    catch (Student & s) { ... } // every try matches first type.
    catch (Grad g) { ... }
    catch (Ugrad ug) { ... }
    catch (All aug) { ... }
}

Use of reference in catch

1. Efficiency?

2. Object slicing in case the exception is rethrown.

Catching ALL exceptions

catch (...) { ... } // should be the last catch block
Retrying Exceptions

```cpp
try {
    ...}
catch (SomeException & e)
{
    cout << "An exception was thrown" << endl;
    throw; // re-throws the exception
}
```

Behind the Scenes
- Objects that were automatically allocated will be deallocated.
- Any objects whose constructor was completed will have its destructor called.

Q: What if no one catches the exception in function hierarchy?
A: `terminate()` is called.
  * You can call `set_terminate()` as a replacement function.
C++ Exception Classes

```cpp
#include <exception>

using namespace std;

try { // your code
  catch (exception e)
  {
    cout << e.what() << endl; // what() is a virtual method which can be overloaded.
  }
}
```

```cpp
#include <stdexcept> // many standard exceptions

(also #include <exception>)
```

```
Exception

runtime_error             logic_error
                        / | \
beam & them have constructors taking string arguments.

range_error             bad_alloc
                        / | \

overflow_error
```

Ex:
```
double MyDivide(double num, double den)
{
  if (den == 0.0)
    throw invalid_argument("den is zero");
  else
    return num/den;
}
```
Declaring Exceptions

```java
void MyFunction(Arguments) throw (type1, type2, ...);
```

```java
double MyDivide(double num, double den)
    throw (invalid_argument)
```

Q: Can we throw something that we did not declare?

A: Yes, but unexpected() is called whose default behavior is to call the terminate() function. You can also define set-unexpected().

Caveats:
- Class D derives from B
  - For a given method
    - D must not increase the types of exceptions beyond what are declared in B
    - D is allowed to reduce the number of exception types.

Why? Note that D object can be accessed through a B reference/pointer.

- It is okay to declare a derived exception thrown from the derived class.