CISC 3142
Programming Paradigms in C++

Ch1 – Introduction to Functional Programming

(From: *Ivan Čukić – Functional Programming in C++, Manning Publications Co. ©2019*)

What is functional programming?

- *Functional programming* (FP) is an old programming paradigm born in academia during the 1950s
- Representative languages are Haskell and Lisp
- While these languages have never been as popular as C/Java/C++, the more popular languages have started introducing features inspired by FP
- No widely accepted definition for FP exists
- A try at definition (`comp.lang.functional`): FP is a style of programming that
  - Emphasizes the evaluation of expressions, rather than execution of commands
  - The expressions are formed by using functions to combine basic values
  - A functional language is one that supports and encourages programming in a functional style
- The author leaves the definition, and what's essential for FP to the readers

Basic idea

- To calculate the sum of a list of numbers
- *Imperative* approach
  - Iterating over the list and adding the numbers to the accumulator variable
  - You explicitly provides the step-by-step process of the summing
- Functional approach
  - Define only what a sum of a list of numbers is
    - A sum of a list of number equals the first element of the list added to the sum of the rest of the list (recursive definition)
    - The sum is zero if the list is empty
  - You effectively defined what the sum is without explaining how to calculate it
- Functional programming belongs to *declarative* programming, which means you state what should be done, not how to do it.

Relationship with object-oriented programming

- OOP is the most popular imperative paradigm whereas FP is the most commonly used declarative one
- OOP is based on creating abstractions for data where you hide the inner representation, and only expose its interface
- FP creates abstractions on the functions, this lets you create more-complex control structures than what the language natively provides
  - It's possible to implement the (`foreach`) loop without changing the compiler much
  - The range-based `for` loop (since C++11) is defined by enumerating via iterators
- Sometimes, one paradigm is more suitable than the other. Often a combination of two works best – the trend of multiparadigm languages

A concrete example of imperative vs declarative programming

- To write a function that takes a list of files and calculates the number of lines in each
- The imperative approach
  - Define a counter to store the # of lines
  - Open each file
  - Read the file one character at a time, and increase the counter every time the '\n' is read
  - At the end of a file, store the # of lines calculated
- Keys
  - Two nested loops and variables to keep the current state
  - Potential errors: uninitialized variable, an improperly updated state, or a wrong loop condition
  - Some will be reported by compiler, some won't

- Sample code

```
std::vector<int> count_lines_in_files(const
                    std::vector<std::string>& files) {
    std::vector<int> results;
    char c = 0;
    for (const auto& file : files) {
        int line_count = 0;
        std::ifstream in(file);
        while (in.get(c)) {
            if (c == '\n') {
                line_count++;
            }
        }
        results.push_back(line_count);
    }
    return results;
}
```

A concrete example (cont')

- The functional approach – taking advantage of std::count

```
int count_lines(const std::string& filename) {
    std::ifstream in(filename);
    return std::count( std::istreambuf_iterator<char>(in), std::istreambuf_iterator<char>(), '\n');
}

std::vector<int> count_lines_in_files(const std::vector<std::string>& files) {
    std::vector<int> results;
    for (const auto& file : files) {
        results.push_back(count_lines(file));
    }
    return results;
}
```

- With this solution, the implementation detail of how counting is done is hidden away. Instead you only declare you want to count newlines
- The functional style – use abstractions that define the intent instead of the how
- FP goes hand in hand with generic programming: both let you think on a higher level of abstraction
- The only task of the count_lines function is to convert filenames to the type std::count can understand (iterators)

A concrete example (cont')

- Take this even further – a common pattern for FP

```
std::vector<int> count_lines_in_files(const std::vector<std::string>& files) {
    std::vector<int> results(files.size());
    std::transform(files.cbegin(), files.cend(), results.begin(), count_lines); // no more loops
    return results;
}
```

- Where std::transform traverses the items in the files, transforms them using the count_lines function, and stores the values in results
- This code doesn't specify algorithm steps, but rather how input should be transformed in order to get the output
- This code can be further simplified (it uses the ranges library, a C++20 feature)

```
std::vector<int> count_lines_in_files(const std::vector<std::string>& files) {
    return files | transform(open_file) /* pipe operator to push a collection thru a transformation */
                 | transform(count_lines);
}
```

Pure functions

- One of the most significant sources of software bugs is program state
- The OOP paradigm groups parts of the state into objects – making it easier to manage, but it doesn't significantly reduce number of possible states
- Mutable states (even for local variables) can create dependencies between different parts of the function
- Ideally a pure function only uses (but doesn't modify) the arguments passed to it in order to calculate the result
  - If a pure function is called multiple times with the same arguments, it must return the same result every time without trace that it was ever invoked (no *side effects*)
  - Strictly immutable-state function might be of little use, as it can't do things like creating/deleting files
- Refined definition: a *pure function* is any function that doesn't have observable (at a higher level) side effects
  - We won't limit ourselves to pure functions only, but strive to cut down on non-pure ones

Avoiding mutable state

- The imperative count_lines_in_files() function should qualify as a pure function. As repeated calls of the same function should produce same result every time
- Let's take another look:

```
for (const auto& file : files) {
    int line_count = 0;
    std::ifstream in(file);
    while (in.get(c)) {
      if (c == '\n') {
        line_count++;
      }
    }
    results.push_back(line_count);
}
```

- The following states got changed
  - Calling .get() changes the input stream, and the variable c
  - line_count is changed via incrementing, and vector results is changed via appending
- But which ones are these *observable state changes*? None of them! (refer to slide #5 to see that they are all local). Therefore you can consider this function pure, though its implementation is not.
- The other implementations (slide #6 and #7) progressively get purer, with the last one completely eliminating local variables

Thinking functionally

- Writing code imperatively then changing it to functional is inefficient
- Instead of thinking about the algorithm steps, you should consider what the input and output are, and what transformations map one to the other
- The count_lines example can be considered as two mappings:

- Lifting these two functions so that they handle a collection of values is conceptually what std::transform does
- From this example, the functional approach is shown to split bigger programming problems into smaller, independent tasks – similar to a moving assembly line
- Each function is highly specialized to do one simple task well
  - It requires only valid input, doesn't care where it comes from

Benefits of functional programming

- Primary benefits (the main takeaway)
  - The code is much shorter (FP constructs are simple but highly expressive)
  - Purity improves the correctness of the code
- Code brevity and readability
  - While it's subjective, one can't argue that we effectively shrink the count_lines_in_files functionality from some 20 lines to about 6 lines combined
  - This is aided by the higher-level abstraction provided by the FP parts of STL
  - But many C++ programmers still stay away from STL algorithms
    - Maybe from thinking about being able to write more efficient code manually
    - Maybe from avoiding writing code that their colleagues can't easily understand
  - Not availing yourself of more-advanced features of the programming language reduces the power and expressiveness of the language
  - According to the author, doing away with loops/branching in favor of higher-level FP constructs is very similar to getting rid of GOTO statement in favor of *structured* programming

Benefits of FP (cont')

- Concurrency and synchronization
  - The main problem with developing concurrent systems is the shared mutable state
  - Parallelizing programs written with pure functions is trivial because they don't mutate anything, thus there is no need for explicit synchronization with atomics or mutexes
  - Code written for a single-threaded system can run on multiple threads with almost no changes
  - A sum algorithm that can be easily parallelized:

```
std::vector<double> xs = {1.0, 2.0, ...};
auto result = sum(xs | transform(sqrt));
```

Benefits of FP (cont')

- Continuous optimization
  - Using higher-level programming abstractions from STL or other trusted libraries means your program will improve over time
    - You gain the benefit without changing a single line
    - You also benefit from the improvement in the language itself
  - Writing low-level performance-critical code manually can have benefits but the optimization is only limited for targeted platforms
    - i.e. it could slow down in untargeted platforms

Evolution of C++ as a FP language

- Even with the initial nickname of “C with classes”, it’s difficult to call the language *object-oriented*
- With the introduction of templates and the creation of STL, which only sparsely uses inheritance and virtual member functions, C++ became a proper multiparadigm language
- Specifically with the design and implementation of STL, one could argue that C++ is not primarily an OO language, but a generic programming language
  - That is, it’s more about static or compile-time polymorphism, as opposed to the dynamic or runtime polymorphism
- For FP in C++, the importance of templates is mainly in the creation of STL algorithms, where behaviors of sorting/counting can be changed by a function argument
- The capability to pass functions as arguments to these algorithms effectively made C++ an FP language, and the evolution of C++ makes writing in functional style much easier, with C++20 adding more FP-inspired features

Summary

- The main philosophy of functional programming is
  - Don’t concern yourself with how something should work, rather with what it should do
- Both approaches – FP and OOP – have a lot to offer. You just learn when to use which, and when to combine them
- C++ is a multiparadigm programming language, combined with generic programming
- FP goes hand-in-hand with generic programming, both inspire programmers to think above the hardware level
- FP works like a pipeline – passing a value through a chain of transformations, and lifting lets your functions operate on collections of values
- Avoiding mutable states improves the correctness of code and removes the need for mutexes in multithreaded code