

Using C++ Exceptions

Exceptions 1

Exceptions

- Dodging an Error
- Handling an Error with an Exception
- C++ try-catch Mechanism
- throw Not Caught Locally
- Remote catch
- Stack Unwinding
- Multi-Level Unwinding
- Specifying Potential Throws
- Thrown Value May Be an Object
- Throwing an Object
- Throwing a Useful Object
- Inheritance in Exceptions
- Stack with Exceptions

Exceptions

Exceptions 2

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

Exceptions 3

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

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

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

Handling an Error with an Exception

Exceptions 4

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

```
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;  
    }  
}
```

try block...

On error: throw a value.

catch thrown value, if any.

C++ try-catch Mechanism

Exceptions 5

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.

throw Not Caught Locally

Exceptions 6

An exception may be thrown in one function and caught in another:

```
void Rational::SetDenominator(int Denom) {  
    if (Denom != 0) {  
        DenominatorValue = Denom;  
    }  
    else {  
        throw (Denom);  
    }  
}
```

no try block this time

On error: **throw** a value.

The thrown value, if any, must be caught elsewhere.

Where? Resolved via the runtime stack's record of the call sequence.

Remote catch

Exceptions 7

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

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

Call may result in a **thrown** value, so we wrap it in a try block.

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

Stack Unwinding

Exceptions 8

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

Exceptions 9

```

void createList(int* Array, int Size);
int getUserInput( );

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;
    }
}

```

Exception thrown in a called function.

Computer Science Dept Va Tech January 2000 OO Software Design and Construction ©2000 McQuain WD

Multi-Level Unwinding

Exceptions 10

```

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
    }
}

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

```

Computer Science Dept Va Tech January 2000 OO Software Design and Construction ©2000 McQuain WD

Specifying Potential Throws

Exceptions 11

The potential for a function to throw an exception may be explicitly shown:

```

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;
}

```

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

Computer Science Dept Va Tech January 2000 OO Software Design and Construction ©2000 McQuain WD

Thrown Value May Be an Object

Exceptions 12

In the simplest case, we may declare a trivial class simply to throw instances of it:

```

class BadDimension { };

int getUserInput( );

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

```

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

Computer Science Dept Va Tech January 2000 OO Software Design and Construction ©2000 McQuain WD

Throwing an Object

Exceptions 13

```
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;  
}
```

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

Throwing a Useful Object

Exceptions 14

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

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

Object can store a relevant message.

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

Throwing a Useful Object

Exceptions 15

Now the encapsulated message may be displayed or even analyzed:

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

Catch object and display message --- useful for localizing and classifying errors.

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

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

Throwing a Useful Object

Exceptions 16

```
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;  
}
```

Set appropriate message, depending on error found.

Computer Science Dept Va Tech January 2000

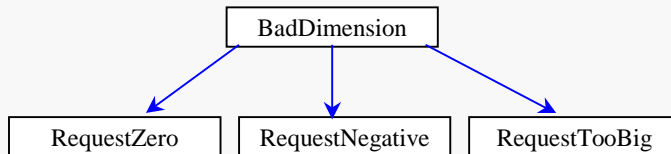
OO Software Design and Construction

©2000 McQuain WD

Inheritance in Exceptions

Exceptions 17

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



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.

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

Stack with Exceptions

Exceptions 18

```
class StackException {};  
class StackOverflow : public StackException {};  
class StackUnderflow : public StackException {};
```

```
#include "StackExceptions.h"  
  
class Stack {  
private:  
    int Capacity; // stack array size  
    int Top; // first available index  
    int* Stk; // stack array  
public:  
    Stack(int InitSize) throw (StackException);  
    bool Push(int toInsert) throw (StackException);  
    int Pop() throw (StackException);  
    int Peek() throw (StackException);  
    bool isEmpty() const;  
    bool isFull() const;  
    ~Stack();  
};
```

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

Stack with Exceptions

Exceptions 19

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:

```
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.

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

Stack with Exceptions

Exceptions 20

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...

```
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;  
}
```

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

We can first consider specific exceptions and then have a final, generic catch:

```
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;
        }
    }
}
```