EDAF30 – Programming in C++

7. Error handling

Sven Gestegård Robertz

Computer Science, LTH

2020

Outline

1. Error handling
   - Exceptions
   - Catching exceptions
   - Throwing exceptions
   - Exceptions and resource management
   - Specification of exceptions
   - Static assert
   - assert
2. type casts
3. Multidimensional arrays
4. Namespaces and ADL

Error handling
Three levels of error handling

1. Direcly handle the error locally and continue execution
2. Categorize and pass error to another module that is expected to handle it
3. Identify the error, give an error message, and crash the program *("fail-fast", e.g.,* `assert`*)*

Level 2: exceptions (or return values)

Throwing exceptions

Example: checking arguments in the Vector class

```
Vector::Vector(int size) {
  if(size < 0) throw length_error("negative size");
  elem = new double[size];
  sz = size;
}

int& Vector::operator[] (int i) {
  if (i<0 || i>=sz) throw out_of_range("Vector::operator[]");
  return elem[i];
}
```

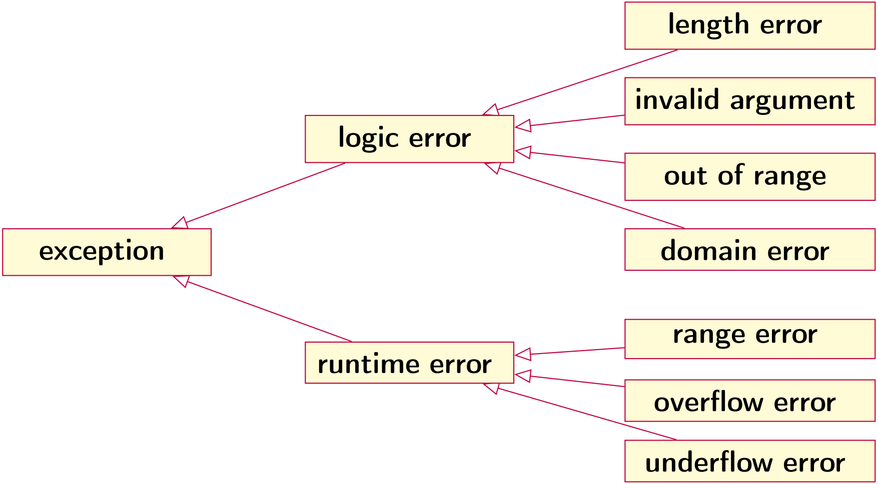
- *NB: to allow checking arguments, we use a* **signed** *integer type for values that should always be positive*
- Vector cannot reasonably handle the error locally, only the caller can know why it passed a certain argument
- std::vector::**operator**[] does no checks (at() does)

Exceptions

- Error handling is done with **`throw`** and **`catch`**. Like Java.
- *"stack unwinding"* until a matching **`catch`** is found.
- When an excepion is thrown, activation records are popped off the stack until a function containing a matching **`catch`** is found. (*"stack unwinding"*)
- If an exception is not caught, the program crashes. (by calling `std::terminate()`.)
- Standard classes for exceptions: `#include <stdexcept>`

The exception classes of the standard library

Class hierarchy for classes in `<stdexcept>`



```
try {
        // Code that may throw
}
catch (some_exception&) {
        // Code handling some_exception
}
catch (another_exception&) {
        // Code handling another_exception
}
catch (...) {
        // default/generic exception handling
}
```

The first **catch** clause with a matching type is selected.
⇒ Catch derived classes before the base class.

... is valid C++, matches anything

Catching exceptions

Example:

```
try {
        cout << "Enter a number: ";
        int i;
        if (cin >> i) {
            int r = f(i);
            cout << "Result: " << r << endl;
        }
}
catch(std::overflow_error&) {
        cout << "Overflow error\n");
}
catch(std::exception& e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
}
```

Catching exceptions

Example:

```
try {
        cout << "Enter a number: ";
        int i;
        if (cin >> i) {
            int r = f(i);
            cout << "Result: " << r << endl;
        }
}
catch(std::overflow_error&) {
        cout << "Overflow error\n");
}
catch(std::exception& e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
}
```

predefined function in the class exception

Catching exceptions
... and rethrowing

```
try{
  do_something();
}
catch {std::ength_error& le) {
  // handle length error
}
catch {std::out_of_range&) {
  // handle out_of_range
}
catch (...) {
  throw;   // defult: pass on
}
```

Throwing exceptions

Creating own exceptions as subclasses

```
#include<stdexcept>

class communication_error : public runtime_error {
public:
        communication_error(const string& mess = "")
                : runtime_error(mess) {}
};
```

Throwing

```
throw communication_error("Checksum error");
```

Throwing exceptions

Creating custom exceptions

```
struct MyOwnException{
  MyOwnException(const std::string& msg, int val)
            : m{msg},v{val} {}
    std::string m;
    int v;
};
```

Using custom exceptions

```
void f() {
    throw MyOwnException("An error occurred", 17);
}

void test1() {
    try{
        f();
    } catch(MyOwnException &e) {
        cout << "Exception: " << e.m << " - " << e.v << endl;
    }
}
```

Catching exceptions
Resource mangement: destructors and “*stack unwinding*”

```
struct Foo {
  int x;
  Foo(int ix) :x{ix} {
    cout << "Foo("<<x<<")\n";}
  ~Foo() {
    cout << "~Foo("<<x<<")\n";}
};

void test(int i)
{
  Foo f(i);
  if(i == 0) {
    throw std::out_of_range("noll?");
  } else {
    Foo g(100+i);
    test(i-1);
    cout << "after call to test("
         << i-1 << ")\n";
  }
}
```

```
int main() {
 Foo f(42);
 try{
   Foo g(17);
   test(2);
 } catch(std::exception& e) {
    cout<<e.what()<< endl; }
}
```

```
Foo(42)
Foo(17)
Foo(2)
Foo(102)
Foo(1)
Foo(101)
Foo(0)
~Foo(0)
~Foo(101)
~Foo(1)
~Foo(102)
~Foo(2)
~Foo(17)
noll?
~Foo(42)
```

Catching exceptions
Resource mangement: destructors and "*stack unwinding*"

```
struct Foo {
  int x;
  Foo(int ix) :x{ix} {
    cout << "Foo("<<x<<")\n";}
  ~Foo() {
    cout << "~Foo("<<x<<")\n";}
};

void test(int i)
{
  Foo f(i);
  if(i == 0) {
    throw std::out_of_range("noll?");
  } else {
    Foo g(100+i);
    test(i-1);
    cout << "after call to test("
         << i-1 << ")\n";
  }
}
```

```
int main() {
 Foo f(42);
 try{
   Foo g(17);
   test(2);
 } catch(std::exception& e) {
    cout<<e.what()<< endl; }
}
```

```
Foo(42)
Foo(17)
Foo(2)
Foo(102)
Foo(1)
Foo(101)
Foo(0)
~Foo(0)
~Foo(101)
~Foo(1)
~Foo(102)
~Foo(2)
~Foo(17)
noll?
~Foo(42)
```

Catching exceptions
Resource mangement: destructors and "*stack unwinding*"

```
struct Foo {
  int x;
  Foo(int ix) :x{ix} {
    cout << "Foo("<<x<<")\n";}
  ~Foo() {
    cout << "~Foo("<<x<<")\n";}
};

void test(int i)
{
  Foo f(i);
  if(i == 0) {
    throw std::out_of_range("noll?");
  } else {
    Foo g(100+i);
    test(i-1);
    cout << "after call to test("
         << i-1 << ")\n";
  }
}
```

```
int main() {
 Foo f(42);
  try{
    Foo g(17);
    test(2);
  } catch(std::exception& e) {
     cout<<e.what()<< endl; }
}
```

```
Foo(42)
Foo(17)
Foo(2)
Foo(102)
Foo(1)
Foo(101)
Foo(0)
~Foo(0)
~Foo(101)
~Foo(1)
~Foo(102)
~Foo(2)
~Foo(17)
noll?
~Foo(42)
```

Catching exceptions
Resource mangement: destructors and "*stack unwinding*"

```
struct Foo {
  int x;
  Foo(int ix) :x{ix} {
    cout << "Foo("<<x<<")\n";}
  ~Foo() {
    cout << "~Foo("<<x<<")\n";}
};

void test(int i)
{
  Foo f(i);
  if(i == 0) {
    throw std::out_of_range("noll?");
  } else {
    Foo g(100+i);
    test(i-1);
    cout << "after call to test("
         << i-1 << ")\n";
  }
}
```

```
int main() {
 Foo f(42);
 try{
   Foo g(17);
   test(2);
 } catch(std::exception& e) {
    cout<<e.what()<< endl; }
}
```

```
Foo(42)
Foo(17)
Foo(2)
Foo(102)
Foo(1)
Foo(101)
Foo(0)
~Foo(0)
~Foo(101)
~Foo(1)
~Foo(102)
~Foo(2)
~Foo(17)
noll?
~Foo(42)
```

Catching exceptions

Resource mangement: destructors and "*stack unwinding*"

```
struct Foo {
  int x;
  Foo(int ix) :x{ix} {
    cout << "Foo("<<x<<")\n";}
  ~Foo() {
    cout << "~Foo("<<x<<")\n";}
};

void test(int i)
{
  Foo f(i);
  if(i == 0) {
    throw std::out_of_range("noll?");
  } else {
    Foo g(100+i);
    test(i-1);
    cout << "after call to test("
         << i-1 << ")\n";
  }
}
```

```
int main() {
 Foo f(42);
 try{
   Foo g(17);
   test(2);
 } catch(std::exception& e) {
    cout<<e.what()<< endl; }
}
```

```
Foo(42)
Foo(17)
Foo(2)
Foo(102)
Foo(1)
Foo(101)
Foo(0)
~Foo(0)
~Foo(101)
~Foo(1)
~Foo(102)
~Foo(2)
~Foo(17)
noll?
~Foo(42)
```

Catching exceptions

Resource mangement: destructors and "*stack unwinding*"

```
struct Foo {
  int x;
  Foo(int ix) :x{ix} {
    cout << "Foo("<<x<<")\n";}
  ~Foo() {
    cout << "~Foo("<<x<<")\n";}
};

void test(int i)
{
  Foo f(i);
  if(i == 0) {
    throw std::out_of_range("noll?");
  } else {
    Foo g(100+i);
    test(i-1);
    cout << "after call to test("
         << i-1 << ")\n";
  }
}
```

```
int main() {
 Foo f(42);
 try{
   Foo g(17);
   test(2);
 } catch(std::exception& e) {
    cout<<e.what()<< endl; }
}
```

```
Foo(42)
Foo(17)
Foo(2)
Foo(102)
Foo(1)
Foo(101)
Foo(0)
~Foo(0)
~Foo(101)
~Foo(1)
~Foo(102)
~Foo(2)
~Foo(17)
noll?
~Foo(42)
```

Catching exceptions
Resource mangement: destructors and "*stack unwinding*"

```
struct Foo {
  int x;
  Foo(int ix) :x{ix} {
    cout << "Foo("<<x<<")\n";}
  ~Foo() {
    cout << "~Foo("<<x<<")\n";}
};

void test(int i)
{
  Foo f(i);
  if(i == 0) {
    throw std::out_of_range("noll?");
  } else {
    Foo g(100+i);
    test(i-1);
    cout << "after call to test("
         << i-1 << ")\n";
  }
}
```

```
int main() {
 Foo f(42);
 try{
   Foo g(17);
   test(2);
 } catch(std::exception& e) {
    cout<<e.what()<< endl; }
}
```

```
Foo(42)
Foo(17)
Foo(2)
Foo(102)
Foo(1)
Foo(101)
Foo(0)
~Foo(0)
~Foo(101)
~Foo(1)
~Foo(102)
~Foo(2)
~Foo(17)
noll?
~Foo(42)
```

Specifying exceptions in C++11

The keyword **noexcept** specifies if a function may throw or not.
No specification is equal to **noexcept**(**false**).

In the function declaration

```
struct Foo {
    void f();
    void g() noexcept;
};
```

and in the function definition

```
#include<stdexcept>
void Foo::f() {
    throw std::runtime_error("f failed");
}
void Foo::g() noexcept{
    throw std::runtime_error("g lied and failed");
}
```

Exception specification

Example usage

```
#include<typeinfo> // for typeid

void test_noexcept()
{
    Foo f;

    try {
        f.f();
    }catch(std::exception &e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
    }
    try {
        f.g();
    }catch(std::exception &e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
    }
    cout << "done\n";
}
```

```
St13runtime_error: f failed
terminate called after throwing an instance of 'std::runtime_error'
  what():  g lied and  failed
```

Exception specification
older C++, do not use

Older C++ had “exception lists” for a function: the types of exceptions that may be generated by the function are specified with the keyword **throw**.

Example of exception list:

```
int f(int) throw(typ1, typ2, typ3) {
        //...
        throw typ1("Error of type 1 occurred");
        //...
        throw typ2("Error of type 2 occurred");
        //...
        throw typ3("Error of type 3 occurred");
}
```

No list	$\Rightarrow$	Any type of exception may be thrown
Empty list (`throw()`)	$\Rightarrow$	No exceptions may be thrown

Rules of thumb for exceptions

- Consider error handling early in the design process
- Use specific exception types, not built-in types.
  (do not `throw 17;`, `throw false;` , etc.)
- "Throw by value, catch by reference"
- If a function should not throw, declare `noexcept`.
- Specify *invariants* for your types
  - The constructor establishes the invariant, or throws.
  - Member functions can rely on the invariant.
  - Member functions must not break the invariant.
  - Example: `Vector`
    - the size `sz` is a positive number
    - the array `elem` points to has size `sz`
    - if the allocation of the array fails `std::bad_alloc` is thrown

If something can be checked at compile-time, use `static_assert`.

Static assert

If something can be checked at compile-time, use static_assert.

```
static_assert ( bool_constexpr , message )         (since C++11)
    message can be omitted.                          (since C++17)

constexpr int some_param = 10;

int foo(int x)
{
    static_assert(some_param > 100, "");
    return 2*x;
}

int main()
{
    int x = foo(5);

    std::cout << "x is " << x << std::endl;
    return 0;
}
```

```
  error: static assertion failed:
      static_assert(some_param > 100, "");
```

Static assert
Type traits

The standard library provides (meta)functions to query properties of types.

```
#include <type_traits>

template <class T>
T foo(const T& t)
{
  static_assert(std::is_copy_constructible<T>::value &&
                std::is_copy_assignable<T>::value,
                "foo requires copying");

    ... // the code of the function
}
```

assert
in <cassert>

A C standard macro for *fail fast* (at run-time).

```
void assert(some expression);
```

Prints an error message and calls `std::abort()`
if the value of the expression is **`false`**.

Example

```
#include <cassert>

std::vector<int> create_vector(int i)
{
    assert(i>=0);

    return std::vector<int>(i);
}
```

Typically only used during development.
If `NDEBUG` is defined, it generates no code.

Type casts
Implicit conversions

Automatic conversions

- Expressions of the type $x \odot y$, for some binary operator $\odot$
  E.g.: **`double + int ==> double`**
  `float + long + char ==> float`
- Assignments and initialization: The value of the right-hand-side is converted to the type of the left-hand-side
- Conversion of an argument to the type of the (formal) parameter
- Expresions in **`if`** statements, etc. $\Rightarrow$ **`bool`**
- built-in array $\Rightarrow$ pointer (*array decay*)
- `0` $\Rightarrow$ `nullptr` (empty pointer in C++11, previously the constant `NULL` was defined)

type casts
Explicit, named type casts (C++-11)

- **static_cast**<new_type> (expr)
  - convert between compatible types (*does not do range check*)
  - "the inverse of a *standard* implicit conversion" (mostly)
- **reinterpret_cast**<new_type> (expr)
  - no safety net, same as C-style cast
- **const_cast**<new_type> (expr) - add or remove **const**
- **dynamic_cast**<new_type> (expr) - use for pointers to objects in class hierarchies. Uses *run-time type info*, like `instanceof` in Java.

Example

```
char c;                // 1 byte
int *p = (int*) &c;    // pointer to int: 4 bytes

*p = 5; // undefined behaviour, stack corruption

int *q = static_cast<int*> (&c); // compiler error
```

Type casting
Explicit type casts, C style

Syntax in C and in C++, like in Java

`(type) expression` , e.g. `(float) 10`

- Greater risk of mistakes - use named casts
  - makes the code clearer, e.g., **`const_cast`** can only change **`const`**
  - easy to search for: casts are among the first to look for when debugging
- Warning in GCC: `-Wold-style-casts`
- Common in older code

Alternative syntax in C++

```
type(expression)
```

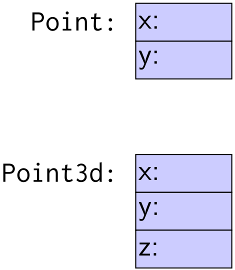
`type` must be *a single word*,

`int *(...)` eller i.e., `unsigned long(...)` is not OK.

Type casts
Warning example

```
struct Point{
    int x;
    int y;
};

struct Point3d {
    int x;
    int y;
    int z;
};
```



Data types and variables

- some concepts:
  - a *type* defines the set of possible values and operations (for an *object*)
  - an *object* is a place in memory that holds a *value*
  - a *value* is a sequence of bits interpreted according to a *type*.

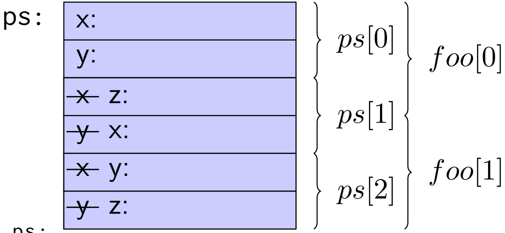
A typecast changes the ***value*** *of a particular memory location by changing how* ***it should be interpreted****.*

Type casts
Warning example

```
struct Point{
    int x;
    int y;
};

Point ps[3];

struct Point3d{
     int x;
     int y;
     int z;
};
Point3d* foo = (Point3d*) ps;
```



With *named casts*, this requires a **reinterpret_cast**<Point3d*>

With **static_cast**<Point3d*> the compiler gives the error
invalid **static_cast** from type 'Point[3] to type 'Point3d*'

special case: **void** pointer

A **void*** can point to an object of any type

In C
: a **void*** is implicitly converted to/from any pointer type.

In C++
: a T* is implicitly converted to **void***. The other direction requires an explicit *type cast*.

Multidimensional arrays

multi-dimensional arrays

- Does not (really) exist in C++
  - are arrays of arrays
  - Look like in Java
- Java: array of *references to arrays*
- C++: two alternatives
  - Array of arrays
  - Array of *pointers* *(to the first element of an array)*

Multi-dimensional arrays

Initializing a matrix with an initializer list:

3 rows, 4 columns

```
int a[3][4] = {
  {0, 1, 2, 3} ,    /*  initializer list for row 0 */
  {4, 5, 6, 7} ,    /*  initializer list for row 1 */
  {8, 9, 10, 11}    /*  initializer list for row 2 */
};
```

Instead of nested lists one can write the initialization as a single list:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

- Multi-dimensional arrays are stored like an array in memory.
- The dimension *closest to the name* is the size of the array
- The remaining dimensions belong to the element type

An array `T array[4]` is represented memory by a sequence of four elements of type `T`: `| T | T | T | T |`

An **int**[4] is represented as

```
|   int   |   int   |   int   |   int   |
```

Thus, **int**[3][4] is represented as

```
| int | int | int | int | int | int | int | int | int | int | int | int |
```

Multi-dimensional arrays

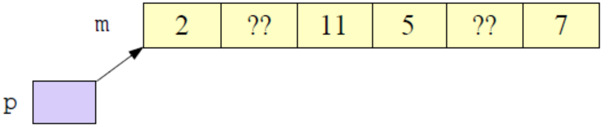
Examples

```
int m[2][3]; // A 2x3-matrix

m[1][0] = 5;

int* e = m;        // Error! cannot convert 'int [2][3]' to 'int*'
int* p = &m[0][0];
*p = 2;

p[2] = 11;
int* q=m[1];       // OK: int[3] decays to int*
q[2] = 7;
```



Multi-dimensional arrays

Parameters of type multi-dimensional arrays

```
// One way of declaring the parameter
void printmatr(int (*a)[4], int n);

// Another option
void printmatr(int a[][4], int n) {
{
   for (int i=0;i!=n;++i) {
      for (const auto& x : a[i]) { The elements of a are int[4]
         cout << x << " ";
      }
      cout << endl;
   }
}
```

Multi-dimensional arrays

Initialization and function call

```
int a[3][4] {1,2,3,4,5,6,7,8,9,10,11,12};
int b[3][4] {{1,2,3,4},{5,6,7,8},{9,10,11,12}};

printmatr(a,3);
cout << "-----------------" << endl;
printmatr(b,3);
```

```
       1        2        3        4
       5        6        7        8
       9       10       11       12
-----------------
       1        2        3        4
       5        6        7        8
       9       10       11       12
```

Argument Dependent Lookup (ADL)

Name lookup is done in *enclosing scopes*, but...

```
namespace test{
    struct Foo{
        Foo(int v) :x{v} {}
        int x;
    };
    std::ostream& operator<<(std::ostream& o, const Foo& f) {
        return o << "Foo(" << f.x << ")";
    }
} // namespace test

int main()
{
    test::Foo f(17);
    cout << f << endl;
}
```

- The function `operator<<(ostream&, const Foo&)` is not visible in `main()`.
- Through ADL it is found in the namespace of its argument (`test`).

Argument Dependent Lookup (ADL)

```
namespace test{
    struct Foo;
    std::ostream& operator<<(std::ostream& o, const Foo& f);

    void print(const Foo& f)
    {
        cout << f << endl;
    }
    void print(int i)
    {
        cout << i << endl;
    }
} // namespace test

int main()
{
    test::Foo f(17);

    print(f);
    print(17);
    test::print(17);
}
```

- The functions `test::operator<<()` and `test::print(const Foo&)` are found through ADL.
- The function `test::print(int)` is not found.
- unless **using** `test::print`.

Suggested reading

References to sections in Lippman

Error handling, exceptions (5.6, 18.1.1)

Namespaces 18.2

static assert *not in Lippman*

assert 6.5.3

Type casts 4.11

const_cast and const overloading 6.2 (p 232–233)

Multi-dimensional arrays 3.6

Next lecture

References to sections in Lippman

swap	13.3
Copying and moving objects	13.4, 13.6
(allocators)	12.2.2
(Classes, dynamic memory allocation)	13.5
Container Adapters	9.6
Pairs	11.2.3
Tuples	17.1