Exceptions

**Exception**: a program error that occurs during execution, or a “signal” generated (“thrown”) when a program execution error is detected.

Exceptions may be thrown by hardware or software; we consider only the latter.

If a software exception is thrown, and an exception-handler code segment is in effect for that exception, then flow of control is transferred to the handler.

If there is no handler for the exception, the program will be terminated.

**Dodging an Error**

```cpp
void Rational::SetDenominator(int Denom) {
    if (Denom != 0) {
        DenominatorValue = Denom;
    } else {
        cerr << "Illegal denominator: " << Denom
            << ", using 1" << endl;
        DenominatorValue = 1;
    }
}
```

Frequently code will be designed to detect and avoid anticipated errors:

Here we see a simple test and response, all handled locally.

**Handling an Error with an Exception**

```cpp
void Rational::SetDenominator(int Denom) {
    try {
        if (Denom != 0) {
            DenominatorValue = Denom;
        } else {
            throw (Denom);
        }
    } catch (int d) {
        cerr << "Illegal denominator: " << d
            << ", using 1" << endl;
        DenominatorValue = 1;
    }
}
```

Here’s the same situation, handled now by throwing an exception:

On error: **throw** a value.

Catch thrown value, if any.
A *try block* is simply a compound statement preceded by the keyword `try`.

One, or more, of the statements in a try block can be a throw statement. A throw statement resembles a function invocation, with information regarding the detected error wrapped within parentheses.

A copy of the information in the throw statement may be passed via the throw statement to an exception handler that is keyed to the *type* thrown.

The value thrown may be of a simple type (as on the previous slide), or a more complex structured type, including an object.

That makes it possible to throw diagnostic information about the error.

### Remote catch

The exception may be caught in the function that called the one performing the throw:

```cpp
void Rational::Rational(int Numer, int Denom) {
    SetNumerator(Numer);
    try {
        SetDenominator(Denom);
    } catch (int d) {
        cerr << "Illegal denominator: " << d << "", using 1" << endl;
        SetDenominator(1);
    }
}
```

Alternatively, the exception may be caught further back up the call sequence.

### Stack Unwinding

If a function throws an exception, and does not catch it, then control is transferred to the calling function, which is now given an opportunity to catch the exception.

When the exception is caught, the catch block is executed and then the catching function may resume execution.

This process continues until either a function catches the exception or all calls have been unwound. In the latter case, the program is terminated.
Multi-Level Unwinding

```cpp
void createList(int* Array, int Size);
void main() {
    int* Array; int Dimension;
    try {
        createList(Array, Dimension);
    } catch (int e) {
        cerr << "Cannot allocate: " << e << endl;
        return;
    } catch (bad_alloc b) {
        cerr << "Allocation failed" << endl;
        return;
    }
}
```

Multi-Level Unwinding

```cpp
void createList(int* Array, int Size) {
    Size = getUserInput();
    try {
        Array = new int[Size];
    } catch (bad_alloc b) { // you can catch an exception
        throw (b); // and re-throw it
    }
}
```

Specifying Potential Throws

```cpp
int getUserInput() throw(int) {
    int Response;
    cout << "Please enter the desired dimension" << endl;
    cin >> Response;
    if (Response <= 0) {
        throw (Response); // caught in main()
    }
    return Response;
}
```

Thrown Value May Be an Object

```cpp
class BadDimension {}; int getUserInput();
void main() {
    int Value;
    try {
        Value = getUserInput();
    } catch (BadDimension e) {
        cerr << "User is an idiot." << endl;
        return;
    }
}
```

Exception thrown in a called function.

Warns user that a value may be thrown and also restricts what type may be thrown.

Thrown value is an object of a trivial class – this IS legal.
Throwing an Object

```cpp
int getUserInput() {
    int Response;
    cout << "Please enter the desired dimension" << endl;
    cin >> Response;
    if (Response <= 0) {
        BadDimension e;  // declare trivial object
        throw (e);       // throw it
    }
    return Response;
}
```

Throwing a Useful Object

```cpp
class BadDimension {
private:
    string Msg;
public:
    BadDimension() {Msg = "Unspecified";}
    BadDimension(string m) {Msg = m;}
    string getMsg() {return Msg;}
};
```

We may also design a class to store more specific information, such as:

```cpp
int getUserInput() {
    int Response;
    cout << "Please enter the desired dimension" << endl;
    cin >> Response;
    if (Response <= 0) {
        BadDimension e("no space requested");
        throw (e);
    }
    if (Response < 0) {
        BadDimension e("negative allocation requested");
        throw (e);
    }
    if (Response > 1000000) {
        BadDimension e("excessive space requested");
        throw (e);
    }
    return Response;
}
```

Throwing a Useful Object

Now the encapsulated message may be displayed or even analyzed:

```cpp
void main() {
    int Value;
    try {
        Value = getUserInput();
    } catch (BadDimension e) {
        cerr << "BadDimension: " << e.getMsg() << endl;
        return;
    }
    return;
}
```

...of course more useful information could also be incorporated...

```cpp
... of course more useful information could also be incorporated...
```
Inheritance in Exceptions

You may also create a hierarchy of exception classes, taking advantage of the type-conversion (and even polymorphic behavior) discussed earlier.

![Exception Hierarchy Diagram]

Now a catch looking for a BadDimension object would catch any of the derived type exceptions as well.

Handy trick for allowing for code libraries to have an extensible, internal scheme of exceptions without breaking client code.

Stack with Exceptions

In the Stack implementation, we can throw objects of the appropriate type, based upon the error that just occurred.

The inheritance hierarchy allows some flexibility in specifying the type that is to be thrown:

```cpp
int Stack::Pop() throw (StackException) {
    if ( (Top > 0) && (Top < Capacity) ) {
        Top--;
        return Stk[Top];
    }
    throw StackUnderflow();
    return 0;
}
```

Legal throw due to inheritance relationship between StackUnderflow and StackException.

Stack with Exceptions

Here, we could just allow `new` to throw a `bad_alloc` exception, but the use of a custom exception hierarchy simplifies the interface and the catch logic…

```cpp
bool Stack::Push(int toInsert) throw (StackException) {
    if (Top == Capacity) {
        int* tmpStk = new(nothrow) int[2*Capacity];
        if (tmpStk == NULL) {
            throw StackOverflow();
        }
        for (int Idx = 0; Idx < Capacity; Idx++) {
            tmpStk[Idx] = Stk[Idx];
        }
        delete [] Stk;
        Stk = tmpStk;
        Capacity = 2*Capacity;
    }
    Stk[Top] = toInsert;
    Top++;
    return true;
}
```
We can first consider specific exceptions and then have a final, generic catch:

```cpp
void main() {
    Stack s1(5);
    s1.Push(99); s1.Push(345); s1.Push(235);
    for (int Idx = 0; Idx < 5; Idx++) {
        try {
            s1.Pop();
        } catch (StackUnderflow) {
            cout << "Error: stack underflow" << endl;
        } catch (StackException) {
            cout << "Error: unclassified stack exception" << endl;
        }
    }
}
```