Exceptions

How do we handle errors?

- Consider the following function:

```
// compute the roots of a second order polynomial a x^2 + b x + c = 0
pair<double, double> roots(double a, double b, double c)
{
    double delta = b * b - 4.0 * a * c;
    double disc = sqrt(delta);  // this can cause an error if (delta < 0)
    double a2 = 2 * a;

    return make_pair((-b + disc) / a2, -(b + disc) / a2);
}
```

- How do we tell the caller that something went wrong?

Using Error Codes (old way)

```
enum ErrorCodes {NoError, NegativeDiscriminant, SomeOtherError };

// compute the roots of a second order polynomial a x^2 + b x + c = 0
ErrorCodes roots(double a, double b, double c, pair<double, double>& result)
{
    double delta = b * b - 4.0 * a * c;
    if (delta < 0) return NegativeDiscriminant;
    double disc = sqrt(delta);  // this can cause an error if (delta < 0)
    double a2 = 2 * a;

    result = make_pair((-b + disc) / a2, -(b + disc) / a2);
    return NoError;
}

// caller function
...
if (NoError == roots(a, b, c, result))
    // use result
else
    // manage the error (e.g. close the app, try other values, display message, ask user to
    correct input,...)
...
```

- Note lot of C libraries (e.g. LAPACK) use `#define` for error codes instead of `enum`

How to add messages?

```
 // note the extra element
enum ErrorCodes {NoError, NegativeDiscriminant, SomeOtherError, ErrorCodesEnd };

const char * ErrorMessags[ErrorCodesEnd] = // same size as (number of enums - 1)
    { ""                                                    // NoError
    , "the given coefficients imply a negative discriminant"    // NegativeDiscriminant
    , "some other error"                                    // SomeOtherError
    };

ErrorCodes roots(double a, double b, double c, pair<double, double>& result)
{  /* body */ }


// caller
...
ErrorCodes retCode = roots(a, b, c, result);
if (NoError == retCode)
    // use result
else {
    cout << "Houston we have a problem: " << ErrorMessages[retCode] << "\n";
    // manage the error (e.g. close the app, try other values, ask user to correct input,...)
}
...
```

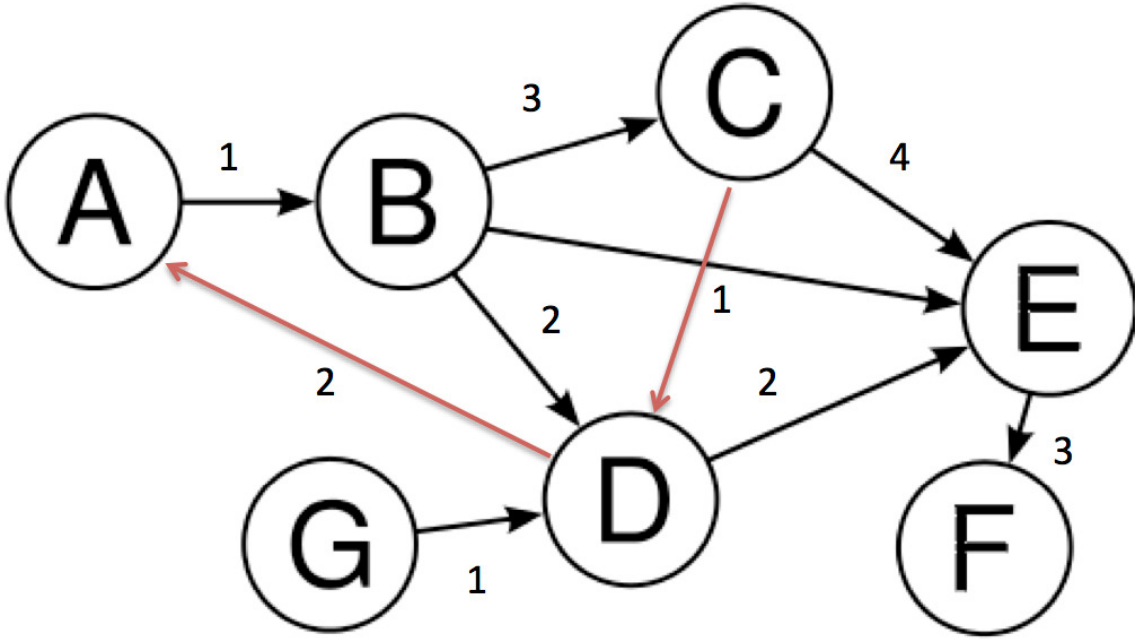
- We store the error messages in a global static array
- Getting custom messages, embedding custom info on the failure, is also possible, although more complicate

It gets complicated!

```
ErrorCodes foo(int a, int& res) { ... }
ErrorCodes bar(int b, int& res) { ... }

ErrorCodes zoo(int a, int& res)
{
    int res_foo, res_bar, err_code;

    err_code = foo(a, res_foo);
    if (err_code != NoError)
        return err_code;

    err_code = bar(res_foo, res_bar);
    if (err_code != NoError)
        return err_code;

    // do something with res_foo and res_bar
    // if another error occurs return the proper error code

    // all good
    return NoError;
}
```

- In zoo I have too manage errors from foo, bar or from zoo itself. We have to write lot of extra code.

And more complicated!



- The above represent a possible calling graph. Nodes are function. Arrows are function calls.
- Entry point is G
- Imagine having to write code in every function to manage error of any possible sub-calls

Exception

- The C++ solution to this problem are exceptions
- It is a construct we use to surround a code block which accounts for the possibility that something can go wrong, either in that function or in any other function called from that code block
- http://www.cplusplus.com/doc/tutorial/exceptions/

Exceptions

```
try
{
    throw 20;
}
catch (int e)
{
    cout << "An exception occurred. Exception Nr. " << e << '\n';
}
```

- The problematic code block is surrounded in a `try {...}` block and the error handling happens in a `catch(...) {...}` block
- An error is reported by throwing an exception using the keyword `throw`.
- When an exception is thrown, flow control is passed to the matching exception handler
- All objects belonging to the try{} code block, go out of scope and gets destroyed

Multiple handler

```
try {
    if (problem1)
        throw 20;
    ...
    if (problem2)
        throw 2.0;
    ...
    if (problem3)
        throw “something else”;
}
catch (int param) { cout << "int exception"; }
catch (double param) { cout << "double exception"; }
catch (...) { cout << "default exception"; }
```

- Exception handlers can be concatenated
- Handler type matching is evaluated in sequence
- The (...) catches any residual exception.

Handling can happen in the caller function

```
// compute the roots of a second order polynomial a x^2 + b x + c = 0
pair<double, double> roots(double a, double b, double c)  // keep clean signature
{
    double delta = b * b - 4.0 * a * c;
    if (delta < 0)
        throw "negative discriminant“;
    double disc = sqrt(delta);  // this can cause an error if (delta < 0)
    double a2 = 2 * a;

    return make_pair((-b + disc) / a2, -(b + disc) / a2);
}

// caller
try {
    result = roots(a, b, c);
}
Catch(const char *msg) { // handle possible errors occurred in roots
    cout << "Houston we have a problem: " << msg << "\n";
    // manage the error (e.g. close the app, try other values, ask user to correct
    input,...)
}
```

Any type can be thrown

```
struct BadIntException  // create my own exception class
{
    BadIntException(int x) : m_x(x) {}
    int m_x;
};

void test(int y)
{
    if (y < 0)
        throw BadIntException{ y };
}

void foo(int y)
{
    try {
        test(y);
    }
    catch(const BadIntException& b)
    {
        cout << “Bad integer: " << b.m_x << “\n";
    }
}
```

- Typically we use std exception classes

A base class handler matches a derived class

```
struct A{};
struct B : A {}

try {
    if (problem1)
        throw A{};
    ...
    if (problem2)
        throw B{};
    ...
}
catch (A& a) { cout << "A exception"; }  // this intercepts both A and B exceptions
catch (B& b) { cout << "B exception"; }  // this will never be used
```

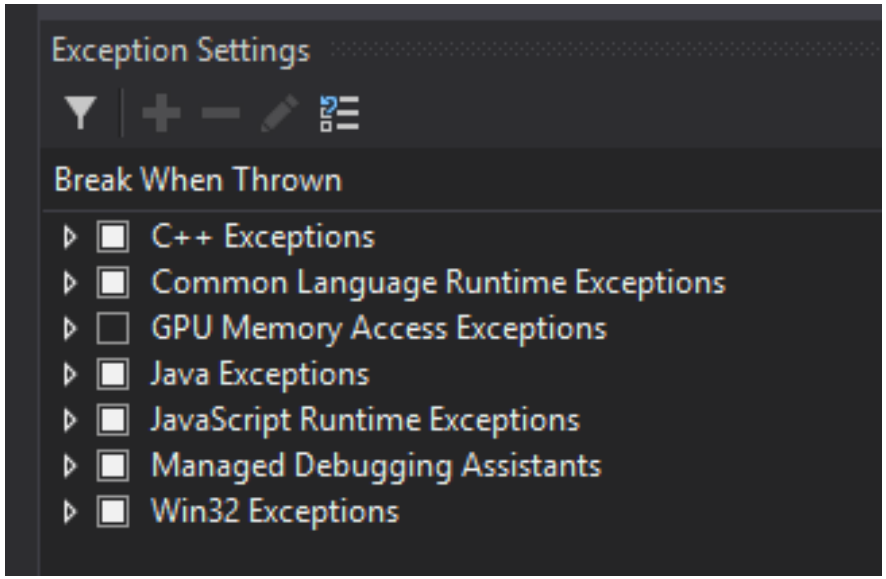
Exceptions must be handled or cause a crash

```
struct A{};
struct B{};
struct C{};

void foo(int x)
{
    if (x>10) throw A{};
    else if (x < -10) throw B{};
    else throw C{};
}

void bar(int x)
{
    try {
        foo(x);
    }
    catch (A& a) { cout << "A exception"; }  // catch A exceptions, others pass through
}

int main()
{
    try { foo((rand(100) % 50) - 25 ); }
    catch (B& a) { cout << “B exception"; }  // catch B exceptions, others pass through
    catch (...) { cout << “Some other exception"; }  // catch everything else. No special handler for C

    return 0;
}
```

- Always put a catch(...) in your main, to catch any unhandled exception and avoid a crash

debugger

- You can ask to the debugger to break when an exception occurs
- Use menu: Debug/Window/ExceptionSettings



STL exceptions

- Defined in: `#include <exception>`
- https://en.cppreference.com/w/cpp/error/exception

- And you can also inherits from those your own exception classes (quite rare: worth doing if you want to define special handling for a particular type of error)

STL exception example

- Most commonly used errors classes are:
  - `std::logic_error`
    - an error probably due to a programming bug occurred (e.g. We forgot to handle a valid case in a switch)
  - `std::invalid_argument`
    - an error probably due to invalid input arguments occurred (e.g. sqrt of a negative number)
  - `std:: runtime_error`
    - an error due to unexpected situation occurred (e.g. internet connection to some web server failed)
- They all inherit from `std::exception`, so catch(`std::exception&`) {} will catch any of these

Good Practice

- Throw as early as an error occur and only manage it later
- Use meaningful error messages: help users identify the problem with their inputs
- Unhandled exceptions will crash the app, so use try catch in main with a catch-all clause, to handle anything that escaped