Introduction

• Exceptions
  – Indicates problem occurred in program
  – Not common
    • An “exception” to a program that usually works

• Exception Handling
  – Resolve exceptions
  – Program may be able to continue
    • Controlled termination
  – Write fault-tolerant programs

Exception-Handling Overview

• Consider pseudocode
  Perform a task
  If the preceding task did not execute correctly
    Perform error processing
  Perform next task
  If the preceding task did not execute correctly
    Perform error processing

• Mixing logic and error handling
  – Can make program difficult to read/debug
  – Exception handling removes error correction from “main line” of program

Exception handling

– For synchronous errors (divide by zero, null pointer)
  • Cannot handle asynchronous errors (independent of program)
  • Disk I/O, mouse, keyboard, network messages
– Easy to handle errors

Terminology

– Function that has error throws an exception
– Exception handler (if it exists) can deal with problem
  • Catches and handles exception
– If no exception handler, uncaught exception
  • Could terminate program
Exception-Handling Overview

- C++ code
  ```cpp
  try {
      code that may raise exception
  } catch (exceptionType){
      code to handle exception
  }
  ```
  - `try` block encloses code that may raise exception
  - One or more `catch` blocks follow
  - Catch and handle exception, if appropriate
  - Take parameter, if named, can access exception object

- Try point
  - Location in `try` block where exception occurred
  - If exception handled
    - Program skips remainder of `try` block
    - Resumes after `catch` blocks
  - If not handled
    - Function terminates
    - Looks for enclosing `catch` block (stack unwinding, 13.8)
  - If no exception
    - Program skips `catch` blocks

Other Error-Handling Techniques

- Ignore exception
  - Typical for personal (not commercial) software
  - Program may fail
- Abort program
  - Usually appropriate
  - Not appropriate for mission-critical software
- Set error indicators
  - Unfortunately, may not test for these when necessary
- Test for error condition
  - Call exit `<cstdlib>` and pass error code

- `setjmp` and `longjmp`
  - `<setjmp>`
  - Jump from deeply nested function to call error handler
  - Can be dangerous
- Dedicated error handling
  - `new` can have a special handler
Simple Exception-Handling Example: Divide by Zero

- **Keyword throw**
  - Throws an exception
  - Use when error occurs
  - Can throw almost anything (exception object, integer, etc.)
    - `throw myObject;`
    - `throw 5;`

- **Exception objects**
  - Base class `runtime_error (<stdexcept>)`
  - Constructor can take a string (to describe exception)
  - Member function `what()` returns that string

Upcoming example

- Handle divide-by-zero errors
- Define new exception class
  - `DivideByZeroException`
  - Inherit from `runtime_error`
- In division function
  - Test denominator
  - If zero, throw exception (throw object)
- In `try` block
  - Attempt to divide
  - Have enclosing `catch` block
    - Catch `DivideByZeroException` objects

```
#include <iostream>
#include <exception>

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

// DivideByZeroException objects should be thrown by functions
// upon detecting division-by-zero exceptions
class DivideByZeroException : public runtime_error {
public:
  DivideByZeroException() : exception( "attempted to divide by zero" ) {}  
};  // end class DivideByZeroException

// perform division and throw DivideByZeroException object if
// divide-by-zero exception occurs
double quotient( int numerator, int denominator )
{
  // throw DivideByZeroException if trying to divide by zero
  if ( denominator == 0 )
    throw DivideByZeroException(); // terminate function

  // return division result
  return static_cast< double >( numerator ) / denominator;
}

int main()
{
  int number1;    // user-specified numerator
  int number2;    // user-specified denominator
  double result;  // result of division

  cout << "Enter two integers (end-of-file to end): " << endl;
  cin >> number1 >> number2;

  // attempt to divide
  try
  {  
    result = quotient( number1, number2 );
  }
  catch ( DivideByZeroException )
  {  
    cout << "Enter two integers (end-of-file to end): ");

    // handle exception
    // ...
### Rethrowing an Exception

#### Rethrowing exceptions
- Use when exception handler cannot process exception
  - Can still rethrow if handler did some processing
  - Can rethrow exception to another handler
    - Goes to next enclosing `try` block
    - Corresponding `catch` blocks try to handle
- To rethrow
  - Use statement `throw;`
    - No arguments
    - Terminates function

```cpp
// throw, catch and rethrow exception
// throw exception and catch it immediately
try {
    throw exception();  // generate exception
} // end try

// handle exception
catch ( exception &caughtException ) {
    throw;  // rethrow exception for further processing
}
```

Notice the structure of the `try` and `catch` blocks. The `try` block can catch `DivideByZeroException` exceptions, and print an error message. If no exception occurs, the `catch` block is skipped.

Member function `what` returns the string describing the exception.

```cpp
// throw, catch and rethrow exception
// throw exception and catch it immediately
try {
    throw exception();  // generate exception
} // end try

// handle exception
catch ( exception &caughtException ) {
    throw;  // rethrow exception for further processing
}
```

Exception handler generates a default exception (base class `exception`). It immediately catches and rethrows it (note use of `throw;`).
Exception Specifications

- List of exceptions function can throw
  - Also called throw list
  - Can only throw `ExceptionA`, `ExceptionB`, and `ExceptionC` (and derived classes)
    - If throws other type, function `unexpected` called
    - By default, terminates program (more 13.7)
    - If no throw list, can throw any exception
    - If empty throw list, cannot throw any exceptions

Processing Unexpected Exceptions

- Function `unexpected`
  - Calls function registered with `set_unexpected`
    - `<exception>`
      - Calls `terminate` by default
    - `set_terminate`
      - Sets what function `terminate` calls
      - By default, calls `abort`
        - If redefined, still calls `abort` after new function finishes
  - Arguments for set functions
    - Pass pointer to function
    - Function must take no arguments
    - Returns `void`
Stack Unwinding

- If exception thrown but not caught
  - Goes to enclosing try block
  - Terminates current function
    • Unwinds function call stack
  - Looks for try/catch that can handle exception
    • If none found, unwinds again
- If exception never caught
  - Calls terminate

Outline

Fig. 13.3: fig13_03.cpp

```cpp
// Fig. 13.3: fig13_03.cpp
// Demonstrating stack unwinding.

#include <iostream>

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

#include <stdexcept>

using std::runtime_error;

// function3 throws run-time error
void function3() throw ( runtime_error )
{
    throw runtime_error( "runtime_error in function3" ); // fourth
}

// function2 invokes function3
void function2() throw ( runtime_error )
{
    function3(); // third
}

// function1 invokes function2
void function1() throw ( runtime_error )
{
    function2(); // second
}

// demonstrate stack unwinding
int main()
{
    // invoke function1
    try {
        function1(); // first
    } // end try
    // handle run-time error
    catch ( runtime_error &error ) // fifth
    {
        cout << "Exception occurred: " << error.what() << endl;
    } // end catch
    return 0;
}
```

Constructors, Destructors and Exception Handling

- Error in constructor
  - new fails; cannot allocate memory
  - Cannot return a value - how to inform user?
    • Hope user examines object, notices errors
    • Set some global variable
  - Good alternative: throw an exception
    • Destructors automatically called for member objects
    • Called for automatic variables in try block
- Can catch exceptions in destructor
Exceptions and Inheritance

• Exception classes
  – Can be derived from base classes
    • i.e., runtime_error::exception
  – If catch can handle base class, can handle derived classes
    • Polymorphic programming

Processing new Failures

• When new fails to get memory
  – Should throw bad_alloc::exception
    • Defined in <new>
  – Some compilers have new return 0
  – Result depends on compiler

---

fig13_04.cpp

```cpp
#include <iostream>

using std::cout;

int main()
{
  double *ptr[50];

  for ( int i = 0; i < 50; i++ ) {
    ptr[i] = new double[5000000];
    if ( ptr[i] == 0 ) {
      cout << "Memory allocation failed for ptr[ " << i << " ]\n";
      break;
    }
  }

  return 0;
}
```

---

fig13_04.cpp

```cpp
int main()
{
  double *ptr[50];

  for ( int i = 0; i < 50; i++ )
    ptr[i] = new double[5000000];

  if ( ptr[4] == 0 ) {
    cout << "Memory allocation failed for ptr[ 4 ]\n";
    break;
  }

  cout << "Allocated 5000000 doubles in ptr[ " << i << " ]\n";
}
```

---

fig13_04.cpp

```cpp
int main()
{
  double *ptr[50];

  for ( int i = 0; i < 50; i++ )
    ptr[i] = new double[5000000];

  if ( ptr[4] == 0 ) {
    cout << "Memory allocation failed for ptr[ 4 ]\n";
    break;
  }

  cout << "Allocated 5000000 doubles in ptr[ 0 ]\n";
  cout << "Allocated 5000000 doubles in ptr[ 1 ]\n";
  cout << "Allocated 5000000 doubles in ptr[ 2 ]\n";
  cout << "Allocated 5000000 doubles in ptr[ 3 ]\n";
  cout << "Memory allocation failed for ptr[ 4 ]\n";
}
```
// Fig. 13.5: fig13_05.cpp
// Demonstrating standard new throwing bad_alloc when memory
// cannot be allocated.

#include <iostream>
#include <new>
using std::cout;
using std::endl;
using std::bad_alloc;

int main()
{
    double *ptr[50];

    // attempt to allocate memory
    try {
        // allocate memory for ptr[ i ]; new throws bad_alloc
        // on failure
        for (int i = 0; i < 50; i++) {
            ptr[i] = new double[5000000];
            cout << "Allocated 5000000 doubles in ptr[ " << i << " ]\n";
        }
    }
    // end try

    // handle bad_alloc exception
    catch (bad_alloc &memoryAllocationException) {
        cout << "Exception occurred: " << memoryAllocationException.what() << endl;
    }

    return 0;
}  // end main

Processing new Failures

- **set_new_handler**
  - Header `<new>`
  - Register function to call when `new` fails
  - Takes function pointer to function that
    - Takes no arguments
    - Returns `void`
  - Once registered, function called instead of throwing exception
// specify that customNewHandler should be called on failed memory allocation
set_new_handler( customNewHandler );

// allocate memory for ptr[i]; customNewHandler will be called on failed memory allocation
for ( int i = 0; i < 50; i++ ) {
    ptr[i] = new double[5000000];
    cout << "Allocated 5000000 doubles in ptr[ ", i, " ]
"; } // end for

return 0;

} // end main

Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
customNewHandler was called

---

Class auto_ptr and Dynamic Memory Allocation

• Declare pointer, allocate memory with new
  – What if exception occurs before you can delete it?
  – Memory leak
• Template class auto_ptr
  – Header <memory>
  – Like regular pointers (has * and ->)
  – When pointer goes out of scope, calls delete
  – Prevents memory leaks
  – Usage
    auto_ptr<MyClass> newPointer( new MyClass() );
    *newPointer points to dynamically allocated object
Create an auto_ptr object that points to an Integer
Constructor for Integer 7
Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99
Destroying program
Destructor for Integer 99

Standard Library Exception Hierarchy

- Exception hierarchy
  - Base class exception(<exception>)
    - Virtual function what, overridden to provide error messages
    - Sample derived classes
      - runtime_error, logic_error
      - bad_alloc, bad_cast, bad_typeid
        - Thrown by new, dynamic_cast and typeid
  - To catch all exceptions
    - catch(...) 
    - catch ( exception AnyException )
      - Will not catch user-defined exceptions