Exception Handling

C++ Object Oriented Programming

Pei-yih Ting

NTOU CS

Contents

Error Handling Strategies

- Error handling is a major source of programmer mistakes. Usually error handling is not considered in the design process and evolve chaotically in an ad hoc manner.
- In C, most programs use return codes to handle error conditions. In UNIX, there is also a global errno variable to indicate the type of errors.

```
int *ptr = (int *) malloc(sizeof(int)*100);
if (ptr==0) {
    cout << "Memory allocation failure!\n";
    // some other resource management tasks, ex. Freeing some memory
    return 0; // return an error code to be handled by the calling program
}
```

- Problems:
  - Return code is a nice-guy approach; it allows the caller to do something when an error occurs but it doesn't *require* the caller to do anything.

Error Handling Strategies (cont'd)

- Problems: (cont'd)
  - Explicit if-check: this type of error processing code is many times longer than the normal processing code. It repeats quite often. However, the probability that these codes get executed may be 1 out of 1000.
    - Cause code-explosion (each error has a different handling program segment, even the same error, ex. File cannot be opened, has different handling strategies at each occurrence. ex. At each occurrence of memory allocation error, the segments of memory to be deallocated are different.)
    - These error handling codes are hard to test and maintain.
    - Obscure the normal program logic, make them hard to test and maintain also.
    - Cause efficiency loss, especially with OO fine-grained member functions.

Error Handling Strategies (cont'd)

- Problems: (cont'd)
  - Cannot handle errors in constructors.
  - It is clumsy to return from a deep function call and handling the resources gracefully during its return. Intermediate callers might not know how to handle the error but it need to pass the error code and information back to the calling routine.
  - Error code is a simple integer, can not carry sufficient information with it.
  - Sometimes the return value is needed for some other important things (normal logics which occur more frequently than errors).
- C++ supports a better error handling mechanism

try { … throw … } catch (…) { … }

Throwing an Exception

```
void Stack::push(int element) throw(int){
    if (isFull()) throw element;
    m_top++;
    m_array[m_top] = element;
}
void main() {
    Stack stack;
    try {
        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);
    }
    catch (int element) {
        cout << "Stack overflow with element " << element << ".\n";
    }
}
```

❷ copied (copy ctor)

❸ copied (copy ctor)

❶ create an exception object (variable) in a special exception area (not on calling stack)

❹ this exception object stays alive till this exception is handled completely by a catch segment (and not re-thrown), destructed thereafter

```
Output:
Stack overflow with element 3.
```

Error Handling Strategies (cont'd)

- Advantages of **try-throw-catch** mechanism
  - Force the occurred errors to be handled
  - The handling position of an exception can be far away from where it happens.
  - Normal logic is separated from exception handling logic.
  - Transmit arbitrarily large amount of information from the error occurring position to the error handling position.
  - Automatically handle the destruction of objects on the stack frames until the place where exception is ultimately handled.
  - Allow different error handlers to be defined for different types of objects. One can design a hierarchy of different types of errors.

In summary: $\boxed{\text{Reduced coding and testing costs}}$

Throwing an Exception (cont'd)

- What happens if you don't *try* to catch the exception?

```
void main() {
    Stack stack;
    stack.push(1);
    stack.push(2);
    stack.push(3);
    stack.push(4);
}
```

  The exception will still be thrown and the program terminates (with a runtime error).

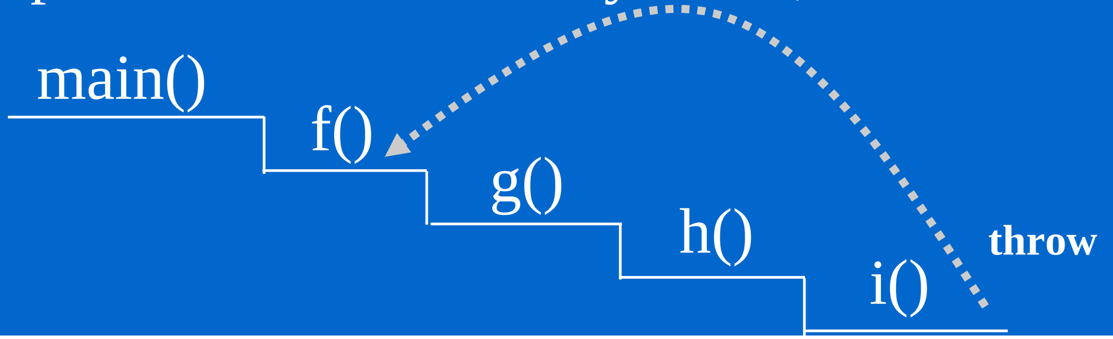
- What happens if you don't *catch* the exception?

```
void main() {
    Stack stack;
    try {
        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);
    }
}
```

Compiler error!!

How it works

- When an exception is raised by the “throw” statement, the nearest catch block that handles the specified exception is executed.
- The execution continues with the next statement after the catch block.
- In a try block, when throw is executed, the try block finalizes right away and every object created within the try block is destroyed.
- Anywhere in a program, when throw is executed, the routine finalized right away and every local object on the stack is destroyed.
- The remaining codes after throw statement are skipped, just as in the case of the return statement in a function.
- When no exception is raised in the try block, the catch block is skipped.



Throwing Exceptions

- ✧ You can throw exceptions anywhere in a try block and actually anywhere in the program.
- ✧ Exceptions can be thrown in functions called within a try block and caught by a catch block following a try block.
- ✧ The expression following the throw keyword determines the type of the exception.
  - ✶ Use a temporary object as the throw argument, e.g.

    ```
    throw std::runtime_error("open file error");
    ```
  - ✶ Do not throw the address of a locally defined variable, array, or object, C++ does not guarantee whether the stack is unwound before (g++) the catch block executes or after (vc2010) the catch block finishes.
- ✧ Uncaught exceptions go up towards the main program.

Throwing Multiple Exceptions

- A function can throw as many types of errors as it pleases. The value thrown back determines which catch handler is invoked

```
void Stack::push(int element) throw(int, char *) {
    if (isFull()) throw element;
    if (m_top<0) throw "Stack is underrun!";
    m_array[m_top++] = element;
}
```

// g++ demands this
// exception specifier
// but vc2010 ignore it

- Within the calling function

```
try {
    …
    stack.push(1);
}
catch (int element) {
    cout << "Stack overflow with element " << element << ".\n";
}
catch (char *errorMsg) {
    cout << errorMsg;
}
```

Catching Exceptions

- The type of the parameter to the catch statement is defined as in function declaration, usually use reference type of parameters, e.g. catch (overflow_error &e), to avoid another copy and enjoy the polymorphic behaviours.
- You must supply at least one catch block for a try block.
- Catch blocks must immediately follow the try block without any program code between them.
- Catch blocks will catch exceptions of the correct type that occur in the code in the immediately preceding try block, including the ones thrown by functions called within the try block.

Exceptions vs. assert()

- assert():
  - Catches situations that SHOULD NOT happen (but did happen). E.g. promise made by other classes. Basically these are cases you don't want to handle (at least not specified in the program specification).
  - Typically disabled before product delivery!
  - Should not be seen by the end customer!
  - Checking to track down programmer's own bugs
- Exception:
  - Should be seen by people using our code. Not disabled in the final released version.
  - Indicates user errors (e.g. invalid argument errors)
  - Indicates some system errors (e.g. file not found)

Problems with assert()

- Your program stops immediately. Usually used in debugging.
- Why should your program continue if an error has occurred?

1. Non-fatal errors

```
void Stack::push(int element) {
    assert(!isFull());
    m_top++;
    m_array[m_top] = element;
}
```

The failure of the call to push may be non-fatal to the rest of the program.

2. Failing gracefully

```
p = new int[kBigArraySize];
assert(p!=0);
```

Although the memory is insufficient, the user may want to save the existing data before quitting.

3. Safety-critical programming

The patient will die if the software crashes. / System will be hacked.

Error Handling in C++

- Three levels:
  - **assert() statements**: those errors that the specification of the program excludes. You don't want it to be handled automatically by your program.
  - **If statements**: those expected situations that happened normally and quite often, eg. user enter incorrect data, file not opened, …
  - **Exceptions**: those expected/unexpected situations that happened rarely (say 1 out of 100), eg. disk access errors, … Or, you want to avoid long/ugly error handling codes…

Rule of thumb: If in doubt, use exceptions

Sometimes, there are still practices of using a single goto statement to handle all sorts of memory deallocation after program fails. In general, this mechanism can be replaced by the exception handling.

Separate Normal and Error Handling

- Example:

```
#include <stdexcept>
using namespace std;
Matrix add(const Matrix &a, const Matrix &b)
                                        throw(overflow_error, underflow_error);
Matrix sub(const Matrix &a, const Matrix &b)
                                        throw(overflow_error, underflow_error);


   …
void solutionA(const Matrix &a, const Matrix &b) throw(underflow_error) {
   try {
      cout << "a + b is " << add(a, b) << "\n";
      cout << "a - b is " << sub(a, b) << "\n";
      cout << "a * b is " << mul(a, b) << "\n";
      cout << "a / b is " << div(a, b) << "\n";
   }
   catch (overflow_error &e) {
      cout << "overflow: " << e.what() << "\n";
   }
```

Normal logic

Error handling logic

Tangled Normal and Error Handling

```
ReturnCode solutionB(const Matrix &a, const Matrix &b)  {
    Matrix result;
    ReturnCode rc;
    rc = add(result, a, b);
    if (rc == OK) {
        cout << "a + b is " << result << "\n";
    } else if (rc == OVERFLOW_ERROR) {
        cout << "overflow error: Matrix + Matrix\n";
        return OK; // error has been handled
    } else {
        return rc; // leaving the unhandled error to calling function
    }
    rc = sub(result, a, b);
    if (rc == OK) {
        cout << "a - b is " << result << "\n";
    } else if (rc == OVERFLOW_ERROR) {
        cout << "overflow error: Matrix - Matrix\n";
        …
}
```

```
enum ReturnCode {
    OK,
    OVERFLOW_ERROR,
    UNDERFLOW_ERROR
};
```

Exception and Termination

- What happens if we don't catch some types of exception?
  The program terminates by calling a global (predefined) *terminate*().
- You can override the built-in *terminate*() function

```
void myTerminate() {
    cout << "I am terminated.\n";
    abort();
}
void main() {
    set_terminate(myTerminate);
    Stack stack;
    try {
        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);
    }
    catch (char *errMsg) {
        cout << errMsg;
    }
}
```

Generic Catch Handler

- You can specify a handler to catch any type of exception that is thrown in a try block:

```
catch (…)
{
    // code to handle any exception
}
```

- This catch block must appear last after all other types of exception handlers if you have other catch blocks defined for the try block.

```
catch (int element)
{
    // code to handle any exception
}
catch (…)
{
    // code to handle any exception
}
```

Standard Exception Classes

- Some predefined exceptions in C++ standard library <stdexcept>

  Root: exception

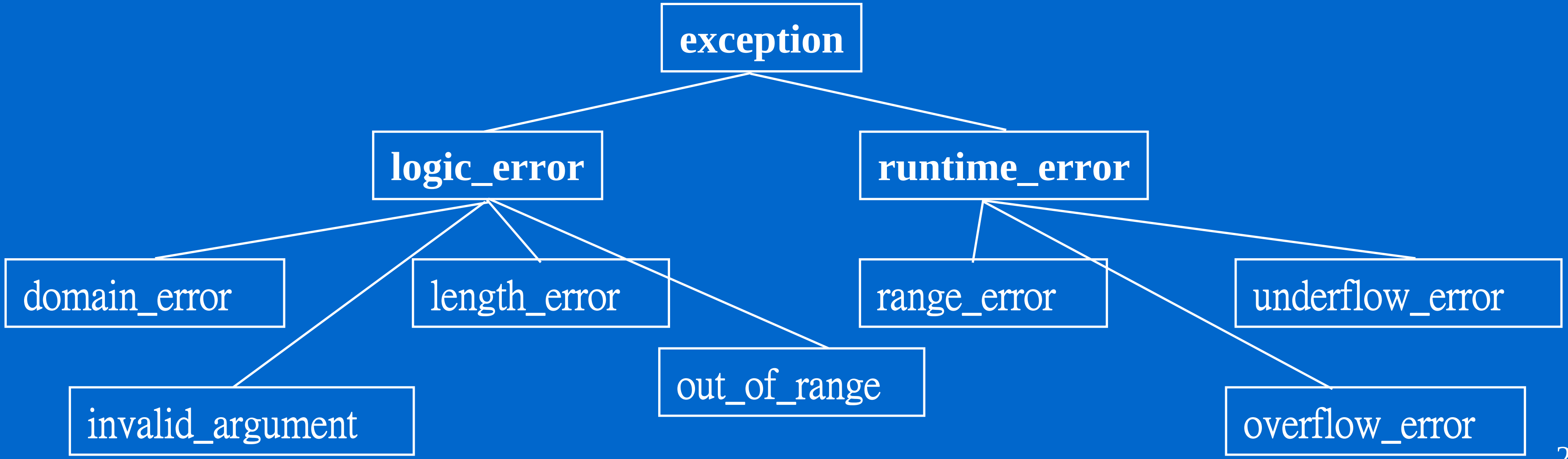
  Main categories:

  logic_error
  runtime_error

  exception::what()
  returns the error message

  Exceptions derived from the above:

  domain_error, invalid_argument, length_error, out_of_range
  range_error, overflow_error, underflow_error



Exception Classes (cont'd)

- You can use the standard exception classes

```
int div(int i1, int i2) throw(invalid_argument) {
    if (i2 == 0) throw invalid_argument("Divided by 0!\n");
    return i1/i2;
}
```

This temporary object will be copied to an exception object and destroyed immediately. At the end of the catch statement that handles this exception, the exception object will be destructed.

- You can derive your own exception class from the standard exception classes if the category matches.

```
class NoGas: public runtime_error {
public:
    NoGas(const string &what) throw();
};
NoGas::NoGas(const string &what) throw():
    runtime_error(what) {
}
```

- Define your own exception class hierarchy: A monolithic hierarchy of exception classes works better than a forest. This allows all uncaught exceptions be caught by *catch (exception &root_class)*, where *root_class* is a polymorphic reference, instead of *catch(…)* statement, inside which you can not even print the cause of error.

Exception Classes (cont'd)

```
class Fred {};

class Wilma {};

void sample() throw(Fred, Wilma) {
    switch (rand()%3) {
        case 0:
            cout << "throwing a Fred\n";
            throw Fred();
        case 1:
            cout << "throwing a Wilma\n";
            throw Wilma();
        default:
            cout << "returning normally\n";
    }
}
```

```
void main() {
    for (int i=0; i<10; i++) {
        try {
            cout << "trying: \n";
            sample();
            cout << "no exception thrown\n";
        }
        catch (Fred &) {
            cout << "caught a Fred\n";
        }
        catch (Wilma &) {
            cout << "caught a Wilma\n";
        }
        catch (…) {
            cout << "this should "
                        "never happen\n";
        }
    }
}
```

Exception Specifier (Throw List)

- A function can *optionally* declare what it throws in exception specifiers (also called a "throw list")

```
class Stack {
public:
    …
    void push(int element) throw(int, char*);
    …
};
```

- Including the throw list with the function definition

```
void Stack::push(int element) throw(int, char *) {
    …
}
```

This is optional but in general should be provided.

Violating the Throw List

- What happens if you try to throw a different type than in the throw specification? The program calls a global function *unexpected*(). The default implementation of *unexpected*() calls *terminate*() to end the program.
- You can override *unexpected*(): rethrow an exception which you handle

```
void myUnexpected() {
    throw("an unexpected exception has occurred.");
}
```

- Setting the function

```
void main() {
    set_unexpected(myUnexpected);
    …
}
```

VC6 seems to have problem

- Overriding unexpected() allows the client to protect itself against errors in a server class.

Dynamically Allocated Memory

- What happens to a segment of dynamically allocated memory that is passed over by an exception?

```
void Foo() {
    Stack *stack = new Stack;
    int *temp = new int[100];
    stack.push(1); // exception is thrown
    delete [] temp;
    delete stack;
}
void main() {
    try {
        Foo();
    }
    catch (int element) {
        cout << "Stack overflow with element " << element << endl;
    }
}
```

- The compiler will automatically call the destructor for all objects on the unwound stack frames BUT not dynamically allocated objects.

Dynamically Allocated Memory (cont'd)

- Solutions to the problem:
  - Use automatic objects on the stack
  - Use container classes instead of raw C arrays
  - Wrap the raw C arrays or dynamically allocated objects within an object that is allocated on the stack. That wrapper object deletes the dynamically allocated resources in its destructor.
  - Use the managed pointer object instead of the raw pointer.

  - Sometimes, opening files is similar to dynamically allocated memory. If you met an exception while the file is opened explicitly with fopen() calls, the try-catch mechanism does not call fclose() automatically. The solution is still wrap the file operations inside a file object, for example, ifstream.

Propagation of Exceptions

- Uncaught exceptions will go up the calling stack till it find a matched handler.
- Exceptions may also pass through levels of handlers

```
void SomeClass::Foo() {
    try {
        m_stack.push(4);
    }
    catch (int element) {
        if (m_size<10000) {
            m_stack.expandStack();
            m_stack.push(4);
        }
        else
            throw;
            // rethrows the original exception object
    }
}
```

```
void main() {
    try {
        Foo();
    }
    catch (int element) {
        cout << "Stack not expanded "
        "after element " << element << endl;
    }
}
```

Pragmatics of Exceptions

- Exceptions should be exceptional (i.e. true errors). Unusual cases should be handled by normal logic using return value and if-statement. Ex. Reaching EOF in reading data
- A function should throw an exception when anything occurs that prevents it from fulfilling its promises (i.e. its contract).
- Responsibilities of the Server class and the Client class
  - Server: give the client of the class the means to avoid exceptions if possible. For example, provide an isFull() function for the Stack class.
  - Client: consider preventing exceptions rather than catching them.
- The hardest part of using exception handling mechanism is to decide what is an error and when an exception should be thrown.
  - This is specification (or contract) dependent.
  - If the specification said that data should always be saved. Then problems in opening files are not errors and should be handled by regular program segments.

Pragmatics of Exceptions (cont'd)

- When should a function catch an exception? When it knows what to do with it. When it has the sufficient information to recover from it.
- Should a catch block fully recover from an error? If possible. But sometimes the best that can be done are some cleanup and a rethrow.
- A constructor should throw an exception when it meets an error.
- What should a composed object do when its member object throws an exception at its construction? Nothing. If after successful construction of m_obj1, the constructor of m_obj2 fails and throws an exception, the composed object is not fully constructed and the destructor of m_obj1 is called. (Note: the dtor ~Composed() will not be invoked. However, m_data was not allocated.)

```
class Composed {
    …
private:
    Ingredient1 m_obj1;
    Ingredient2 m_obj2;
    int *m_data;
};
```

```
Composed::Composed(int size):m_obj1(), m_obj2() {
    m_data = new int[size];
}
```

failure

Pragmatics of Exceptions (cont’d)

- Should destructors throw exceptions when they fail? NO

  If a dtor throws an exception during the stack-unwinding process of another exception, terminate() is invoked, which kills the program. At the same time, if a dtor calls some routines that may throw exceptions, it should catch all possible errors.