سلسلة تعلم البرمجة بلغة C++ الحديثة

Learn Modern C++ Programming Course

إعداد المهندس أحمد الديب

#25: Exceptions & Resource Management

Resource Management

```
void use_file(const char* filename) {
  std::FILE* file = std::fopen(filename, "r");
  // use the file
  std::fclose(file);
}
```

Dangerous

```
void use_file(const char* filename) {
  std::FILE* file = std::fopen(filename, "r");
  try {
    // use the file
  } catch (...) {  // catch every possible exception
    std::fclose(file);
    throw;
  }
  std::fclose(file);
}
```

verbose, tedious, and potentially expensive. Worse still, such code becomes significantly more complex when several resources must be acquired and released.

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

Resource Management Using RAII

```
class FileHandler {
 public:
  FileHandler(const char* filename, const char* mode)
      : file{std::fopen(filename, mode)} {
    if (file == nullptr) throw std::runtime_error{"Can't open file"};
  }
  // TODO: implement other constructors, e.g. move & copy
  ~FileHandler() { std::fclose(file); }
  operator std::FILE*() { return file; }  // implicit conversion operator

 private:
  std::FILE* file;
};


void use_file(const char* filename) {
  FileHandler fh{filename, "r"};
  // use the file
}
```

Managing resources using local objects is usually referred to as "Resource Acquisition Is Initialization" (RAII). This is a general technique that relies on the properties of constructors and destructors and their interaction with exception handling.

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

Finally Using RAII

```
template <typename F>
struct Final {
  Final(F f) : clean{f} {}
  ~Final() { clean(); }
  F clean;
};

template <class F>
Final<F> finally(F f) {
  return Final<F>(f);
}

void test() {
  int* p = new int{7};
  auto act1 = finally([&] {
    delete p;
    std::cout << "Goodbye!\n";
  });
  {
    auto act2 = finally([] { std::cout << "finally!\n"; });
  }
}
```

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

noexcept Functions

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


void try_test() {
  try {
    test();
  } catch (std::length_error&) {
    std::cerr << "length error \n";
  }
}


int main() {
  //
  try_test();
}
```

- To indicate that function shouldn’t throw.
- Not completely checked by the compiler and linker.
- If it will throw an exception that wasn’t caught before leaving, the program terminates by invoking `std::terminate()`.

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

Function try-Blocks

```
void test()
try {
  //
  std::vector<int> vec(-10);
} catch (std::length_error &) {
  std::cerr << "length error\n";
}


int main() { test(); }
```

```
class MyVector {
 public:
  MyVector(int size) try : _vec(size) {
  } catch (std::exception&) {
    std::cerr << “exception from MyVector\n";
  }  // implicit "throw;"


 private:
  std::vector<int> _vec;
};
```

By default, if an exception is thrown in a base-or-member initializer, the exception is passed on to whatever invoked the constructor for the member’s class. However, the constructor itself can catch such exceptions by enclosing the complete function body – including the member initializer list – in a try-block.

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

Termination

- Don’t throw an exception while handling an exception. E.g. throwing in class destructors.
- Don’t throw an exception that can’t be caught.

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

Static Assertions

- Exceptions report errors found at run time. If an error can be found at compile time, it is usually preferable to do so.

```
template <int size>
class MyVector {
 public:
  MyVector() : _vec(size) { static_assert(size >= 0, "size is negative"); }

 private:
  std::vector<int> _vec;
};

int main() {
  //
  MyVector<-10> vec;
}
```

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

Example

```
enum class error_action { ignore, throwing, terminating, logging };
constexpr error_action default_error_action = error_action::throwing;


enum class error_code { default_exception, range_error, length_error };
const std::vector<std::string> error_code_name{"default exception",
                                               "range error", "length error"};
constexpr error_code default_error_code = error_code::default_exception;


template <error_action action = default_error_action,
          error_code code = default_error_code, class C>
constexpr void expect(C cond) {
  if constexpr (action == error_action::logging)
    if (!cond())
      std::cerr << "expect() failure: " << int(code) << " "
                << error_code_name[int(code)] << '\n';


  if constexpr (action == error_action::throwing)
    if (!cond()) throw code;


  if constexpr (action == error_action::terminating)
    if (!cond()) std::terminate();
}

int main() {
  expect<error_action::logging, error_code::length_error>([] { return false; });
}
```

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

Thank you