Object Oriented Programming - Lecture 7

Diana Borza - diana.borza@ubbcluj.ro

April 12, 2023

Content

- Exceptions
- Generic programming - templates - compile time polymorphism
- What's new in C++ (cont'd) - range based for loop

Exceptions I

- **Exceptions** provide a mechanism to decouple handling of errors or other exceptional circumstances from the typical control flow of your code.
- When writing reusable code, error handling is a necessity.
- Exceptions allow programmers the freedom to handle errors when and how ever is most useful for a given situation.

Exceptions II

- Error handling without exceptions:
  - Error flags (global variables).
  - Return codes (error codes).
- Problems:
  - By default, the error flags are ignored (unless you write the code to verify the error flag or the error code).
  - Handling code is tedious to write and sometimes hard to understand;
  - What happens if the caller is not able to equipped to handle the error? Or if the caller just ignores the error?
  - What happens if the abnormal situation occurs in a constructor?
  - Error handling code is intricately linked to the normal control flow of the code.

Exceptions in C++ I

- In C++ we use 3 keywords, which operate in conjunction to each other:
  - **throw** - signals that an exception or error case has occurred; used to *raise* an exception.
  - **try** - marks an instruction block that might cause problems (i.e. throw exceptions). Acts like an *observer*, looking for the errors.
  - **catch** - immediately follows the try block and contains the code to handle the error.
    - print an error;
    - return a value or error code back to the caller;
    - may throw another exception.

Exceptions in C++ II

```
void exceptionExample(){

    try {
        // code that may throw an exception
    } catch (ExceptionClass &e) {
        // error handling , if the thrown error is of type ErrorClass ,
        // or any other type derived from ErrorClass
    } catch(...){
        // error handling - any type of error
    }
}
```

Exceptions - execution flow

- If no exception is thrown during execution of the try guarded section, the catch clauses that follow the try block are not executed.
- When an exception occurs, control moves from the throw statement *to the first* catch statement that can handle the thrown type: **stack unwinding**.
- This search continues "down" the function-call stack. If an exception is never caught, the program halts.
- After an exception has been handled the program, execution resumes after the try-catch block, not after the throw statement.
- Exceptions may be re-thrown with throw.
  Rule: When rethrowing the same exception, use the throw keyword by itself.

Catch clauses I

- The type of the argument in the catch block is checked against the type that was thrown.
- Multiple handlers can be chained, with different parameter types.
- The exception is caught by the handler only if the types match (directly or by inheritance). For primitives type, no casting is performed: e.g. an exception thrown with a char cannot be handled by a catch block of int.
- In case of multiple handlers, these are matched in the order of appearance.
- When the thrown objects belong to the same class hierarchy, **the most derived types should be handled first**.

Catch clauses II

- A catch block with an ellipsis (...) as parameter will catch any type of exception.
- A good coding standard is to throw by value and **catch by reference** (to avoid copying the object and to preserve polymorphism).
- The exception object is destroyed only after the exception has been handled (when the (last) catch block completes).
- The exception object is used to transmit information about the error that occurred.
- It is a good practice to create exception classes for certain kinds of exceptions that occur in your programs.

noexcept specifier

- noexcept - is an exception specifier, which is used to indicate that a function cannot throw an exception;
- destructors are generally implicitly noexcept (as they can't throw an exception);
- noexcept is not a compile-time check
  - it is merely a method for a programmer to inform the compiler whether or not a function throws exceptions;
  - the compiler can use this information to enable certain optimizations.
- If a noexcept function does try to throw an exception, then std::terminate is called to terminate the application.

```
void myFunction1() noexcept; // myFunction1 does not throw any exceptions
void myFunction2() noexcept(false); // myFunction2 MIGHT throw exceptions
```

User defined exceptions I

- The exception object thrown can be just about any kind of data structure you like.
- It is a good practice to create exception classes for certain kinds of exceptions that might occur in one's programs.
- The C++ STL provides a base class specifically designed to declare objects to be thrown as exceptions: class std::exception in the header <exception>.
- As of C++17, in STL there are 25 different exception classes that can be thrown (subclasses of std::exception).
  `https://en.cppreference.com/w/cpp/error/exception`

User defined exceptions II

- nothing throws a std::exception directly, and neither should you;
- std::runtime_error (included as part of the stdexcept header) is a popular choice, because it has a generic name, and its constructor takes a customizable message
- You can create a class that inherits from std::exception, the what() method can be overriden (it returns a const char*).

Exceptions

"What good can using exceptions do for me? The basic answer is: Using exceptions for error handling makes you code simpler, cleaner, and less likely to miss errors. But what's wrong with "good old errno and if-statements"? The basic answer is: Using those, your error handling and your normal code are closely intertwined. That way, your code gets messy and it becomes hard to ensure that you have dealt with all errors (think "spaghetti code" or a "rat's nest of tests")." – **Bjarne Stroustrup.**

Exceptions

"Counter-intuitively, the hard part of coding exceptions is not the explicit throws and catches. The really hard part of using exceptions is to write all the intervening code in such a way that an arbitrary exception can propagate from its throw site to its handler, arriving safely and without damaging other parts of the program along the way."" – **Tom Cargill.**

Advantages of using exceptions

- separation of the error handling code from the normal flow of control;
- different types of errors can be handled in one place (inheritance);
- the program cannot ignore the error, it will terminate unless there is a handler for the exception;
- functions will need fewer arguments and return values $\rightarrow$ this makes them easier to use and understand; any amount of information can be passed with the exception.

Downsides of using exceptions

- Exceptions do come with a small performance price to pay.
  - they increase the size of your executable;
  - they may also cause it to run slower due to the additional checking that has to be performed.
- Exception handling is best used when all of the following are true:
  - The error being handled is likely to occur only infrequently.
  - The error is serious and execution could not continue otherwise.
  - The error cannot be handled at the place where it occurs.
  - There isn't a good alternative way to return an error code back to the caller.

Exception-safe code

- If an exception occurs, an exception-safe code means that:
  - **there are no resource leaks;**
  - **there are no visible effects;**
  - **the operation is either completely executed, or not executed at all (transaction).**
- For every managed resource (e.g. memory, file) - create a class.
- Any pointer will be encapsulated in an object which is automatically managed by the compiler.
- Such objects will be created locally in the function (not allocated on the free store).
- This way - one ensures that the destructor is called when the execution leaves scope (even if an exception is thrown).
- Use RAII - Resource Acquisition Is Initialization.

Lambda expressions I

- A **lambda expression** (**lambda** or **closure**) allows the definition of an anonymous function inside another function.
- The anonymous function is defined in the function where it is called.
- Very useful for some algorithms defined in the STL std::find_if, std::count_if, std::transform.
- Syntax:

```
[ captureClause ] ( parameters ) -> returnType
{ function body;}
[captureClause] (parameter list) {function body;}
```

  - *captureClause* and *parameters* can be empty if not needed;
  - *returnType* is optional, and if omitted, auto will be assumed.

Lambda expressions II

- The **captureClause** clause is used to (indirectly) give a lambda access to variables available in the surrounding scope that it normally would not have access to.
- We need to list all the variables we want to access from within the lambda as part of the capture clause.
- For each variable in **captureClause**, **a clone!** of that variable is made (with an identical name) inside the lambda.
- A default capture (also called a capture-default) captures all variables that are mentioned in the lambda.
  - To capture all used variables by value, use a capture value of $=$.
  - To capture all used variables by reference, use a capture value of &.

STL algorithms

- algorithms are implemented as functions that operate using iterators'
- std::min_element() and std::max_element() algorithms find the min and max element in a container class;
- std::find() algorithm to finds a value in a container; returns the end of the iterator if the element is not present in the container;
- std::sort() - sorts a container; doesn't work on list container classes!
- std::reverse() - reverses a container;

Templates I

- *Generic programming* - algorithms are written with generic types, that are going to be specified later.
- Generic programming is supported by most modern programming languages.
- Templates allow working with generic types.
- Provide a way to reuse source code. The code is written once and can then be used with many types.
- Allow defining a function or a class that operates on different kinds of types (is parametrized with different types).

Templates - the most famous program that didn't compile

- The first concrete demonstration templates of this was a program written by *Erwin Unruh*, which computed prime numbers although it did not actually finish compiling: **the list of prime numbers was part of an error message generated by the compiler on attempting to compile the code**.

http://aszt.inf.elte.hu/~gsd/halado_cpp/ch06s04.html#Static-metaprogramming

Unruh example

```
// Prime number computation by Erwin Unruh
template <int i> struct D { D(void*); operator int(); };

template <int p, int i> struct is_prime {
    enum { prim = (p%i) && is_prime<(i > 2 ? p : 0), i -1> :: prim };
    };

template < int i > struct Prime_print {
    Prime_print<i-1> a;
    enum { prim = is_prime<i, i-1>::prim };
    void f() { D<i> d = prim; }
    };

struct is_prime<0,0> { enum {prim=1}; };
struct is_prime<0,1> { enum {prim=1}; };
struct Prime_print<2> { enum {prim = 1}; void f() { D<2> d = prim; } };
#ifndef LAST
#define LAST 10
#endif
main () {
    Prime_print<LAST> a;
    }
```

```
01 | Type `enum{}' can't be converted to type `D<2>' ("primes.cpp",L2/C25).
02 | Type `enum{}' can't be converted to type `D<3>' ("primes.cpp",L2/C25).
03 | Type `enum{}' can't be converted to type `D<5>' ("primes.cpp",L2/C25).
04 | Type `enum{}' can't be converted to type `D<7>' ("primes.cpp",L2/C25).
05 | Type `enum{}' can't be converted to type `D<11>' ("primes.cpp",L2/C25).
06 | Type `enum{}' can't be converted to type `D<13>' ("primes.cpp",L2/C25).
07 | Type `enum{}' can't be converted to type `D<17>' ("primes.cpp",L2/C25).
08 | Type `enum{}' can't be converted to type `D<19>' ("primes.cpp",L2/C25).
09 | Type `enum{}' can't be converted to type `D<23>' ("primes.cpp",L2/C25).
10 | Type `enum{}' can't be converted to type `D<29>' ("primes.cpp",L2/C25).
```

Function templates

```
template<typename T>
T maxVal(T v1, T v2){
    if(v1 > v2)
        return v1;
    return v2;
}

template<class T>
T minVal(T v1, T v2){
    if(v1 < v2)
        return v1;
    return v2;

}
```

- T is the template parameter, a type argument for the template;
- The template parameter can be introduced with any of the two keywords: typename, class.

Function templates

- The process of generating an actual function from a template function is called **instantiation**.

```
double minD = minVal<double>(4.5, 23);
int maxI = maxVal<int>(2, 3);
Coin c = maxVal<Coin>(Coin{10}, Coin{50}); // we must overload the comparison operators for the Coin class
```

Class templates

- A template can be seen as a skeleton or macro.
- When specific types are added to this skeleton (e.g. double), then the result is an actual C++ class.
- When instantiating a template, the compiler creates a new class with the given template argument.
- The compiler needs to have access to the implementation of the methods, to instantiate them with the template argument.
- Simple solution: **place the definition of a template in a header file.**

What's new in C++ (cont'd) - range based for

- Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container.
- syntax:
  for (range_declaration : range_expression)
    loop_statement