

## **ChartDirector Finance Projects C#**

### **Ответ**

**Для реализации проекта ChartDirector на платформе .NET (C#) вам потребуется библиотека ChartDirector, которая позволяет создавать разнообразные графики и диаграммы высокого качества прямо внутри вашего приложения. Вот пошаговая инструкция по созданию простого графического представления финансовых данных с использованием ChartDirector и C#.**

### **Шаг 1: Установка библиотеки**

- 1. Скачайте библиотеку ChartDirector для Windows (.NET). Это платная библиотека, однако доступна бесплатная пробная версия.**
- 2. После установки добавьте ссылку на библиотеку (NChartDir.dll) в ваш проект Visual Studio. Откройте Solution Explorer → щелкните правой кнопкой мыши на проекте → выберите Add Reference → перейдите на вкладку Browse и найдите файл NChartDir.dll.**

### **Шаг 2: Создание графика**

**Создаем простое приложение WPF или WinForms, где будем отображать график изменения акций или валюты за определенный период.**

**Пример кода для визуализации линейного графика цен акций:**

```
using System.Windows.Forms;  
using ChartDirector;  
  
public class FinancialGraph : Form  
{  
    private PictureBox pictureBox = new PictureBox();  
  
    public FinancialGraph()  
    {  
        // Настройка формы  
        this.Text = "Financial Graph";  
        this.Width = 800;  
        this.Height = 600;  
  
        // Создаем PictureBox для отображения графика  
        pictureBox.Dock = DockStyle.Fill;  
        Controls.Add(pictureBox);  
  
        DrawChart();  
    }  
}
```

```

}

private void DrawChart()
{
    // Данные акции (пример)
    double[] prices = { 100, 102, 105, 103, 107 };
    string[] labels = { "Jan", "Feb", "Mar", "Apr", "May" };

    XYChart chart = new XYChart(600, 400); // Размер графика

    // Добавляем названия осей
    chart.addTitle("Акции Компании XYZ");
    chart.xAxis().setTitle("Месяцы");
    chart.yAxis().setTitle("Цена ($)");

    // Устанавливаем значения на оси X
    chart.xAxis().setLabels(labels);

    // Рисуем линию графика
    LineLayer lineLayer = chart.addLineLayer(prices, 0xff0000, "Цены
    акций"); // Цвет линии красный

    // Генерируем картинку
    Bitmap bitmap = chart.makeBitmap();

    // Отображаем графику в PictureBox
    pictureBox.Image = bitmap;
}
}

// Запуск программы
static class Program
{
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new FinancialGraph());
    }
}

```

**Что сделано в коде?**

- 1. Мы создали форму с элементом PictureBox, куда будем выводить готовый график.**

2. Использовали класс XYChart из библиотеки ChartDirector для рисования двухосевого графика.
3. Через метод addLineLayer() добавили слой линий для наших значений цен акций.
4. Задали метки на оси X и заголовки для осей.
5. Наконец, преобразовали объект chart в растровый рисунок и показали его в форме.

## Возможности расширения

Библиотека ChartDirector поддерживает огромное количество типов графиков и диаграмм: гистограммы, круговые диаграммы, поверхности, пузырьковые графики и многое другое. Вы можете легко расширить этот пример, добавив интерактивность (например, вывод всплывающих подсказок при наведении мышью).

Таким образом, ChartDirector — мощный инструмент для построения профессиональных графиков на платформе .NET/C#, позволяющий эффективно представить финансовые данные вашему приложению.

Ты все правильно понял. Давай посмотрим Пример

```
private void
createChart(RazorChartViewer viewer)    {           // Create a finance chart
demo containing 100 days of data        int noOfDays = 100;

    // To compute moving averages starting from the first day, we need to get
extra data
    // points before the first day
    int extraDays = 30;

    // In this exammple, we use a random number generator utility to simulate
the data. We
    // set up the random table to create 6 cols x (noOfDays + extraDays) rows,
using 9 as
    // the seed.
    RanTable rantable = new RanTable(9, 6, noOfDays + extraDays);

    // Set the 1st col to be the timeStamp, starting from Sep 4, 2002, with
each row
    // representing one day, and counting week days only (jump over Sat and
Sun)
    rantable.setDateCol(0, new DateTime(2002, 9, 4), 86400, true);

    // Set the 2nd, 3rd, 4th and 5th columns to be high, low, open and close
data. The open
    // value starts from 100, and the daily change is random from -5 to 5.
    rantable.setHLOCCols(1, 100, -5, 5);
```

```
// Set the 6th column as the vol data from 5 to 25 million
ratable.setCol(5, 50000000, 250000000);

// Now we read the data from the table into arrays
double[] timeStamps = ratable.getCol(0);
double[] highData = ratable.getCol(1);
double[] lowData = ratable.getCol(2);
double[] openData = ratable.getCol(3);
double[] closeData = ratable.getCol(4);
double[] volData = ratable.getCol(5);

// Create a FinanceChart object of width 640 pixels
FinanceChart c = new FinanceChart(640);

// Add a title to the chart
c.addTitle("Finance Chart Demonstration");

// Set the data into the finance chart object
c.setData(timeStamps, highData, lowData, openData, closeData, volData,
extraDays);

// Add the main chart with 240 pixels in height
c.addMainChart(240);

// Add a 5 period simple moving average to the main chart, using brown
color
c.addSimpleMovingAvg(5, 0x663300);

// Add a 20 period simple moving average to the main chart, using purple
color
c.addSimpleMovingAvg(20, 0x9900ff);

// Add HLOC symbols to the main chart, using green/red for up/down days
c.addHLOC(0x008000, 0xcc0000);

// Add 20 days bollinger band to the main chart, using light blue (9999ff) as
the border
// and semi-transparent blue (c06666ff) as the fill color
c.addBollingerBand(20, 2, 0x9999ff, unchecked((int)0xc06666ff));

// Add a 75 pixels volume bars sub-chart to the bottom of the main chart,
using
// green/red/grey for up/down/flat days
c.addVolBars(75, 0x99ff99, 0xff9999, 0x808080);
```

```

// Append a 14-days RSI indicator chart (75 pixels high) after the main
chart. The main
// RSI line is purple (800080). Set threshold region to +/- 20 (that is, RSI =
50 +/-
// 25). The upper/lower threshold regions will be filled with red
(ff0000)/blue
// (0000ff).
c.addRSI(75, 14, 0x800080, 20, 0xff0000, 0x0000ff);

// Append a 12-days momentum indicator chart (75 pixels high) using blue
(0000ff) color.
c.addMomentum(75, 12, 0x0000ff);

// Output the chart
viewer.Image = c.makeWebImage(Chart.SVG);
}

```

---

Что мы здесь видим. Для демонстрации нормально. Но для реального времени надо это исправлять. И так реальное время. Сильно упрощенно. Здесь сразу выполняется Генерация 100 + 20 свечей и потом выводится весь массив загружается в график. В реальном времени все по-другому. В единицу времени к нам прилетает одна свеча. Далее эта свеча добавляется к текущему массиву свечей и этот массив выводится в Chart. То есть генератор должен генерить одну свечу, добавлять в массив первой, убирать последнюю свечу и выводить весь массив свечей в График. Таким образом у нас кадр предыдущий от кадра текущего отличается одной новой свечой и убранной последней свечой. Сюда настоятельно просится CircularBuffer. Начальный момент. Вот в Начальный момент, например, в начале торгов должны загружаться свечи, кол-во корых должно закрывать весь график, а потом переходим к текущему моменту, описанному выше. То есть Здесь надо запрограммировать Начальный момент, где работает Random генератор, и уже Текущую торговлю с добавлением первой свечи (самую левую) и убиранием последней свечи. Получается, что генератор свеч нужен в Начальный момент для генерации Массива свеч на весь графики далее в текущей торговле Генератор нужен для генерации одной свечи. Как думаешь Генераторы, которые представлены в этом Примере подойдут для выполнения двух кейсов: 1. Генерация массива свечей, Генерация одной свечи. В Текущей торговле мы все же можем использовать CircularBuffer обязательно с очередью и с массивом уже готовых свечей, которые в каждый интервал будут переписываться в буфера для рендинга (здесь надо применить механизмы синхронизации).

**Как тебе такая схема ? Выскажи свое мнение на словах. Пока без кода.**

**Ответ**

**Предложенная тобой схема выглядит вполне разумной и соответствует общим принципам эффективной обработки потоковых данных в режиме реального времени. Давай разберём её подробнее:**

**1. Начальная загрузка (Генерация массива свечей):**

- **Сначала генерируется начальный массив свечей, достаточный для заполнения всего графика. Для этого отлично подойдёт случайный генератор (Random Generator), представленный в примере. Его задача проста: создать заранее заданное число свечей для начальной загрузки графика.**
- **Если речь идёт о демо-версии или симуляции исторических данных, такой подход абсолютно оправдан.**

**2. Обработка новых данных (Режим реальной торговли):**

- **По мере поступления новых данных (каждая новая свеча) старые данные постепенно отбрасываются, обеспечивая постоянное обновление экрана. Ты верно заметил, что тут идеально подходит структура типа кольцевого буфера (Circular Buffer), так как она позволяет быстро обновлять данные, поддерживая постоянный размер массива.**
- **Каждый новый тик добавляет новую свечу слева (первое значение в массиве), а последняя свеча справа удаляется автоматически благодаря механизму циклических сдвигов.**
- **Такая реализация обеспечивает плавное движение графика вправо и минимизирует затраты памяти, поскольку длина буфера фиксирована.**

**Синхронизация:**

- **Важно учитывать возможные проблемы параллелизма, особенно если обновления происходят одновременно с процессом рендера. Нужно предусмотреть меры синхронизации (локи, семафоры и т.п.) для предотвращения некорректного чтения или записи данных в моменты переключения кадров.**

**Механизм рендеринга:**

- **Каждое новое изменение данных вызывает перерисовку графика, но важно минимизировать частоту таких изменений, чтобы избежать излишней нагрузки на систему. Например, обновление можно**

**производить только тогда, когда новые данные существенно отличаются от предыдущих.**

#### **Заключение:**

**Идея использовать разные генераторы для начальных и текущих данных верна и эффективна. Так как требования к генераторам различны: первый предназначен для быстрой генерации большого объёма данных, второй – для точной обработки отдельных поступающих элементов.**

**Общий подход выглядит следующим образом:**

- Начальные данные: Random Generator создаёт полный набор данных для первоначального наполнения графика.**
- Реальное время: Кольцевой буфер хранит ограниченное количество последних свечей, постоянно дополняясь новыми данными и избавляясь от устаревших.**

**Эта архитектура позволит достичь плавного и эффективного отображения динамики данных в реальном времени, сохраняя хорошую производительность даже при больших объёмах данных.**

**В одну секунду необходимо: Генерить Новую свечу, Убрать старую. Закинуть Новую свечу в CircularBuffer (CB), Самая старая свеча сама исчезнет из CB. Выгрузить из CB массив свечей и загрузить этот массив в массив для Рендинга. Если возможно брать свечи для рендинга прямо из CB, то надо это сделать. Также надо подумать как мы будем хранить свечи в CB в 6-ти массивах (DT, H,L,O, C, Volume) или хранить свечи будем в виде class CandleStick полями, а для рендинга будем разбрасывать CandleStick на шесть массивов - так как это нужно для рендинга. Так как задача вполне подъемная, может быть надо будет использовать BackgroundService. В BackgroundService мы вычисляем каждый кадр, заполняем Буфера для рендинга. А Рендинг будет запускаться через каждую секунду. У него задача простая - создать картинку - график из шести массивов и нарисовать его в Browser. Вот такая задача.**

- 1. Выскажи свое мнение.**
- 2. Покорректируй мой текст для лучшего его понимания. Можно уже даже в новом тексте имена переменным давать. Вообще подробный проект необходимо создать в тексте**

#### **Ответ**

**Задача действительно интересная и требует тщательного подхода к организации потоков данных и синхронизации процессов. Попробую сформулировать проект более чётко и понятно, основываясь на твоих комментариях.**

---

## **Проект: Реалтайм график с обработкой свечей в кольцевом буфере**

### **Задача:**

Необходимо построить систему, которая в реальном времени принимает котировки фондового рынка, формирует на их основе "свечи" и выводит график на экран браузера. Система должна поддерживать обработку потока данных таким образом, чтобы каждая секунда соответствовала новому значению (новой свече), которое заменяет самую старшую запись (старую свечу) в графике.

### **Подход:**

**Система состоит из двух основных компонентов:**

1. **Background Service**, который занимается расчётом новых данных и обновляет внутренний кольцевой буфер.
2. **Rendering Component**, который каждые секунды берёт актуальные данные из кольцевого буфера и производит рендеринг нового состояния графика.

---

## **Компоненты системы**

### **1. Background Service (обновление данных)**

Этот компонент обрабатывает входящие данные (котировки, объемы и т.д.). Каждые  $n$  секунд (например, каждую секунду) он вычисляет новую свечу и вставляет её в начало кольцевого буфера (Circular Buffer). Одновременно самая старая свеча выбрасывается из буфера, так как буфер имеет фиксированный размер.

### **Структура данных в буфере:**

Каждый элемент буфера представляет собой структуру типа **Candlestick**. Она включает поля:

- **Timestamp (dt)**,
- **High price (h)**,
- **Low price (l)**,
- **Open price (o)**,
- **Close price (c)**,
- **Volume (v)**.



Структуру можно реализовать так: `class Candlestick{ public DateTime dt; public double h; public double l; public double o; public double c; public long v;}`

Буфер представляет собой коллекцию объектов типа `Candlestick`, организованных в кольцеобразной структуре. Длина буфера выбирается исходя из количества свечей, необходимых для полного покрытия окна графика.

Процесс добавления данных:

Каждую секунду сервис вычисляет новую свечу и помещает её в начало буфера, смещая остальные элементы вперёд. Последняя свеча вытесняется и теряется.

Пример метода для добавления новой свечи: `void UpdateCandle(CircularBuffer<Candlestick> buffer, Candlestick newCandle){ if (!buffer.IsFull()) { buffer.Enqueue(newCandle); // Добавляем в конец } else { buffer.Dequeue(); // Удаляем первую (самую старую) buffer.Enqueue(newCandle); // Добавляем новую }}`

## 2. Rendering Component (визуализация графика)

Этот компонент отвечает за создание графика и его регулярное обновление. Каждые `N` миллисекунд (обычно `~1000` мс) он извлекает содержимое кольцевого буфера и отправляет его на рендеринг.

График строится с использованием данных непосредственно из кольцевого буфера. Важно обеспечить согласованность операций чтения и записи данных, используя примитивы синхронизации (например, `lock` или семафоры).

Архитектура взаимодействия компонентов:

1. Данные поступают в `background service`. Каждая секунда формируется новая свеча.
2. Новая свеча записывается в кольцевой буфер (одновременно удаляя самую старую свечу).
3. `Renderer` периодически запрашивает состояние буфера и строит новый кадр графика.
4. Готовая картинка отправляется в браузер для отображения.

---

Детали реализации:

### 1. Circular Buffer:

Реализуется стандартным кольцевым буфером с поддержкой фиксированного размера. Ключевая операция – последовательное смещение данных.

Пример структуры:

```
class CircularBuffer<T>{    private readonly T[] _data;
private int _head, _tail, _count;    public bool IsFull => _count >= _data.Length;
public CircularBuffer(int capacity)    {        _data = new T[capacity];    }    public
void Enqueue(T item)    {        lock (_data)        {            if (_count ==
_data.Length)                Dequeue(); // Автоматически удаляем старый элемент
перед добавлением нового            _data[_tail++] = item;            _tail %=
_data.Length;            ++_count;        }    }    public T Dequeue()    {        lock
(_data)        {            var result = _data[_head++];            _head %= _data.Length;
--_count;            return result;        }    }}
```

## 2. Хранение данных:

Свечи хранятся в виде объектов класса **Candlestick**. В каждом таком объекте содержится вся необходимая информация для отображения свечи на графике.

При рендеринге графа отдельные массивы для каждого параметра (high, low, open, close, volume) формируются динамически на основе содержимого буфера.

Например, для рендеринга достаточно вызвать такую функцию:

```
(double[], double[], double[], double[], double[])
GetArraysForRender(CircularBuffer<Candlestick> buffer){    List<double> highs
= new(), lows = new(), opens = new(), closes = new(), volumes = new();
foreach(var candle in buffer.ToArray())    {        highs.Add(candle.h);
lows.Add(candle.l);        opens.Add(candle.o);        closes.Add(candle.c);
volumes.Add(candle.v);    }    return (highs.ToArray(), lows.ToArray(),
opens.ToArray(), closes.ToArray(), volumes.ToArray());}
```

## 3. Рендеринг графика:

Используя библиотеку вроде **ChartDirector**, можно организовать рендеринг на основе полученных массивов данных. Библиотека позволяет строить широкий спектр графиков и диаграмм, включая свечные графики (OHLC).

Пример рендеринга OHLC-графика:

```
private void
RenderChart(RazorChartViewer viewer, CircularBuffer<Candlestick> buffer){    //
Получаем массивы для рендеринга    var (highs, lows, opens, closes, _) =
GetArraysForRender(buffer);    // Время берётся из массива timestamp'ов
double[] timestamps = buffer.Select(x => x.dt.Ticks /
TimeSpan.TicksPerSecond).ToArray();    // Строим график    XYChart chart =
new XYChart(640, 400);    chart.addTitle("Real-time Stock Price");
```

```
chart.xAxis().setTitle("Time"); chart.yAxis().setTitle("Price"); // Дочерний
график OHLC chart.addCandleStick(highs, lows, opens, closes, 0x008000,
0xcc0000); // Выводим SVG-изображение viewer.Image =
chart.makeWebImage(Chart.SVG);}
```

**Итоговая схема:**

- 1. Пользователь открывает страницу.**
  - 2. Браузер запрашивает у сервера первое заполнение графика (начальные данные).**
  - 3. Сервер возвращает исходные данные (инициализация графика).**
  - 4. Background Service регулярно получает свежие данные и обновляет кольцевой буфер.**
  - 5. Renderer каждые N секунд читает данные из буфера и отправляет готовую картинку обратно браузеру.**
- 

**Оптимизации и улучшения:**

- Использование оптимизированных структур данных для быстрого доступа к данным (например, поддержка ленивых копирований массивов).**
  - Организация кеширования промежуточных результатов для ускорения последующих рендеров.**
  - Повышение производительности рендеринга путём оптимизации библиотеки ChartDirector (например, настройка кэш-полей, выбор наиболее производительных режимов рендеринга).**
- 

**Проект хорошо масштабируется и обладает гибкостью для дальнейшего развития, будь то интеграция дополнительных индикаторов или улучшение алгоритмов анализа данных.**

**Еще одна важная деталь по BackgroundService** Кроме описанных выше функций он еще будет делать следующее:

**Этот Сервис будет работать в нескольких режимах.1А.**

- Генерация quotes(Ticker, DT, Bid, ASK, Last, VolumeInLast - Random**
- Построение Candle по сгенеренным quotes1В.Генерация Candle по RandomGenerator**

**2А.**

- Получение quotes(Ticker, DT, Bid, ASK, Last, VolumeInLast из Внешних источников (MAssTRanzit)
- Построение Candle по сгенеренным quotes2В.Получение: Candle из Внешних источников (MAssTRanzit)Вот такой у нас должен быть функционал.

Оцените мысль.Подрихтуйте этот текст и добавьте его к Описанию BackGroundService

Ответ

Извините, произошла ошибка

Еще одна важная деталь по BackgroundServiceКроме описанных выше функций он еще будет делать следующее:

Этот Сервис будет работать в нескольких режимах.1А.

Генерация quotes(Ticker, DT, Bid, ASK, Last, VolumeInLast - Random  
Построение Candle по сгенеренным quotes1В.Генерация Candle по RandomGenerator

2А.

Получение quotes(Ticker, DT, Bid, ASK, Last, VolumeInLast из Внешних источников (MAssTRanzit)

Построение Candle по сгенеренным quotes2В.Получение: Candle из Внешних источников (MAssTRanzit)Вот такой у нас должен быть функционал.

Оцените мысль.Подрихтуйте этот текст и добавьте его к Описанию BackGroundService