Exceptions

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Motivation

Programs encounter errors during execution ...

- Mal-formed or incomplete input
- Filesystem errors (files not found, ...)
- Network connection errors
- ...

Errors do not happen all the time, but do happen sometimes! How to handle them programmatically?

We will see how to use **exceptions** for this purpose. Discussed examples are available at GitHub.

Error handling - possible strategies ...

- Ignore the error.
- Use assertions.
- `abort` / `exit` the program.
- Use error codes.
- **Use exceptions**

Error handling - possible strategies ... (cont.)

Ignore the error!

- Program has no support for dealing with the error.
- It assumes the error will not occur. But ... when the error does occur, the program will likely have an abrupt (e.g. crash) or erroneous behavior.

Error handling - possible strategies ... (cont.)

```
assert condition;
```

Assertions

`assert condition` terminates the program if the `condition` is false ((see documentation)

Assertions are very common (and useful!) to validate that certain logical conditions *never* happen. The use of assertions is not suitable for handling errors *that can* occur.

Error handling is too simplistic: a program stops immediately when an error is detected. Assertions are also enabled if code is generated in "debug mode".

Error handling - possible strategies ... (cont.)

```
if (error_condition) exit(1);
```

Using `exit` or `abort`

Another possible error handling strategy is to call `exit` or `abort` in case an error is detected.

The program ends immediately. This strategy is not very suitable in many cases, e.g. in the context of a library that must provide client code with a convenient way of dealing with errors.

Error handling - possible strategies ... (cont.)

Using error codes

```
int err = func();
if (err != 0) {
   . . . // handle error
}
```

Possible error is inferred by the value returned by a function.

This mechanism is employed by C library functions or POSIX functions of the operating system. For example, `printf` returns a negative value in case of a writing error or invalid print format.

Error handling must also occur immediately upon exit from a function, which can lead to verbose code and very scattered error handling logic.

Using exceptions

In C++ (and also Python, C#, Java ...) we can use **exceptions** to deal with errors in structured manner.

- Errors are flagged at the point they are detected by **throwing an exception** using the `throw` statement.
- **Exceptions thrown can be handled** at another code point using `try-catch` blocks.

Example 1

```
try {
  std::cout << "Enter a positive integer: ";
  int n; std::cin >> n;
  if (n <= 0)
    throw std::logic_error("expected positive integer");
  std::cout << "The number is " << n << std::endl;
}
catch(std::logic_error& e) {
  std::cout << "Error: " << e.what() << std::endl;
}
```

Normal execution

```
Enter a positive integer: 1
The number is 1.
```

Execution with exception thrown

```
Enter a positive integer: 0
Error: expected positive integer
```

Example 1 (cont.)

```
try {
  . . .
  if (n <= 0)
    throw std::logic_error("expected positive integer");
  std::cout << "The number is " << n << std::endl;
}
catch(std::logic_error_error& e) {
  std::cout << "Error: " << e.what() << std::endl;
}
```

Exception is thrown if `n <= 0` using `throw`. What happens?

- remaining instructions within the `try` block are not executed; and
- and the execution flow continues to the `catch` block, which is called the **exception handler**.

In this case the exception is an object of type `std::logic_error`, one of the exception classes defined by the C++ library (to be discussed later), and `what()` returns an error message for the exception.

Example 2

```
int read_positive_int() {
  int n;
  std::cin >> n;
  if (n <= 0)
    throw std::logic_error("expected positive integer");
  return n;
}
int main() {
   try {
     std::cout << "Enter a positive integer: ";
     int n = read_positive_int();
     std::cout << "The number is " << n << std::endl;
   }
   catch(std::logic_error& e) {
     std::cout << "Error: " << e.what() << std::endl;
   }
}
```

Example 2 (cont.)

```
int read_positive_int() { . . .
  throw std::logic_error("expected positive integer");
. . . }
int main() {
   try {
     . . . int n = read_positive_int(); . . .
   }
   catch(std::logic_error& e) {
     std::cout << "Error: " << e.what() << std::endl;
   }
}
```

In this example, an exception (`std::logic_error`) can be thrown by a function (`read_positive_int()`) and handled by the caller of the function (`main`). When thrown, `read_positive_int()` terminates and execution continues in the `catch` block of `main`. Subsequent instructions in the `try` block in `main` are not executed (`read_positive_int` does not return a value).

Example 3

```
class time_of_day {
. . .
public:
  time_of_day(int h, int m) {
    if (h < 0 || h > 23 || m < 0 || m > 59)
      throw std::logic_error("invalid args");
    . . .
  }
```

The constructor of a `time_of_day` class validates if supplied `int` arguments `h` and `m` define a valid time. `std::logic_error` is thrown if that does not happen, and the object construction is aborted in this case.

Example 3 (cont.)

```
try {
  time_of_day a(23, 59); // normal execution
  time_of_day b(24, 2);  // exception thrown
  time_of_day c(23, 12); // not executed
} catch (std::logic_error& e) {
  std::cout << "Error: " << e.what() << std::endl;
}
```

Only object `a` gets defined (constructed), `b` and `c` do not.

Another important aspect is that the destructor of `a` is called before the `catch` block is executed. For all objects defined until an exception is thrown, there is the guarantee that their destructor is invoked.

The `throw` instruction in summary

When we execute

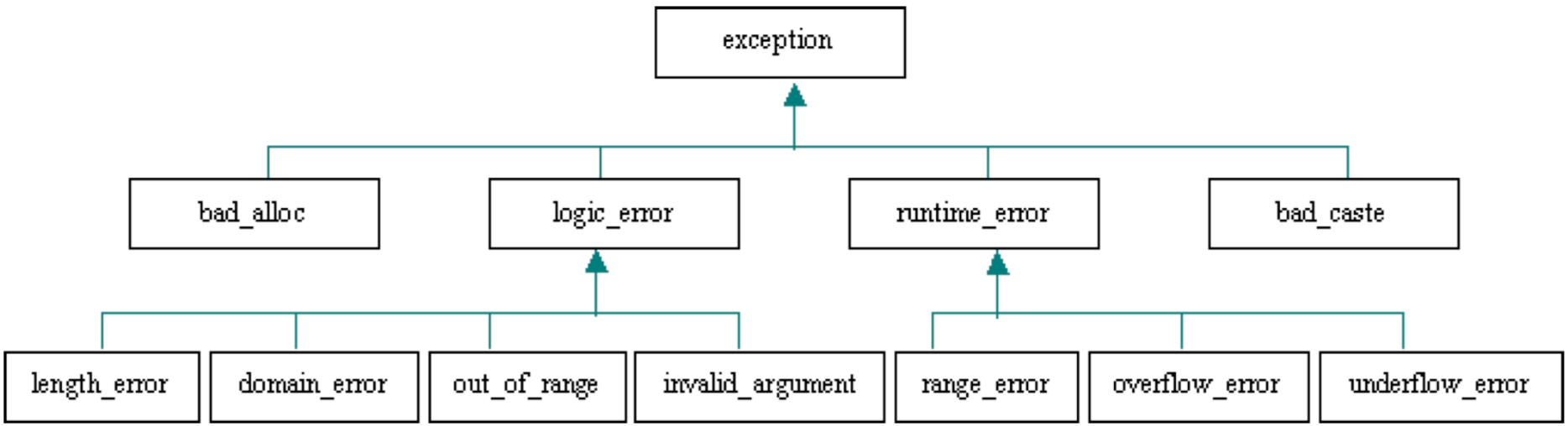
```
throw e;
```

- Exception `e` is thrown, terminating the current sequential control flow.
- Execution resumes in the first function of the call stack that has a `catch` block for the type of `e`. For each function that is terminated by the exception, destructors are invoked for all objects that were created before.
- When there is no `catch` block, the program prints an error message by default. This default behavior can be changed using terminate_handler.

Exception types

```
throw e;
```

`e` can have any type (for instance even an `int` value), but usually an exception class is employed, such as those defined in the `<stdexcept>` header defining a hierarchy of standard exception classes.



Exception types (cont.)

`logic_error`	exception class to indicate violations of logical preconditions or class invariants (class)
`invalid_argument`	exception class to report invalid arguments (class)
`domain_error`	exception class to report domain errors (class)
`length_error`	exception class to report attempts to exceed maximum allowed size (class)
`out_of_range`	exception class to report arguments outside of expected range (class)
`runtime_error`	exception class to indicate conditions only detectable at run time (class)
`range_error`	exception class to report range errors in internal computations (class)
`overflow_error`	exception class to report arithmetic overflows (class)
`underflow_error`	exception class to report arithmetic underflows (class)

The `<stdexcept>` exceptions are used by the C++ library. For instance `at` in `std::vector` may throw `std::out_of_range`:

Exceptions

`std::out_of_range` if `!(pos < size())`.

Exception types (cont.)

Member function `what()` returns a string explaining the cause of error:

```
try {
   . . .
}
catch(std::exception& e) {
  std::cout << "Error: " << e.what() << std::endl;
}
```

Example 4

New exception classes can be defined as derived classes of those in `<stdexcept>`.

```
class invalid_time : public std::logic_error {
public:
    invalid_time()
      : logic_error("invalid time") { }
};
class time_of_day {
. . .
public:
    time_of_day(int h, int m) {
        if (h < 0 || h > 23 || m < 0 || m > 59)
            throw invalid_time();
        . . .
    }
. . .
};
```

Multiple `catch` blocks

```
try {
  code_that_may_throw();
}
catch (type_1& e) {
  handle_type_1_exception(e);
}
catch (type_2& e) {
  handle_type_2_exception(e);
}
catch(...) {
  handle_other_exceptions();
}
```

We can have several catch blocks corresponding to different types of exceptions. A `catch(...)` block can be specified last to handle exceptions of any other type.

Example 5

Suppose `f` can throw `std::runtime_error`, `std::logic_error` and `std::out_of_range`. We can individually define `catch` blocks for each of the exception types:

```
try {
  f(123);
}
catch(std::out_of_range& e) { . . . }
catch(std::logic_error& e) { . . . }
catch(std::runtime_error& e) { . . . }
```

Example 5 (cont.)

Variant 2 (see provided code): we can group the handling of `logic_error` and `out_of_range` in a single `catch` block for `logic_error`, since `out_of_range` is a subclass of `logic_error`.

```
try {
  f(123);
}
catch(std::logic_error& e) { . . . }
catch(std::runtime_error& e) { . . . }
```

Example 5 (cont.)

Variant 3 (see provided code): we can handle all exception types using a single `catch` block for `std::exception`, as it is a base class for all the others in the example.

```
try {
  f(123);
}
catch(std::exception& e) { . . . }
```

Variant 4 We can also use a `catch(...)` block that catches any type of exception.

```
try {
  f(123);
}
catch(...) { . . . }
```