Exception handling

Comp Sci 1575 Data Structures

Exceptions

There are no exceptions to the rule that everybody likes to be an exception to the rule.
- Charles Osgood

Outline

Outline

Problem

- When writing an algorithm to solve a problem in the general case, often special cases must be addressed
- Some might occur only rarely
- Adding rare cases to the algorithm may increase it's complexity or clarity
- Error handling return statements end up intricately linked to the normal control flow of the code, constraining both how the code is laid out, and how errors can be reasonably handled.
- Ideally, express the algorithm in general form including any common special cases.
- Rare exceptional situations, along with a strategy to handle them, could appear elsewhere, like an annotation to the algorithm.

Outline

Solution

- Exception handling is a process of handling exceptional situations that may occur in a program
- Allows programmers to cleanly separate the code that implements the focused algorithm from the code that deals with exceptional situations that the algorithm may face.
- After giving the proper message stating the reason of the exception, the program continues to execute after correcting the error, or will terminate gracefully i.e., it will give a proper message and then will terminate the program.
- Exception handling communicates the existence of a run-time problem or error from where it was detected to where the issue can be handled.
- Un-handled exceptions stop program execution.
- Also commonly called raising an exception.

Outline

Keywords: try, throw, and catch


```
try
{
   statements;
   throw exception-type;
}

catch(exception-type var)
{
   statements;
}
```

Keywords: try, throw, and catch

- **try** block acts watches for any exceptions that are thrown by any of the statements within the try block (protected code)
  - **throw** keyword signals that an exception or error case has occurred, and is followed by an: {error code, description of the problem, or a custom exception class}
  - Should an error occur in the try block, an exception is thrown (raised), which then causes the current scope to be exited, and also each outer scope (propagation) until a suitable handler (catch block) is found, calling in turn the destructors of any objects in these exited scopes.
- **catch** keyword is used to define a catch block that handles exceptions for a single exception type; catch must be in the same scope as try

Outline

1 Introduction
Problem
Solution
Keywords
Benefits
2 Examples
3 Stack unwinding
4 std exceptions

Benefits

- Functions don't need to return error codes, freeing their return values for program logic.
- Fewer functions to deal with errors
- An exception jumps to the point in the call stack that can handle the error.
- Exception stack-unwinding mechanism destroys all objects in scope according to well-defined rules after an exception is thrown.
- Clean separation between code that detects errors and the code that handles errors
- Algorithm is kept focused on solving the problem, and exceptional cases are handled elsewhere.
- Approach is more modular and encourages the development of code that is cleaner and easier to maintain and debug.

Outline

Examples

Check out the code

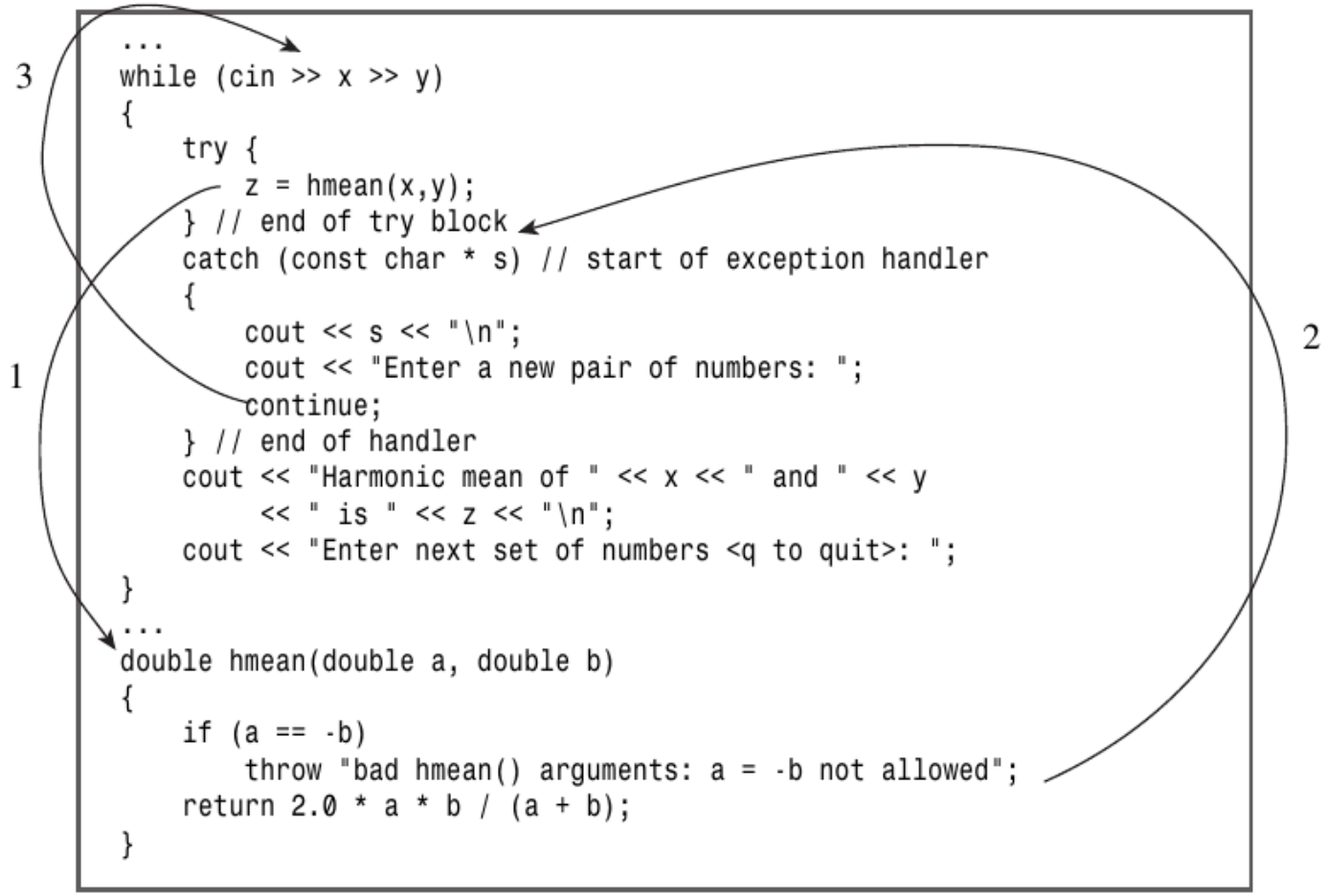
Outline

Trace the execution


```
void g() { throw "exception";}
void f()
{
  std::string str = "Hello";
  g();
}
int main()
{
  try { f(); }
  catch(...)   { }
}
```

main() calls f(); f() creates a local variable named str; str constructor allocates a memory chunk to hold the string "Hello"; f() calls g(); g()throws an exception; f() does not catch the exception; Because the exception was not caught, we now need to exit f() in a clean fashion. All the destructors of local variables previous to the throw are called, called 'stack unwinding'; destructor of str is called, releasing memory occupied by it. main() catches the exception; The program continues; 'stack unwinding' guarantees destructors of local variables (stack variables) will be called when we leave its scope.

Trace the execution

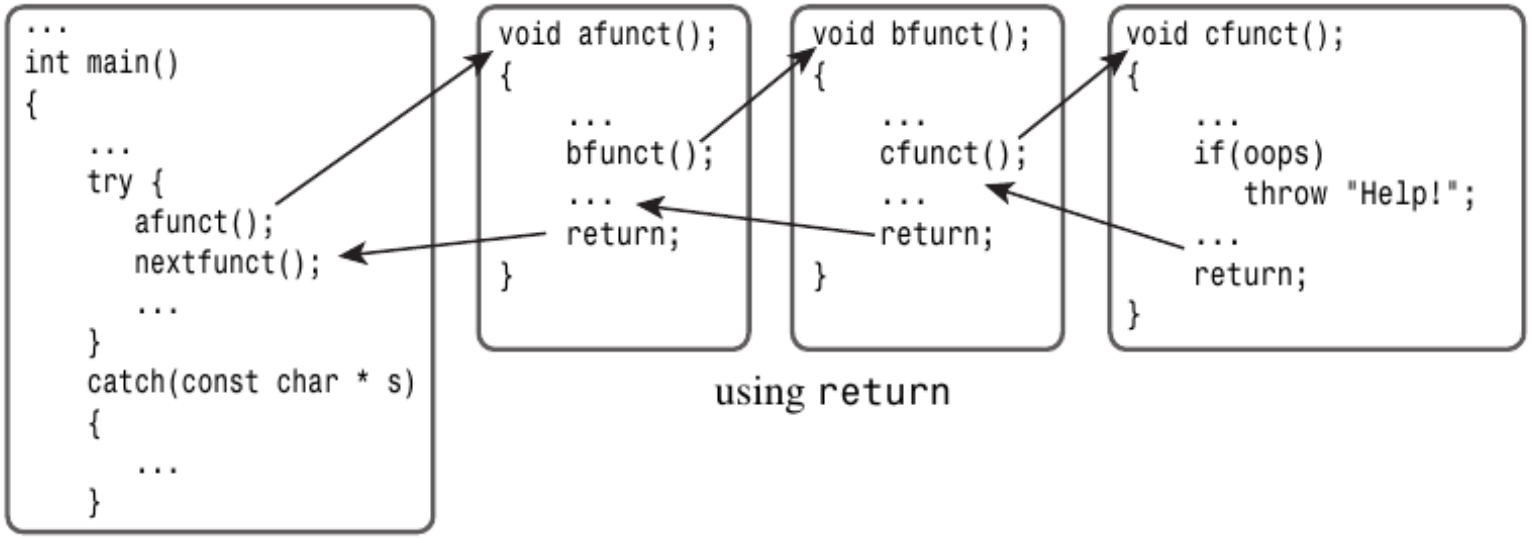


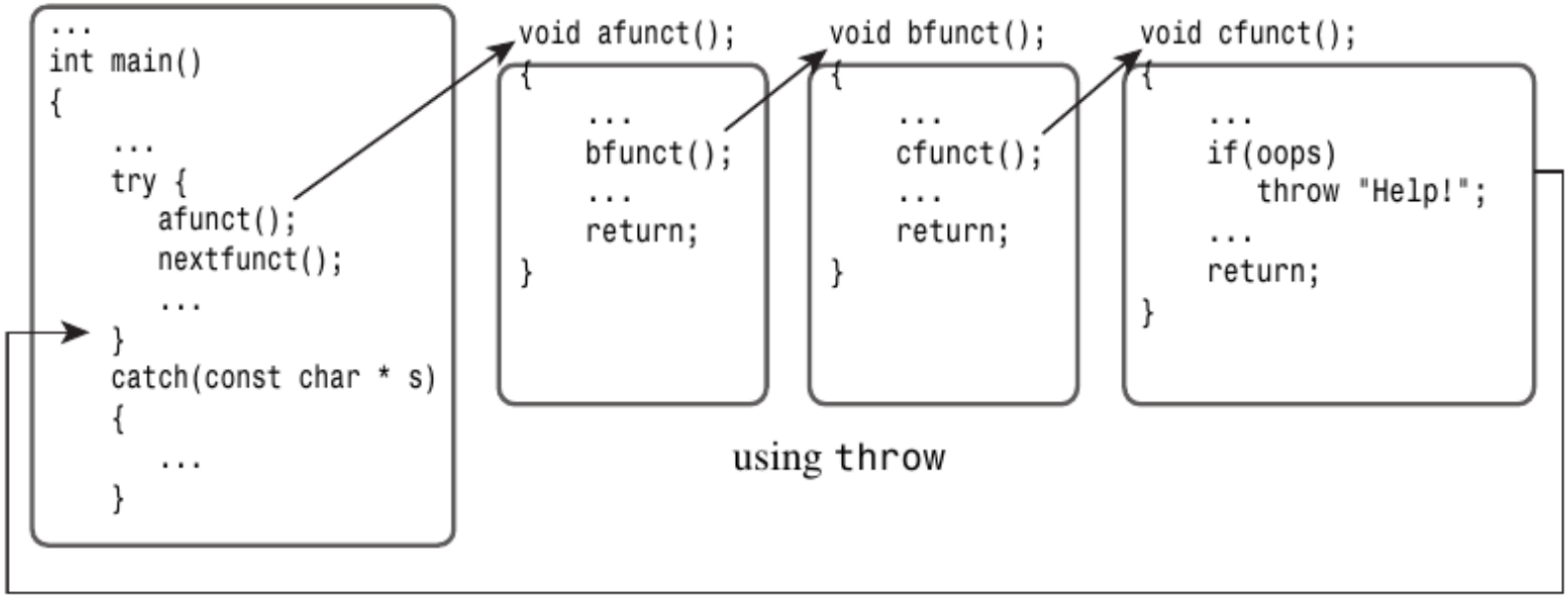
1. The program calls `hmean()` within a try block.

2. `hmean()` throws an exception, transferring execution to the catch block, and assigning the exception string to `s`.

3. The catch block transfers execution back to the `while` loop.

Trace of try/ catch versus return





Order of unwinding

Exception handling must be designed carefully. For example, if execution unwinds before a delete statement in a local scope or in a destructor, memory leaks can occur.

Outline

std exceptions

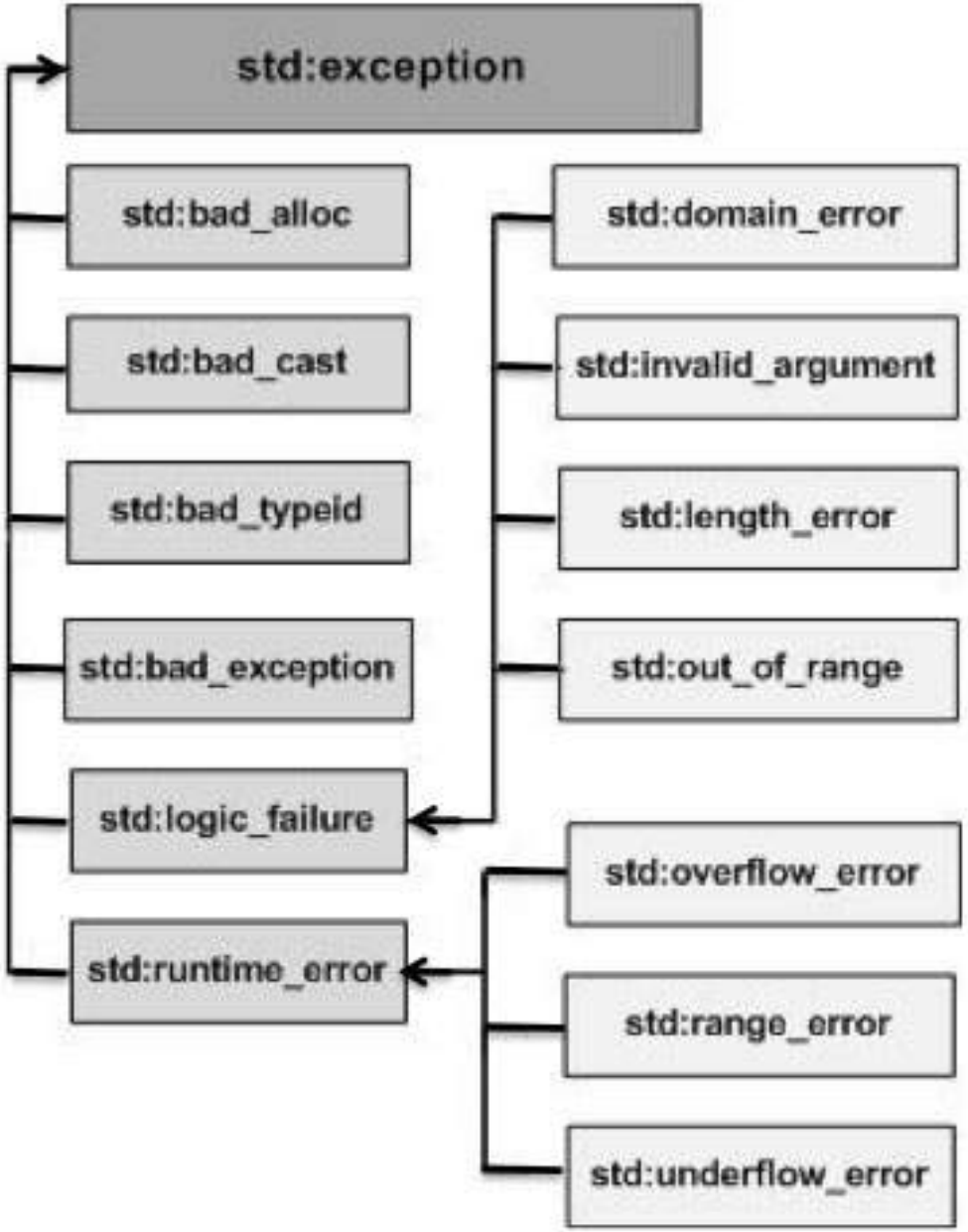

```
#include <exception>
int main(){
  try {
    // do something (might throw an exception)
  }
  catch(const std::exception &e){
    // handle exception e
  }
  catch(...){
    // catches all exceptions,
    // not already caught by a catch block
    // before can be used to catch exception
    // of unknown or irrelevant type
  }
}
```

std exception class


```
class exception{
public:
   exception() throw();
   exception(const exception&) throw();
   exception& operator= (const exception&) throw();
   virtual ~exception() throw();
   virtual const char* what() const throw();
}
```

what() is commonly used – check out the code

Exception family tree (inheritance)



Limiting which errors (if any) a function can throw


```
// no exceptions allowed from f: C++98 style
int f(int x) throw();

// only int exceptions allowed from f: C++98 style
int f(int x) throw(int);

// no exceptions allowed from f: C++11 style
int f(int x) noexcept;
```

C++98 does not enforce exception specifications at compile time. For example, the following code is legal:

```
void DubiousFunction(int iFoo) throw()
{
    if (iFoo < 0)
    {
        throw RangeException();
    }
}
```