Introduction

- Exception handling - catch errors before they occur
  - Deals with synchronous errors (i.e., Divide by zero)
  - Does not deal with asynchronous errors - disk I/O completions, mouse clicks - use interrupt processing
  - Used when system can recover from error
    - Exception handler - recovery procedure
  - Typically used when error dealt with in different place than where it occurred
  - Useful when program cannot recover but must shut down cleanly
- Exception handling should not be used for program control
  - Not optimized, can harm program performance

Exception handling improves fault-tolerance
- Easier to write error-processing code
- Specify what type of exceptions are to be caught

Most programs support only single threads
- Techniques in this chapter apply for multithreaded OS as well (Windows NT, OS/2, some UNIX)

Exception handling another way to return control from a function or block of code

When To Use Exception Handling?

- Error handling should be used for
  - Processing exceptional situations
  - Processing exceptions for components that cannot handle them directly
  - Processing exceptions for widely used components (libraries, classes, functions) that should not process their own exceptions
  - Large projects that require uniform error processing

Other Error-Handling Techniques

- Use assert
  - If assertion false, the program terminates
- Ignore exceptions
  - Use this “technique” on casual, personal programs - not commercial!
- Abort the program
  - Appropriate for nonfatal errors give appearance that program functioned correctly
  - Inappropriate for mission-critical programs, can cause resource leaks
- Set some error indicator
  - Program may not check indicator at all points there error could occur
Other Error-Handling Techniques

- Test for the error condition
  - Issue an error message and call exit
  - Pass error code to environment
- set jmp and longjump
  - In <csetjmp>
  - Jump out of deeply nested function calls back to an error handler.
  - Dangerous - unwinds the stack without calling destructors for automatic objects
- Specific errors
  - Some have dedicated capabilities for handling them
  - If new fails to allocate memory new_handler function executes to deal with problem

Basics of C++ Exception Handling: 

try, throw, catch

- A function can throw an exception object if it detects an error
  - Object typically a character string (error message) or class object
  - If exception handler exists, exception caught and handled
  - Otherwise, program terminates

Basics of C++ Exception Handling:

try, throw, catch

- Format
  - Enclose code that may have an error in try block
  - Follow with one or more catch blocks
    - Each catch block has an exception handler
  - If exception occurs and matches parameter in catch block, code in catch block executed
  - If no exception thrown, exception handlers skipped and control resumes after catch blocks
  - throw point - place where exception occurred
    - Control cannot return to throw point

A Simple Exception-Handling Example:

Divide by Zero

- Look at the format of try and catch blocks

1. Class definition
1.1 Function definition

1. Class definition
1.2 Initialize variables
2. Input data
2.1 try and catch blocks
2.2 Function call
3. Output result
Program Output

Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): 33 9
The quotient is: 3.66667

Enter two integers (end-of-file to end): 14

Throwing an Exception

- `throw` - indicates an exception has occurred
  - Usually has one operand (sometimes zero) of any type
  - If operand an object, called an exception object
  - Conditional expression can be thrown
  - Code referenced in a `try` block can throw an exception
  - Exception caught by closest exception handler
  - Control exits current try block and goes to `catch` handler (if it exists)
  - Example (inside function definition)
    ```cpp
    if (denominator == 0)
      throw DivideByZeroException();
    ```
    - Throws a `dividebyzeroexception` object

Catching an Exception

- Exception handlers are in `catch` blocks
  - Format: `catch (exceptionType parameterName) {
    exception handling code
  }
  ` - Caught if argument type matches `throw` type
  - If not caught then `terminate` called which (by default) calls `abort`
  - Example:
    ```cpp
    catch (DivideByZeroException ex) {
      cout << "Exception occurred: " << ex.what() << '
    
    ```
    - Catches exceptions of type `DivideByZeroException`

- Catch all exceptions
  ```cpp
  catch (Exception e) - catches all exceptions
  ```
  - You do not know what type of exception occurred
  - There is no parameter name - cannot reference the object

- If no handler matches thrown object
  ```cpp
  Searches next enclosing `try` block
  ```
  - If none found, `terminate` called
  - If found, control resumes after last `catch` block
  - If several handlers match thrown object, first one found is executed

- `catch` parameter matches thrown object when
  ```cpp
  They are of the same type
  ```
  - Exact match required - no promotions/conversions allowed
  - The `catch` parameter is a `public` base class of the thrown object
  - The `catch` parameter is a base-class pointer/reference type and the thrown object is a derived-class pointer/reference type
  - The `catch` handler is `catch (......)`
  - Thrown `const` objects have `const` in the parameter type

- Exception not required to terminate program
  - However, terminates block where exception occurred

- Exception caught by closest exception handler
  - Control exits current try block and goes to `catch` handler (if it exists)

- Example (inside function definition)
  ```cpp
  if (denominator == 0)
    throw DivideByZeroException();
  ```
  - Throws a `dividebyzeroexception` object
Catching an Exception (IV)

- Unreleased resources
  - Resources may have been allocated when exception thrown
  - `catch` handler should delete space allocated by `new` and close any opened files
- `catch` handlers can throw exceptions
  - Exceptions can only be processed by outer `try` blocks

Rethrowing an Exception

- Rethrowing exceptions
  - Used when an exception handler cannot process an exception
  - Rethrow exception with the statement:
    - `throw;`
      - No arguments
  - If no exception thrown in first place, calls `terminate`
  - Handler can always rethrow exception, even if it performed some processing
  - Rethrown exception detected by next enclosing `try` block

---

```cpp
// Demonstration of rethrowing an exception.
#include <iostream>
#include <exception>

using std::cout;
using std::endl;

void throwException() {
    // Throw an exception and immediately catch it.
    try {
        cout << "Function throwException
";
        throw exception();  // generate exception
    }
    catch ( exception e ) {
        cout << "Exception handled in function throwException
";
    }
}

int main() {
    try {
        throwException();
        cout << "This should not print
";
    }
    catch ( exception e ) {
        cout << "Exception handled in main
";
    }
    cout << "Program control continues after catch in main"
     << endl;
    return 0;
}
```

1. Load header
2. Function call
3. Output

Exception Specifications

- Exception specification (`throw list`)
  - Lists exceptions that can be thrown by a function
  - Example: `int g( double h ) throw ( a, b, c )`:
    - Function can throw listed exceptions or derived types
    - If other type thrown, `function unexpected` called
    - `throw()` (i.e., no `throw` list) states that function will not throw any exceptions
      - In reality, function can still throw exceptions, but calls `unexpected` (more later)
    - If no `throw` list specified, function can `throw` any exception

Processing Unexpected Exceptions

- Function `unexpected`
  - Calls the function specified with `set_unexpected`
    - Default: `terminate`
- Function `terminate`
  - Calls function specified with `set_terminate`
    - Default: `abort`
- Prototypes in `<exception>`
- Take pointers to functions (i.e., Function name)
  - Function must return `void` and take no arguments
  - Returns pointer to last function called by `terminate` or `unexpected`
Stack Unwinding

- Function-call stack unwind when exception thrown and not caught in a particular scope
  - Tries to catch exception in next outer try/catch block
  - Function in which exception was not caught terminates
    - Local variables destroyed
    - Control returns to place where function was called
  - If control returns to a try block, attempt made to catch exception
    - Otherwise, further unwinds stack
  - If exception not caught, terminate called

Constructors, Destructors and Exception Handling

- What to do with an error in a constructor?
  - A constructor cannot return a value - how do we let the outside world know of an error?
    - Keep defective object and hope someone tests it
    - Set some variable outside constructor
  - A thrown exception can tell outside world about a failed constructor
  - catch handler must have a copy constructor for thrown object

Constructors, Destructors and Exception Handling

- Thrown exceptions in constructors
  - Destructors called for all completed base-class objects and member objects before exception thrown
  - If the destructor that is originally called due to stack unwinding ends up throwing an exception, terminate called
  - If object has partially completed member objects when exception thrown, destructors called for completed objects

Constructors, Destructors and Exception Handling

- Resource leak
  - Exception comes before code that releases a resource
  - One solution: initialize local object when resource acquired
    - Destructor will be called before exception occurs
  - catch exceptions from destructors
  - Enclose code that calls them in try block followed by appropriate catch block

Exceptions and Inheritance

- Exception classes can be derived from base classes
- If catch can get a pointer/reference to a base class, can also catch pointers/references to derived classes

Processing new Failures

- If new could not allocate memory
  - Old method - use assert function
    - If new returns 0, abort
    - Does not allow program to recover
  - Modern method (header <new>)
    - new throws bad_alloc exception
  - Method used depends on compiler
    - On some compilers: use new(nothrow) instead of new to have new return 0 when it fails
      - Function set_new_handler(functionName) - sets which function is called when new fails.
      - Function can return no value and take no arguments
    - new will not throw bad_alloc
Processing new Failures

- new
  - Loop that tries to acquire memory
- A new handler function should either:
  - Make more memory available by deleting other dynamically allocated memory and return to the loop in operator new
  - Throw an exception of type bad_alloc
  - Call function abort or exit (header <cstdlib>) to terminate the program

Program Output

```
Allocated 5000000 doubles in ptr[0]
Allocated 5000000 doubles in ptr[1]
Allocated 5000000 doubles in ptr[2]
Exception occurred: Allocation Failure
```

1. Load headers
   1.1 Function definition
   1.2 Initialize large arrays
2. Use all available memory
3. Output

Class auto_ptr and Dynamic Memory Allocation

- Pointers to dynamic memory
  - Memory leak can occur if exceptions happens before delete command
  - Use class template auto_ptr (header <memory>) to resolve this
- auto_ptr objects act just like pointers
  - Automatically deletes what it points to when it is destroyed (leaves scope)
  - Can use * and -> like normal pointers
// Example.cpp
// Demonstrating auto_ptr

#include <iostream>

using std::cout;
using std::endl;

#include <memory>

using std::auto_ptr;

// 1. Load header
1.1 Class definition
1.2 Function definitions

class Integer {
public:
    Integer( int i = 0 ) : value( i )
    { cout << "Constructor for Integer " << value << endl; }

    ~Integer()
    { cout << "Destructor for Integer " << value << endl; }

    void setInteger( int i ) { value = i; }

    int getInteger() const { return value; }

private:
    int value;
};

int main() {
    cout << "Creating an auto_ptr object that points "
    "to an Integer
"

    auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );

    cout << "Using the auto_ptr to manipulate the Integer
";
    ptrToInteger->setInteger( 99 );
    cout << "Integer after setInteger: "
    << ( *ptrToInteger ).getInteger() 
    << "\nTerminating program" << endl;

    return 0;
}

1. Initialize auto_ptr pointer
2. Manipulate values
3. Output
4. Program Output

Creating an auto_ptr object that points to an Integer
Constructor for Integer 7
Using the auto_ptr to manipulate the Integer
Destructor for Integer 99
Terminating program

Standard Library Exception Hierarchy

- Exceptions fall into categories
  - Hierarchy of exception classes
    - Base class exception (header <exception>)
    - Derived classes: runtime_error and logic_error
      (header <stdexcept>)
  - Class logic_error
    - Errors in program logic, can be prevented by writing proper code
    - Derived classes:
      - invalid_argument - invalid argument passed to function
      - length_error - length larger than maximum size allowed
      - out_of_range - out of range subscript

- Other classes derived from exception
  - Exceptions thrown by C++ language features
    - new - bad_alloc
    - dynamic_cast - bad_cast
typedef - bad_typedef
  - Puh std::bad_exception in throw list
    - unexpected() will throw bad_exception instead of calling
      function set by set_unexpected