C++ PROJECT: Implementing Exotic Option Valuation:

BarrierOption Up_and_Out and BarrierOption Down_and_Out using Monte Carlo Simulation in C++

Barrier Options

Barrier options are a Category of exotic option where the payoff depends not only on the underlying asset price at expiration but also on whether the asset price breaches a predefined barrier level during the life of the option.

Knock-Out Options

- A **knock-out option** ceases to exist (becomes worthless) if the underlying asset price crosses the barrier level during its lifetime.
  - **Up-and-Out Call**: A call option where the option becomes void if the underlying price rises **above** a certain barrier.
  - **Down-and-Out Call**: A call option where the option becomes void if the underlying price falls **below** a certain barrier.

Knock-In Options

- A **knock-in option** only comes into existence if the underlying asset price crosses the barrier level during its lifetime.
  - **Up-and-In Call**: A call option where the option is activated if the underlying price rises **above** a certain barrier.
  - **Down-and-In Call**: A call option where the option is activated if the underlying price falls **below** a certain barrier.

Monte Carlo Approach

Monte Carlo methods simulate the price paths of the underlying asset to estimate the value of these options.

Monte Carlo Up-and-Out Call Option

- The option starts as valid.
- If at any time during the simulation the underlying price **exceeds** the barrier:
  - The option is immediately **knocked out** (worth zero).
- If the barrier is never breached:
  - The payoff at expiration is: max(ST−K,0)

where ST is the terminal price of the underlying asset, and KKK is the strike price.

Monte Carlo Down-and-Out Call Option

- The option starts as valid.
- If at any time during the simulation the underlying price **falls below** the barrier:
    - The option is immediately **knocked out** (worth zero).
- If the barrier is never breached:
    - The payoff at expiration is: max(ST−K,0)

Monte Carlo Up-and-In Call Option

- The option starts as **inactive**.
- If at any time during the simulation the underlying price **exceeds** the barrier:
    - The option is **activated** and behaves like a standard call option.
- If the barrier is never breached:
    - The option remains worthless.

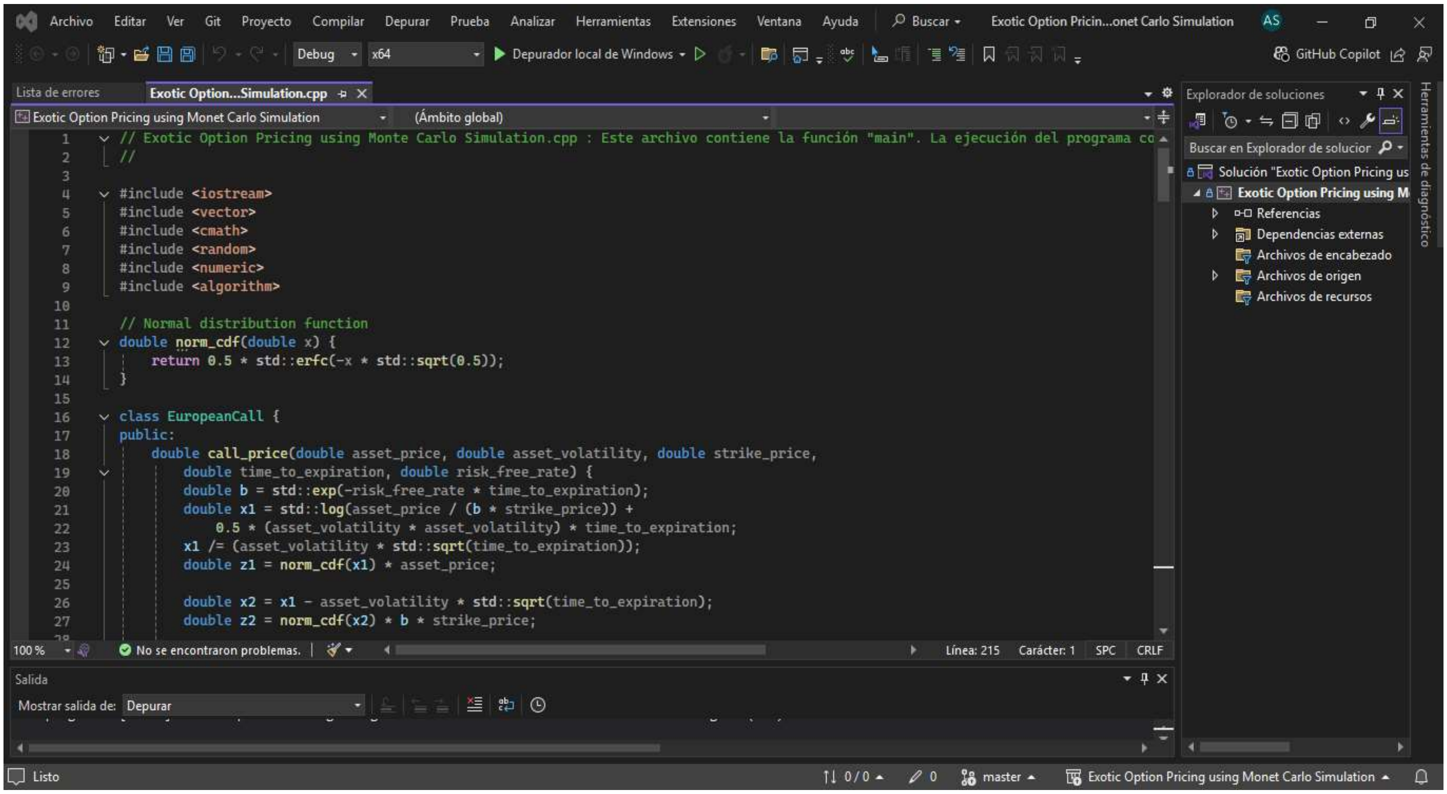
Monte Carlo Down-and-In Call Option

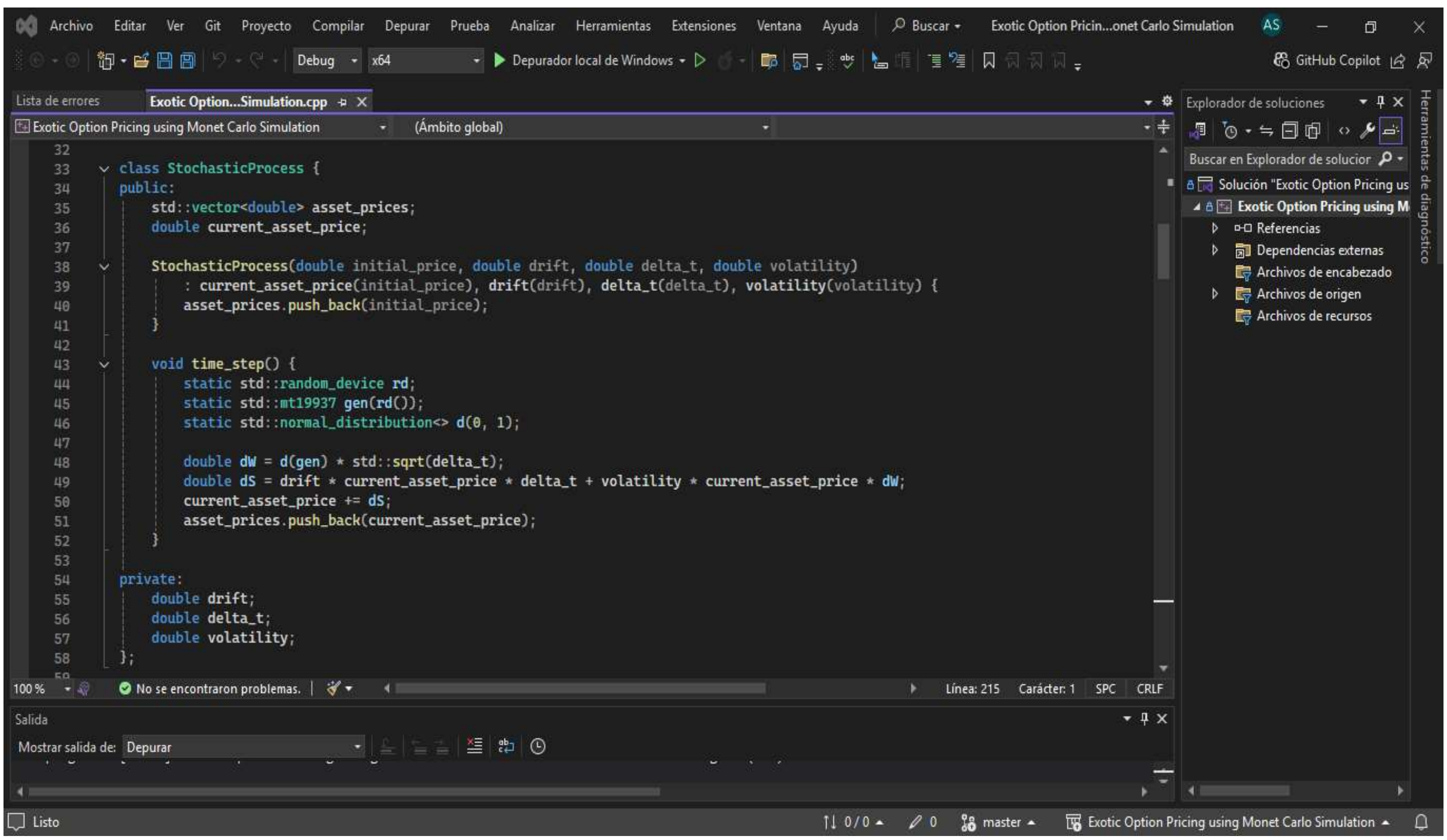
- The option starts as **inactive**.
- If at any time during the simulation the underlying price **falls below** the barrier:
    - The option is **activated** and behaves like a standard call option.
- If the barrier is never breached:
    - The option remains worthless.

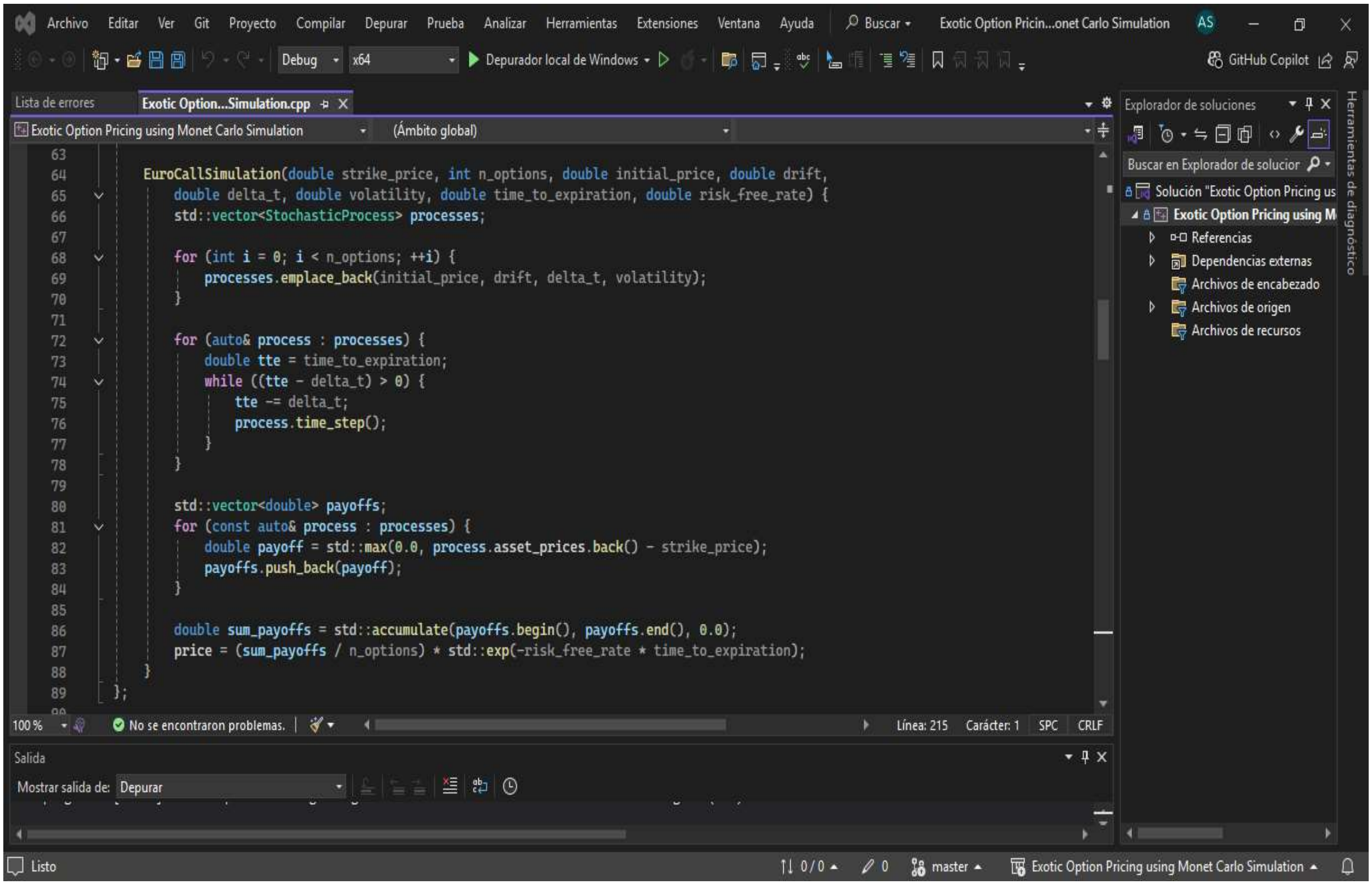
Key Factors in Monte Carlo Simulations

1. **Barrier Monitoring Frequency**:
    - Continuous Monitoring: The barrier is checked at every time step of the simulation (most realistic).
    - Discrete Monitoring: The barrier is checked at specific intervals (e.g., daily, weekly).
2. **Path Dependency**:
    - Barrier options are path-dependent, meaning the entire price path of the underlying asset matters, not just the terminal price.
3. **Numerical Convergence**:
    - A large number of simulated paths are required for convergence to an accurate estimate.
4. **Discounting**:
    - The expected payoff is discounted to the present value using the risk-free rate: Option Price=e^−rT*Expected Payoff

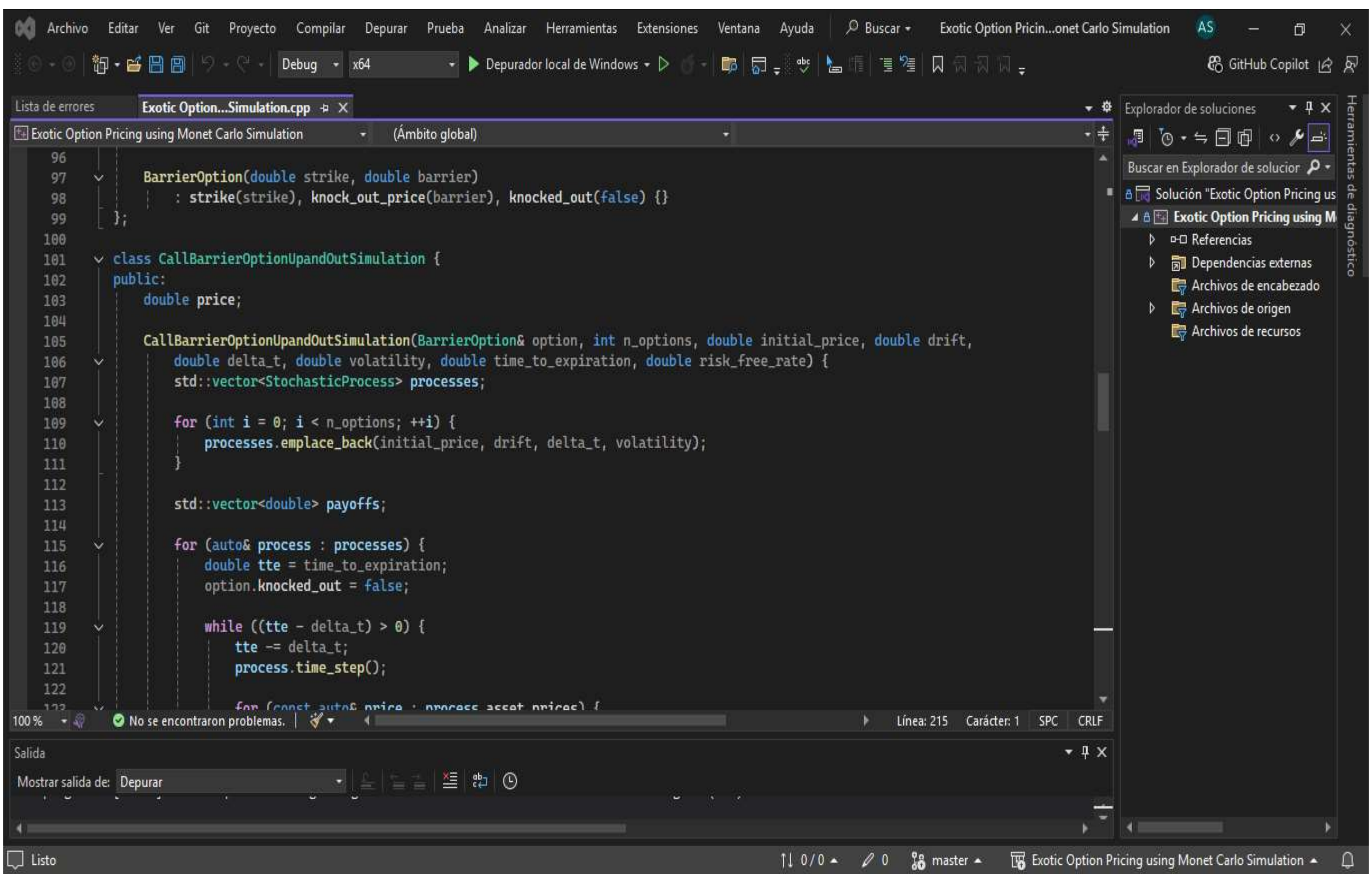
IMPLEMENTATION in C++ Using Visual Studio Code Editor



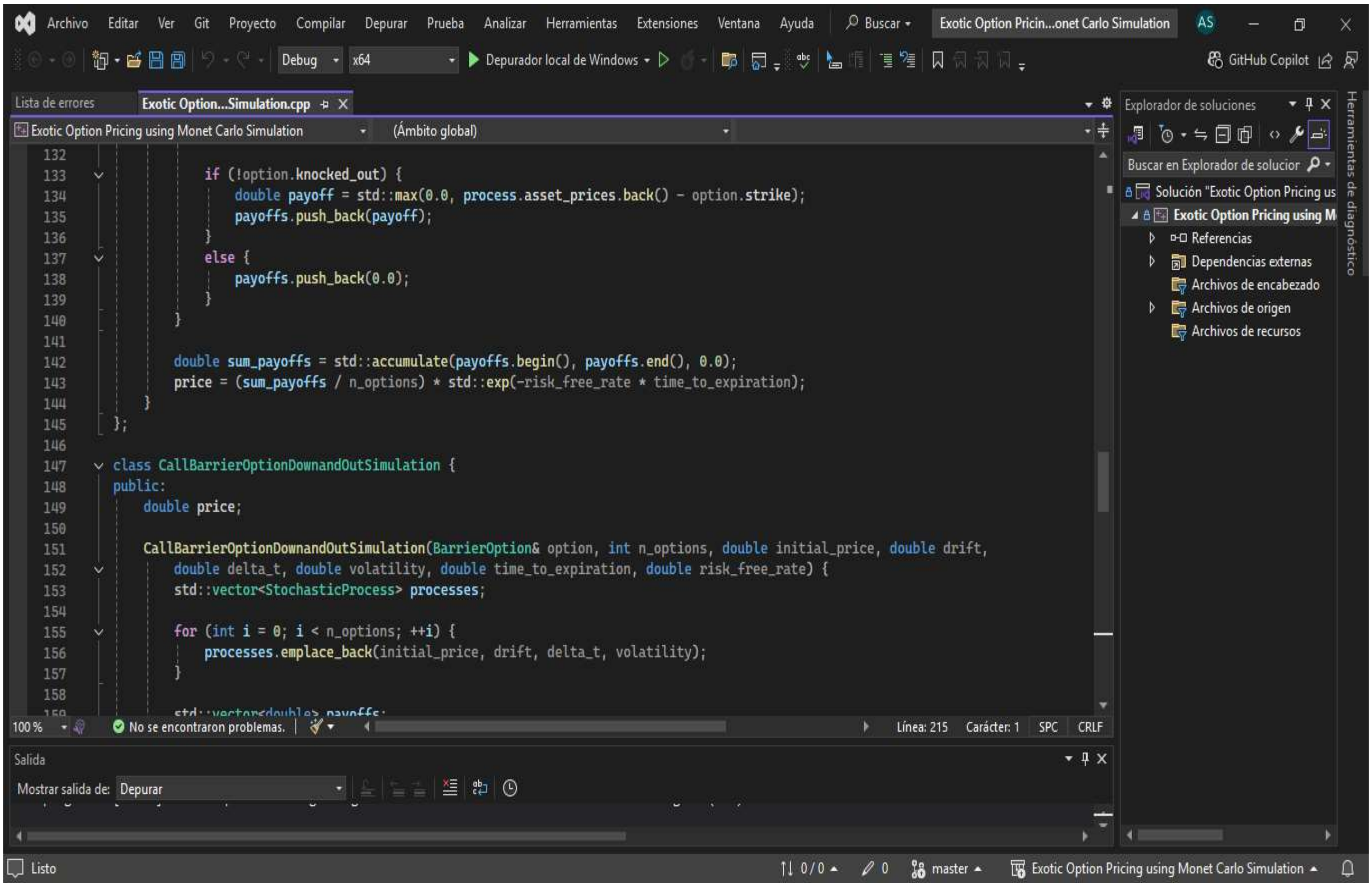




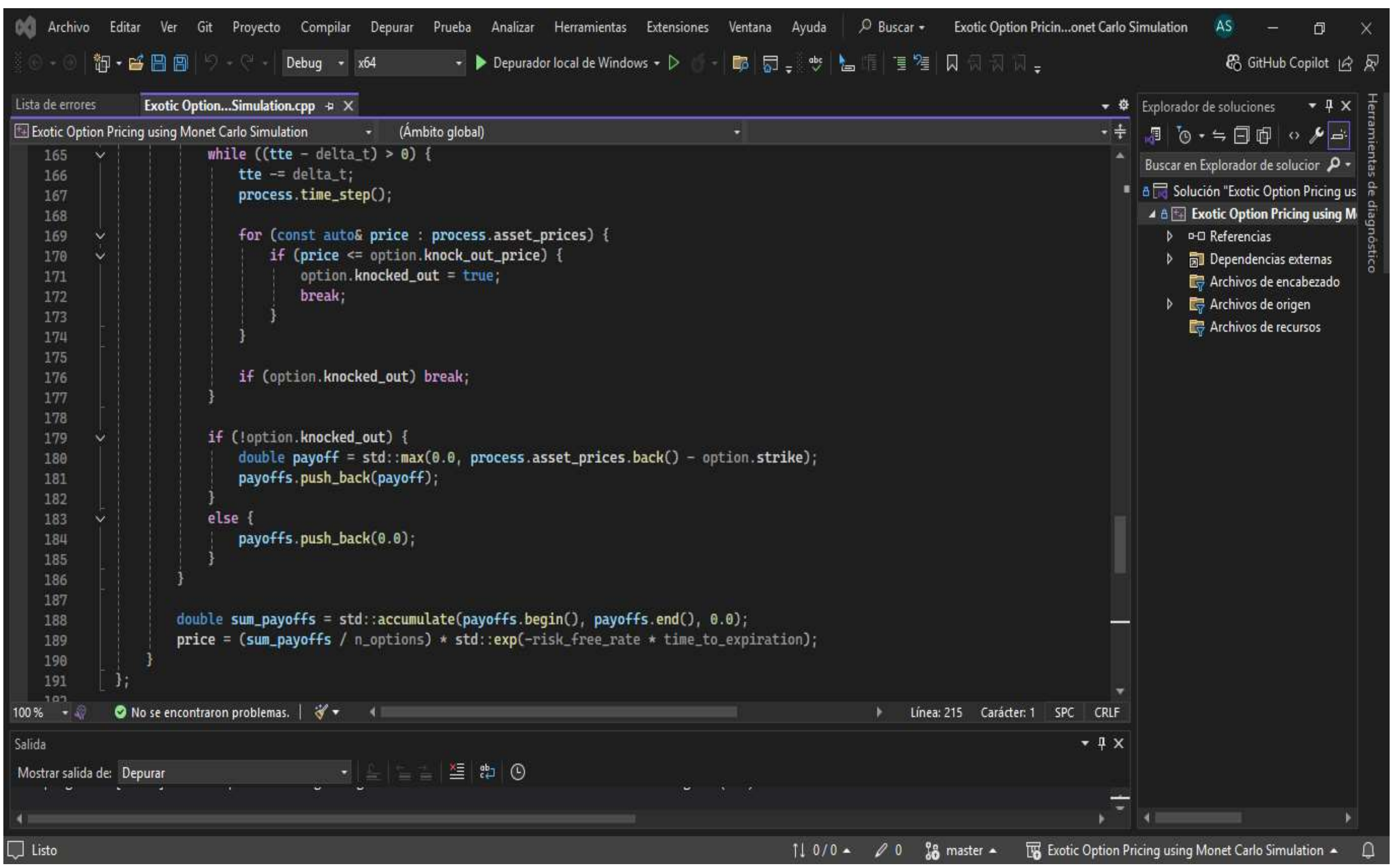
```cpp
    EuroCallSimulation(double strike_price, int n_options, double initial_price, double drift,
        double delta_t, double volatility, double time_to_expiration, double risk_free_rate) {
        std::vector<StochasticProcess> processes;

        for (int i = 0; i < n_options; ++i) {
            processes.emplace_back(initial_price, drift, delta_t, volatility);
        }

        for (auto& process : processes) {
            double tte = time_to_expiration;
            while ((tte - delta_t) > 0) {
                tte -= delta_t;
                process.time_step();
            }
        }

        std::vector<double> payoffs;
        for (const auto& process : processes) {
            double payoff = std::max(0.0, process.asset_prices.back() - strike_price);
            payoffs.push_back(payoff);
        }

        double sum_payoffs = std::accumulate(payoffs.begin(), payoffs.end(), 0.0);
        price = (sum_payoffs / n_options) * std::exp(-risk_free_rate * time_to_expiration);
    }
};
```



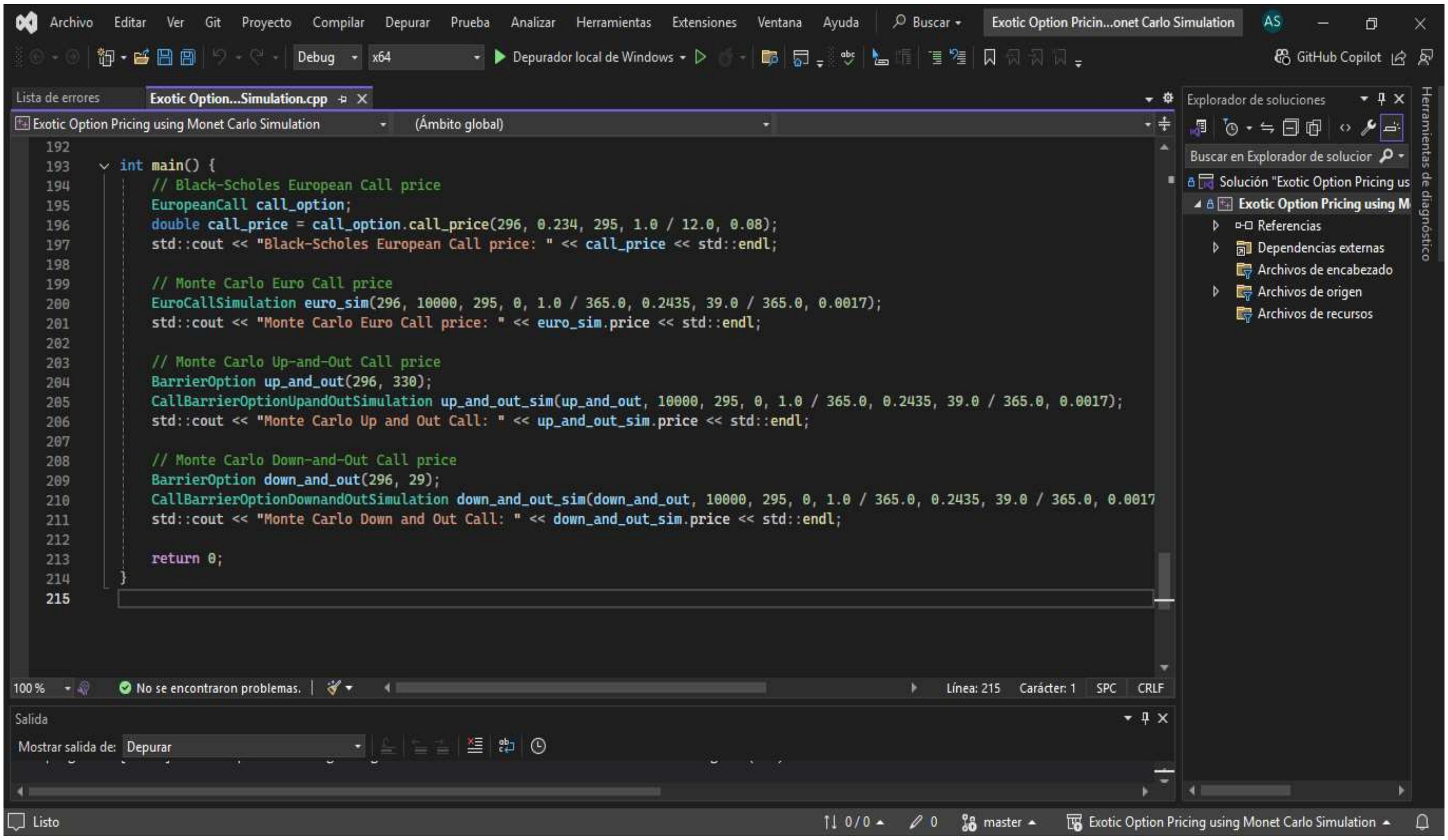
```cpp
    BarrierOption(double strike, double barrier)
        : strike(strike), knock_out_price(barrier), knocked_out(false) {}
};

class CallBarrierOptionUpandOutSimulation {
public:
    double price;

    CallBarrierOptionUpandOutSimulation(BarrierOption& option, int n_options, double initial_price, double drift,
        double delta_t, double volatility, double time_to_expiration, double risk_free_rate) {
        std::vector<StochasticProcess> processes;

        for (int i = 0; i < n_options; ++i) {
            processes.emplace_back(initial_price, drift, delta_t, volatility);
        }

        std::vector<double> payoffs;

        for (auto& process : processes) {
            double tte = time_to_expiration;
            option.knocked_out = false;

            while ((tte - delta_t) > 0) {
                tte -= delta_t;
                process.time_step();

                for (const auto& price : process.asset_prices) {
```



```cpp
            if (!option.knocked_out) {
                double payoff = std::max(0.0, process.asset_prices.back() - option.strike);
                payoffs.push_back(payoff);
            }
            else {
                payoffs.push_back(0.0);
            }
        }

        double sum_payoffs = std::accumulate(payoffs.begin(), payoffs.end(), 0.0);
        price = (sum_payoffs / n_options) * std::exp(-risk_free_rate * time_to_expiration);
    }
};

class CallBarrierOptionDownandOutSimulation {
public:
    double price;

    CallBarrierOptionDownandOutSimulation(BarrierOption& option, int n_options, double initial_price, double drift,
        double delta_t, double volatility, double time_to_expiration, double risk_free_rate) {
        std::vector<StochasticProcess> processes;

        for (int i = 0; i < n_options; ++i) {
            processes.emplace_back(initial_price, drift, delta_t, volatility);
        }

        std::vector<double> payoffs;
```



```cpp
            while ((tte - delta_t) > 0) {
                tte -= delta_t;
                process.time_step();

                for (const auto& price : process.asset_prices) {
                    if (price <= option.knock_out_price) {
                        option.knocked_out = true;
                        break;
                    }
                }

                if (option.knocked_out) break;
            }

            if (!option.knocked_out) {
                double payoff = std::max(0.0, process.asset_prices.back() - option.strike);
                payoffs.push_back(payoff);
            }
            else {
                payoffs.push_back(0.0);
            }
        }

        double sum_payoffs = std::accumulate(payoffs.begin(), payoffs.end(), 0.0);
        price = (sum_payoffs / n_options) * std::exp(-risk_free_rate * time_to_expiration);
    }
};
```



RESULTS Below