سلسلة تعلم البرمجة بلغة C++ الحديثة

Learn Modern C++ Programming Course

إعداد المهندس أحمد الديب

#24: Exception Handling

Error Handling

- The discussion of errors focuses on errors that cannot be handled locally (within a single small function), so that they require separation of error-handling activities into different parts of a program.
- Library author can detect error but does not know how to handle them.
- User of a library knows how to handle errors but can not detect them.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Traditional Error Handling

- Terminate the program
  - drastic approach
- Return an error value.
  - every call must be checked for the error value
  - callers often ignore return value
  - some operations simply do not have return values; e.g. a constructor
- Return a legal value and leave the program in an error state
  - many standard C library functions set the nonlocal variable `errno` to indicate an error
- Call an error-handler function

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Exceptions

```
struct Some_error {
  std::string what;
};

int do_task() {
  int result = 0;
  if (result) {
    return result;
  } else {
    throw Some_error{"problem !!"};
  }
}

void taskmaster() {
  try {
    auto result = do_task();
    // use result
  } catch (Some_error error) {
    // failure to do_task: handle problem
    std::cout << error.what << std::endl;
  }
}
```

- The returning function must leave the program in a good state and not leak any resources.
- The exception-handling mechanism is integrated with the constructor/destructor mechanisms and the concurrency mechanisms to help ensure that.
- An exception is an object thrown to represent the occurrence of an error. It can be of any type that can be copied, but it is strongly recommended to use only user-defined types.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Stack Unwinding

```
struct Some_error {
  std::string what;
};

int do_task() {
  int result = 0;
  if (result) {
    return result;
  } else {
    throw Some_error{"problem !!"};
  }
}

void taskmaster() {
  try {
    auto result = do_task();
    // use result
  } catch (Some_error error) {
    // failure to do_task: handle problem
    std::cout << error.what << std::endl;
  }
}
```

- The exception object caught is in principle a copy of the one thrown.
- The exception is passed (back) from called function to calling function until a suitable handler is found.
- The type of the exception is used to select a handler in the catch-clause of some try-block.
- In each scope exited, the destructors are invoked so that every fully constructed object is properly destroyed.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Invariants

- What is assumed to be true for a class is called a class invariant.
- It is the job of a constructor to establish the invariant for its class (so that the member functions can rely on it) and for the member functions to make sure that the invariant holds when they exit.

```
void test() {
  try {
    std::vector<int> vec(-10);

  } catch (std::length_error&) {
    std::cerr << "test failed: length error\n" << std::endl;
    throw;  // rethrow

  } catch (std::bad_alloc&) {
    std::cerr << "test failed: memory exhaustion\n" << std::endl;
    std::terminate();  // terminate the program
  }
}
```

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Exception-safe

- We call an operation exception-safe if that operation leaves the program in a valid state when the operation is terminated by throwing an exception.
- We assume that a class has a class invariant.
- We assume that this invariant is established by its constructor and maintained by all functions with access to the object's representation until the object is destroyed.

Source: The C++ Programming Language (4th Edition), Bjarne Stroustrup

Thank you