CSC230

Intro to C++ Lecture 17

Outline

- Lab 7 discussion
- Template / Exceptions in C++ (continued)

Lab 7 discussion

- Implement an insert function based on your lab 7.
- Do we need to sort the linked list?
  - String compare
- Difference between append and insert.

Exceptions handling

When something goes wrong in one function, how should we notify the function caller?

- Return a special value to the caller?
- Return a boolean value to the caller?
- Set a global variable? (Toyota, is it your style?)
- Print out a message?
- Print out a message and exit the program?
- Handle the problem without telling the caller?
- Set a failure flag?
- Example : divide-1.cpp

```
#include <iostream>     // std::cerr
#include <fstream>      // std::ifstream

int main () {
  std::ifstream is;
  is.open ("test.txt");
  if ( (is.rdstate() & std::ifstream::failbit ) != 0 )
    std::cerr << "Error opening 'test.txt'\n";
  return 0;
}
```

What is the problem with these options?

All these options are passive (the caller need to check whether there is a problem).

- The function with problem/error should always notify the caller. Do not keep quiet.
- If constructor has a problem
  - It cannot return a value. A constructor does not have a return value.
- The error happens inside a function that does not know how to handle it.
- Example : divide-2.cpp

Exception handling

- Caller has a choice on how to handle the problem.
  - The function caused the error does not need to guess what to do.
- The normal control flow and the exception handling are separated.
- The program is easy to read.

```
try
{
   // protected code
}catch( ExceptionName e1 )
{
   // catch block
}catch( ExceptionName e2 )
{
   // catch block
}catch( ExceptionName eN )
{
   // catch block
}
```

assert

The assert statement checks certain boolean condition is true or not. If it is false, the program will be terminated.

- Good for developing/testing
- Not good for final product
- assert is usually used for testing / you can turn on or off the assertion\
- What is the difference between assert and exception?

```
#include <iostream>
#include <cassert>

int main()
{
  assert(2+2==4);
  std::cout << "Execution continues past the first assert\n";
  assert(2+2==5);
  std::cout << "Execution continues past the second assert\n";
}
```

Why exception?

- With exception handling, a program can continue executing (rather than terminating) after dealing with a problem.
- This helps to support robust applications that contribute to *mission*
  - *-critical* computing or *business-critical* computing
- When no exceptions occur, there is no performance reduction
- Example : divide-3.cpp

throw statement

- Exception can be thrown anywhere within a code block
- throw statement creates an exception
- The value (operand) of the throw statement determines the type of exception
- The operand of the throw statement can be any expression

```
double division(int x, int y)
{
    if( y == 0 )
    {
        throw "Division by zero condition!";
    }
    return (x/y);
}
```

`try` Blocks

- Keyword `try` followed by braces (`{ }`)
- What should enclose?
  - Statements that might cause exceptions
  - Statements that should be skipped in case of an exception
  - revisit : divide-3.cpp

`Catch` Handlers

- Immediately follow a **`try`** block
  - One or more **`catch`** handlers for each **`try`** block
- Keyword **`catch`**
- Exception parameter enclosed
  - Represents the type of exception to process
  - Can provide an optional parameter name to interact with the caught exception object
- Executes if exception parameter type matches the exception thrown in the **`try`** block
  - Could be a base class of the thrown exception's class

Catching exceptions

- You can specify what type of exception to catch

```
try
{
   // protected code
}catch( ExceptionName e )
{
  // code to handle ExceptionName exception
}
```

- Above code will catch an exception of ExceptionName type.
- If you want to catch any exceptions, you must put an ellipsis, ….

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

Exception example


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

double division(int x, int y)
{
  if( y == 0 )
    {
      throw "Divided by zero!";
    }
  return (x/y);
}
```

```
int main ()
{
  int m = 230;
  int n = 0;
  double r = 0;

  try {
    r = division(m, n);
    cout << r << endl;
  }catch (const char* msg) {
    cerr << msg << endl;
  }

  return 0;
}
```

Throw a char array
Catch a char array

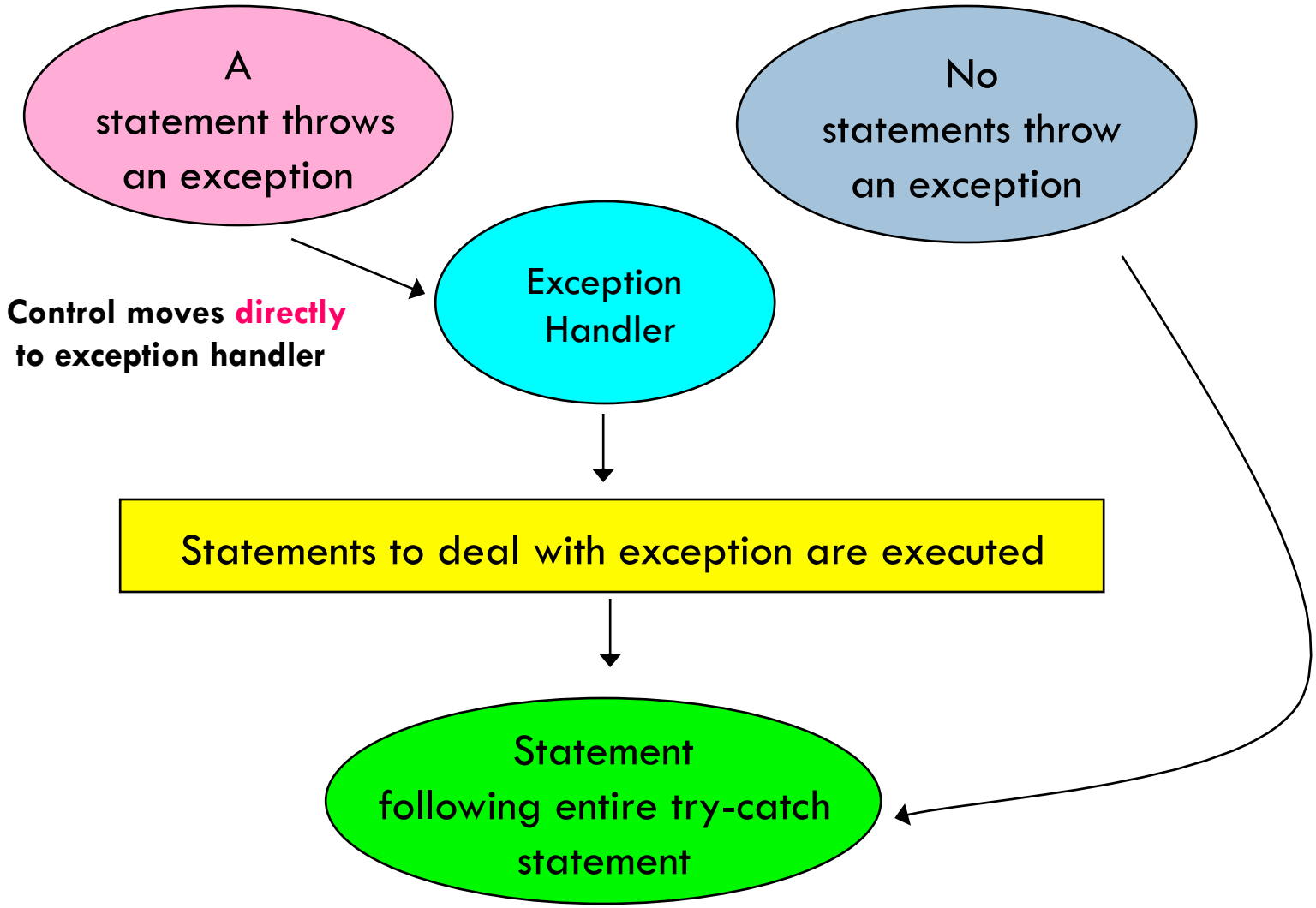
try-blocks and if-else

- try-blocks are very similar to if-else statements
  - If everything is normal, the entire try-block is executed
  - else, if an exception is thrown, the catch-block is executed
- A big difference between try-blocks and if-else statements is the try-block's ability to send a message to one of its branches

Example of a `try-catch` Statement

```
 try
{
        // Statements that process personnel data and may throw
        // exceptions of type int, string, and SalaryError
}
catch ( int )
{
        // Statements to handle an int exception
}
catch ( string s )
{
       cout << s << endl;  // Prints "Invalid customer age"
        // More statements to handle an age error
}
catch ( SalaryError )
{
       // Statements to handle a salary error
}
```

Execution of try-catch



Who will catch

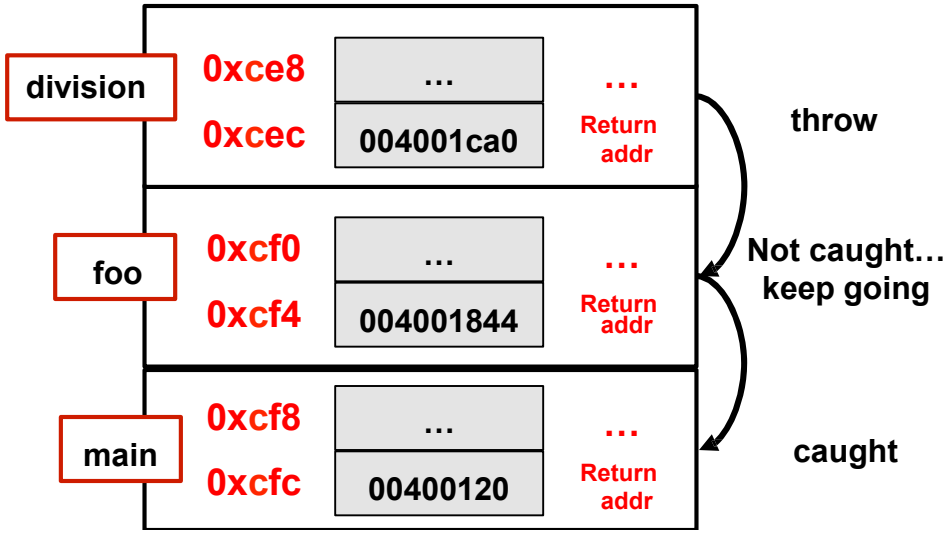

```
#include <iostream>
using namespace std;

double division(int x, int y)
{
  if( y == 0 )
    {
      throw “Divided by zero!";
    }
  return (x/y);
}

double foo(int x, int y)
{
  return division(x, y);
}
```

```
int main ()
{
  int m = 230;
  int n = 0;
  double r = 0;

  try {
    r = foo(m, n);
    cout << r << endl;
  }catch (const char* msg) {
    cerr << msg << endl;
  }
  return 0;
}
```

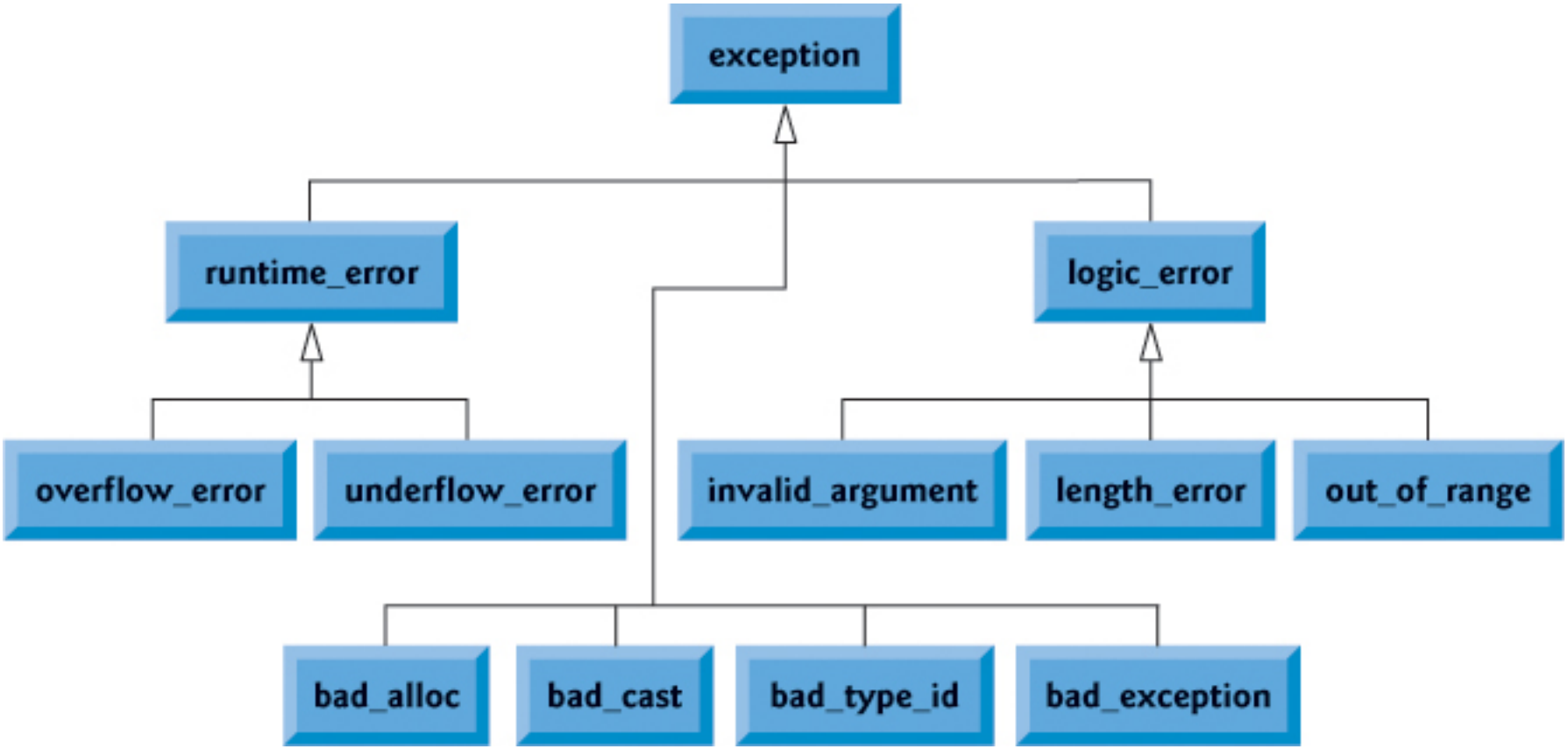


Throw something meaningful

- In general, do not throw primitive values, such as int or float
  - *throw 200*
  - It is hard for other function to figure out the meaning of the number
  - It does not provide context information
- In general, do not throw a string
  - It is easy for human being to understand
  - But it is hard for other function to figure out

- Use a **class**, especially those defined in <stdexcept>
  - *throw std::invalid_argument("value is negative");*
  - *throw std::runtime_error("Failed");*
  - Method what() with extra details

std::exception



std::exception

Excetption	**Description**
std::bad_alloc	Can be thrown by **new**
std::bad_cast	Can be thrown by **dynamic_cast**
std::bad_typeid	Can be thrown by **typeid**
std::logic_error	An exception can be detected by READING the code
std::domain_error	Caused by Mathematically invalid domain
std::invalid_argument	Caused by invalid arguments
std::length_error	Cause by a too big std::string
std::out_of_length	Caused by std::vector, std::bitset<>operator[]()
std::runtime_error	An exception can not be detected by reading the code
std::overflow_error	Caused by mathematical overflow

Define new exceptions


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

struct NewException : public exception
{
  const char * what ()
  {
    return "Exception";
  }
};
```

Inherits and overrides exception class

what() is defined in exception class, and overridden by every child exception class

```
int main()
{
  try
    {
      throw NewException();
    }
  catch(NewException& e)
    {
      std::cout << "NewException caught" << std::endl;
      std::cout << e.what() << std::endl;
    }
  catch(std::exception& e)
    {
      //Other errors
    }
}
```

Examples

- Check out some examples