Learning Objectives

• Exception Handling Basics
  – Defining exception classes
  – Multiple throws and catches
  – Exception specifications

• Programming Techniques for Exception Handling
  – When to throw exceptions
  – Exception class hierarchies
Introduction

• Typical approach to development:
  – Write programs assuming things go as planned
  – Get "core" working
  – Then take care of "exceptional" cases

• C++ exception-handling facilities
  – Handle "exceptional" situations
  – Mechanism "signals" unusual happening
  – Another place in code "deals" with exception
Exception-Handling Basics

• Meant to be used sparingly
  – In "involved" situations
• Difficult to teach such large examples
• Approach:
  – Simple toy examples, that would not normally use exception-handling
  – Keep in mind "big picture"
Toy Example

Imagine: people rarely run out of milk:
```cpp
cout << "Enter number of donuts:";
cin >> donuts;
cout << "Enter number of glasses of milk:";
cin >> milk
dpg = donuts/static_cast<double>(milk);
cout << donuts << "donuts.\n";
   << milk << "glasses of milk.\n";
   << "You have " << dpg
   << "donuts for each glass of milk.\n";
```

Basic code assumes never run out of milk
Toy Example if-else

• Notice: If no milk \(\rightarrow\) divide by zero error!
• Program should accommodate unlikely situation of running out of milk
  – Can use simple if-else structure:
    ```cpp
    if (milk <= 0)
        cout << "Go buy some milk!\n";
    else
        {...}
    ```
• Notice: no exception-handling here
Toy Example with Exception Handling: **Display 18.2**  Same Thing Using Exception Handling

```cpp
9 try
10 {
11     cout << "Enter number of donuts:\n";
12     cin >> donuts;
13     cout << "Enter number of glasses of milk:\n";
14     cin >> milk;
15
16     if (milk <= 0)
17         throw donuts;
18     dpg = donuts/static_cast<double>(milk);
19     cout << donuts << " donuts.\n"
20         << milk << " glasses of milk.\n"
21         << "You have " << dpg
22         << " donuts for each glass of milk.\n";
23 }
24 catch(int e)
25 {
26     cout << e << " donuts, and No Milk!\n"
27         << "Go buy some milk.\n";
28 }
```
Toy Example Discussion

• Code between keywords *try* and *catch*
  – Same code from ordinary version, except if statement simpler:
    if (milk <= 0)
    throw donuts;
  – Much cleaner code
  – If "no milk" → do something exceptional

• The "something exceptional" is provided after keyword *catch*
Toy Example try-catch

• Try block
  – Handles "normal" situation

• Catch block
  – Handles "exceptional" situations

• Provides separation of normal from exceptional
  – Not big deal for this simple example, but important concept
try block

• Basic method of exception-handling is try-throw-catch

• Try block:
  try
  {
    Some_Code;
  }

  – Contains code for basic algorithm when all goes smoothly
• Inside try-block, when something unusual happens:

```java
try {
    Code_To_Try
    if (exceptional_happened)
        throw donuts;
    More_Code
}
```

– Keyword `throw` followed by exception type
– Called "throwing an exception"
catch-block

• When something thrown → goes somewhere
  – In C++, flow of control goes from try-block to catch-block
    • try-block is "exited" and control passes to catch-block
  – Executing catch block called "catching the exception"

• Exceptions must be "handled" in some catch block
catch-block More

• Recall:
  catch(int e)
  {
    cout << e << " donuts, and no milk!\n";
    << " Go buy some milk.\n";
  }

• Looks like function definition with int parameter!
  – Not a function, but works similarly
  – Throw like "function call"
catch-block Parameter

• Recall: catch(int e)

• "e" called catch-block parameter
  – Each catch block can have at most ONE catch-block parameter

• Does two things:
  1. type name specifies what kind of thrown value the catch-block can catch
  2. Provides name for thrown value caught; can "do things" with value
Defining Exception Classes

• throw statement can throw value of any type

• Exception class
  – Contains objects with information to be thrown
  – Can have different types identifying each possible exceptional situation
  – Still just a class
    • An "exception class" due to how it’s used
Exception Class for Toy Example

• Consider:
  class NoMilk
  {
  public:
   NoMilk() {}  
   NoMilk(int howMany) : count(howMany) {}  
   int getcount() const { return count; }
  private:
   int count;
  };

• throw NoMilk(donuts);
  – Invokes constructor of NoMilk class
Multiple Throws and Catches

• try-block typically throws any number of exception values, of differing types

• Of course only one exception thrown
  – Since throw statement ends try-block

• But different types can be thrown
  – Each catch block only catches "one type"
  – Typical to place many catch-blocks after each try-block
    • To catch "all-possible" exceptions to be thrown
Catching

• Order of catch blocks important
• Catch-blocks tried "in order" after try-block
  – First match handles it!
• Consider:
  catch (...) {   }
  – Called "catch-all", "default" exception handler
  – Catches any exception
  – Ensure catch-all placed AFTER more specific exceptions!
  • Or others will never be caught!
Trivial Exception Classes

• Consider:
  class DivideByZero
  {
  }
• No member variables
• No member functions (except default constructor)
• Nothing but it’s name, which is enough
  – Might be "nothing to do" with exception value
  – Used simply to "get to" catch block
  – Can omit catch block parameter
Throwing Exception in Function

• Function might throw exception
• Callers might have different "reactions"
  – Some might desire to "end program"
  – Some might continue, or do something else
• Makes sense to "catch" exception in calling function’s try-catch-block
  – Place call inside try-block
  – Handle in catch-block after try-block

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.
Throwing Exception in Function Example

• Consider:
  
  ```java
  try {
      quotient = safeDivide(num, den);
  }
  catch (DivideByZero) {
      ... 
  }
  ```

• `safeDivide()` function throws `DividebyZero` exception
  – Handled back in caller’s catch-block
Exception Specification

• Functions that don’t catch exceptions
  – Should "warn" users that it could throw
  – But it won’t catch!

• Should list such exceptions:
  double safeDivide(int top, int bottom)
  throw (DividebyZero);
  – Called "exception specification" or "throw list"
  – Should be in declaration and definition
  – All types listed handled "normally"
  – If no throw list → all types considered there
Throw List

• If exception thrown in function NOT in throw list:
  – No errors (compile or run-time)
  – Function unexpected() automatically called
    • Default behavior is to terminate
    • Can modify behavior

• Same result if no catch-block found
Throw List Summary

• void someFunction()
  throw(DividebyZero, OtherException);
  //Exception types DividebyZero or OtherException
  //treated normally. All others invoke unexpected() 

• void someFunction() throw ();
  //Empty exception list, all exceptions invoke unexpected() 

• void someFunction();
  //All exceptions of all types treated normally
Derived Classes

• Remember: derived class objects also objects of base class

• Consider:
  D is derived class of B

• If B is in exception specification ➔
  – Class D thrown objects will also be treated normally, since it’s also object of class B

• Note: does not do automatic type cast:
  – double will not account for throwing an int
unexpected()

- Default action: terminates program
  - No special includes or using directives
- Normally no need to redefine
- But you can:
  - Use set_unexpected
  - Consult compiler manual or advanced text for details
When to Throw Exceptions

• Typical to separate throws and catches
  – In separate functions

• Throwing function:
  – Include throw statements in definition
  – List exceptions in throw list
    • In both declaration and definition

• Catching function:
  – Different function, perhaps even in different file
Preferred throw-catch Triad: throw

• void functionA() throw (MyException)
  {
    ...
    throw MyException(arg);
    ...
  }

• Function throws exception as needed
Preferred throw-catch Triad: catch

- Then some other function:
  
  ```
  void functionB()
  {
      ...
      try
      {
          ...
          functionA();
          ...
      }
      catch (MyException e)
      {
          // Handle exception
      }
      ...
  }
  ```
Uncaught Exceptions

• Should catch every exception thrown
• If not → program terminates
  – terminate() is called
• Recall for functions
  – If exception not in throw list: unexpected() is called
    • It in turn calls terminate()
• So same result
Overuse of Exceptions

• Exceptions alter flow of control
  – Similar to old "goto" construct
  – "Unrestricted" flow of control
• Should be used sparingly
• Good rule:
  – If desire a "throw": consider how to write program without throw
  – If alternative reasonable → do it
Exception Class Hierarchies

• Useful to have; consider:
  DivideByZero class derives from:
    ArithmeticError exception class
    
    – All catch-blocks for ArithmeticError also catch DivideByZero
    
    – If ArithmeticError in throw list, then DividebyZero also considered there
Testing Available Memory

• new operator throws bad_alloc exception if insufficient memory:
  try
  {
    NodePtr pointer = new Node;
  }
  catch (bad_alloc)
  {
    cout << "Ran out of memory!";
    // Can do other things here as well...
  }

• In library <new>, std namespace
Rethrowing an Exception

• Legal to throw exception IN catch-block!
  – Typically only in rare cases
• Throws to catch-block "farther up chain"
• Can re-throw same or new exception
  – rethrow;
    • Throws same exception again
  – throw newExceptionUp;
    • Throws new exception to next catch-block
Summary 1

- Exception handling allows separation of "normal" cases and "exceptional" cases
- Exceptions thrown in try-block
  – Or within a function whose call is in try-block
- Exceptions caught in catch-block
- try-blocks typically followed by more than one catch-block
  – List more specific exceptions first
Summary 2

• Best used with separate functions
  – Especially considering callers might handle differently
• Exceptions thrown in but not caught in function, should be listed in throw list
• Exceptions thrown but never caught → program terminates
• Resist overuse of exceptions
  – Unrestricted flow of control
Exception Handling
#include <iostream>
using namespace std;

int main()
{
    double x;
    cout << "enter a number" << endl;
    cin >> x;

    if (x < 0)
    
        cout << "x is a negative value" << endl;
    cout << "end of main" << endl;

}
Basic Exception Handling

```cpp
#include <iostream>
#include <string>
using namespace std;

int main()
{
    double x;
    cout<< "enter a number"<<endl;
    cin>>x;
    try
    {
        if(x<0)
            throw "x is negative value";
    }
    catch (int x)
    {
        cout<<"int exception: " << x<<endl;
    }
    catch (double x)
    {
        cout<<"double exception: " << x << endl;
    }
    catch (const char* x)
    {
        cout<<"string exception: " << x << endl;
    }
    cout<<"end of main"<<endl;
}
```
#include <iostream>
#include <string>
using namespace std;

void checkValue(double x)
{
    if(x<0)
        throw x;
}

int main()
{
    double x;
    cout << "enter a number" << endl;
    cin >> x;
    try
    {
        cout << "Checking value of x" << endl;
        checkValue(x);
    }
    catch (int x)
    {
        cout << "int exception: " << x << endl;
    }
    catch (double x)
    {
        cout << "double exception: " << x << endl;
    }
    catch (const char* x)
    {
        cout << "string exception: " << x << endl;
    }
    cout << "end of main" << endl;
}
Stack Unwinding

• Each function is terminated in sequence until a handler for the exception is found, or until main() is terminated without the exception being handled
```cpp
#include <iostream>
using namespace std;

void last() // called by third()
{
    cout << "Start last" << endl;
    cout << "last throwing int exception" << endl;
    throw -1;
    cout << "End last" << endl;
}

void third() // called by second()
{
    cout << "Start third" << endl;
    last();
    cout << "End third" << endl;
}

void second() // called by first()
{
    cout << "Start second" << endl;
    try{third();}
    catch(double)
    {
        cout << "second caught double exception" << endl;
    }
    cout << "End second" << endl;
}

void first() // called by main()
{
    cout << "Start first" << endl;
    try{second();}
    catch (int)
    {
        cout << "first caught int exception" << endl;
    }
    catch (double)
    {
        cout << "first caught double exception" << endl;
    }
    cout << "End first" << endl;
}

int main()
{
    cout << "Start main" << endl;
    try
    {
        first();
    }
    catch (int)
    {
        cout << "main caught int exception" << endl;
    }
    cout << "End main" << endl;
}
```
#include <iostream>
using namespace std;

int last() // called by third()
{
    cout << "Start last" << endl;
    return -1;
    cout << "End last" << endl;
}

int third() // called by second()
{
    cout << "Start third" << endl;
    int rc=last();
    if (rc<0)
    {cout<<"last(): rc is less than 0"<<endl;
     return rc;}
    else
    {cout<<"last(): rc is greater than 0"<<endl;
     return 0;}
    cout << "End first" << endl;
}

int main()
{
    cout << "Start main" << endl;
    int rc= first();
    if (rc<0)
    {cout<<"main: rc is less than 0"<<endl;
     return 0;}
    else
    {cout<<"main: rc is greater than 0"<<endl;
     cout << "End main" << endl;
Uncaught Exceptions

```cpp
#include <iostream>
using namespace std;

void checkValue(double x)
{
    if(x<0)
    throw x;
}

int main()
{
    double x;
    cout<<"enter a number"<<endl;
    cin>>x;

    checkValue(x);
    cout<<"end of main"<<endl;
    return 0;
}
```
#include <iostream>
using namespace std;

void checkValue(double x)
{
    if(x<0)
        throw "x is less than 0";
}

int main()
{
    double x;
    cout<<"enter a number"<<endl;
    cin>>x;
    try{
        cout<<"Checking value of x"<<endl;
        checkValue(x);
    }
    catch(int x)
    { cout<<"int exception:"<< x<<endl;}
    catch(double x)
    { cout<<"double exception:"<<x<<endl;}
    catch(...)
    { cout<<"undetermined exception"<<x<<endl;}
    cout<<"end of main"<<endl;
}
Exception Specifiers

- a mechanism that allows us to use a function declaration to specify whether a function may or will not throw exceptions.
Empty throw statement

• use an empty throw statement to denote that a function does not throw any exceptions outside of itself

```cpp
void checkValue(double x) throw()
{
    if(x<0)
        throw "x is less than 0";
}

int main()
{
    double x;
    cout<<"enter a number"<<endl;
    cin>>x;
    try{
        cout<<"Checking value of x"<<endl;
        checkValue(x);
    }
    catch(int x)
    { cout<<"int exception:"<<x<<endl; }
    catch(double x)
    { cout<<"double exception:"<<x<<endl; }
    catch(...)
    { cout<<"undetermined exception"<<x<<endl; }

    cout<<"end of main"<<endl;
}
Example 2

```cpp
#include <iostream>
using namespace std;

void f()
{
    throw "x is less than 0";
}
void checkValue(double x) throw()
{
    if(x<0)
        f();
    //throw "x is less than 0";
}
int main()
{
    double x;
    cout<<"enter a number"<<endl;
    cin>>x;
    try{
        cout<<"Checking value of x"<<endl;
        checkValue(x);
    }
    catch(int x)
    {
        cout<<"int exception:"<<x<<endl;
    }
    catch(double x)
    {
        cout<<"double exception:"<<x<<endl;
    }
    catch(...)
    {
        cout<<"undetermined exception"<<x<<endl;
    }
    cout<<"end of main"<<endl;
}``
Specific throw statement

```cpp
void checkValue(double x) throw (const char*)
{
    if (x < 0)
        throw "x is less than 0";
}

int main()
{
    double x;
    cout << "Enter a number" << endl;
    cin >> x;
    try{
        cout << "Checking value of x" << endl;
        checkValue(x);
    }
    catch(int x)
        { cout << "int exception:" << x << endl; }
    catch(double x)
        { cout << "double exception:" << x << endl; }
    catch(...)
        { cout << "Undetermined exception" << x << endl; }

    cout << "End of main" << endl;
}
Example 3

```cpp
void checkValue(double x) throw (double)
{
    if (x < 0)
        throw "x is less than 0";
}

int main()
{
    double x;
    cout << "enter a number" << endl;
    cin >> x;
    try{
        cout << "Checking value of x" << endl;
        checkValue(x);
    } catch (int x) {
        cout << "int exception:" << x << endl;
    } catch (double x) {
        cout << "double exception:" << x << endl;
    } catch (...) {
        cout << "undetermined exception" << x << endl;
    }
    cout << "end of main" << endl;
}
```
# Exception Class

```cpp
#include <iostream>
using namespace std;

class ArrayException
{
    private:
        string m_error;

    public:
        ArrayException(string error):m_error(error){}
        string what(){return m_error;}
};

class IntArray
{
    private:
        int m_data[3];
    public:
        IntArray(){}

        int& operator[](const int index)
        {
            if(index<0||index>=3)
                throw ArrayException("Invalid index");
            return m_data[index];
        }
};

int main()
{
    IntArray array;
    try
    {
        int value=array[10];
    }
    catch (ArrayException &obj)
    {
        cout<<"An array exception occured ("<<obj.what()<<")"<<endl;
    }
}
Exception and Inheritance

```cpp
#include <iostream>
using namespace std;

class Base
{
public:
    Base() {}
};

class Derived: public Base
{
public:
    Derived() {}
};

int main()
{
    try
    {
        throw Derived();
    }
    catch (Base &base)
    {
        cout << "caught Base";
    }
    catch (Derived &derived)
    {
        cout << "caught Derived";
    }
}
```
Exception and Inheritance

```cpp
#include <iostream>
using namespace std;

class Base {
  public:
    Base() {}
};

class Derived: public Base {
  public:
    Derived() {}
};

int main() {
  try {
    throw Derived();
  } catch (Derived &derived) {
    cout << "caught Derived";
  } catch (Base &base) {
    cout << "caught Base";
  }
}
```
Standard Exceptions

- std::bad_alloc
- std::bad_cast
- std::bad_typeid
- std::bad_exception
- std::logic_failure
- std::runtime_error
- std::domain_error
- std::invalid_argument
- std::length_error
- std::out_of_range
- std::overflow_error
- std::range_error
- std::underflow_error
bad_alloc

- type of the object thrown as exceptions by the allocation functions to report failure to allocate storage

```cpp
#include <iostream>
#include <exception> // for std::exception
#include <string>    // for this example
using namespace std;

int main()
{
    try
    {
        // Your code using standard library goes here
        // We'll trigger one of these exceptions intentionally for the sake of example
        string s;
        s.resize(-1); // will trigger a std::bad_alloc
    }
    // This handler will catch std::exception and all the derived exceptions too
    catch (exception &exception)
    {
        cout << "Standard exception: " << exception.what() << endl;
    }
    return 0;
}
```

Standard exception: basic_string::resize
what()

• exception has a virtual function named what() that returns a C-style string description of the exception.

• derived classes can override the what() function to change the message.
Deriving own classes from exception class

```cpp
#include <iostream>
#include <string>
using namespace std;

class ArrayException: public exception
{
    private:
        string m_error;
    
    public:
        ArrayException(string error):m_error(error){}
        string what(){return m_error;}
};

class IntArray
{
    private:
        int m_data[3];
    public:
        IntArray(){

        }

        int& operator[](const int index)
        {
            string s;
            s.resize(-1);
            //if (index<0||index>=5)
            //throw ArrayException("Invalid index");
            return m_data[index];
        }

};

int main()
{
    IntArray array;
    try
    {
        int value=array[10];
    }
    catch (ArrayException &obj)
    {
        cout<"An array exception occurred (" <<obj.what()<<")"<<endl;
    }
    catch (exception &obj)
    {
        cout<"Some other exception occurred (" <<obj.what()<<")"<<endl;
    }
}
Rethrowing Exceptions

```cpp
// ArrayException class
class ArrayException: public exception
{
    private:
        string m_error;

    public:
        ArrayException(string error): m_error(error){}
        string what(){return m_error;}
};

class IntArray
{
    private:
        int m_data[3];
    public:
        IntArray(){}

    int& operator[](const int index)
    {
        try
        {
            if (index<0||index>=5)
                throw ArrayException("Invalid index");
            return m_data[index];
        }
        catch (ArrayException &obj)
        {
            cout<<"An array exception occurred in ArrayException (" <<obj.what()<<")"<<endl;
            throw;
        }
    }
};

int main()
{
    IntArray array;
    try
    {
        int value=array[10];
    }
    catch (ArrayException &obj)
    {
        cout<<"An array exception occurred in main (" <<obj.what()<<")"<<endl;
    }
    catch (exception &obj)
    {
        cout<<"Some other exception occurred (" <<obj.what()<<")"<<endl;
    }
}
```
Example 4

class A
{
private:
    int m_x;
public:
    A(int x) : m_x(x)
    {
        if (x <= 0)
            throw 1;
    }
};

class B : public A
{
public:
    B(int x) : A(x)
    {// What happens if creation of A fails and we want to handle it here?}
};

int main()
{
    try
    {
        B b(0);
    }
    catch (int)
    {
        cout << "Oops\n";
    }
}
Function try blocks

class B : public A
{
public:
    B(int x)try : A(x)
    {
    }
    catch(...)
    {
        cout<<"catch in B"<<endl;
    }
};
Example 5

```cpp
#include <iostream>
using namespace std;

class Test {
public:
    Test() { cout << "Constructor of Test " << endl; }
    ~Test() { cout << "Destructor of Test " << endl; }
};

int main() {
    try {
        Test t1;
        throw 10;
    } catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```
Answer

• When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block
• What is the advantage of exception handling?
Example 6

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x = -1;
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n"
    }

    cout << "After catch \n";
    return 0;
}
```
Example 7

```cpp
#include <iostream>
using namespace std;

int main()
{
    try
    {
        throw 10;
    }
    catch (...)
    {
        cout << "default exception\n";
    }
    catch (int param)
    {
        cout << "int exception\n";
    }
    return 0;
}
```
Example 8

```cpp
#include <iostream>
using namespace std;

class Test {
    static int count;
    int id;

public:
    Test() {
        count++;
        id = count;
        cout << "Constructing object number " << id << endl;
        if(id == 4)
            throw 4;
    }
    ~Test() { cout << "Destructing object number " << id << endl; }
};

int Test::count = 0;

int main() {
    try {
        Test array[5];
    } catch(int i) {
        cout << "Caught " << i << endl;
    }
}
```