

Exceptions

Exceptions 1

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.

Computer Science Dept Va Tech March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

Dodging an Error

Exceptions 2

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.

Computer Science Dept Va Tech March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

Handling an Error with an Exception

Exceptions 3

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.

Computer Science Dept Va Tech March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

C++ try-catch Mechanism

Exceptions 4

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, or a call to a function that contains 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.

Computer Science Dept Va Tech March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

throw Not Caught Locally

Exceptions 5

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.

Computer Science Dept Va Tech March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

Remote catch

Exceptions 6

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
            << "\n, 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.

Computer Science Dept Va Tech March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

Stack Unwinding

Exceptions 7

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.

H()
 G()
 F()
 main()

runtime stack

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.

Computer Science Dept Va Tech March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

Multi-Level Unwinding

Exceptions 8

```
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 March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

Multi-Level Unwinding

Exceptions 9

```

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 March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

Specifying Potential Throws

Exceptions 10

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 March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

Thrown Value May Be an Object

Exceptions 11

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 March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

Stack with Useful Exceptions

Exceptions 12

```

class StackException {
private:
    string Msg;
public:
    StackException(string M = "unspecified");
    string getMessage() const;
};

class Stack {
private:
    int Capacity; // stack array size
    int Top; // first available index
    string* Stk; // stack array
public:
    Stack(int InitSize = 0) throw (StackException);
    bool Push(string toInsert) throw (StackException);
    string Pop() throw (StackException);
    bool isEmpty() const;
    bool isFull() const;
    ~Stack();
};
    
```

Computer Science Dept Va Tech March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

Using Exceptions

Exceptions 13

Exceptions are particularly useful in library code, such as generic data structures and algorithms.

The library will throw an exception and allow the client code to catch it and deal with it in a problem-specific context.

A thrown exception will only be caught if there is a catch block whose function is on the runtime stack and which is looking for an exception of the type that has been thrown.

`catch(...)` will catch an exception of ANY type.

Caught exceptions can be re-thrown if desired.

Control does NOT return to the point where the exception was thrown.

Computer Science Dept Va Tech March 2001

OO Software Design and Construction

©2001 McQuain WD & Keller BJ

Stack with Exceptions

Exceptions 14

In the `Stack` constructor implementation, we can throw an object holding an appropriate message, if the initial allocation fails:

```
Stack::Stack(int InitSize) throw (StackException) {
    if (InitSize <= 0) {
        Capacity = Top = 0;
        Stk = NULL;
        return;
    }
    Capacity = InitSize;
    Top = 0;
    Stk = new(nothrow) string[InitSize];
    if (Stk == NULL) {
        throw (StackException("stack allocation failed"));
    }
}
```

Computer Science Dept Va Tech March 2001

OO Software Design and Construction

©2001 McQuain WD & Keller BJ

Stack with Exceptions

Exceptions 15

In the `Stack::Pop()` implementation, we can throw an object holding an appropriate message, if the stack is empty:

```
string Stack::Pop() throw (StackException) {
    if ( (Top > 0) && (Top < Capacity) ) {
        Top--;
        return Stk[Top];
    }
    throw StackException("stack underflow");
    return string("");
}
```

Computer Science Dept Va Tech March 2001

OO Software Design and Construction

©2001 McQuain WD & Keller BJ

A Memory-management Exception

Exceptions 16

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

```
bool Stack::Push(string toInsert) throw (StackException)
{
    if (Top == Capacity) {
        string* tmpStk = new(nothrow) string[2*Capacity];
        if (tmpStk == NULL) {
            throw StackException("stack overflow");
        }
        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 March 2001

OO Software Design and Construction

©2001 McQuain WD & Keller BJ

Driver

Exceptions 17

```
int main() {
    const int Size = 10;
    Stack s1(Size);

    s1.Push("First");
    s1.Push("Second");
    s1.Push("Third");
    s1.Push("Fourth");

    for (int Idx = 0; Idx < Size; Idx++ ) {
        try {
            s1.Pop();
        }
        catch (StackException e) {
            cout << "Error: " << e.getMessage() << endl;
            cout << " occurred calling s1.Pop() in main()"
                << " with index " << Idx << endl;
            return 1;
        }
    }
    return 0;
}
```

Computer Science Dept Va Tech March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ

Conclusions

Exceptions 18

Do not view exceptions as simply another control mechanism — the cost of a throw/catch action is too high and the alteration of flow control is too difficult to understand. (It's almost as bad as a `goto`.)

Design your exceptions to provide useful information; every thrown exception needs to be chased back to a generating error, so it's useful to know exactly where the exception was thrown, triggering values of local variables and/or parameters, etc.

It is very common to design a hierarchy of exception classes, using inheritance. We will examine that later...

Computer Science Dept Va Tech March 2001 OO Software Design and Construction ©2001 McQuain WD & Keller BJ