High Level Parallel Programming

C++ Exceptions

Alessandro Savino and Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Exceptions

- Errors can be dealt with at place where error occurs
  - It is easy to see if a proper error checking has been implemented
- Exception handling
  - Makes programs clearer, more robust, and fault-tolerant
- Common failures translated into exceptions
  - Function **new** does not allocate memory
  - Out of bounds array subscript
  - Division by zero
  - Invalid function parameters

It is easy to explicitly check this condition

It is **not** possible to explicitly check these conditions

Exceptions

- Exceptions deal with code problems that cannot be easily captured within the standard code flow
- Exceptions catch errors before they occur
  - Deal with synchronous errors (i.e., divide by zero)
  - Do not deal with asynchronous errors
    - Disk I/O completions, mouse clicks, use interrupt processing
  - Used when system can recover from error
    - The run through an exception handler, i.e., a recovery procedure
  - Useful when program cannot recover but must shut down cleanly

Exceptions

- Exception 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
  - Write large projects that require uniform error processing

Exceptions

- Exception handling should not be used for program control
  - The code is not optimized
  - It can reduce program performance

Exception handling

- C++ supports exceptions similarly to other languages
  - Exceptions transfer control and information up the call stack
  - While transferring control up the call stack, C++ performs stack unwinding
    - Properly cleans up all objects with automatic storage duration
    - Ensures correct behavior, e.g., of RAII classes
  - Can be thrown by throw-expressions, dynamic_cast, new-expressions and some standard library functions

Exception handling

- Exceptions use three keywords
  - **throw** e:
    - Constructs an exception object, e, and takes it out of an enclosing context defined by a try block
  - **try** {…}:
    - Defines, for thrown exceptions, an enclosing context with specified catch handlers
  - **catch** (E &e) {…}:
    - Exception handler catch(E &e) responds to an exception object of type “E”
    - Notice that the catch is preferable by reference

Example

Generic example
No errors

If no exception is thrown in the try section the code in the try block is executed, the catch clause is skipped, and computation resumes after the catch clause

```
try {
  // some code that may throw an exception
} catch(exception &e) {
  // some processing to attempt to recover from
  // error based on information carried
  // by the exception
}
```

Example

Generic example
With an error

If an exception is thrown in the try block the remaining code in the try block is skipped, and a matching catch clause is entered, if found. Computation resumes after the last statement in matching catch clause. Matching is based on the type of the exception.

```
try {
  // some code that may throw an exception
} catch(exception &e) {
  // some processing to attempt to recover from
  // error based on information carried
  // by the exception
}
```

Bad Allocation Example

Example on allocation
With an error

```
#include <iostream>
#include <new>

using namespace std;
int main() {
   double *ptr[50];
   try {
     for (int i=0; i<50; i++) {
        ptr[i] = new double[5000000];
     }
   }
   catch (bad_alloc &exception) {
     cout << "Exception occurred: "
          << exception.what() << endl;
   }
   return 0;
}
```

If I get a problem (e.g., memory exhausted)

Reference mode (most used)

Warning: In here we should free all previously allocated memory

All exception have a what method to be used to describe the exception

Multiple Error Handling

- Exception handlers are often chained at the end of a try block to catch all possible desired exceptions

```
try {
  // some code that may throw an exception
  ...
}
catch (T1 &t1) {
  // processing for type T1
}
catch (T2 &t2) {
  // processing for type T2
}
```

Reference mode (most used)

Exceptions which are not specified in the cath list are not captured

Exception Handling

- C++ exception handling guarantees that
  - When an exception forces the execution to leave a scope, all objects built in that scope that have been successfully constructed will have their destructors called
- However, there may be unreleased resources
  - Resources may have been allocated when the exception has been thrown
  - The catch handler should explicitly
    - Delete space allocated by new instruction
    - Close all opened files

Exception Handling

- If there is no exception handler matching the throwning object
  - 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

Exception Handling

- 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
- Always follow the rule
  - Throw by value
  - Catch by reference

Exception Specifications

- A function can declare exception specifications

```
void f() throw (E1, E2, E3);
```

  - Declares that f may throw any of E1, E2, or E3

```
void f() throw ();
```

  - Declares that f does not throw any exception

```
void f();
```

  - Declares that f does can throw any type of exception

Example

A simple exception handling example: divide-by-zero

```
#include <iostream>

using namespace std;


class DivideByZeroException {

  public:
    DivideByZeroException()
      : message("attempted to divide by zero") { }
    const char *what() const {return message;}

  private:
    const char *message;
};
```

The user's class handling the error

Default constructor

The object message is assigned a constant pointer to a char

Example

```
double quotient (int numerator, int denominator) {

  if (denominator == 0)
    throw DivideByZeroException();

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

It checks for a divide-by-zero exception

I throw an object of that class

Example

```
int main() {
  int number1, number2;
  double result;
  cout << "Enter 2 integers: ";
  while (cin >> number1 >> number2) {
    try {
      result = quotient (number1, number2);
      cout << "Quotient: " << result << endl;
    } catch ( DivideByZeroException &ex ) {
      cout << "Exception: " << ex.what() << '\n';
    }


    cout << endl << "Enter 2 integers: ";
  }
  cout << endl;
  return 0;
}
```

End-of-file to endr

Exception handler

Example

- Output (example)

```
Enter two integers: 100 7
Quotient: 14.2857

Enter two integers: 100 0
Exception: attempted to divide by zero

Enter two integers: 33 9
Quotient: 3.66667

Enter two integers:
```

Standard Exceptions

- Standard Exception class
  - Can be used when we want to throw an exception which is not defined by the user

```
namespace std {
  class exception {

    public:
      virtual const char* what() const throw();

    // create, copy, assign, destroy
    // exception objects
  }
}
```

This function does not throw anything

The "what" function cannot throw an exception otherwise we create a recursive throw mechanism

Standard Exceptions

- Standard exceptions can be thworn in the following way

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

using namespace std;

int main() {
   try {
      // do something that can throw a standard e
   } catch(exception& e) {
      std::cout << e.what() << std::endl;
   }
}
```

Function terminate

- Terminate is a function pointer with default value the C library function abort()
- You can define your own terminate handler using
  - set_terminate(void(*)());
- Example

```
void Arnold() { std::cout << "I'll be back" }

int main() {
  set_terminate(Arnold);
  ...
}
```

Specification Violations

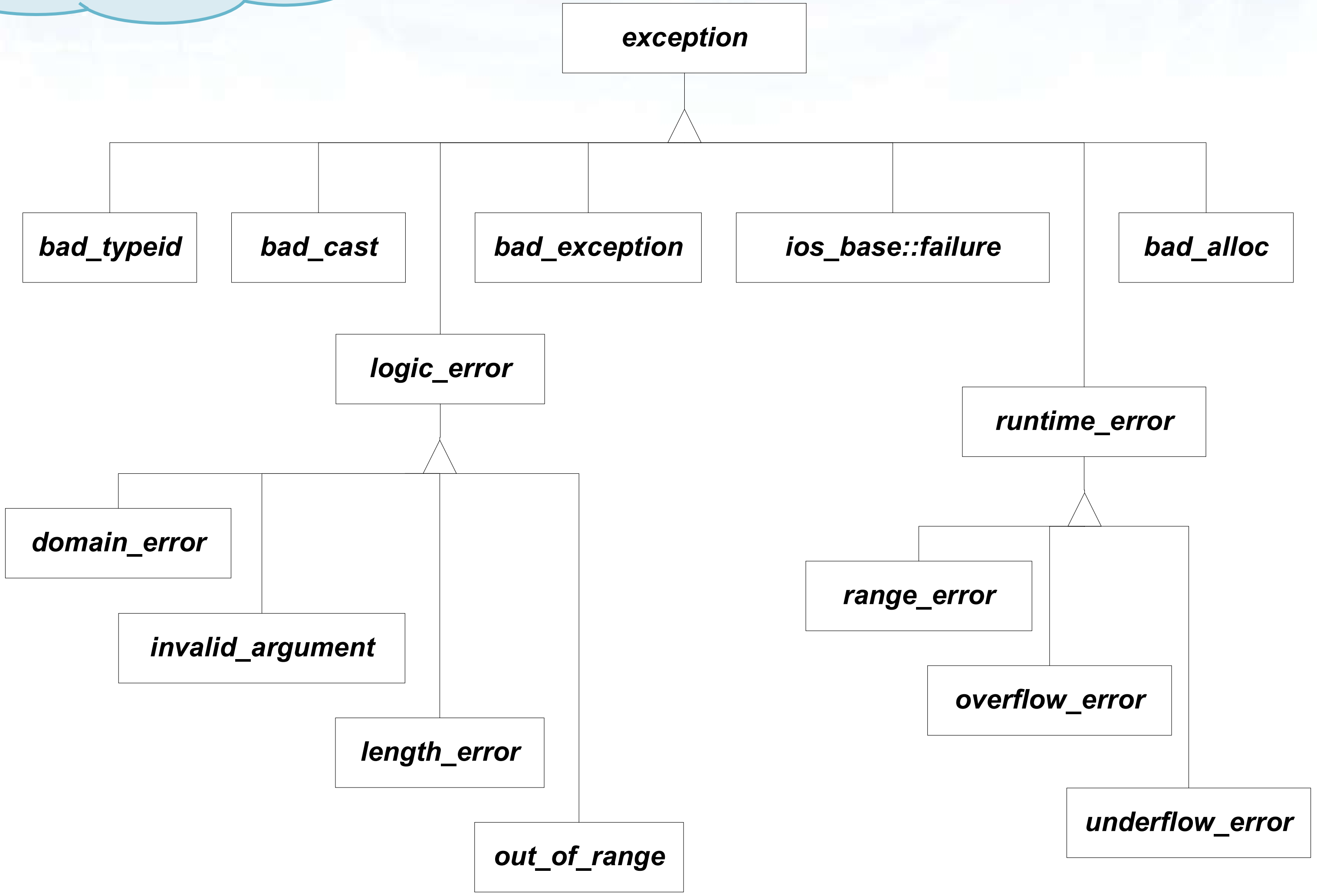
- If an exception specification is violated, the special function unexpected() is called when the exception is thrown
- By default unexpected() terminates execution
  - However, you may change that behavior by defining your own

```
void FreddyKrueger() { … }

int main(){
   set_unexpected(FreddyKrueger);
   ...
}
```

Standard Exceptions

Exception full set



Exception Safety

- Sutter et al. [2000] define different levels of exception safety
- Basic guarantee
  - In the presence of exceptions thrown by called global functions, object messages, template parameters, or library calls, the code
    - Will not leak resources
    - Will maintain in a consistent state, even if this state can be unpredictable

Exception Safety

- Strong guarantee
  - If an operation terminates because of an exception, program state will remain unchanged
  - This implies commit-or-rollback semantics, including that no references or iterators will become invalid if an operation fails

Exception Safety

- Nothrow guarantee
  - A function will not emit an exception under any circumstances
  - Exception safety isn't possible unless certain functions are guaranteed not to throw exceptions

Exception Safety

- To implement strong exception safety
  - In each function, take all the code that might emit an exception and do all its work safely off-to-the-side
    - Only when you know that work has succeeded should you modify program state, by swapping current state with the off-to-the-side state, using only non-throwing operations like pointer swaps
  - Destructors must always provide the nothrow guarantee, since destructors are called in the scope of an exception and a second active exception will always immediately call **terminate** without further cleanup