C++ PROGRAMMING

Lecture 5

Secure Software Engineering Group
Philipp Dominik Schubert

CONTENTS

1. Error handling
   A. Return codes
   B. Assertions
   C. Exceptions
2. Function pointers
3. `std::function`

Error handling

- How to handle program errors?
  - Depends on your problem(s)
- More important
  - How to detect, recognize, and handle errors?
- Three (four) important mechanisms

  A. Ignore
     - Do not ignore errors

  B. Return codes

  C. Assertions (static and dynamic)

  D. Exceptions
- Error handling is a very important part of computer programming!

Error handling

- Remember our `scalar_product()` function

```
double scalar_product(const vector<double> &u, const vector<double> &v);
```

- What if `u` and `v` do not have the same length?
- Imagine you are a maths library implementer (who charges a lot of money ;-)
- Your costumers want code that works reliably
  - They want to know when something goes wrong
    - Rather than getting non-sense results

Error handling

- Users of your library would like to know about an error or misuse of the `scalar_product()` function
- Why could this even happen?
  - User has no clue about mathematics
  - User has made a typo
  - User has created the vectors dynamically (and something went wrong)
  - User read data from ill-formatted file
  - …
  - There are lots of sources for errors
    - Because we are humans!

Why our world does not crash

- Our world heavily depends on critical software systems
  - Nuclear power plants
  - Planes
  - Credit institutes
  - Cars
  - Trains
  - Does your grandma use software?
    - Yes, at the grocery store → Cash registers
  - When critical software fails
    - People get injured
    - People get financial ruined

Why our world does not crash

- But how can you board an airplane without fear then?

1. Such systems are heavily restricted and standardized
   - No `new` or `delete` after take-off (planes)
   - No dynamic memory allocation at all (cars)
2. Use error handling (which we will cover today)
3. Use excessive testing
4. Use methods for formal verification, static and dynamic analysis
   - Remember the valgrind memory analysis tool and Clang’s sanitizers
   - Our group focuses on secure software engineering; I work in static analysis (later on)
5. Proving software is usually impossible (sometimes it is possible within a certain scope)
   - Some credit institutes use languages like Haskell (a functional language)
6. Get the best people for the job
   - Bjarne Stroustrup is managing directory for technology at Morgan Stanley

Method I: Using return codes

Use a cleverly designed return code to report a problem:

```
#include <cmath>

double scalar_product(std::vector<double>& u,
                      std::vector<double>& v){
  if (u.size() != v.size()) { return NAN; }
  double result = 0;
  for (size_t i = 0; i < v.size(); ++i){
    result += u[i] * v[i];
  }
  return result;
}
```

```
// a caller might check if the result
// is nan (not a number)

// user generate some data
std::vector<double> a = {1 ,2, 3};
std::vector<double> b = {4, 5};
// user calls your function
double result = scalar_product(a, b);
// user checks for success
if (std::isnan(result)) {
  std::cout << "something went wrong!\n";
} else
  std::cout << "success\n";
}
```

Introduction to special floating point numbers

- When working with floating point types
  - `NAN` is quite common
    - `double value = pow(-1.0, NAN);`
    - `NAN` propagates through calculations
    - Indicates that a value is not a number
  - `inf`
    - `double value = 1.0 / 0.0;`
    - Represents positive infinity
  - `-inf`
    - `double other = -(1.0 / 0.0);`
    - Represents negative infinity

- Useful functions to checks for these values

```
#include <cmath>
std::isnan()
std::isfinite()
std::isinf()
std::isnormal()
```

- Have a look at:

  http://en.cppreference.com/w/cpp/header/cmath

Introduction to special numbers

- Other important values? (on my 64 bit machine)

```
#include <cstddef>
#include <limits>
#include <iostream>
using namespace std;
int main() {
  std::cout << "min int: " << std::dec << std::numeric_limits<int>::min() << '\n';
  std::cout << "max int: " << std::dec << std::numeric_limits<int>::max() << '\n';
  std::cout << "min unsigned: " << std::dec << std::numeric_limits<unsigned>::min() << '\n';
  std::cout << "max unsigned: " << std::dec << std::numeric_limits<unsigned>::max() << '\n';
  std::cout << "double epsilon: " << std::dec << std::numeric_limits<double>::epsilon() << '\n';
  // min int: -2147483648
  // max int: 2147483647
  // min unsigned: 0
  // max unsigned: 4294967295
  // double epsilon: 2.22045e-16
  return 0; }
```

Method I: Using return codes

- Common way of reporting success or failure
- The C programming language makes heavy use of it
- Functions that provide a return value are documented with an error code table
  - Handle an error according to its type
- Return codes are quite common in C++ too
  - That was not always the case
  - Return codes are recommended in google's internal C++ coding guidelines
- Sometimes return codes are not intuitive (remember `scalar_product()`)
  - Maybe `scalar_product()` returns NAN because one of the vectors' entries was NAN
  - Idea: change the signature to

```
int scalar_product(const vector<double> &u,
                   const vector<double> &v, double& result);
```

  - Not smart!

Method I: Using return codes

- Using a smarter version: C++17 `std::optional`

```
#include <iostream>
#include <optional>
#include <vector>
std::optional<double> scalar_product(
                     const std::vector<int> &u,
                     const std::vector<int> &v){
  if (u.size() != v.size()) {
    return std::nullopt;
  }
  double result = 0;
  for (int i = 0; i < u.size(); ++i) {
    result += u[i] * v[i];
  }
  return result;
}
```

```
int main() {
  std::vector<int> a = {1, 2, 3};
  std::vector<int> b = {4, 5, 6};
  std::optional<double> r =
       scalar_product(a, b);
  if (r.has_value()) {
    std::cout << r.value() << '\n';
  }
  std::optional<double> s =
       scalar_product(a, {42, 43});
  std::cout << "has value: "
            << s.has_value();
  return 0;
}
```

Method II: Using assertions

- Find bugs using assertions
- Check if a certain condition holds
- If not, a hard error is reported

- Dynamic assert
  - Evaluated at runtime
  - Can be switched on and off
    - Using the symbol: `NDEBUG`
  - Affects (runtime) performance
  - How to use dynamic assertions?
    a) Develop code using dynamic assertions
    b) Remove them with when you ship your product

```
#include <iostream>
// uncomment to disable assert()
// #define NDEBUG
#include <cassert>

int main() {
  assert(2 + 2 == 5);
  return 0;
}
```

assert's implementation

```
#ifdef NDEBUG
#define assert(condition) ((void)0)
#else
#define assert(condition) /* implementation defined */
#endif
```

- How to print an error message, too?

```
int main() {
  assert(2 + 2 == 5);
  return 0;
}
```

```
int main() {
  assert((2 + 2 == 5) && "This is false!");
  return 0;
}
```

Method II: Using assertions

- Static assert
  - Evaluated at compile time
  - Compiler aborts compilation if a static assertion fails

```
static_assert ( bool_constexpr ,
                message )
static_assert ( bool_constexpr )
```

    - If `bool_constexpr` returns …
      - `true`, this declaration has no effect
      - `false`, a compile-time error is reported and the message is displayed
      - Message has to be a string literal
    - Does not affect (runtime) performance

```
#include <iostream>

int main() {
  static_assert(2 + 2 == 5,
                "This is just false!");
  return 0;
}
```

Dynamic assertions versus static assertions

- Think about the following
  - Errors are bad
  - But an early error is a good error
    - At least better than a late error
  - C/C++: everything that can be done at compile time should be done at compile time!
  - Discover an error early saves
    - Time
    - Money
    - Nerves
    - People

Contracts, functions and invariants

- Functions …
  - Get some input
  - Do some useful work and produce a result
  - Return some output
- A function can be viewed as a contract
  - Preconditions
    - Conditions that hold for the input before processing
  - Postconditions
    - Conditions that hold for the output after processing
  - (`class` / `struct`) invariants
    - Conditions that hold before and after processing
  - If conditions are violated, the application of a function rarely makes sense

Enforcing contracts using assertions

- A function is a contract
- Contracts can be enforced
- Conditions are checked using assertions
- Some conditions are hard or even impossible to express
  - Use a comment in natural language then!
  - Comment your functions!

```
class Car {
 private:
  bool engine_running;

 public:
  bool is_running() {
    return engine_running;
  }
  void stop() {
    assert(is_running());
    stop_engine();
    assert(!is_running());
  }
  void start() {
    assert(!is_running());
    start_engine();
    assert(isrunning());
  }
};
```

Type traits

- Introduced in C++11
- “Type traits defines a compile-time template-based interface to query or modify the properties of types.” [http://en.cppreference.com]
- Use `#include <type_traits>`
- Often implemented using **SFINAE** (later on)
- Type properties and different categories
  1. Primary type categories
  2. Composite type categories
  3. Type properties
  4. Supported operations
  5. Property queries
  6. Type relationships

- Example

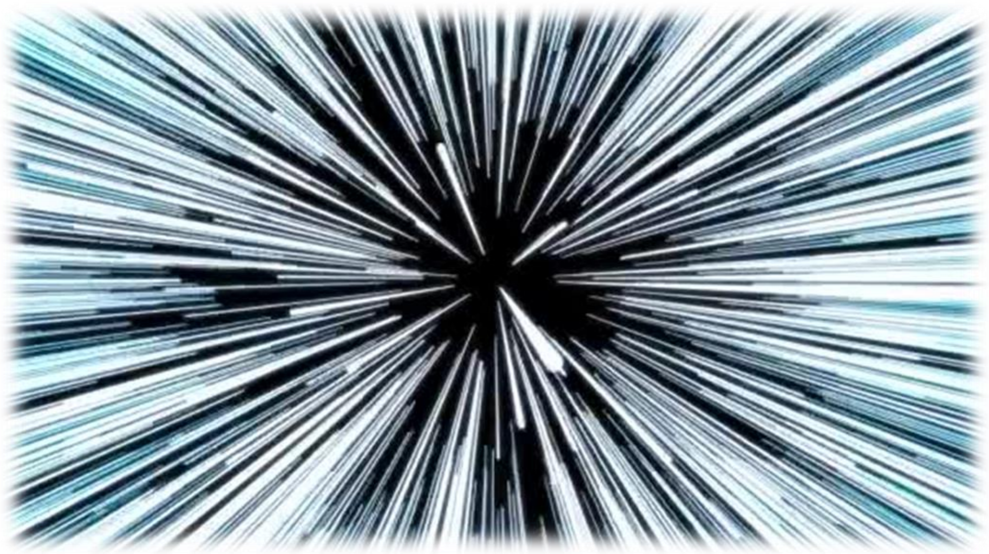
```
#include <iostream>
#include <type_traits>

struct A {};
class B {};

int main() {
  std::cout << std::boolalpha;
  std::cout << std::is_class<A>::value << '\n';
  std::cout << std::is_class<B>::value << '\n';
  std::cout << std::is_class<int>::value << '\n';
  return 0;
}
```

Method III: Using exceptions

- "Exception handling provides a way of transferring control and information from some point in the execution of a program to a handler associated with a point previously passed by the execution …"
- "… in other words, exception handling transfers control up the call stack."



- An exception can be thrown by
  - Throw-expression
  - Dynamic cast
  - Typeid
  - New-expression
  - Allocation function
  - And any of the STL functions specified to throw exceptions to signal a certain error condition

[http://en.cppreference.com/w/cpp/language/exceptions]

Method III: Using exceptions

- … so an exception can be **thrown** to indicate an error
- An exception can be **caught** to handle the error
- In order for an exception to be caught …
  - The throw-expressions has to be contained within a try-block
    - Or inside a function that is called in a try-block
  - And the catch clause has to match the type of the exception

    [http://en.cppreference.com/w/cpp/language/exceptions]
- In summary
  - A certain type of an exception can be thrown to indicate an error
  - An exception can be caught with a catch clause
  - The control flow is transferred to an “earlier” point at which the error can be handled
  - There are some places where you should not throw!
    - This is necessary to guarantee resource safety

Method III: Using exceptions

```
#include <iostream>
#include <stdexcept>
int main() {
  std::vector<double> v = {11, 12, 13};
  double& value = get(v, 0);
  std::cout << value << '\n';
  try {
    double& other = get(v, 100);
    std::cout << other << '\n';
  } catch (std::out_of_range& e) {
    std::cout << "error: " << e.what();
  }
  return 0;
}
```

```
ble& get(std::vector<double>& v,
                 size_t idx) {
  f (idx >= v.size()) {
    // at this point we are in trouble
    throw std::out_of_range("idx: "
     + to_string(idx) + "out of range!");
  } else {
    return v[idx];
  }
}
```

- The `out_of_range` exception transfers the control flow back to the callers catch block!
- The catch block is called exception handler!

[Figure taken from images.google.de]

Method III: Using exceptions

- An exception is a class that contains all information necessary to perform the job
- `#include <stdexcept>` defines, among others, the following useful exception types
  - `std::logic_error`
  - `std::invalid_argument`
  - `std::domain_error`
  - `std::length_error`
  - `std::out_of_range`
  - `std::runtime_error`
  - `std::range_error`
  - `std::overflow_error`
  - `std::underflow_error`
  - You can write your own exception as well
    - Inherit from an exception class and adjust it to your needs (maybe later on)

Method III: Using exceptions

- Please don't

```
#include <iostream>
#include <stdexcept>

int main() try {
  std::cout << "I am trying\n";
  throw std::runtime_error("error");
} catch (std::runtime_error &e) {
  std::cout << “Something went wrong!\n";
  return 0;
}
```

[Figure taken from images.google.de]

What about our scalar product?

- What handling would be adequate?
  - Return codes / assertions / exceptions?

```
#include <stdexcept>
#include <cmath>
double scalar_product(std::vector<double> &u,
                      std::vector<double> &v){
  if (u.size() != v.size()){
    throw std::logic_error("wrong imensions");
  }
  double result = 0;
  for (size_t i = 0; i < u.size(); ++i){
    result += u[i] * v[i];
  }
  return result;
}
```

```
// user generate some data
std::vector<double> a = {1 ,2, 3};
std::vector<double> b = {4, 5};
// user calls your function
double result;
try {
  result = scalar_product(a, b);
} catch (std::logic_error& e) {
  // perform adequate steps
  // perhaps inform the user
  std::cout << "scalar_product has
                thrown!\n";
  std::cout << e.what();
}
```

Never catch like this

```
#include <stdexcept>
#include <cmath>
double scalar_product(std::vector<double> &u,
                      std::vector<double> &v){
  if (u.size() != v.size()){
    throw std::logic_error("wrong imensions");
  }
  double result = 0;
  for (size_t i = 0; i < u.size(); ++i){
    result += u[i] * v[i];
  }
  return result;
}
```

```
// user generate some data
std::vector<double> a = {1 ,2, 3};
std::vector<double> b = {4, 5};
// user calls your function
double result;
try {
  result = scalar_product(a, b);
} catch (std::logic_error& e) {
  // ah, just ignore
}
```

- These things can be seen in real-world code

Re-throwing is possible as well

```
#include <stdexcept>
#include <cmath>
double scalar_product(std::vector<double> &u,
                      std::vector<double> &v){
  if (u.size() != v.size()){
    throw std::logic_error("wrong imensions");
  }
  double result = 0;
  for (size_t i = 0; i < u.size(); ++i){
    result += u[i] * v[i];
  }
  return result;
}
```

```
// user generate some data
vector<double> a = {1 ,2, 3};
vector<double> b = {4, 5};
// more code
// user calls your function
double result;
try {
  result = scalar_product(a, b);
} catch (std::logic_error& e) {
  // the next try - catch - block
  // should take care
  throw;
  // now we go even further upwards
  // and look for another matching
  // catch (std::logic_error e)
}
// more code
```

Stack unwinding

- There are books and papers on this topic
- See http://en.cppreference.com/w/cpp/language/throw
- The principle is not that complicated

```
#include <iostream>
#include <stdexcept>
struct A {
  A(size_t size) : mem(new int[size]) {}
  ~A() { delete[] mem; }
  int *mem;
};
```

- There are no leaks!

```
int main() {
  try {
    A a(2);
    A b(4);
    // more code …
    throw std::runtime_error("crash");
  } catch (std::runtime_error &e) {
    std::cout << "gotcha\n";
  }
  return 0;
}
```

Problems when unwinding the stack

- When exception handling fails and the stack cannot be unwound ➔ `terminate()` is called
  - `terminate()` is called whenever
    - an exception is not caught
    - an exception is thrown while exception handling
    - … there are some more cases
  - `std::terminate()` calls
    - `terminate_handler()`
      - The terminate handler usually leads to hard program termination
    - But you can install your own terminate handler with `set_terminate()`

```
void myHandler() { std::cout << "My own termination handler!"; std::abort(); };
int main() {
  // set terminate handler
  std::set_terminate(&myHandler);
  throw std::runtime_error("crash");
}
```

Specifying functions as `noexcept`

- Functions can be specified to be guaranteed not to throw an exception
- For example small and simple functions that do not throw

```
int add(int a, int b) noexcept {
    return a + b;
}
```

- This keyword is first about semantics
  - You can immediately see that this function does not throw
  - As useful as specifying a member function as `const` if it does not modify its data members
- May lead to a performance increase, compilers may generate faster code
  - Please do not use it blindly → caution: transitivity
- If you lie to the compiler and you throw in a function marked as `noexcept`?
  - `std::terminate()` will be called, which causes program termination
  - Do not lie to the compiler

Specifying functions as `throw`?

- Functions can be specified to indicate that they may throw
- Consider

```
double& give_me(std::vector<double> &v, size_t idx) throw(std::out_of_range) {
  if (idx >= v.size()) {
    throw out_of_range("idx out of range");
  }
  return v[idx];
}
```

- It is about semantics
  - You know what it throws
    - You know that you must have a corresponding exception handler
- **But** it is not good practice to use it → don't do it
  - Compiler cannot check if `std::out_of_range` is thrown or something else
  - The annotation was a bad idea

Why should I care about all the specifiers and qualifiers?

- Reading code is not always easy
- Using specifiers and qualifiers helps
- Good code should document itself
  - Code should immediately tell you what it does
    - Otherwise rewrite it
  - Find useful names for variables, functions, structs, classes, unions, enums

Why should I care about all the specifiers and qualifiers?

- You might have noticed:
  - When you read a function declaration you should immediately …
    - know what it does
    - know how it has to be used
    - know how it behaves
    - but not necessarily how it does its job
  - Otherwise rewrite your code

```
matrix matrix_multiply(const matrix &a, const matrix &b);
int add(int a, int b) noexcept;
class Vec3 {
 private:
  double x, y, z;
 public:
  constexpr Vec3(double a, double b, double c) noexcept;
  constexpr double euclidean_length() const noexcept;
};
```

Pros and cons exceptions [found on stack overflow]

- Pro
  - Separate error-handling code from normal program flow
  - Throwing exceptions is the only clean way to report an error in constructors
  - Hard to ignore
  - Easily propagated from deeply nested functions
  - Carry much more information than an error code
  - Exception objects are matched to the handlers using the type system
  - Automatic stack unwinding
- Con
  - Break code structure by creating invisible exit points that make code hard to read
  - Easily lead to resource leaks when used wrong
  - Learning to write exception safe code is hard
  - Expensive and break the paradigm: only pay for what you use
  - Hard to introduce to legacy code
  - Easily abused for performing tasks that belong to normal program flow

When to use what?

- A rule of thumb (found on stack overflow)
  - Use assertions to catch your own errors
    - Use assertions for functions and data that are internal to your system
  - Use exceptions to catch other peoples errors
    - To check preconditions in public API’s
      - API = application programming interface
    - When dealing with external data that is not under your control
  - Return codes are the poor man’s exceptions

“You cannot throw in destructors and you should not throw in constructors!”

- You cannot throw in destructors
- Think of a dynamically allocated array of variables of user defined types
  - `delete[]`
  - What happens if an exception is thrown while destructing the 2th element?
    - Abort?
      - ➔ You leak!
    - Ignore and continue destructing the remaining variables?
      - **C++ can only have one outstanding exception!**
      - If another exception is thrown you are doomed
        - ➔ You leak!
    - “Do you feel lucky […]?”

“You cannot throw in destructors and you should not throw in constructors!”

- You should not throw in constructors
- What happens if an exception is thrown within the constructor?
  - You fail half way
  - The variable is not set up correctly
    - ➔ Destructor cannot be called
    - No stack unwinding can be performed
    - You have to do it yourself
- Using a return code is not possible
  - Constructors do not have a return value

Pointers again

- Remember pointers

```
int i = 42;
int *i_ptr = &i; // i_ptr points to i
```

- So far we have only seen pointers to variables
- But more bizarre pointers are possible → functions have addresses, too
  - Pointers to functions ☺
  - How does that look like?

```
int (*f)(int, int) = nullptr;
```

    - `f` is a variable of type function pointer to function of type `int (int, int)`
    - In other words: `f` can point to every function that matches this signature
      - Getting two integers as parameters
      - Returning an integer

Pointers to functions

```
#include <iostream>

int mult(int a, int b) {
  return a * b;
}

int perform_binary_operation(int a,
                             int b,
                             int (*f)(int, int)) {
  return f(a, b);
}
```

```
int main() {
  int result =
    perform_binary_operation(4,
                             5,
                             &mult);
  std::cout << result << '\n';
  return 0;
}
```

Why is that useful?

- Now we can pass functions as parameters
- Remember our integrator program integrator.cpp

```
#include <iostream>
#include <cmath> // we use the 'abs()' function
long double integrate(const long double from, const long double to,
                      const size_t N, long double (*function) (long double)) {
  long double integral_val = 0.0;
  long double x = from;
  const long double step_width = std::abs(from-to) / static_cast<long double>(N);
  for (size_t n = 0; n < N; ++n) {
    integral_val += function(x);
    x += step_width;
  }
  return integral_val / N; }
```

- We have abstracted away a concrete function
- A user of `integrate()` can just pass a function pointer
- We can know integrate everything that matches the signature

The `std::function` wrapper

- Fiddling with raw function pointers is not very handy
- Use a wrapper type

```
#include <functional>
int add(int a, int b) { return a + b; }
int perform_binary_operation(int a, int b, std::function<int(int, int)> f) {
  return f(a, b);
}
int main() {
  int result = perform_binary_operation(2, 6, add);
  std::cout << result << '\n';
  return 0;
}
```

- When using function pointers you do not need the '&'

Recap

- Why error handling is important
- Return codes
- Assertions
- Exceptions
- Special floating point values
- Functions and templates to check types and their properties
- When to use what kind of error handling
- Function pointers
- `std::function`

Thank you for your attention
Questions?