The value of `largestShortPrime` must be set at initialization time because it is a **`const`** variable with static storage duration. The loop inside the lambda expression computes the desired value, using a local variable, `v`, and a **`for`** loop. Note that the call operator for the resulting closure object is immediately invoked via the `()` argument list at the end of the lambda expression; the closure object is never stored in a named variable and goes out of scope as soon as the full expression is completely evaluated. This computation would formerly have been possible only by creating a single-use *named* function.

Use with `std::function`

As convenient as lambda expressions are for passing functors to algorithm *templates*, each closure having an unnamed and distinct typemakes it difficult to use them outside of a generic context. The C++11 Standard Library class template, `std::function`, bridges this gap (at a cost of runtime overhead) by providing a polymorphic invocable type that can be constructed from any type with a compatible invocation prototype, including but not limited to closure types.

As an example, consider a simple interpreter for a postfix input language that stores a sequence of instructions in an `std::vector`. Each instruction can be of a different type, but they all accept the current stack pointer as an argument and return the new stack pointer as a result. Each instruction is typically a small operation, ideally suited for being expressed as a lambda expression:

```
#include <cstdlib>     // std::strtol
#include <functional>  // std::function
#include <string>      // std::string
#include <vector>      // std::vector

using Instruction = std::function<long*(long* sp)>;

std::vector<Instruction> instructionStream;

std::string nextToken();                // Read the next token.
char tokenOp(const std::string& token);  // operator for token

void readInstructions()
{
    std::string token;
    Instruction nextInstr;
    while (!(token = nextToken()).empty())
    {
        switch (tokenOp(token))
        {
            case 'i':
            {
```