

Chapter 12

C++ Exceptions

1

Exceptions

An exception is a program error that occurs during execution.

If an exception occurs, 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 terminate.

2

Dodging an Error

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

3

Handling an Error with 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.

4

C++ try-catch Mechanism

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.

5

throw not caught locally

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

no **try** block this time

On error: **throw** a value.

thrown value, if any, must be caught elsewhere.

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

6

Remote catch

```
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.

7

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.

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

8

Multi-level Unwinding

```
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;
    }
}
```

9

Multi-level Unwinding

```
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
    }
}
```

10

Multi-level Unwinding

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

11

Specifying potential throws

```
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 what may be thrown and also restricts what may be thrown.

12

Thrown value may be an object

```
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.

13

Thrown value may be an object

```
int getUserInput( ) {

    int Response;

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

    return Response;
}
```

14

Throwing a useful object

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

Object can store a relevant message.

15

Throwing a useful object

```
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.

16

Throwing a useful object

```
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.

17

Inheritance in Exceptions

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

18