```
                // push integer literal
                long v = std::strtol(token.c_str(), nullptr, 10);
                nextInstr = [v](long* sp){ *sp++ = v; return sp; };
                break;
            }
            case '+':
            {
                // + operation: pop 2 longs and push their sum
                nextInstr = [](long* sp){
                    long v1 = *--sp;
                    long v2 = *--sp;
                    *sp++ = v1 + v2;
                    return sp;
                };
                break;
            }
            // ... more cases
        }

        instructionStream.push_back(nextInstr);
    }
}
```

The `Instruction` type alias is an `std::function` that can hold, through a process called **type erasure**, any invocable object that takes a `long*` argument and returns a `long*` result. The `readInstructions` function reads successive string tokens and switches on the operation represented by the token. If the operation is `i`, then the token is an integer literal. The string token is converted into a `long` value, `v`, which is captured in a lambda expression. The resulting closure object is stored in the `nextInstr` variable; when called, it will push `v` onto the stack. Note that the `nextInstr` variable outlives the primary `v` variable, but, because `v` was captured by copy, the **captured variable**'s lifetime is the same as the closure object's. If the next operation is `+`, `nextInstr` is set to the closure object of an entirely different lambda expression, one that captures nothing and whose call operator pops two values from the stack and pushes their sum back onto the stack.

After the **`switch`** statement, the current value of `nextInstr` is appended to the instruction stream. Note that, although each closure type is different, they all can be stored in an `Instruction` object because the prototype for their call operator matches the prototype specified in the instantiation of `std::function`. The `nextInstr` variable can be created empty, assigned from the value of a lambda expression, and then later reassigned from the value of a different lambda expression. This flexibility makes `std::function` and lambda expressions a potent combination.

One specific use of `std::function` worth noting is to return a lambda expression from a nontemplate function: