#7 Assignment - CMPT 383

Luiz Fernando Peres de Oliveira - 301288301 - lperesde@sfu.ca

I. Memory

Allocating memory in C++ (and most of languages) can be done statically or dynamically. In static allocated memory, all memory allocation calculations can be done before the program runs, while dynamic allocated memory can only be determined in runtime. Static memory is allocated by the system from the **stack** and dynamic memory is allocated by the system from the **heap**.

In C++, dynamic memory is usually allocated using the operator **new** or by using the C library *stdlib* directly. When you allocate dynamic memory in C++, the system returns a pointer to beginning of the sequence of bytes (block of memory). It is important to notice that memory is a finite resource and therefore, there are limitations on dynamic memory allocation. For this reason, a request to dynamic allocation of memory cannot be 100% guaranteed by the system.

Moreover, if an user tries to allocated more memory than it is physically available (the system can also limit a program's memory), it will be thrown an exception *bad_alloc*, informing that the system did not granted the requested memory. It is however possible to suppress such exception in C++ by using the keyword *nothrow*. Eg.: $mem = new(nothrow)int[3];$

II. Paradigms

The lowest level abstraction of a computer programs is **machine code**, which is the binary assembly representation of a program (a machine code or *object code* contains literally 0's and 1's). Usually, high-level compilers (such as clang or clang++) compile their source code to Assembly language. From there, the Assembly code is then assembled to machine code by a program called Assembler. The Assembler assembles Assembly into machine code by mapping opcodes (Assembly instructions) in an usually one-to-one conversion, depending on the level of abstractions that the Assembly code may have and the Assembler may support.

Assembly Language (and therefore machine code) falls under the imperative paradigm because, Assembly, as any imperative language uses statements or instructions that change a program's state. In order to program in Assembly, one must describe all the steps the machine must take in order to accomplish a given task. In fact, high-level imperative programming languages such as *Fortran* and *C* are abstractions of Assembly Language.

Speaking of high-level and low-level programming languages, one might understand their main characteristics by their main differences. That is to say, a high-level language is a programming language (such as *Java* and *Python*) that allows programmers to write code that is not hardware specific, while a low-level programming is a very specific language that is possibly the closest mean of communication between a human and the bare metal, for instance Assembly and machine code. Moreover, high-level programming languages are more human-readable than low-level languages. Some say however that it exists another category between high-level and low-level so called mid-level programming languages, a category that puts *C*, *C++* and most system programming languages. This category would take a few characteristics of both low-level and high-level programming languages.

III. Functional Programming

Lazy evaluation is a method that evaluates expressions (possibly *"infinite"* elements) in runtime. The elements in a lazy evaluation are not bound to variables by default. Instead, the program waits until these values are needed by other computations and only them generates and evaluates them "on the fly". Lazy evaluation is also called called-by-name and it is the opposite of eager evaluation or **call-by-value**, which is the most used strategy for evaluation of expressions in programming languages.

Another type of evaluation in languages such as Haskell is **strictly**, which is an annotation to force evaluation that might be needed when a given function calls another function, e.g $x : (h(gx)$. This kind of annotations become very handy when dealing with partial functions, for example.

Lastly, as some aspects of functional programming are becoming very popular, such as the **lambda expressions**, some multiparadigms languages like *C++*, *Python* and *Java* have become to give more and more support to functional programming. In C++, for instance, it is already possible to write *functional combinators* as well as *lambdas expressions*. Likewise, **Java 8** has included *higher order functions* as native values, making *chain call* possible for functions.

IV. Programming Questions

1. In **Haskell**, define the logical disjunction operator (i.e., boolean OR) four different ways using pattern matching.

```
data MyBool = True | False
OR1 :: MyBool -> MyBool -> MyBool
OR1 False False = False
OR1 _ _         = True

OR2 :: MyBool -> MyBool -> MyBool
OR2 False False = False
OR2 False True  = True
OR2 True  False = True
OR2 True  True  = True

OR3 :: MyBool -> MyBool -> MyBool
OR3 True _ = True
OR3 _ True = True
OR3 _ _    = False

OR4 :: MyBool -> MyBool -> MyBool
OR4 False False = False
OR4 _ True      = True
OR4 True _      = True
```

2. In an Object-oriented language (i.e., C++) create a function class. The idea is this class can accept functions as arguments as well as return functions. The class should be an abstract base class which must be inherited/extended/interfaced from in order to derive a concrete class.

```
class InputIt {
public:
  virtual void f() = 0;
};
class UnaryFunction {
public:
  virtual void f2() = 0;
};
template<class InputIt,
         class UnaryFunction>
UnaryFunction for_each(InputIt first,
                       InputIt last,
                       UnaryFunction f)
{
  for (; first != last; ++first) {
    f(*first);
  }
  return f;
}

int main() {
  std::vector<double> v{
    1.0, 2.2, 4.0, 5.5, 7.2
  };
  double r = 4.0;
  std::for_each(v.begin(),
                v.end(),
                [&](double & v) {
                  v += r;
  });
  std::for_each(v.begin(),
                v.end(),
                [](double v) {
                  std::cout << v << " ";
  });
```

3. In C create a function that returns a function. Create another function that takes functions as arguments.

```
#include <stdio.h>

typedef void (*Callback)(char *value);

void executeBlock(Callback block) {
  block("Awesome callback");
}

void returnThisOne(char *value) {
  printf("Value: %s", value);
}

Callback returnBlock() {
  return returnThisOne;
}

int main() {
  executeBlock(returnBlock());
  return 0;
}
```

References

[1] "Function Types", Bartosz Milewski's Programming Cafe, 2017. [Online]. Available: https://bartoszmilewski.com/2015/03/13/function-types/. [Accessed: 29- Nov- 2017].

[2] "Dynamic memory - C++ Tutorials", Cplusplus.com, 2017. [Online]. Available: http://www.cplusplus.com/doc/tutorial/dynamic/. [Accessed: 04- Dec- 2017].

[3] A. code?, "Assembly code vs Machine code vs Object code?", Stackoverflow.com, 2017. [Online]. Available: https://stackoverflow.com/questions/466790/assembly-code-vs-machine-code-vs-object-code. [Accessed: 04- Dec- 2017].

[4] "Programming paradigm", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Programming_paradigm. [Accessed: 04- Dec- 2017].

[5] "Imperative programming", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Imperative_programming. [Accessed: 04- Dec- 2017].

[6] "Computers for Beginners/Programming - Wikibooks, open books for an open world", En.wikibooks.org, 2017. [Online]. Available: https://en.wikibooks.org/wiki/Computers_for_Beginners/Programming. [Accessed: 04- Dec- 2017].

[7] "Lazy evaluation - HaskellWiki", Wiki.haskell.org, 2017. [Online]. Available: https://wiki.haskell.org/Lazy_evaluation. [Accessed: 05- Dec- 2017].

[8] "Functional Programming Lazy Evaluation", www.tutorialspoint.com, 2017. [Online]. Available: https://www.tutorialspoint.com/functional_programming/functional_programming_lazy_evaluation.htm. [Accessed: 05- Dec- 2017].

[9] ”Eager evaluation”, En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Eager_evaluation. [Accessed: 05- Dec- 2017].

[10] [16]”Performance/Strictness - HaskellWiki”, Wiki.haskell.org, 2017. [Online]. Available: https://wiki.haskell.org/Performance/Strictness#Evaluating _expressions_strictly. [Accessed: 05- Dec- 2017].

[11] ”Functional programming in C++”, Blog.madhukaraphatak.com, 2017. [Online]. Available: http://blog.madhukaraphatak.com/functional-programming-in-c++/. [Accessed: 05- Dec- 2017].

[12] ”Flying Bytes An Introduction to Functional Programming in Java 8: Part 1 - Functions as Objects”, Flyingbytes.github.io, 2017. [Online]. Available: https://flyingbytes.github.io/programming/java8/functional/part1/2017/01/23/Java8-Part1.html. [Accessed: 05- Dec- 2017].