Exceptions

C++ exceptions are similar to Java exceptions
Exceptions are thrown by a throw statement
Exceptions are caught by a catch clause called a handler
In C++ any type may be thrown: int, objects, pointers to objects etc.
You can even throw the thing you just caught

Syntax

Syntax of the throw statement

throw x;

or
throw; // this rethrows the current exception

Syntax of the try statement

try {
    ............
} catch( Type var ) {
    ............
} catch( ... ) { // catch any exception
    ............
}

or
X::X(X &x) try : a(1) {
    ............
} catch( Type var ) {
    ............
} catch( ... ) { // catch any exception
    ............
}

The second form will catch exceptions arising from the constructor initializers
using ... allows catching any exception but you can’t examine the exception
However you can rethrow it

Operation of throw and catch

Steps in throwing an exception

When an exception is thrown, control is transferred to the nearest handler with a matching type
A throw-expression initializes a temporary object which will be used to initialize the variable named in the matching handler
When the last handler being executed for the exception exits by any means other than throw; the temporary object is destroyed
A throw-expression with no operand rethrows the exception being handled
Once an exception is thrown, the function activations through which the exception is passed are terminated (more on this below).

Handling an exception

The exception-declaration in a handler describes the type(s) of exceptions that can cause that handler to be entered
A handler is a match for a throw-expression with an object of type E if the types match
The handlers for a try block are tried in order of appearance
That makes it possible to write handlers that can never be executed

A ... in a handler’s exception-declaration functions similarly to ... in a function parameter declaration - it specifies a match for any exception
If present, a ... handler shall be the last handler for its try block
If no match is found among the handlers for a try block, the search for a matching handler continues in a dynamically surrounding try block
An exception is considered handled upon entry to a handler and the stack will have been unwound at that point
The exception being handled is rethrown if control reaches the end of a handler of the function-try-block of a constructor or destructor
A function returns when control reaches the end of a handler for the function-try-block
At the end of execution of a catch phrase, execution continues past the end of the try-catch group
If the use of a temporary object can be eliminated without changing the meaning of the program, then the optional name can be bound directly to the temporary object specified in a throw-expression causing the handler to be executed

Specifications of what exceptions might be thrown

Syntax

int f() const throw(int, output_error) {
    ............
}

You do not have to state what gets thrown from a function, but if it is mentioned, it must be complete
Our guidelines tell you to declare the exceptions thrown
Use the keyword throw, not throws, and put the list of exception classes in parentheses
If an exception is thrown and passes through a function with an exception specification that is not listed then the function unexpected() is called

There is no library standard that states which classes may or may not be exceptions
You can thrown direct values, which may cause some copying
Destructors are called on any local (automatic) class-typed variables declared in the functions being terminated as the exception is sent up

Proper Use of Exceptions

Most people who write about exception use say that the name, “exception”, is a good guide for when to use it: only in exceptional, i.e., unexpected conditions
But also recall that exceptions are things that remain in the code when it is considered ready for release, unlike assertions
Exceptions and assertions ought to have distinct uses in program development
Exceptions are basically free in terms of run-time penalty when they are not used, but they are rather expensive when used
Here is what happens when an exception is thrown

Typically, the object to throw is not already in existence, so it must be created (“Foo(- -)” or “new Foo(- -)”). A run-time support function now searches the stack for try blocks.

For each try block found, the catch phrases are scanned. Each catch phrase must have its type compared to the type of the thrown object. For each try block encountered and each “scope” passed by, all automatic objects must have their destructors called, and the call stack must be cleaned up.

Are these good uses of exceptions?

A piece of hardware has failed.
Array index is out of bounds.

Out of memory.
Downcasting operation failed
Exit from a complex loop

for example, you’re playing Blackjack, and you run out of cards.

What about this definition of Exception

An exception is raised within a routine that is unable to fulfill the terms of its contract, for reasons beyond its control.

Geigel Notes

When things go wrong

When a program comes upon a problem it cannot solve locally it can:

- Terminate the program
- E.g. assertion
- Return an “error” value
- Return a valid value and leave the program in a “bad” state.
- E.g. iostreams
- Call some “error handling” functions

Enter…the exception

Exceptions allow a method to tell the caller when an error has occurred.

Many times it is the calling function that knows what to do when an error occurs.

Exceptions allow the caller to respond to the error rather than the method itself.

Different callers may wish to respond to particular errors differently.

C++ Exceptions

The idea behind C++ exceptions is very much like Java exceptions.

Like all things C++, though, C++ exceptions do have their quirks.

Throwing exceptions

In C++, exceptions are thrown by using the throw keyword. Unlike Java, there is not a Throwable class.

In C++, any item can be thrown:

- Basic datatypes (int, float, etc.)
- Class objects
- Pointers to class objects
- References to class objects

```cpp
class Stack {
public:
    bool isFull();
    void push();
private:
    int size;
    ...
};

void Stack::push() {
    ...
    if (isFull()) throw size;
}
```

Like in Java, it is more useful to create a hierarchy of Exception classes:

```cpp
class MathError { };
class Overflow : public MathError {};  
class Underflow : public MathError {};  
class DivideByZero : public MathError {};  
```

Exception classes are not special. They can contain methods/data like any other class.

```cpp
class MathError {
    //...
    virtual void printMessage() const;
};
```

They can also be derived from multiple classes:

```cpp
class NetfileError : public NetworkErr, public FileError { ... }
```
### Catching Exceptions

Like in Java, C++ uses a try/catch block for catching exceptions.

```cpp
void f() {
    try {
        // call to a method that may throw something
        } catch (Overflow) {
            // code that handles an overflow error
            ...
        }
    ...}
}
```

### Rules for catching exceptions:

- `try { // something of type E is thrown }`
- `catch (H) { // when is the handler invoked?}

**Handler is invoked if**

- `H` is the same type as `E`
- `E` is derived from `H`
- `H` and `E` are pointers and 1 & 2 apply to the things they point to.
- `H` is a reference and 1 & 2 hold for the type `H` refers to.

```cpp
void f() {
    throw MathError();
}

void g() {
    try { f ();
        } catch (MathError E) { E.printMessage(); }
    }

Copying will occur
void f() {
    throw Overflow();
}

void g() {
    try { f ();
        } catch (MathError E) { E.printMessage(); }
    }

Slicing Will Occur
void f() {
    throw new MathError();
}

void g() {
    try { f ();
        } catch (MathError E) { E.printMessage(); }
    }

No Slicing Will Occur
void f() {
    throw Overflow();
}

void g() {
    try { f ();
        } catch (MathError &E) { E.printMessage(); }
    }

To catch anything, regardless of type, use the ...
syntax.
try {
    // something
    } catch (...) { // catches anything thrown at you
    }
```

### Catching exceptions

Handlers in a try / catch block are tried in the order in which they appear.

```cpp
try {// something
    } catch (Overflow) {
        // handle overflow
    } catch (MathError){
        // handle any math error
    } catch (...) {
        // handle anything
}
```

**Erroneous ordering:**

```cpp
try {// something
    } catch (...) { // handle anything
        } catch (MathError) { // it'll never get here
            } catch (Overflow) { // or here
            }
```

### Rethrowing exceptions

Once caught, an exception can be rethrown by using the throw keyword:

```cpp
try {
    // something
    } catch (...) { // handle anything
        // catches anything thrown at you
        // and throws it back
        throw;
    }
```

### Exception specification

Like Java, what gets thrown by a method can be declared when defining the function.
Unlike Java, this declaration is not required.

But if there, it is guaranteed to throw only what's specified.

```cpp
void f() throw (Overflow, int) {
    if ( ) throw Overflow();
    else throw 7;
}
```

If unspecified (default), the function may throw anything:

```cpp
int f(); // can throw anything
```

To indicate that a function will never throw an exception:

```cpp
int g() throw (); // throws nothing
```

In exception hierarchies, derived classes may only restrict what is thrown.

```cpp
class B {
    virtual void f(); // can throw anything
    virtual void g() throw (X, Y);
    virtual void h() throw (X);
};
class C : public B {
    virtual void f() throw (X); // ok
    virtual void g() throw (X); // ok
    virtual void h() throw (X,Y); // not okay
}
Stack unwinding
When an exception is thrown in C++
Call stack is searched for first function to catch the data
thrown.
If none found, program will terminate.
If one is found:
All local variables from all methods on stack from
method that threw the exception to that which caught it,
will have it’s destructor called.
Note that this is not true for objects allocated on the
heap.
If an exception is caught and handled
Execution continues from next statement after the try/catch
block.
Stuff *pointer( 0 );
try {
    Stuff direct( "direct" );
    pointer = new Stuff( "dynamic" );
direct.test();    // int thrown here
delete pointer;
} catch( int x ) {
    cout << "Exception #" << x << " caught" << endl;
delete pointer;
}

Stack unwinding
Advantageous to wrap system resource calls into class
objects.
The resource can be cleaned up during object destruction.
void use_file (const char *name) {
    FileObject f (…)
    // do something with f
}

Standard Exceptions
There are some standard exceptions
bad_alloc — thrown by new
bad_cast — thrown by dynamic_cast
bad_typeid — thrown by typeid
STL exceptions
out_of_range
invalid_argument
overflow_error
ios_base:failure
There is no guarantee or rule that forces one to derive
their exceptions from this hierarchy.

Enter…the exception
Exceptions allow a method to tell the caller when an
error has occurred
Many times it is the calling function that knows what to do
when an error occurs.
Exceptions allow the caller to respond to the error rather
than the method itself.
Different callers may wish to respond to particular errors
differently.

Exceptions and Constructors
If an exception is thrown during the call to an object’s
constructor, only the data members that have been
completely constructed are destroyed.
class X {
    Y yy;
    Z zz;
public:
    X ( char *foo, int bar ) :
        yy (foo), zz (bar) {}