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.

©2002 McQuain WD & Keller BJ

Dodging an Error

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.

©2002 McQuain WD & Keller BJ

Handling an Error with an Exception

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.

 ©2002 McQuain WD & Keller BJ

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, 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.

©2002 McQuain WD & Keller BJ

throw Not Caught Locally

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.

©2002 McQuain WD & Keller BJ

Remote catch

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.

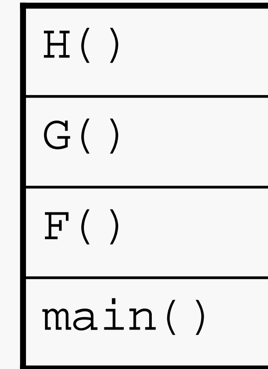
 ©2002 McQuain WD & Keller BJ

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.

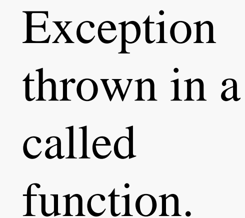


runtime stack

©2002 McQuain WD & Keller BJ

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;
   }
}
```



©2002 McQuain WD & Keller BJ

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
   }
}
```

```
int getUserInput( ) {

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

 ©2002 McQuain WD & Keller BJ

Specifying Potential Throws

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.

©2002 McQuain WD & Keller BJ

Thrown Value May Be an Object

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

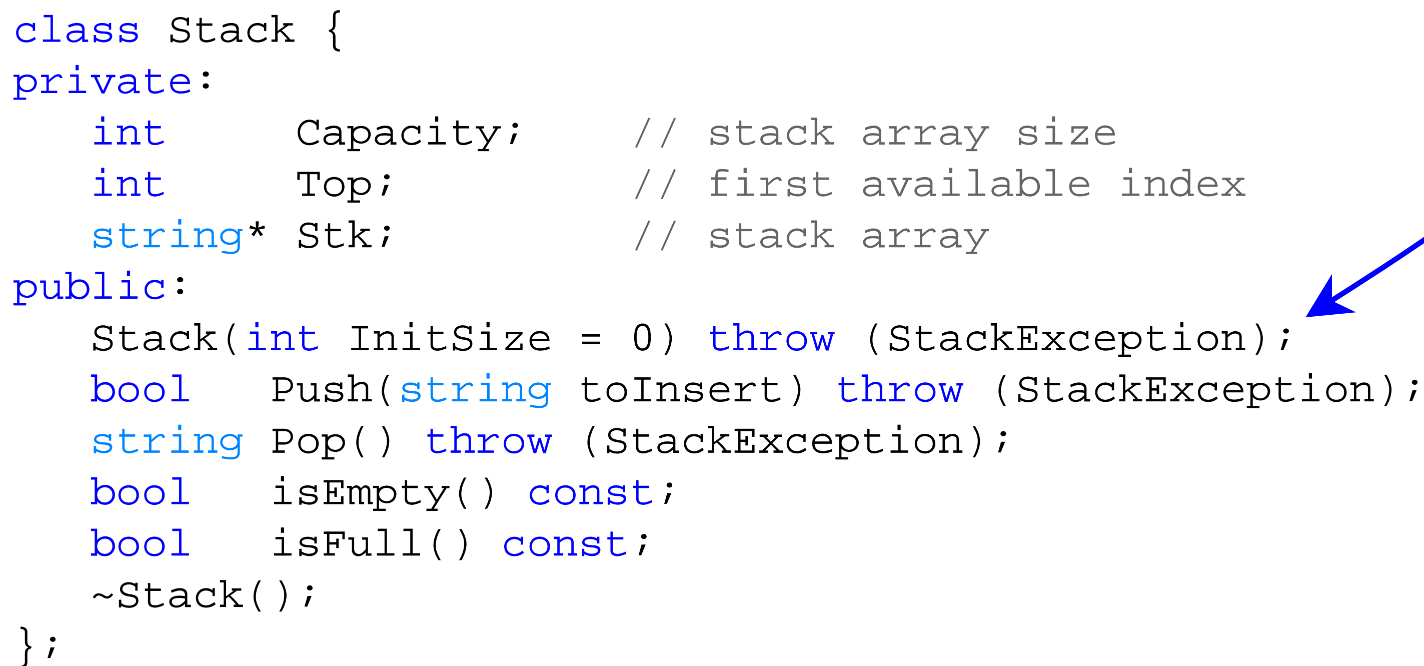
```
class BadDimension { };

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.

©2002 McQuain WD & Keller BJ

Stack with Useful Exceptions


```
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();
};
```


©2002 McQuain WD & Keller BJ

Using Exceptions

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.

©2002 McQuain WD & Keller BJ

Stack with Exceptions

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"));
   }
}
```


©2002 McQuain WD & Keller BJ

Stack with Exceptions

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("");
}
```


©2002 McQuain WD & Keller BJ

A Memory-management Exception

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;
}
```


©2002 McQuain WD & Keller BJ

Driver

```
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;
}
```

Conclusions

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…

©2002 McQuain WD & Keller BJ

Exceptions for Access Violations

The iterator for the DListT template may be modified to throw an exception when it an invalid instance of it is dereferenced:

```
class IllegalAccess {};                     // exception object
. . .

class iterator {
public:
   iterator() { Position = NULL; }
. . .

   T& operator*() {

      if ( Position == NULL )
         throw IllegalAccess();

      return (Position->Element);
   }
. . .
```

Now, dereferencing an invalid iterator will produce behavior consistent with pointers.

©2002 McQuain WD & Keller BJ

Exceptions for Arithmetic Errors

Exceptions may also be used to mimic aborted hardware errors:

```
#include <limits>
using namespace std;

class ArithmeticException {};
class NegativeRadicand    : public ArithmeticException {};
class DivideByZero        : public ArithmeticException {};
class ConvergenceFailure : public ArithmeticException {};

double SquareRoot(double A) throw(ArithmeticException) {

   if ( A < 0.0 )
      throw NegativeRadicand();

   int Iterations = 0;
   const int ITERATIONLIMIT = 100;

   double EPS_DBL = numeric_limits<double>::epsilon();
   double MIN_DBL = numeric_limits<double>::min();

. . .
```


©2002 McQuain WD & Keller BJ

Exceptions for Arithmetic Errors


```
. . .
   double Xk, Xkp1;

   Xkp1 = A;
   while ( fabs(Xkp1 * Xkp1 - A) > EPS_DBL ) {

      Xk = Xkp1;

      if ( fabs(Xk) < MIN_DBL )
         throw DivideByZero();
      else
         Xkp1 = Xk / 2.0 + A / (2.0 * Xk);

      Iterations++;

      if ( Iterations > ITERATIONLIMIT )
         throw ConvergenceFailure();
   }

   return Xkp1;
}
```


©2002 McQuain WD & Keller BJ

Hierarchy of Exceptions

Using an inheritance hierarchy of exception objects allows for flexible control in the handler:

```
#include <limits>
using namespace std;

class ArithmeticException {};
class NegativeRadicand    : public ArithmeticException {};
class DivideByZero        : public ArithmeticException {};
class ConvergenceFailure  : public ArithmeticException {};
. . .
```

```
    try {
       root = SquareRoot(x);
    }
    catch ( NegativeRadicand N ) {
       cout << "caught a NegativeRadicand" << endl;
    }
    catch ( ArithmeticException A ) {
       cout << "caught a general arithmetic exception" << endl;
    }
```


©2002 McQuain WD & Keller BJ