Functional Programming

Student's Name

Department, Institutional Affiliation

Course Name and Number

Professor's Name

Due Date

Functional Programming

Functional Programming is a programming paradigm that treats computation as the evaluation of mathematical functions. FP emphasizes immutability, pure functions, and the avoidance of mutable state and side effects. It provides benefits such as improved code clarity, testability, and concurrency.

Immutability

In functional programming, immutability is preferred to avoid unexpected changes and side effects. To achieve immutability in C++, const variables and references can be utilized (Flovén, 2019). Example in Figure 1 below.

Figure 1

The screenshot below shows the implementation of immutability.

```cpp
#include <iostream>

int main() {
  const int x = 5;
  const int& y = x;

  // Error: x = 10; // Cannot modify a const variable
  // Error: y = 10; // Cannot modify a const reference

  std::cout << "x: " << x << std::endl;
  std::cout << "y: " << y << std::endl;

  return 0;
}
```

The variables "x" and "y" are declared as const, preventing their modification.

Pure Functions

Pure functions have no side effects and consistently provide the same outcome for the same input. They don't alter any external states; they just rely on their input parameters. It is possible to build pure functions in C++ by removing global variables and using const arguments (EventHelix, 2020). Example in Figure 3 below.

Figure 2

The screenshot below shows an implementation of pure functions.

```
#include <iostream>

int square(int x) {
  return x * x;
}

int main() {
  int num = 5;
  int result = square(num);

  std::cout << "Square of " << num << ": " << result << std::endl;

  return 0;
}
```

The "square()" function is a pure function that calculates the square of an integer without modifying any external state.

Higher-Order Functions

Higher-order functions accept parameters from other functions and return results from other functions. They make it possible to compose functions and create strong abstractions. Higher-order functions can be implemented more easily in C++ thanks to function pointers and lambda expressions (McNamara & Smaragdakis, 2000). Example in Figure 5 below.

Figure 3

The screenshot below shows an implementation of higher-Order Functions.

```cpp
#include <iostream>

void applyOperation(int x, int y, int (*operation)(int, int)) {
  int result = operation(x, y);
  std::cout << "Result: " << result << std::endl;
}

int main() {
  int num1 = 10;
  int num2 = 5;

  // Using a lambda expression
  applyOperation(num1, num2, [](int a, int b) { return a + b; });

  // Using a function pointer
  applyOperation(num1, num2, [](int a, int b) { return a * b; });

  return 0;
}
```

The "applyOperation()" function takes a function pointer as an argument and applies the operation to the given numbers.

References

EventHelix. (2020, August 13). *Pure functions in C++*. Medium. https://medium.com/software-design/pure-functions-in-c-fc102fd9c5e0

Flovén, F. (2019, April 29). *Immutability*. Medium. https://medium.com/@ffloven/immutability-4c8e0077fe9a

McNamara, B., & Smaragdakis, Y. (2000). Functional programming in C++. *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. https://doi.org/10.1145/351240.351251