Hanbit eBook
Realtime 70

MFC 프로그래밍

주식 분석 프로그램 만들기

김세훈 지음

한빛미디어 Hanbit Media, Inc.

MFC 프로그래밍

주식 분석 프로그램 만들기

MFC 프로그래밍 주식 분석 프로그램 만들기

초판발행 2014년 06월 24일

지은이 김세훈 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-664-7 15000 / **정가** 12,000원

책임편집 배용석 / **기획** 정지연 / **편집** 정지연·이세진
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 최송실
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 서은옥, 김옥현

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_김세훈

"나이에 비해서 경력이 없네요." 2006년 36살의 나이에 한 회사의 CTO에게 들은 말이다. 실력이 없으면 내 인생을 내 마음대로 결정할 수 없다 생각했는데, 우리나라는 나이가 아주 중요한 요소임을 간과했었다. 사회는 변한다. 고령화 사회를 걱정하며, 먼 훗날 젊은이들이 부양해야 할 노인들이 많다고 언론에선 말한다. 그러나 고령화 사회는 노인이 돼서도 일해야 하는 사회다. 저자는 노인이 돼서도 즐겁게 일하기 위하여 실력이 나이를 극복할 수 있다고 믿으며 현재도 자신을 만들어 가는 중이다. 또한, 인생의 나머지 반을 어떻게 살 것인지 항상 고민한다.

저자 서문

우리나라에서는 외국인들이 프로그램 매매로 주식을 한다고 알고 있다. 프로그램 매매라는 단어에서 전문가적인 느낌을 받지만, 간단히 말하면 주식 매매하는 사람들이 컴퓨터 프로그램도 만들 수 있다는 것이다. 물론, 주식 분석가와 프로그래머 등 여러 사람이 모여 그룹을 형성하여 함께 만들어나간 것이라고 보면 된다. 일반적으로 주식 매매를 하는 사람들은 컴퓨터 프로그램에 대하여 잘 알지 못하고, 프로그래머는 주식을 깊이 있게 알지 못한다. 하지만 앞으로 주식 매매를 하거나 주식을 깊이 있게 이해하기 위해서는 스스로 프로그램을 만들 필요가 있다.

요즘은 증권회사에서 주식 데이터를 가져올 수 있다. 예전에는 증권회사에서 제공하는 데이터와 주식 정보만을 봐야 했다면 이제는 일반 사람들이 가공되지 않은 주식 데이터를 가져와서 스스로 원하는 차트Chart를 만들 수 있다. 즉, 프로그래밍할 줄 안다면 자신이 주로 보는 차트를 직접 만들어 실시간으로 변화를 확인할 수 있다. 선택한 종목의 데이터를 1초마다 계속 가져와서 새로 그려주면(컴퓨터의 속도는 우리가 생각하는 것 이상으로 빠르므로) 사용자는 큰 불편 없이 1초마다 변하는 데이터를 보면서 매매할 수 있다. 이 책에서는 실시간으로 데이터를 가져와 차트를 업데이트하는 방법은 구현하지 않지만, 이 책을 이해한 독자라면 큰 문제 없이 실시간 주식 프로그램을 만들 수 있다.

주식 분석할 때 주로 보는 보조 지표는 MACD와 볼린저 밴드Bollinger Band다. 일반 사람들이 주로 보는 이평선보다 참고자료로 사용하기에 아주 좋다. 그러나 주식을 하는 사람들은 자신이 생각한 것을 믿는 경향이 있다. 예를 들면, MACD라는 보조 지표를 사용하면 30%만이 종목에 맞게 적용되는데도 MACD를 맹신하는 사람들은 자신에게 잘 맞는 보조 지표라고 생각하며 믿곤 한다. 그러나 보조 지표는 말 그

대로 참고용이며, MACD와 볼린저 밴드를 함께 보는 것이 좋다. 주식에서도 철저한 준비가 필요하다. 자신이 사용하는 알고리즘이 있다면 증권회사에서 받은 데이터를 가지고 시뮬레이션을 하는 것이 좋다.

현재 사회는 잘해야 하는 시대이다. 노동으로 먹고살던 시대에는 근면 성실하기만 하면 잘 살 수 있었지만, 지금은 노동의 시간이 아닌 결과물의 시대가 되었다. 주식을 한다는 것은 머리로 생각하며 올바른 분석을 해야 하는 일이다. 또한, 주식에서 프로그래밍한다는 것은 정보의 분석뿐 아니라 남들보다 빨리 정보를 얻을 힘이 생긴 것이라고 볼 수 있다. 주식 프로그램에서 궁극의 목적은 컴퓨터가 알아서 매매하는 것인데, 머지않은 미래에는 프로그램을 누가 더 잘 만들었는지가 중요할 것이며, 주식 시장은 컴퓨터 간의 경쟁이 될 것이다.

이 책을 나에게 주식을 가르쳐 주신 박준근 형님에게 바친다. 또한, 이 책이 출간되기를 가장 많이 기다린 친구 성연준의 둘째 아들 성주호(2008년생)에게 선물로 준다. 주호는 가끔 전화로 나에게 말했다. "삼촌, 내 책은 언제 줄 거야?"

집필을 마치며

김세훈

일러두기

이 책은 'How-to Series'의 두 번째 책으로 MFC를 이용하여 주식 프로그램을 만드는 방법에 대해 알아본다. 주식 프로그램은 증권회사에서 데이터를 가져와 분석한 후, 자동매매를 함으로써 완성된다. 즉, 시스템 트레이딩 프로그램을 잘 구축하는 것이 중요하다. 하지만 이 책은 MFC 프로그래밍에 초점을 맞추어 주식이라는 주제를 다루고자 한다. 이 책을 이해한 독자라면 증권사에서 데이터를 가져오는 API를 이용하여 자신만의 자동매매 시스템을 쉽게 구현할 수 있다. 여기서 다루는 프로그램은 윈도우 MFC에 기반을 두고 있으며, 이 책을 통해 MFC 프로그래밍을 이해할 수 있는 계기가 될 것이다.

이 책에서 다룰 내용은 다음과 같다.

- 주식 그래프 그리기
- 이평선 구현과 알고리즘 검증
- MACD Moving Average Convergence/Divergence, 이동평균 수렴/확산 지수 구현과 자동매매 검증
- 볼린저 밴드 Bollinger Band 구현과 자동매매 검증

'주식 그래프 그리기'에서는 MFC로 그래프를 구현하는 방법과 MFC의 몇 가지 기능을 다룬다. 첫 번째 항목을 완벽히 이해하면 나머지 항목들은 주식 알고리즘을 중점적으로 다루므로 프로그램을 쉽게 이해할 수 있다. 'How-to Series'의 첫 번째 책인 『소수와 RSA 알고리즘으로 배우는 Big Number 연산』(한빛미디어, 2013)[01]에서 구조체 사용에 대해 설명한 대로, 주식 프로그램도 구조체를 이용하여 필요한 데이터를 자유자재로 가져다 쓸 수 있고, 개인적으로 구현한 알고리즘을 검증하는

---

01 http://www.hanbit.co.kr/ebook/look.html?isbn=9788968486074

데 큰 무리가 없다.

주식은 세계 증시, 유가, 회사 공시 등 많은 정보를 가지고 주식을 판단해야 하지만, 이 책에서는 오직 다섯 개의 정보(시가/고가/저가/종가/거래량)만 가지고 프로그램을 구현하고 공개된 알고리즘만 다룬다.

이 책을 읽는 독자가 차트를 분석하는 데 초보자라면 차트 분석을 경험할 수 있을 것이고, 분석이 가능한 중급 이상의 경력자로 자신만의 분석 알고리즘이 있다면 직접 알고리즘을 구현하여 검증할 수 있을 것이다. MACD와 볼린저 밴드에서 다룰 자동매매는 일봉을 가지고 검증하고, 상수값을 다양한 값들로 변경하면서 검증할 것이다. 프로그래밍을 위해서는 구현할 알고리즘을 정확히 이해해야 하므로 여기서 다루는 보조 지표의 특성을 좀 더 깊이 있게 알 수 있을 것이다.

부록에서는 이베스트투자증권의 API인 Xing API를 간단히 설명하고, 이 책에서 사용하는 데이터를 Xing API를 통하여 받아오는 프로그램을 제공한다. 주식 분석은 가장 최근 데이터를 가지고 분석하는 것이 좋으므로 부록에 수록된 프로그램으로 원하는 데이터를 가져오는 것도 좋다. 또한, 일봉 데이터를 가지고 진행하지만, 주봉 데이터를 받는다면 주간 데이터를 가지고 프로그램의 검증이 이루어질 수 있을 것이다. 필자의 경험으로는 현재 이베스트투자증권의 Xing API가 MFC 프로그램을 만들기에는 가장 적당하며 풍부한 자료를 제공한다. 주식을 분석한다는 취지의 프로그램 책이지만, 잘 만들어진 주식 프로그램은 증권사 API를 사용하여 실시간으로 데이터를 가져와 분석하여 자동매매를 하는 것이 궁극의 목표일 수 있다. 필자는 비록 실제로 주식을 하지는 않지만, 오래전부터 데이터를 가지고 분석 작업

을 진행해 왔다. 따라서 이 책의 내용은 분석 작업의 시작일 수 있으며 독자들이 확실하게 자신만의 알고리즘을 만든다면 모의투자 방식으로 검증 후 실제 매매에 적용할 수도 있을 것이다.

이 책은 주식 프로그램을 간단히 다루었지만, 언젠가는 뛰어난 알고리즘을 구현하는 한국 사람이 뉴욕의 주식 시장에서 시스템 트레이딩으로 그들과 경쟁하기를 바라는 마음이다. 현재는 외국의 프로그램 매매를 우리가 따라가고 있지만, 나중에는 우리도 그들의 시장에서 함께 할 수 있을 것이다. 그런 의미에서 'How-to Series'의 두 번째 불을 조용히 지펴주기 바란다.

대상 독자 및 참고사항

초급 | **초중급** | 중급 | 중고급 | 고급

이 책은 C++를 알고 있는 초급자 이상을 대상으로 윈도우 프로그램을 만드는 MFC 프로그래밍을 설명합니다. 주제가 있는 프로그램을 배우자는 목적으로 주식이라는 주제로 프로그램을 만들면서 설명합니다.

주식을 하는 사람들은 자신만의 프로그램을 만들어 운용하기를 원하지만, 방법을 모르기에 어떻게 시작을 해야 할지 모를 수 있습니다. 이 책은 MFC 프로그램의 버튼 만들기부터 시작하여 프로그램으로 구현하는 방법을 단계적으로 설명합니다.

주식 분석은 증권회사에서 제공하는 HTS 프로그램을 이용하는데, HTS 프로그램은 MFC로 만들었습니다. 따라서 이 책을 이해한 독자는 간단한 개인용 HTS를 만들어서 응용할 수 있습니다. 또한, 주식의 보조 지표인 이평선이 어떻게 만들어졌는지를 알 수 있으며, 주식을 좀 더 깊게 분석할 때 주로 사용하는 보조 지표인 MACD와 볼린저 밴드를 구현하는 방법도 이해할 수 있어서 두 보조 지표를 활용하는 데 도움이 될 것입니다.

보조 지표가 어떻게 만들어지는지 아는 것과 모르고 그냥 보는 것에는 큰 차이가 있습니다. 보조 지표를 만들 수 있다는 것은 그만큼 보조 지표를 여러 측면으로 분석하는 힘이 있다는 것입니다. 이 책에서 제공하는 방법의 하나는 데이터를 이용해서 가상 시뮬레이션을 해보는 알고리즘 검증으로, 자신만의 알고리즘이 있다면 주식 데이터로 검증하는 법도 이해할 수 있습니다. MFC 프로그래밍에 대한 책이지

만, 주식 초보자는 주식 분석을 이해하는 데 도움이 되고, 중급자 이상은 자신만의 알고리즘을 검증하는 데 도움이 될 것입니다.

이 책에 수록된 코드들은 윈도우에서 생성하고 실행시켜야 하며 Visual studio 2008에서 구현되었습니다. 'How-to Series'는 방법론적인 부분에 초점을 맞추어 만들어진 책입니다. 일방적인 지식의 전달이라기보다는 "이러한 방법도 있구나" 하고 이해하는 것이 중요합니다.

예제 테스트 환경

사용 프로그램	설명
Visual Studio 2008 이상	예제는 Windows 환경에서 테스트하였다.
Windows OS	Windows 환경에서만 실행할 수 있다.

책에서 사용한 예제 파일은 다음 웹 사이트에서 내려받을 수 있다.

- https://www.hanbit.co.kr/exam/2664

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

   500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. 무료로 업데이트되는 전자책 전용 서비스입니다.

   종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위해 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

1 | MFC 시작하기

현재 가장 많이 사용하는 컴퓨터 운영체제는 윈도우로, 윈도우에서 실행되는 프로그램은 사용하기 쉽도록 그래픽 부분이 강조된다. 마우스가 없던 시절에는 텍스트만 사용해서 컴퓨터를 작동시켰지만, 윈도우 프로그램은 주로 마우스로 업무를 수행할 수 있다. 이렇게 사용자 편의 위주의 프로그램을 만드는 것은 프로그래머에게 더 많은 코딩을 하게 만든다.

MFCMicrosoft Foundation Class는 윈도우 프로그램을 만드는 확장된 C++ 라이브러리Library다. MFC는 개발자가 좀 더 쉽게 윈도우 프로그램을 만들 수 있는 기능을 제공한다. MFC에서 사용하는 언어는 비주얼 C++라고 부른다. 단어에서도 알 수 있듯이 시각적인Visual 윈도우 응용 프로그램Application을 만들기 위한 프로그래밍 언어다. 지금은 MFC를 사용하는 프로그래머가 많지 않으나, 윈도우 프로그램을 다룰 줄 알면 프로그래밍의 범위를 확장할 수 있다.

증권회사에서 제공하는 API는 MFC에 기반을 두고 있어서 증권이나 주식 관련 데이터를 다루려면 좀 더 MFC에 친숙해져야 한다. 물론, Excel의 매크로 기능으로 주식 데이터를 다룰 수 있지만, 주식 데이터를 좀 더 자유자재로 다루기 위해서는 프로그래밍 언어를 사용하는 것이 훨씬 효과적이다.

여기서는 MFC의 기능을 모두 설명하지 않는다. 기능을 전부 다루려면 두꺼운 책 한 권 분량이 되므로 프로그램을 만들면서 필요한 부분만을 설명한다. 그리고 프로그램을 만들기 위해 시중에 나와 있는 MFC 책을 독파할 필요도 없다. 단지, 구현해야 할 기능들만 인터넷이나 안내서를 통하여 습득하는 것이 좋다. 프로그래밍에서는 모든 기능을 알고 만드는 것보다 필요한 것이 무엇인지를 아는 것이 중요하다.

1.1 Visual Studio

비주얼 스튜디오는 윈도우 프로그램을 만드는 데 필요한 응용 프로그램으로, 컴파일 기능이 있는 편집기Editor라고 할 수 있다. 이 책에서는 비주얼 스튜디오 2008을 사용한다. 대부분의 증권회사 API가 비주얼 스튜디오 2008을 기반으로 만들어졌으므로 최신 버전의 비주얼 스튜디오가 굳이 필요하지 않다. 또한, 이 책에서 구현하는 프로그램들은 상위 버전의 비주얼 스튜디오에서도 실행할 수 있다.

여기서는 비주얼 스튜디오의 자세한 사용법을 명시하지 않으며, 프로그램을 만들면서 필요한 부분만을 설명한다. 자세한 내용은 인터넷 등을 찾아서 습득하기 바란다.

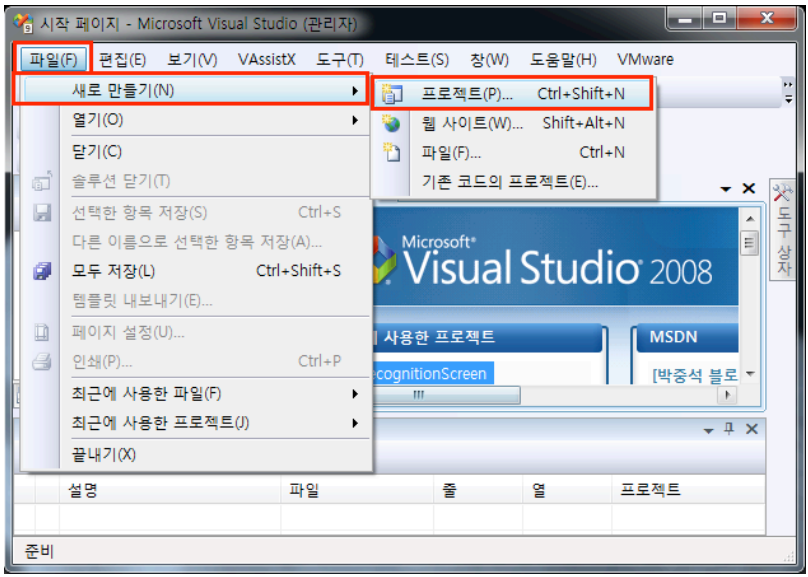
1.2 MFC 프로그래밍을 위한 준비

1.2.1 프로젝트 생성하기

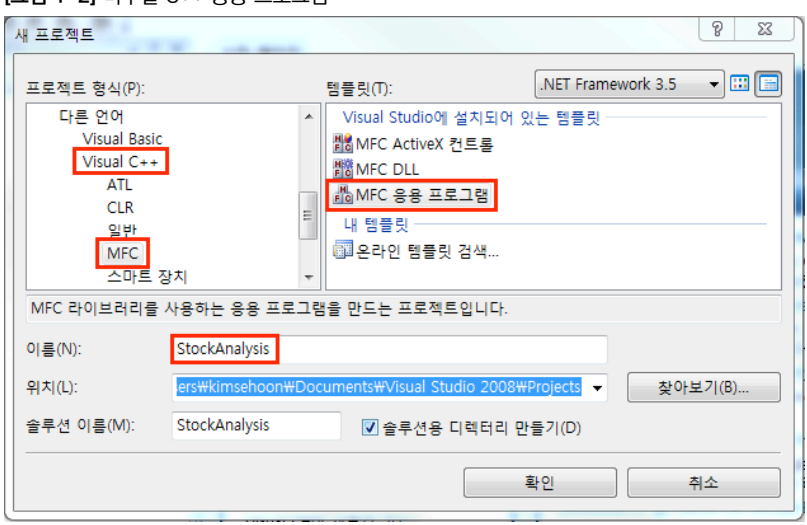
MFC는 비주얼 스튜디오를 실행하고 새 프로젝트를 만든다. [그림 1-1]과 같이 비주얼 스튜디오의 상위 메뉴에서 '파일(F)'을 선택한 후, '새로 만들기(N)'와 '프로젝트(P)'를 선택하여 프로젝트를 생성한다.

새 프로젝트 창이 열리면 [그림 1-2]와 같이 Visual C++을 선택하여 MFC 응용 프로그램을 만들고, '이름(N)'에는 새 프로젝트의 이름을 써준다. 이 책에서 구현하는 모든 프로그램은 StockAnalysis주식 분석이라는 이름을 사용한다.

[그림 1-1] 새 프로젝트 만들기

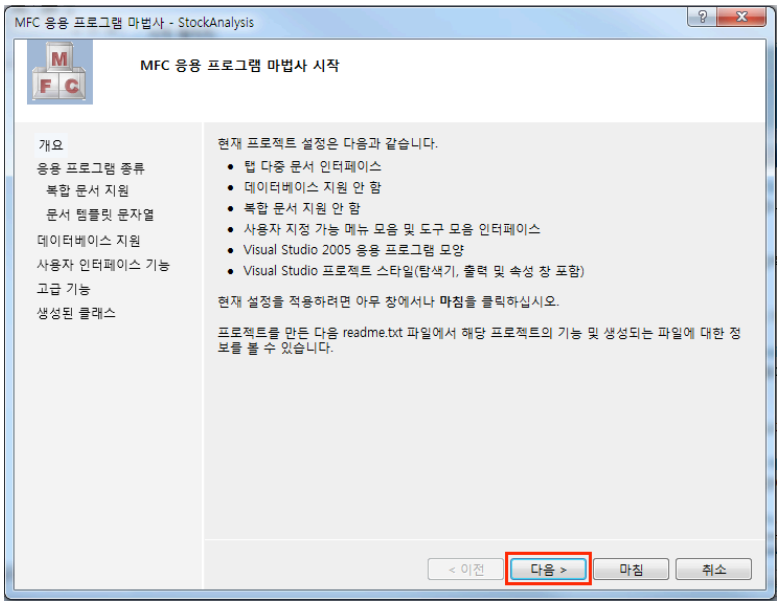


[그림 1-2] 비주얼 C++ 응용 프로그램



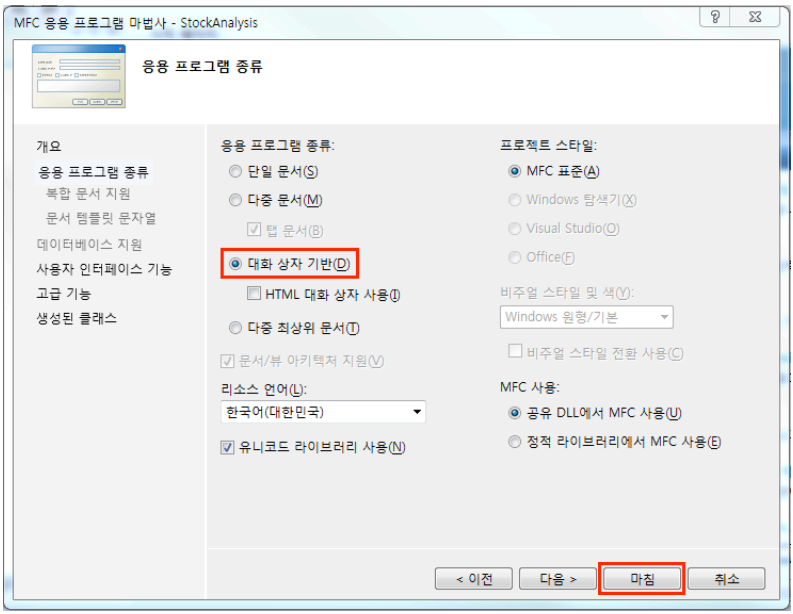
[그림 1-3]처럼 MFC 마법사가 시작되면 '다음'을 눌러서 응용 프로그램의 종류를 선택한다.

[그림 1-3] MFC 마법사 시작



응용 프로그램의 종류는 [그림 1-4]와 같이 '대화 상자 기반(D)'를 선택하고 '마침'을 누르면 새로운 프로젝트가 생성된다. '대화 상자 기반'은 다이얼로그Dialog 프로그램을 만드는 것으로, 응용 프로그램에 메뉴Menu 등을 생성하지 않고 빈Empty 화면에서 프로그램을 만들기 위해서다. 필자는 MFC에서 다이얼로그로 프로그램을 생성하는 것을 좋아하는데, 상업적인 프로그램이 아닌 간단한 프로그램을 만들기에 좋다. 메뉴 등을 만들면 프로그램의 완성도를 높이기 위해 코드 수정이 많이 필요하다. 이 책에서는 내용 전달에 우선을 두었으므로 기능적인 부분들은 생략한다.

[그림 1-4] 대화 상자 기반(다이얼로그) 프로그램 생성

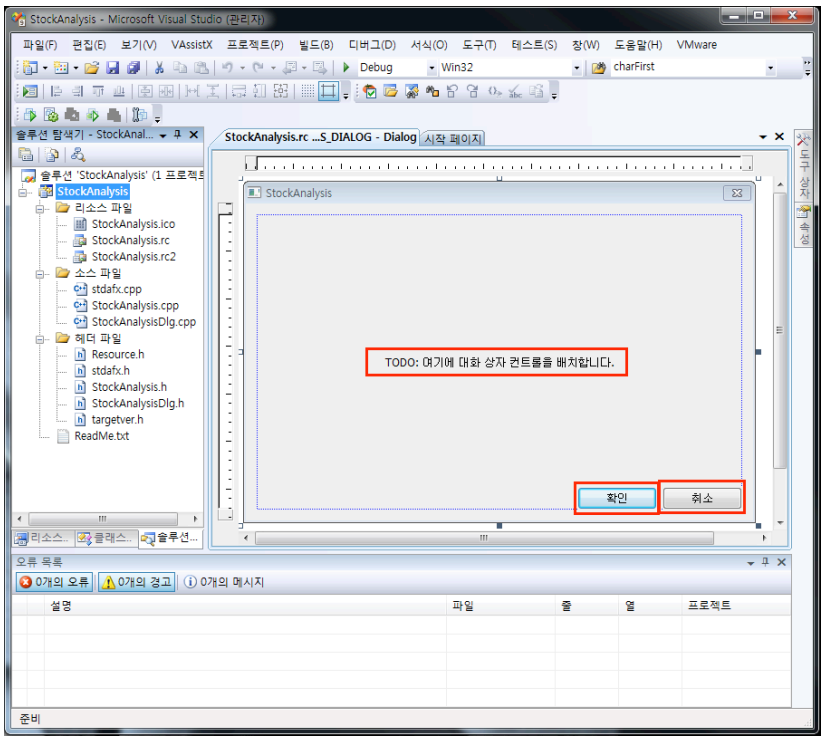


MFC에서 프로젝트를 만들면 기본적으로 생성되는 파일을 보일러플레이트Boilerplate라고 한다. 윈도우 환경에서 프로그램을 구동하는 데 필요한 파일들은 이 보일러플레이트에서 생성하고 필요한 곳에 프로그램을 만들어 나간다.

> **NOTE**
>
> 보일러플레이트를 자세히 알면 좋지만, 윈도우 API를 이해해야 하므로 필자는 권하지 않는다. 마우스나 터치를 이용한 그래픽 응용 프로그램들을 만들 때 기본적으로 생성되는 코드라고 생각하면 된다. 안드로이드나 아이폰에서도 응용 프로그램들을 만들 때 기본적인 코드가 생성된다. 그러므로 기본적으로 생성되는 코드를 어려워할 필요가 없으며 그중에서 무엇이 필요한지만 알면 된다. 고맙게도 인터넷에 많은 자료가 있어서 우리가 원하는 정보를 어렵지 않게 찾을 수 있다. 프로그래밍을 잘하는 데는 필요한 정보를 얼마나 잘 찾아 응용할 수 있는지도 중요하다. 필자는 프로그래밍에서 문제 해결 능력이 가장 중요하다고 강조하고 싶다.

MFC 응용 프로그램 마법사를 끝내면 [그림 1-5]와 같이 다이얼로그 화면이 뜬다. 다이얼로그 바탕에 기본으로 나오는 글자와 버튼들은 삭제한다. 프로그램에 필요한 모든 내용을 직접 구현하므로 하나씩 만들어 가면서 설명한다.

[그림 1-5] 생성된 다이얼로그 기반 프로그램



[그림 1-5]의 왼쪽에 자동으로 생성된 파일 중에서 다음 세 개의 파일이 중요하다.

- StockAnalysisDlg.h
- StockAnalysisDlg.cpp
- stdafx.h

생성된 보일러플레이트에서는 앞의 세 개 파일들만을 가지고 코드 수정이 이루어진다. 파일명 끝에 Dlg라고 이름 붙은 두 파일은 다이얼로그 응용 프로그램에서 그래프를 그리거나 버튼을 눌렀을 때 동작하는 이벤트Event 처리를 위한 코드를 만드는 데 사용된다. 이 책에서는 'StockAnalysisDlg.h'와 'StockAnalysisDlg.cpp' 두 개의 파일과 새로운 클래스Class 파일들을 생성하여 프로그램을 구현한다. 'stdafx.h' 파일은 프로그램 전체에서 전역 변수Global Variable를 사용하는 데 필요하지만, 이 책에서는 사용하지 않는다.

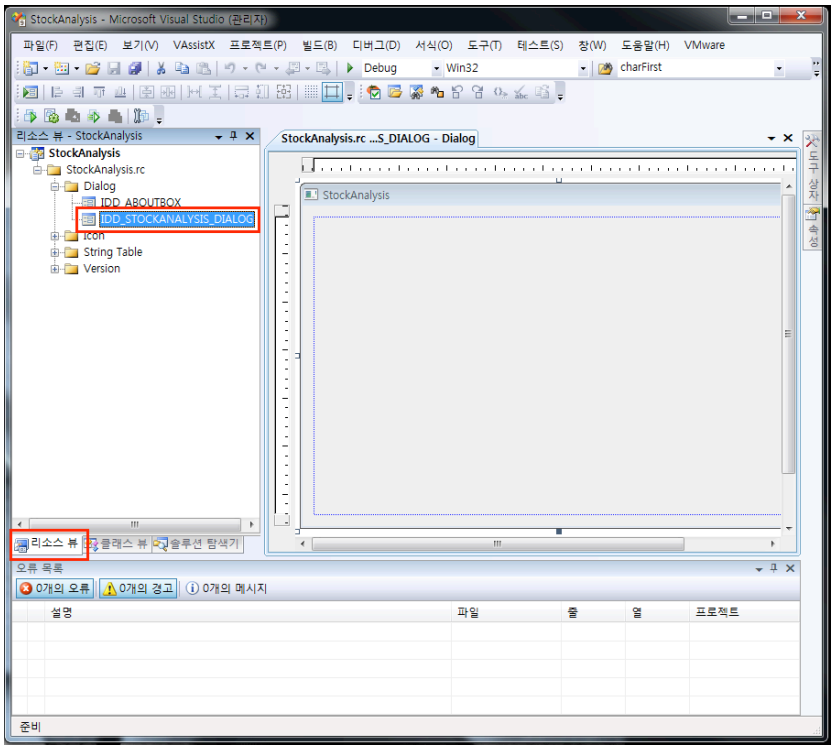
1.2.2 Button 만들기

앞으로 출간할 'How-to Series'에서는 다수의 MFC 프로그램을 다룬다. 이후 출간될 책들은 시리즈의 전작에서 설명한 내용은 생략한다. 따라서 이 책에서 다루는 MFC의 기초 부분은 다음 시리즈에서는 생략될 것이다.

이 절에서는 MFC의 도구 상자를 이용하여 버튼 만드는 법을 자세히 설명하겠다. 여기서 버튼 만드는 법을 이해한다면 다른 도구들도 버튼과 같은 방법으로 만들면 되므로 인터넷이나 기존에 출간된 MFC 관련 서적을 본다 해도 어렵지 않게 내용을 이해할 수 있을 것이다.

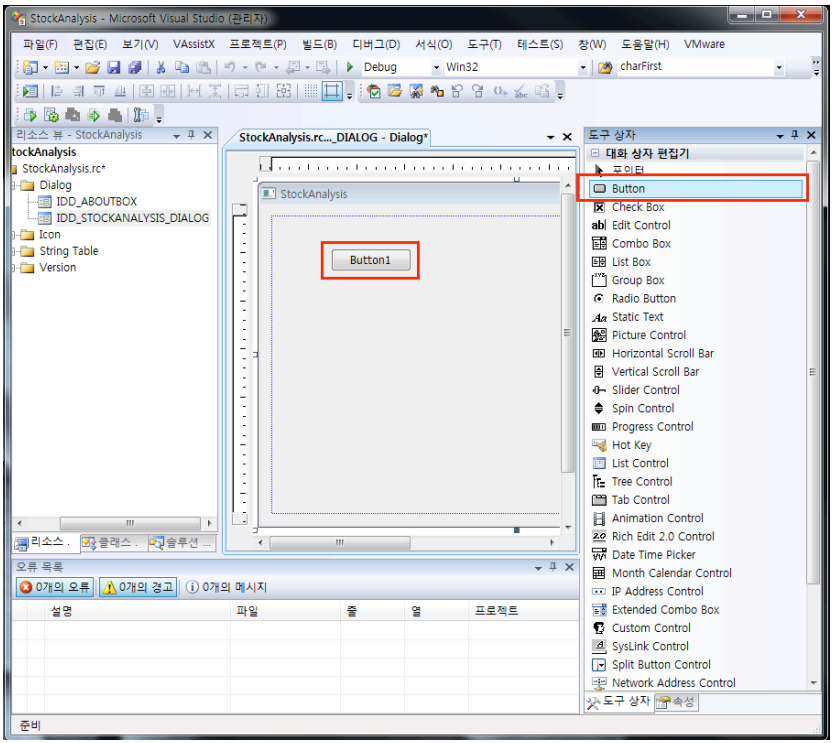
시각적인 프로그램에서 기본으로 생각할 것은 개별 단위들이 속성Property을 가진다는 것이다. 예를 들면, MFC에서는 도구 상자 안의 모든 도구가 속성을 가지며, 웹 프로그램에서는 개별 태그Tag에 속성을 주어 좀 더 보기 좋은 프로그램을 만들 수 있다.

[그림 1-6] 도구를 그리기 위한 다이얼로그



버튼을 그리려면 [그림 1-6]과 같이 '리소스 뷰'를 선택한 후 'Dialog' 폴더의 'IDD_STOCKANAYSIS_DIALOG'를 선택한다. 다이얼로그 창이 열리면 [그림 1-6]의 오른쪽에 위치한 '도구 상자' 탭을 클릭하여 [그림 1-7]과 같이 도구 상자를 연다. 도구 상자에서 Button을 선택한 다음 다이얼로그 창에서 마우스 왼쪽 버튼을 클릭하거나 드래그Drag하면 버튼이 자동으로 화면에 그려진다.

[그림 1-7] 도구 상자의 다양한 도구들

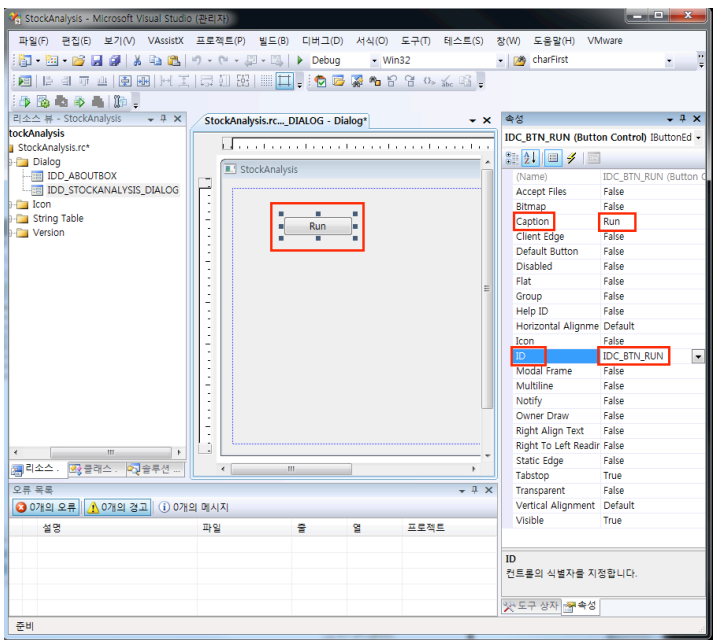


도구 상자에는 매우 다양한 도구들이 있다. 윈도우 프로그램에서 흔히 볼 수 있는 도구들로 원하는 도구를 선택해서 다이얼로그에 그리면 된다. 이 책에서는 다루지 않지만, 도구를 그릴 때 비주얼 스튜디오는 프로그램에 해당 도구를 위한 기본적인 코드들을 자동으로 생성한다. 따라서 프로그램의 깊은 부분까지 이해하지 않아도 프로그램을 쉽게 만들 수 있다.

[그림 1-6]의 오른쪽 '속성' 탭을 선택하면 [그림 1-8]과 같이 속성 창이 열린다. 다이얼로그 창에 그려진 'Button1'을 선택하면 도구 상자에서 해당 'Button'의 속성 정보가 나타나는데, 여기서 중요한 것은 'Caption'과 'ID'다. 'Caption'은 선택한

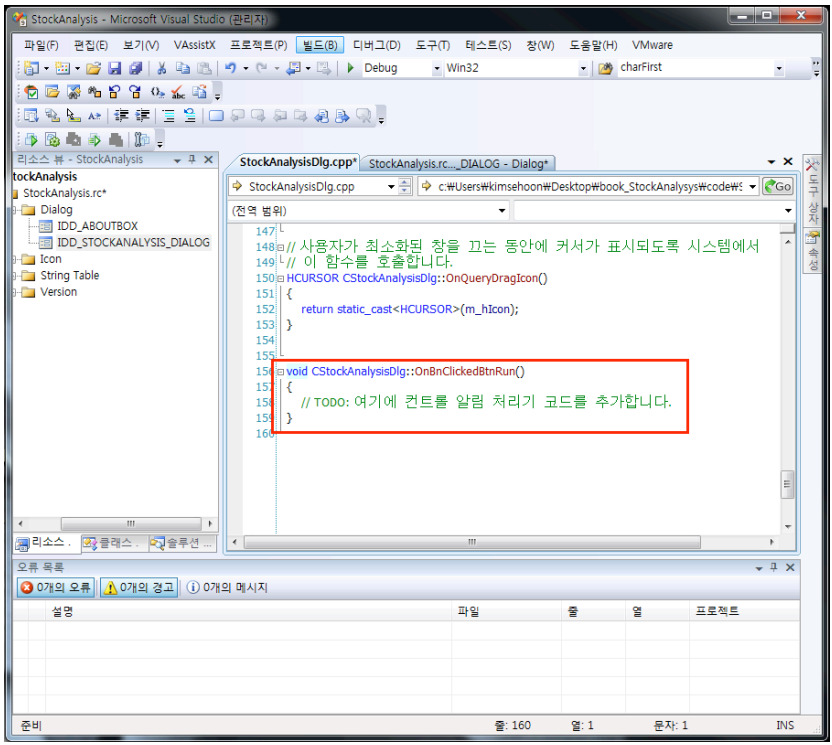
버튼 위에 써지는 텍스트 'Button1'이며(수정 가능), 'ID'는 프로그램에서 사용되는 버튼의 고유 'ID'를 지정한다. 예제에서는 'ID'를 'IDC_BUTTON1'에서 'IDC_BTN_RUN'으로 바꾼다. 이름에서 'BTN'은 'Button'의 약어이고, 뒤에 'RUN'과 같이 버튼의 이름을 써주는 것이 좋다. 프로그램에서 버튼이 많아졌을 때 이처럼 새롭게 이름을 작성해 주면 코드에서 해당 버튼을 쉽게 구분할 수 있다. 속성의 나머지 부분들은 버튼의 시각적인 부분들을 위한 것이므로 여기서는 다루지 않는다.

[그림 1-8] 버튼의 속성 설정하기



다이얼로그 창에 그려진 'Run' 버튼을 마우스로 더블 클릭하면 [그림 1-9]와 같이 코드가 자동으로 생성되고, StockAnalysisDlg.cpp 파일로 이동한다. 코드 아랫부분의 OnBnClickedBtnRun()은 실행 프로그램에서 'Run' 버튼을 클릭했을 때 실행되는 함수다.

[그림 1-9] 버튼을 눌렀을 때 실행되는 함수



실제 버튼 클릭을 위해 새롭게 생성된 코드는 [그림 1-9]에 나와 있는 것 외에 다음과 같은 내용이 더 있다.

[StockAnalysisDlg.h]

```
afx_msg void OnBnClickedBtnRun();
```

[StockAnalysisDlg.cpp]

```
BEGIN_MESSAGE_MAP(CStockAnalysisDlg, CDialog)
```

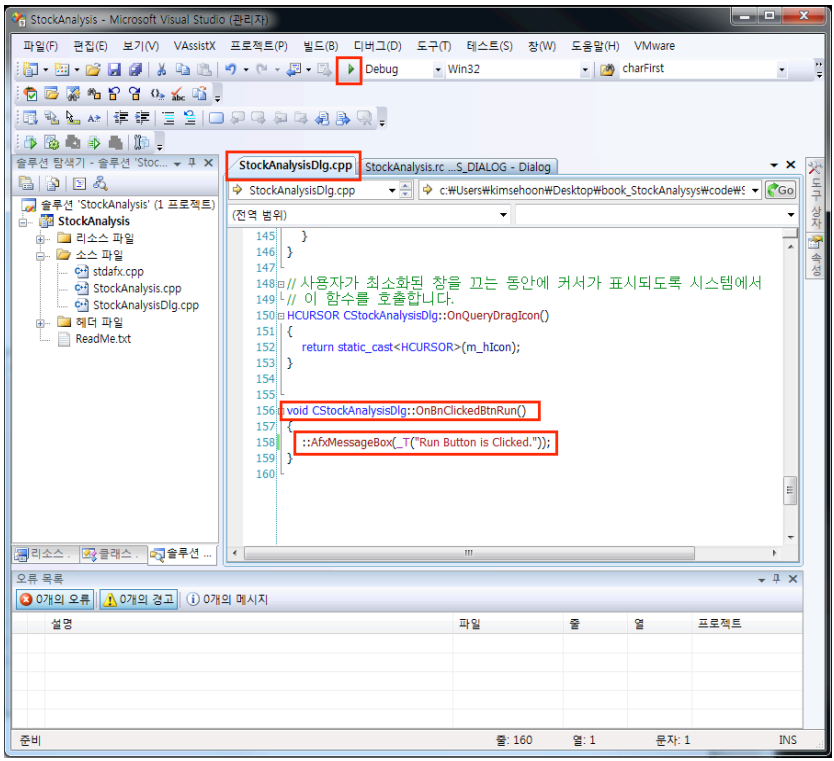
```
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_BTN_RUN, &CStockAnalysisDlg::OnBnClickedBtnRun)
END_MESSAGE_MAP()
```

첫 번째 코드는 버튼을 클릭하면 실행되는 함수를 선언한 것이고, 두 번째 코드의 회색 부분은 버튼을 클릭했을 때 버튼과 해당 함수를 연결한다. 두 번째 코드는 ID가 'IDC_BTN_RUN' 버튼을 클릭했을 때(ON_BN_CLICKED) 'OnBnClickedBtnRun()' 함수를 실행하도록 연결하는 부분이다. 비주얼 스튜디오는 자동으로 이러한 코드를 만들기 때문에 프로그래머가 좀 더 쉽게 프로그래밍을 할 수 있으며, 해당 함수 내에서만 코드 작업을 해주면 된다.

OnBnClickedBtnRun() 함수를 눌렀을 때 메시지 창Message Box를 출력하려면 [그림 1-10]과 같이 OnBnClickedBtnRun() 함수 안에 AfxMessageBox() 함수를 사용한다. 필자는 MFC 프로그램에서 디버깅Debugging할 때 매우 유용한 AfxMessageBox() 함수를 자주 사용한다. AfxMessageBox() 함수의 괄호 안에는 MFC에서만 쓰는 'CString'이라는 문자열 클래스Class를 사용해야 하지만, '_T' 매크로를 이용하여 직접 문자열을 사용해도 된다. AfxMessageBox() 함수는 다음과 같이 사용하여 "Run Button is Clicked."라는 메시지를 출력할 수 있다.

```
::AfxMessageBox(_T( " Run Button is Clicked. " ));
```

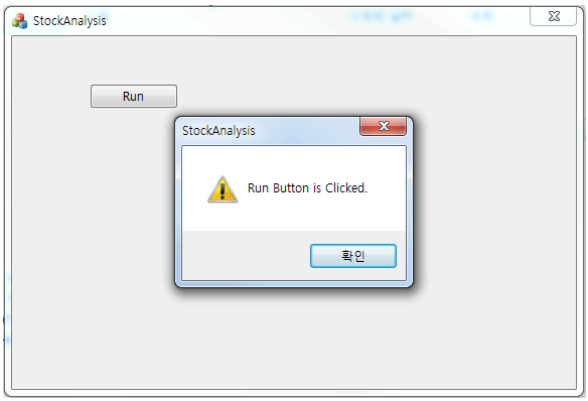
[그림 1-10] 버튼을 눌렀을 때 실행되는 코드



마지막으로 실행 프로그램을 만들어 테스트해 보자. 비주얼 스튜디오의 메뉴에 있는 '플레이Play' 버튼을 눌러도 되고, 단축키로 'Ctrl + F5'를 사용해서 컴파일하고 실행 프로그램을 만들어도 된다. 프로그램을 만들면서 실행 프로그램을 자주 만들어야 하므로 단축키 'Ctrl + F5' 사용을 권한다. 모든 단축키를 숙지하는 것은 힘들지만, 자주 사용하는 단축키 정도는 기억하는 것이 좋다.

프로그램을 실행하면 [그림 1-11]과 같이 'Run' 버튼 하나만 나타나며, 'Run' 버튼을 누르면 "Run Button is Clicked."라는 메시지를 출력하는 창이 생성된다.

[그림 1-11] 실행 프로그램



프로그래밍 언어 기초 책을 보면 항상 나오는 것이 "Hello World"라는 문자열을 출력하는 것이다. 지금까지 버튼을 이용해서 "Hello World"라는 아주 간단한 프로그램을 만들었다고 보면 된다. "Hello World" 프로그램이 항상 먼저 나오는 이유는 오류Error 없이 프로그램을 만들어서 실행되는지를 알 수 있기 때문이며, 이는 중요한 부분이다. 이후부터는 요소를 하나씩 추가하면서 프로그램을 만들어 나가면 된다. 또한, 코드를 조금씩 추가하면서 실행 프로그램을 자주 만들어서 테스트하는 것도 중요하다.[01]

도구 상자 안에 있는 다른 도구들도 이와 같은 방식으로 그려주고 속성을 사용하면 된다. 직접 프로그래밍해 보고 시행착오를 거쳐 알게 된 내용이 오래 기억되므로 여러 가지 도구들을 직접 그려보기 바란다. 또한, 인터넷에 개별 도구들에 대한 자세한 설명이 나와 있으므로 직접 찾아보기를 권한다. 경험상 프로그래밍은 내가 아는 것만을 가지고 프로그램을 만드는 것은 극히 일부분이며, 무엇이 필요한지를 알고 방법을 찾아서 새로운 것을 만드는 일이라고 생각한다.

---

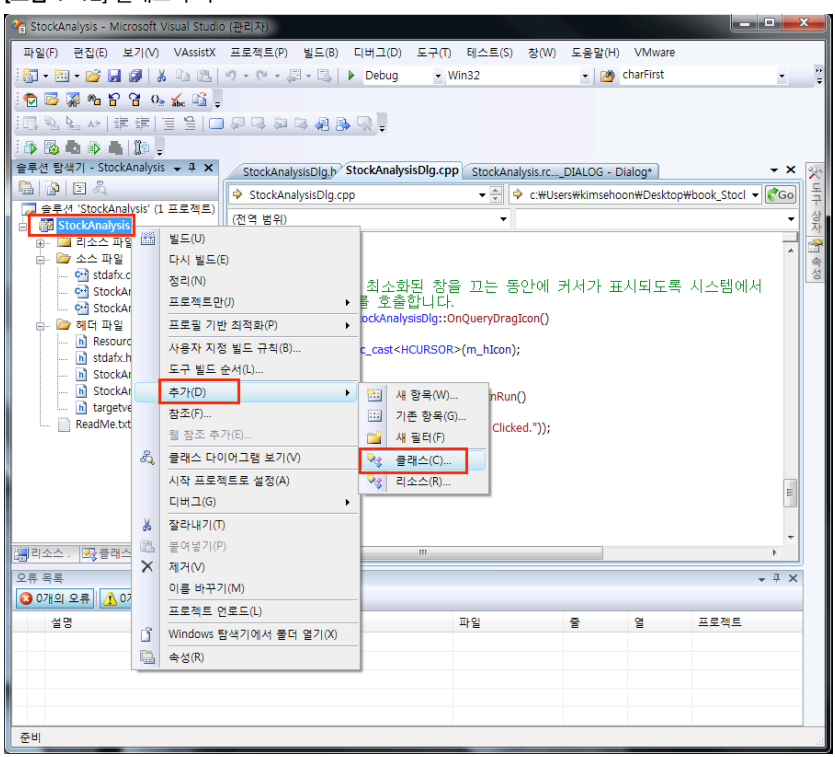
01 'How-to Series'의 첫 번째 책인 『소수와 RSA 알고리즘으로 배우는 Big Number 연산』(한빛미디어, 2013)의 부록에 이 부분이 자세히 설명되어 있으므로 참고하기 바란다.

1.2.3 Class 추가하기

MFC는 C++에 기반을 둔 프로그램이므로 구현하려는 C++ 클래스를 추가하여 프로그램을 만든다. 이 책의 내용 대부분은 새롭게 생성된 클래스 안에서 구현할 것이다. 단지 그래프를 그리거나 버튼 등을 눌렀을 때 동작하는 부분만 StockAnalysisDlg.cpp에 구현한다. 따라서 프로그램의 알고리즘과 같이 중요한 부분들은 추가된 클래스 안에서 새롭게 구현한다.

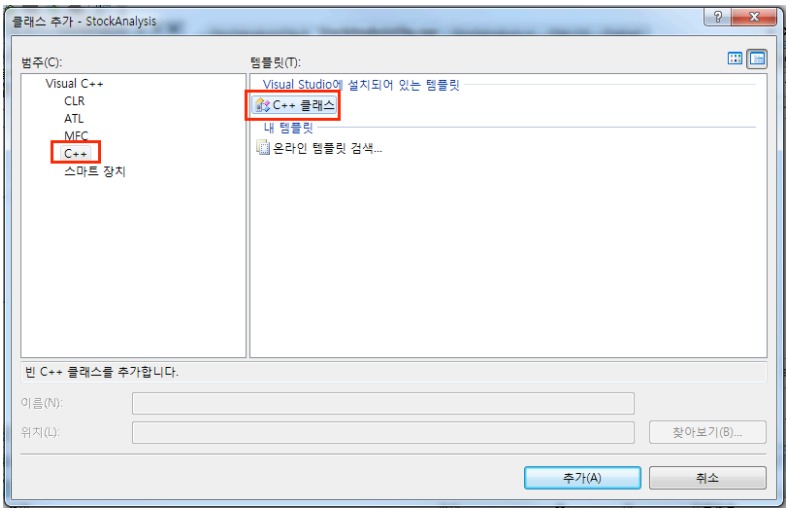
새로운 클래스를 추가하려면 [그림 1-12]와 같이 프로젝트 이름에서 마우스 오른쪽 버튼을 눌러, '추가(D) → 클래스(C)'를 선택한다.

[그림 1-12] 클래스 추가



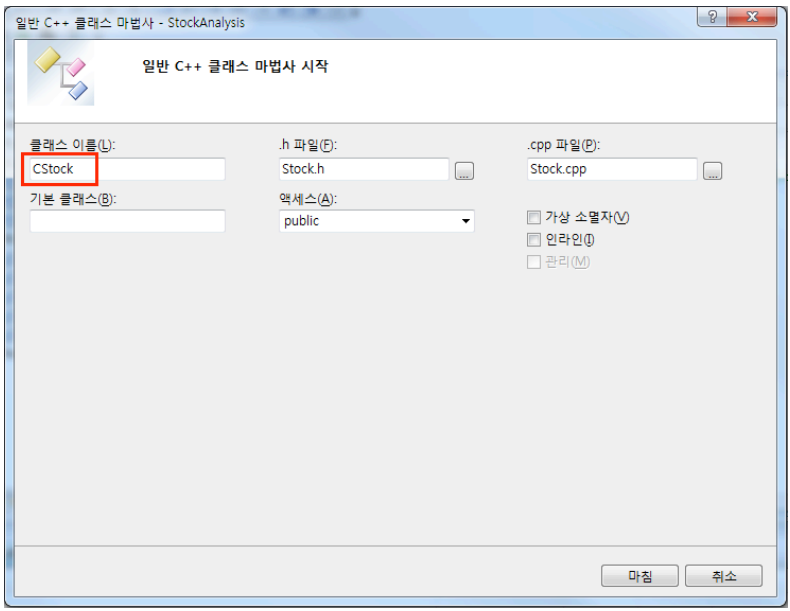
[그림 1-13]과 같이 '클래스 추가' 창이 열리면 'C++ → C++ 클래스'를 선택하고 '추가' 버튼을 누른다.

[그림1-13] C++ 클래스 추가



[그림 1-14]와 같이 C++ 클래스 마법사 창이 열리면 클래스 이름을 'CStock'으로 지정한다.[02] 그러면 첫 문자 C가 생략된 'Stock.h'와 'Stock.cpp' 파일이 생성되고, 'Stock'이라는 클래스에 앞으로 구현하게 될 코드들이 생성된다.

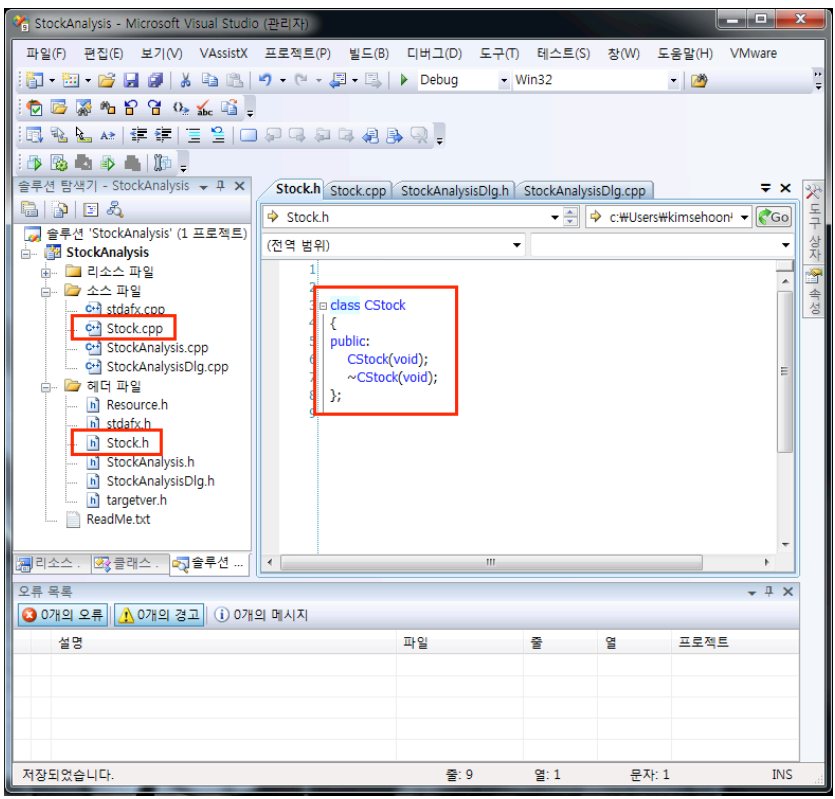
[그림 1-14] 클래스 이름 정하기



02 MFC는 클래스 이름을 정할 때 항상 대문자 'C'를 앞에 써주는 것이 좋다.

[그림 1-15]를 보면 'CStock' 클래스와 관련된 두 개의 파일이 생성되었고, 내용이 없는 빈 클래스가 선언되었음을 알 수 있다. 이 책에서는 두 개의 C++ 클래스에 코드가 구현되는데, 'Stock'이라는 클래스와 그래프를 그리기 위해 계산을 하는 'Graph' 클래스다. 'CGraph' 클래스도 'CStock' 클래스와 같은 방법으로 추가한다.

[그림 1-15] 추가된 클래스



MFC 프로그램을 만들어 본 독자라면 지금까지의 내용은 너무 간단했을 것이다. 하지만 이 책에서는 MFC 프로그래밍을 처음 해보는 독자들도 많을 것으로 판단되어 각 단계를 좀 더 구체적으로 기술하였다.

다음 장은 이 책에서 가장 중요한 부분으로, 데이터를 읽어와서 그래프를 어떻게 그리는지를 설명한다. 여기서 만드는 프로그램은 상업적인 프로그램이 아니므로 간단한 프로그램을 통하여 기본적인 부분을 익히고, 자신만의 기능을 추가하여 개인 프로그램을 만들어 보기 바란다.

2 | Chart 그리기

이번 장에서는 MFC에서 주식 그래프를 그려본다. 주식 데이터를 다루는 'CStock' 클래스와 그래프를 그리기 위해 계산하는 'CGraph' 클래스, 그리고 화면에 그래프를 그려주는 'CStockAnalysisDlg' 클래스에서 프로그램이 구현된다. 여기서 사용되는 데이터는 'data.txt' 파일에 저장되어 있는데, 그 내용은 다음과 같으며 최근 데이터는 부록으로 제공되는 'GetStockData' 프로그램을 사용해서 얻을 수 있다.

```
2130
A000020  동화약품  250
20130823  6140  6220  6020  6120  83828
20130822  6300  6300  6110  6120  119820
……생략……
20120823  5400  5510  5390  5490  84425
A000040  S&T모터스  250
20130823  510  513  502  506  255864
20130822  500  513  488  498  721021
……생략……
```

'data.txt'에서 가장 위에 '2130'은 전체 종목 수를 나타낸다. 즉, 'data.txt'에는 2,130개의 종목 데이터가 저장되어 있다. 두 번째 줄에서 'A000020'은 종목 번호이며, '동화약품'은 종목 이름, 그리고 '250'이라는 숫자는 해당 종목과 관련한 250개의 데이터가 밑에 기록되어 있음을 나타낸다. 250개의 데이터는 250줄에 기록되어 있는데, 한 줄에는 날짜, 시가, 고가, 저가, 종가, 거래량 순으로 저장되어 있다. 'data.txt'에 저장된 형식은 필자가 임의대로 정한 것으로, 개인적으로 프로그램을 만든다면 데이터를 보기 편한 방식으로 저장하면 된다.

2.1 Data Structure

주식 프로그램의 시작은 데이터를 읽어와서 프로그램의 구조체 안에 저장하는 것이다. 그렇다면 데이터를 담기 위한 구조체Structure를 어떻게 만들면 좋을까? 필자는 개별 종목 하나를 하나의 구조체에 담고, 이렇게 만들어진 구조체를 배열Array로 관리한다. 프로그램을 만들 때 같은 자료형과 형태의 데이터는 배열로 만드는 것이 좋다. 종목 데이터를 구조체에 넣는 방법은 여러 가지가 있는데 다음과 같이 두 가지 형태의 구조체로 표현할 수 있다.

[코드 2-1] Stock Data 구조체 (1)

```
struct Company {
      CString strJongMok, strName;      //종목 번호, 종목 명
      int quantity;                     //데이터 개수
      long date[250];                   //날짜
      long startVal[250];               //시가
      long highVal[250];                //고가
      long lowVal[250];                 //저가
      long lastVal[250];                //종가
      long vol[250];                    //거래량
};

Company allCompany[2130];         //2130개의 모든 종목 구조체를 배열로 선언
```

[코드 2-2] Stock Data 구조체 (2)

```
struct Data {
      long date;                        //날짜
      long startVal;                    //시가
      long highVal;                     //고가
      long lowVal;                      //저가
```

```
    long lastVal;                    //종가
    long vol;                        //거래량
};

struct Company {
    CString strJongMok, strName;     //종목 번호, 종목 명
    int quantity;                    //데이터 개수
    Data data[250];                  //주가 데이터
};

Company companies[2130];             //2130개의 모든 종목 구조체를 배열로 선언
```

[코드 2-1]의 구조체는 이전에 개인적으로 만든 프로그램에서 구현한 방법이고, [코드 2-2]의 구조체는 이 책에서 만드는 프로그램에 사용할 방법이다. [코드 2-1]은 주식 데이터를 날짜가 아닌 최소 단위의 데이터 배열로 나타냈고, [코드 2-2]는 날짜에 따른 데이터를 구조체에 담아서 구조체를 배열로 만들었다. [코드 2-2]에서 'Company' 구조체를 배열로 만들었는데 여기의 'companies'라는 변수를 가지고 모든 데이터에 접근할 수 있다.

프로그램을 만들 때 중요한 것 중 하나는 많이 사용하는 상수를 정의해서Define 사용하는 것이다. 헤더 파일에 'Data' 구조체의 배열 크기와 모든 종목 개수의 최대 크기를 정의하는 것이 좋다. 이 책에서는 'Stock.h'에 다음과 같이 상수들을 정의한다.

```
#define  MAX_DATA 250            //한 종목당 250개의 데이터를 가진다.
#define  MAX_COMPANY 2500        //2130개 종목보다 큰 수로 설정한다.
```

이처럼 상수를 정의하면 헤더 파일에서 상수값만 변경해서 프로그램의 크기를 쉽게 바꿀 수 있다. 예를 들어, 종목당 250개의 데이터를 1000으로 바꾸면 종목당

1,000개의 데이터를 가지고 처리할 수 있는 프로그램으로 쉽게 바뀐다. 또한, 상수를 정의하면 프로그램의 가독성을 높여준다. 코드 안에서 250이라는 숫자를 사용하면 이것이 무엇을 의미하는지 알기가 힘들다. 그러나 숫자 대신 'MAX_DATA'라는 새롭게 정의된 상수를 써주면 이름을 통하여 데이터의 최대값임을 쉽게 알 수 있다. 이처럼 사용하면 프로그램 구현이 쉬워지므로 상수를 정의해서 사용하기를 권한다.

다음 [코드 2-3]은 'Stock.h'에서 사용하는 코드다.

[코드 2-3] Data 구조체 (Stock.h)

```
#define MAX_DATA     250
#define MAX_COMPANY  2500

struct Data {
    long date;                    //날짜
    long startVal;                //시가
    long highVal;                 //고가
    long lowVal;                  //저가
    long lastVal;                 //종가
    long vol;                     //거래량
};

struct Company {
    CString strJongMok, strName;  //종목 번호, 종목 명
    int quantity;                 //데이터 개수
    Data data[MAX_DATA];          //주가 데이터
};

struct AllCompany {
    int quantity;                 //전체 종목 개수
    Company companies[MAX_COMPANY]; //2500개 종목 구조체를 배열로 선언
```

```
};

class CStock
{
public:
02      AllCompany allCompanies;
public:
    CStock(void);
    ~CStock(void);
};
```

01 AllCompany 구조체는 이후에 추가되는 이평선Day Moving Average, 이동평균선이나 보조 지표 등 모든 종목의 데이터를 포함하기 위해 추가한다. 여기서는 companies라는 Company 구조체 배열만을 가지고 있지만, 이후에 만들어지는 구조체는 이곳에 변수로 선언된다.

02 CStock 클래스에서 모든 데이터에 접근하려면 allCompanies라는 변수를 public으로 선언한다. 외부에서 CStock 클래스를 생성하면 모든 종목의 데이터에 대한 접근이 가능하다.

이전 책[01]에서 강조했듯이 프로그램에서 구조체를 이해하는 것이 가장 중요하다. 데이터가 어떻게 프로그램에서 구조화되는지를 이해해야만 나머지 부분을 쉽게 만들 수 있다. 특히, 주식 프로그램에서는 데이터를 가져와서 구조화할 수 있다면 프로그램의 반은 끝났다고 볼 수 있다. 물론, 여기서는 증권회사에서 실제 데이터를 가져오지는 않았지만, 직접 증권회사에서 데이터를 가져와서 구조체로 넣는다면 실시간 자동매매 프로그램을 쉽게 구현할 수 있을 것이다.

2.2 File Data 읽기

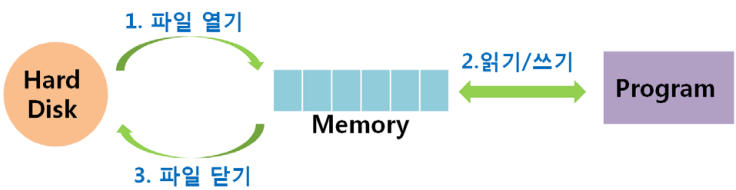
MFC에서 파일을 읽고 쓰는 것은 C++에 기반을 두고 있어서 특별히 고려해야 할 것이 없다. 또한, C++에서 C 함수도 사용할 수 있으므로 이 책에서는 C 함수를 사용하여 파일에 읽고 쓰는 작업을 할 것이다.

01 『소수와 RSA 알고리즘으로 배우는 Big Number 연산(http://goo.gl/fNfOVw)』(한빛미디어, 2013)

MFC에서 C 함수를 사용할 때 고려해야 할 점은 자료형Data Type이다. 즉, 데이터를 MFC에서만 사용하는 문자열 자료형인 'CString'으로 변환하는 작업이 필요하다. 파일에서 데이터를 읽을 때는 C 함수로 읽어서 'char *' 형태의 문자열을 'CString'으로 형변환해야 하며, 파일에 기록할 때는 'CString'을 'char *' 형태로 변환해서 C 함수로 기록한다.

프로그램을 만들다 보면 파일 입출력을 많이 다루게 된다. 잠시 파일 입출력이 어떤 원리로 이루어지는지 알아보자.

[그림 2-1] 파일 입출력



[그림 2-1]과 같이 파일은 하드디스크에 저장되어 있다. 프로그램에서 파일을 열면Open 파일의 모든 내용은 메모리에 올라간다. 파일을 열 때 읽기Read만 가능한지 아니면 쓰기Write도 가능한지를 지정하는데, 읽기만 가능한 파일은 메모리에서 읽을 수만 있고, 쓰기가 가능한 파일은 읽고 쓰기가 가능하다. 또한, 파일의 마지막부터 추가하여 쓰기가 가능한 'Append' 속성을 부여할 수도 있다. 그리고 파일이 메모리에 올려져 있으므로 특정 위치에서 읽기와 쓰기도 할 수도 있다. 이 내용은 프로그램 관련 기초 서적이나 인터넷에서 쉽게 찾을 수 있으므로 여기서는 다루지 않는다.

파일 열기로 파일이 메모리에 올려진 후에 프로그램은 메모리에서만 읽기/쓰기를 하게 된다. 프로그램에서 읽기/쓰기 작업이 다 이루어진 후 '파일 닫기'를 하면, 읽기 속성의 파일은 연결이 끊어지고 쓰기 속성의 파일은 메모리의 내용을 하드디스

크에 저장한다. 간단하게 파일 입출력은 크게 '파일 열기', '메모리에서 파일 내용 작업', '파일 닫기'의 세 단계로 구분하여 이해하는 것이 좋다.

파일에서 읽기를 간단히 표현하면 다음과 같이 나타낼 수 있다.

```
FILE *fp;                              //파일 자료형 - FILE
fp = fopen( " fileName " , "rt");      //파일 열기 - fopen
fscanf(fp, "%d \n", &variable);        //파일 읽기 - fscanf
fclose(fp);                            //파일 닫기 - fclose
```

파일을 열 때 fopen( ) 함수 안의 'rt'로 속성을 주는데, 'r'은 Read를, 't'는 Text로 읽겠다는 의미이다. 모든 줄에 사용되는 변수 fp는 File Pointer로, 메모리에 올려진 파일의 주소를 가리킨다. 즉, 'FILE *fp'는 변수 fp를 파일 포인터로 사용한다는 것이고, 두 번째 줄의 'fp = fopen("fileName", "rt")'는 'fileName'이라는 파일을 하드디스크에서 읽어와 메모리에 올리고 fp가 메모리에 올려진 파일을 가리키고 있다고 생각하면 된다. 세 번째 줄의 'fscanf(fp, "%d \n", &variable)'은 fp 파일에서 정수형(%d) 값을 'variable'이라는 변수로 읽어오는 것이다. 이때 주의할 것은 변수 앞에 '&' 문자를 써주어야 한다. '&' 문자를 사용하면 메모리에서 변수의 주소값을 나타내므로 "메모리에 있는 variable 변수의 공간에 데이터를 집어넣어라"는 의미가 된다. 마지막 줄의 'fclose(fp)'은 처리가 끝난 fp 파일을 닫는다.

파일에 쓰기는 읽기와 비슷하며 다음과 같이 나타낼 수 있다.

```
FILE *fp;                              //파일 자료형 - FILE
fp = fopen( " fileName " , "wt");      //파일 열기 - fopen
fprintf(fp, "%d \n", variable);        //파일 쓰기 - fprintf
fclose(fp);                            //파일 닫기 - fclose
```

다음은 CStock( ) 함수를 선언하고 기본 뼈대를 구현한 부분이다.

[코드 2-4] CStock 함수 선언(Stock.h)

```
class CStock
{
public:
	AllCompany allCompanies;
public:
	CStock(void);
	~CStock(void);

	void Run();
	void ReadDataFromFile();
	void WriteDataToFile();
};
```

[코드 2-5] CStock 함수 구현 전 뼈대(Stock.cpp)

```
void CStock::Run()
{
	ReadDataFromFile();
	WriteDataToFile();
}

void CStock::ReadDataFromFile()
{
}

void CStock::WriteDataToFile()
{
}
```

[코드 2-6] CStock 변수 정의(StockAnalysisDlg.h)

```
#include "Stock.h"

#pragma once

class CStockAnalysisDlg : public CDialog
{
private:
     CStock *stock;
     ##### 생략 #####
};
```

[코드 2-7] CStock 변수 생성과 실행(StockAnalysisDlg.cpp)

```
BOOL CStockAnalysisDlg::OnInitDialog()
{
     ##### 생략 #####
     stock = new CStock();
     ##### 생략 #####
}

##### 생략 #####

void CStockAnalysisDlg::OnBnClickedBtnRun()
{
     stock->Run();
}
```

지금까지 구현한 코드를 'Ctrl + F5' 키로 컴파일이 잘 되는지 확인한다. 이제는 CStock 클래스에 선언된 'ReadDataFromFile()' 함수와 'WriteDataToFile()' 함수 안의 내용을 구현한다. 'ReadDataFromFile()' 함수는 [코드 2-8]과 같이 구현한다.

[코드 2-8] 'data.txt' 파일에서 데이터 읽기(Stock.cpp)

```
void CStock::ReadDataFromFile()
{
    int cn;
    FILE *fp;

01  fp = fopen("data.txt", "rt");

02  if(fp==NULL) {
        ::AfxMessageBox(_T("data.txt file is not opened"));
        exit(1);
    }

03  cn = fscanf(fp, "%d \n", &allCompanies.quantity);

04  if(cn==0) {
        ::AfxMessageBox(_T("Error: can't get allCompanies.quantity."));
        exit(1);
    }

    char jongmok[20];
    char name[50];

    for(int i=0; i<allCompanies.quantity; i++) {
        cn = fscanf(fp, "%s %s %d \n", jongmok, name, &allCompanies.compan
ies[i].quantity);
        if(cn==0) {
            ::AfxMessageBox(_T("Error: fscanf error inside the first for
loop"));
            exit(1);
        }

05      allCompanies.companies[i].strJongMok.Format("%s", jongmok);
        allCompanies.companies[i].strName.Format("%s", name);
```

```
            for( int j =  0; j < allCompanies.companies[i].quantity; j++)
            {
                cn = fscanf(fp, "%ld %ld %ld %ld %ld %ld \n",
                    &allCompanies.companies[i].data[j].date,
                    &allCompanies.companies[i].data[j].startVal,
                    &allCompanies.companies[i].data[j].highVal,
                    &allCompanies.companies[i].data[j].lowVal,
                    &allCompanies.companies[i].data[j].lastVal,
                    &allCompanies.companies[i].data[j].vol);

                if(cn==0) {
                    ::AfxMessageBox(_T("Error: fscanf error inside the second
    for loop"));
                    exit(1);
                }
            }
        }

06      fclose(fp);
    }
```

---

01 'data.txt' 파일을 읽기 위해 메모리에 올리고 파일 포인터 fp로 가리킨다. 'data.txt' 파일은 텍스트 형태이므로 't' 옵션을 준다.

02 'data.txt' 파일이 없거나 열 수 없으면 메모리에 파일을 올릴 수 없으므로 메모리를 가리키는 'fp'는 아무 값도 없는 NULL을 가진다. 이때 오류 메시지를 출력하고 프로그램을 종료한다.

03 'data.txt' 파일에서 지정된 형식으로 데이터를 읽어온다. 이때 변수가 차지하고 있는 메모리의 주소값을 의미하도록 변수 앞에 '&' 문자를 써준다. 따라서 주소값이 가리키는 메모리의 변수 공간에 파일 데이터를 복사한다.

04 'fscanf( )' 함수가 정상적으로 동작하지 않으면 '0'을 반환하며, 오류 메시지를 출력하고 프로그램을 종료한다.

05 'jongmok'이라는 'char *' 변수를 'strJongMok'이라는 'CString' 변수로 형변환한다. 여기서는 'CString'의 'Format( )' 함수를 사용한다.

06 메모리를 가리키는 파일 포인터 'fp'의 연결을 해제하고 파일을 닫는다.

코드에서 구조체의 접근은 단계적으로 이루어진다. 'allCompanies.companies[i].data[j].date'와 같이 'allCompanies' 변수에서 시작하여 단계적으로 하위 구조체로 내려가며 세부 데이터에 접근한다. 구조체 안에 구조체가 존재하므로 하위 구조체를 따라가면서 코드가 길어진다. 이 구조체를 어떻게 정의하느냐에 따라 코드는 다양하게 만들어진다. 처음에 구조체를 잘못 정의하여 개발 중에 구조체를 바꾸면 많은 부분을 변경해야 한다. 이러한 이유로 충분한 시간을 들여서 프로그램의 설계 단계에서 완성도를 높이는 것이 프로그램 개발 시간을 단축하는 방법이다.

[코드 2-9]는 'WriteDataToFile()' 함수를 구현한 것으로, 구조체에 저장된 데이터들을 'data2.txt' 파일에 기록한다.

[코드 2-9] 'data2.txt' 파일에 데이터 쓰기 (Stock.cpp)

```
void CStock::WriteDataToFile()
{
     FILE *fp;
     fp = fopen("data2.txt", "wt");

     if(fp==NULL) {
          ::AfxMessageBox(_T("data.txt file is not opened"));
          exit(1);
     }

     fprintf(fp, "%d \n", allCompanies.quantity);
     for(int i=0; i<allCompanies.quantity; i++){
          fprintf(fp, "%s %s %d \n",
          LPSTR(LPCTSTR(allCompanies.companies[i].strJongMok)),
          LPSTR(LPCTSTR(allCompanies.companies[i].strName)),
01        allCompanies.companies[i].quantity);

          for(int j =  0; j < allCompanies.companies[i].quantity; j++){
               fprintf(fp, "%ld %ld %ld %ld %ld %ld \n",
```

```
                    allCompanies.companies[i].data[j].date,
                    allCompanies.companies[i].data[j].startVal,
                    allCompanies.companies[i].data[j].highVal,
                    allCompanies.companies[i].data[j].lowVal,
                    allCompanies.companies[i].data[j].lastVal,
                    allCompanies.companies[i].data[j].vol);
            }
        }

        fclose(fp);
    }
```

01 파일에 데이터를 쓸 때 '&'를 변수 앞에 붙이지 않는 것은 변수의 값을 기록하므로 메모리 주소를 알 필요가 없기 때문이다. 'LPSTR'과 'LPCTSTR'은 MFC에 정의된 자료형으로, 'CString'을 'char *'로 형변환하기 위해 사용된다.[02]

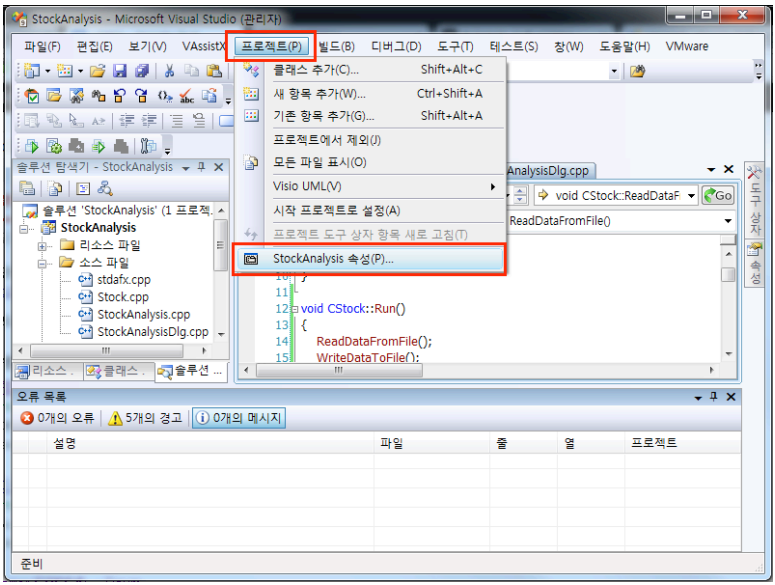
구현한 프로그램을 실행시키고 'Run' 버튼을 누르면 'data.txt' 파일의 내용이 'data2.txt'에 복사된다. 데이터가 정확히 복사되었다면, 'ReadDataFromFile()' 함수가 구조체에 제대로 데이터를 넣은 것이다. 앞으로는 'ReadDataFromFile()'만을 이용하여 주식 데이터를 읽어오고 프로그램을 추가로 구현한다.

앞의 코드를 컴파일할 때 'Error C2664'가 발생할 수 있다. 이것은 윈도우 프로그래밍을 할 때 자주 발생하는 문자열 문제로, 문자 집합Character Set의 설정을 변경해야 한다. 즉, '유니코드'에서 '멀티바이트'로 문자 설정을 변경해야 한다. [그림 2-2]와 같이 메뉴의 '프로젝트(P) → 속성(P)'에서 [그림 2-3]과 같이 '구성 속성 → 일반 → 문자 집합 → 멀티바이트 문자 집합 사용'을 선택하면 문자 집합이 변경된다.[03]

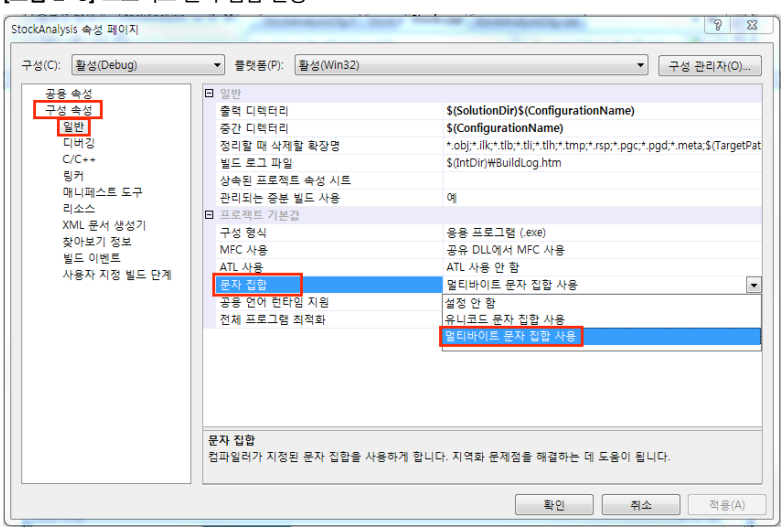
02 'LPSTR'과 'LPCTSTR'은 MFC에서만 사용하는 자료형으로, 이에 대한 자세한 설명은 따로 자료를 찾아보길 바란다.

03 이것은 필자가 MFC를 사용하면서 가장 많이 부딪히는 문제 중 하나다. 오류가 날 때는 '문자 집합'을 변경하면서 컴파일을 해보거나 오류 번호를 가지고 인터넷에서 찾아보는 것이 가장 빠른 해결책이다. 프로그래밍 언어는 규칙이 정해져 있다. 다른 사람이 정해 놓은 규칙을 모두 알 수는 없으므로 그 규칙을 찾아서 해결해야 한다. 논리적인 오류가 아닌 프로그래밍 언어의 규칙 때문에 부딪히는 오류는 인터넷을 통하여 해결하는 것을 권한다.

[그림 2-2] 프로젝트 속성 변경

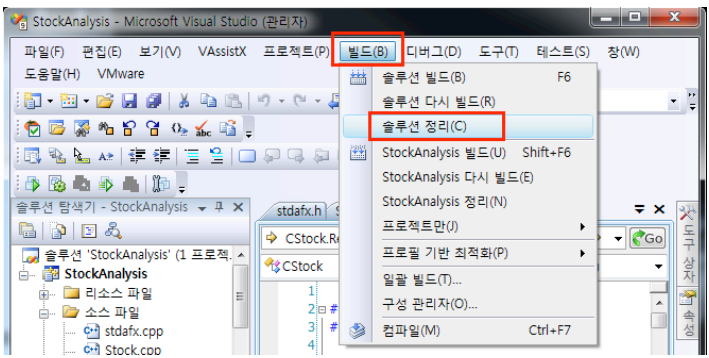


[그림 2-3] 프로젝트 문자 집합 변경



문자 집합을 변경한 후 컴파일이 안 되면 [그림 2-4]처럼 '빌드(B) → 솔루션 정리(C)'를 선택하여 정리한다. 솔루션 정리를 하면 미리 컴파일되어 남아 있는 것들이 지워진다. MFC 프로그램을 구현하다 보면 솔루션 정리를 자주 하게 된다. 특별한 오류 없이 컴파일이 안 되면 솔루션 정리를 하고 컴파일해 보는 것도 좋다.

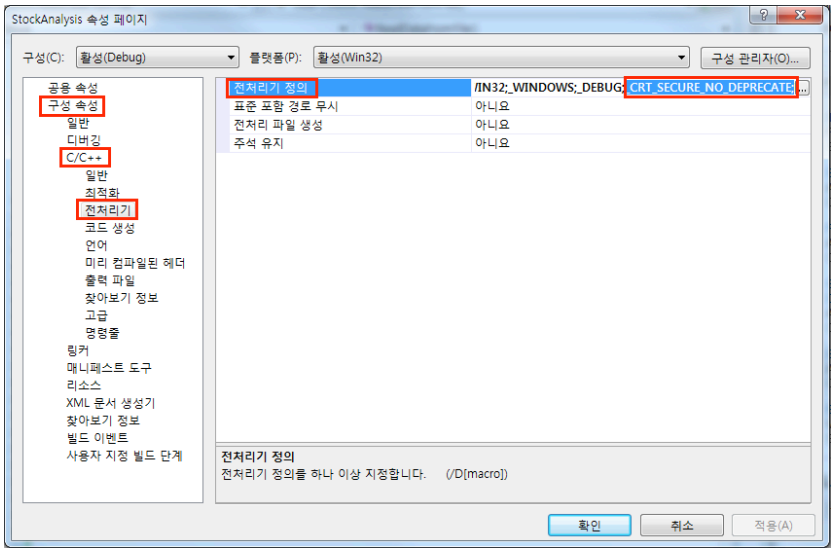
[그림 2-4] 솔루션 정리



MFC에서 C 언어의 함수를 사용하면 경고Warning가 발생하지만, 프로그램을 구동하는 데는 아무 문제가 없다. C/C++의 함수는 잘 알아두면 많은 곳에서 쓰일 수 있어서 MFC의 함수는 어쩔 수 없는 경우에만 사용하고자 한다. 여기서 구현하는 프로그램은 상업적인 용도가 아니므로 파일 입출력 부분에서 C 언어의 함수를 사용한다. MFC 프로그램에서 MFC 함수가 아닌 C 함수를 사용한 점에 대해서 이해를 바란다.

실제로 경고는 무시해도 되므로 MFC의 전처리기에서 경고 메시지를 보이지 않게 설정한다. 경고 번호가 'Warning C4996'이므로 [그림 2-5]와 같이 속성 창에서 C/C++의 전처리기에 "_CRT_SECURE_NO_DEPRECATE;"라는 글자를 추가한다. 경고가 없는 프로그램을 만들어서 완성도를 높이는 것도 중요하다. 예를 들어, 우주로 쏘아 올리는 로켓에 관련된 프로그램을 만든다면 어떠한 오류나 경고도 없어야 한다. 하지만 프로그램을 만들다 보면 접하게 되는 경고를 무시할 때가 많다.

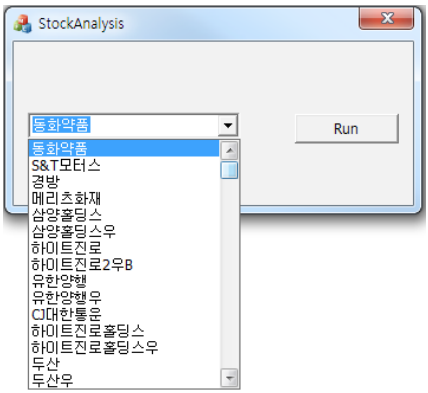
[그림 2-5] C 언어 함수 사용 시 경고 메시지 안 보이게 하기



2.3 Combo Box 만들기

콤보 박스Combo Box를 만드는 이유는 주식 종목들을 목록에 나열하고 콤보 박스에서 하나를 선택하여 해당 종목의 그래프를 그려주기 위해서다. 프로그램에서 2,000개가 넘는 종목들에 대한 접근은 콤보 박스를 통해 이루어진다. 콤보 박스를 마우스로 선택하면 [그림 2-6]과 같이 종목 이름이 나열된다. 이제부터 만드는 프로그램은 이 콤보 박스에서 종목 이름을 선택하면 해당 종목에 대한 그래프를 그린다.

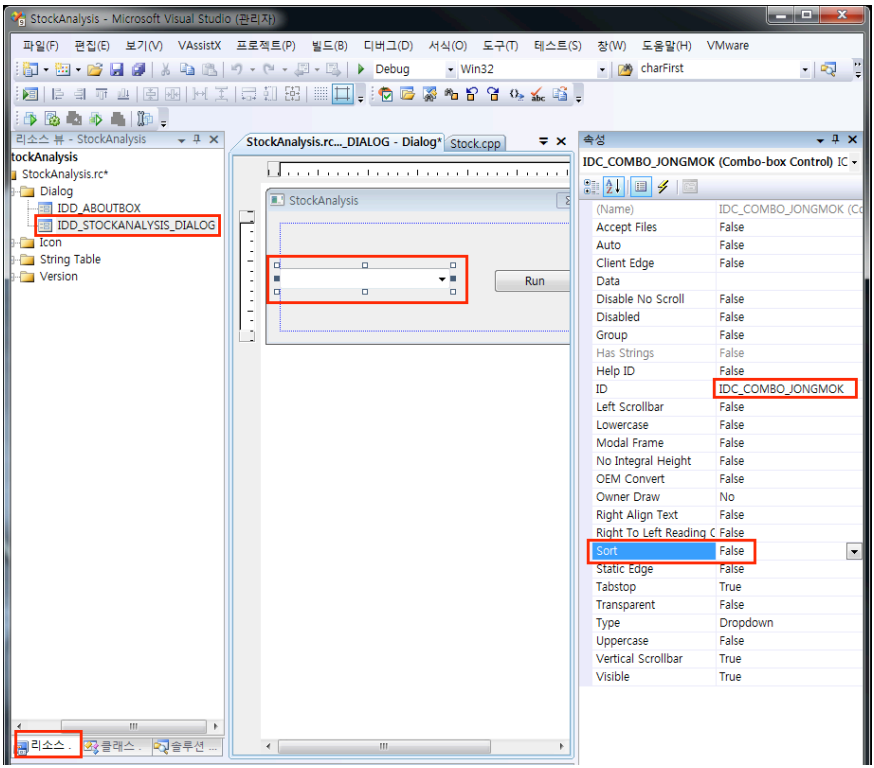
[그림 2-6] 종목 콤보 박스



콤보 박스를 그리려면 버튼을 만든 방법과 같이 '도구 상자'에서 '콤보 박스 Combo Box'를 선택한다. 콤보 박스의 속성에서 [그림 2-7]과 같이 'ID'를 'IDC_COMBO_JONGMOK'로 지정하여 종목에 대한 콤보 박스라는 것을 알 수 있게 한다. 이때 중요한 것은 속성 중 'Sort' 항목을 'False'로 설정하는 것이다. 'Sort'가 'True'면 콤보 박스의 목록은 이름순으로 정렬되며, 'False'면 콤보 박스에 삽입된 순서대로 정렬된다. 여기서 만드는 프로그램은 증권회사에서 제공하는 순서대로 종목이 정렬되므로 반드시 콤보 박스의 'Sort' 항목은 'False'로 설정한다.[04]

04 필자는 이전에 주식 프로그램을 다 만들고 'Sort'를 'False'로 설정하지 않아서 종목을 선택했을 때, 다른 종목의 그래프를 그리는 오류를 범하였다. 한참 뒤에야 이 문제점을 인지하게 되었지만, 프로그램에 대한 자만심에 빠져 증권회사에서 제공하는 데이터에 오류가 있다고 판단하여 증권회사의 게시판에 문의하였다. 독자들은 필자와 같은 오류를 범하여 잘못된 프로그램을 만들지 않기를 바라며 'Sort'의 중요성을 인지하기 바란다.

[그림 2-7] 종목을 위한 콤보 박스 속성



콤보 박스를 마우스로 더블 클릭하면 'StockAnalysisDlg.cpp' 파일에 [코드 2-10]과 같이 'OnCbnSelChangeComboJongmok()' 함수가 추가된다. 콤보 박스의 종목 중 하나를 선택하면 실행되는 함수로, 함수의 내용은 다음 장에서 주식 그래프를 그릴 때 추가한다.

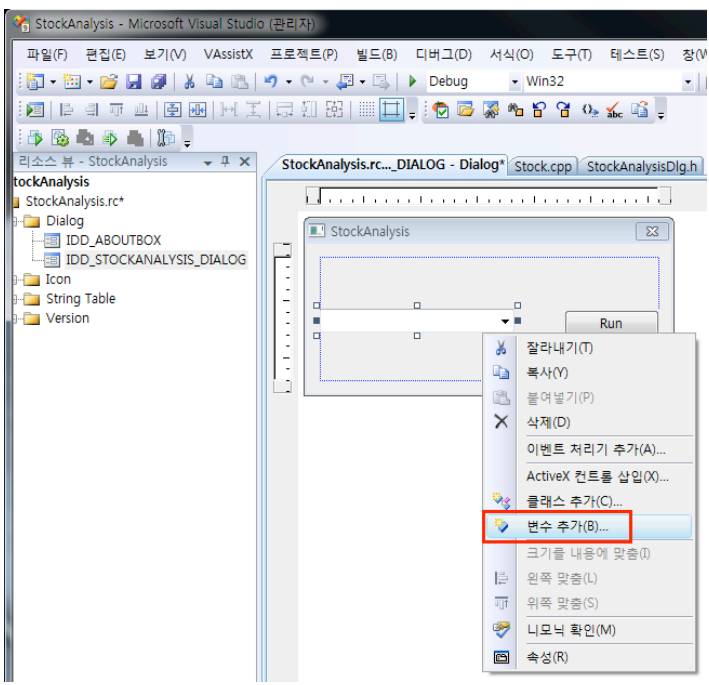
[코드 2-10] 콤보 박스에서 종목이 선택되었을 때 실행되는 함수(StockAnalysisDlg.cpp)

```
void CStockAnalysisDlg::OnCbnSelchangeComboJongmok()
{
```

```
        // TODO: 여기에 제어 알림처리기 코드를 추가한다.
    }
```

콤보 박스를 이용하려면 콤보 박스를 제어하는 제어 변수Control Variable가 추가되어야 한다. 변수에는 값을 저장하는 'Value'와 해당 항목을 제어할 수 있는 'Control' 두 가지가 있다. 콤보 박스는 'Control' 변수가 필요하므로 [그림 2-8]과 같이 콤보 박스에서 마우스 오른쪽 버튼을 눌러 '변수 추가(B)'를 선택한다.

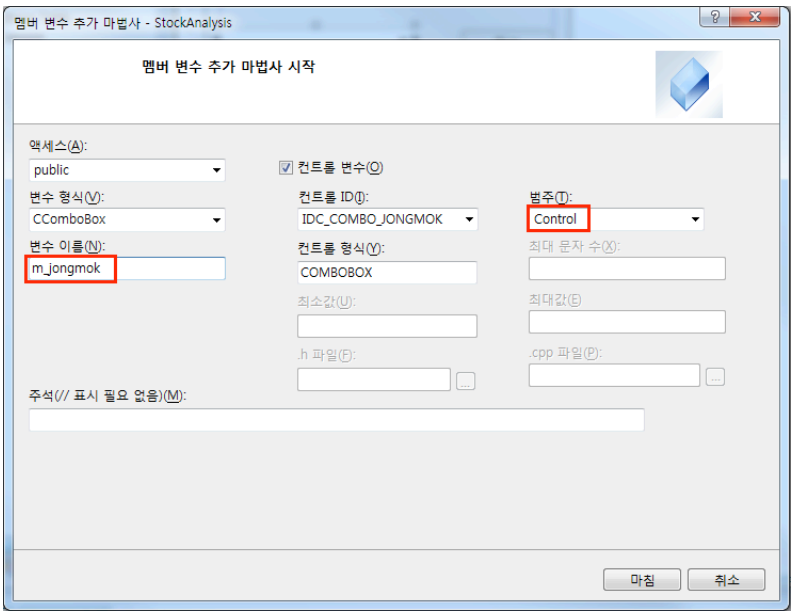
[그림 2-8] 콤보 박스에서 변수 추가



[그림 2-9]와 같이 마법사 창이 뜨면 '범주(T)' 항목에서 'Control'을 선택하고, '변수 이름(N)'에는 'm_jongmok'이라고 적는다. 변수 이름을 정할 때는 멤버Member라는 의미의 'm_'을 이름 앞에 항상 써주는 것이 좋다. 이렇게 하면 클래스 변수와

구별할 수 있고, MFC의 변수라는 것을 쉽게 알 수 있다.

[그림 2-9] 콤보 박스에서 'Control' 변수 추가



[그림 2-9]에서 'Control' 변수인 'm_jongmok'을 추가하면 'StockAnalysisDlg.h'와 'StockAnalysisDlg.cpp' 파일에 다음과 같은 코드가 추가된다. 만약 자동으로 추가되지 않으면 두 개의 파일에 다음 내용을 추가한다.

[StockAnalysisDlg.h](파일의 아랫부분)

```
CComboBox m_jongmok;
```

[StockAnalysisDlg.cpp] (파일의 중간 부분)

```
void CStockAnalysisDlg::DoDataExchange(CDataExchange* pDX)
```

```
{
    CDialog::DoDataExchange(pDX);
    DDX_Control(pDX, IDC_COMBO_JONGMOK, m_jongmok);
}
```

이제 'Control' 변수인 'm_jongnmok'을 이용하여 콤보 박스의 종목들에 대한 제어가 가능하다. 'CComboBox' 클래스의 함수들을 사용하여 목록에서 어떤 종목이 선택되었는지를 알게 되고, 해당 종목에 대한 데이터를 처리하거나 그래프를 그릴 수도 있다. 또한, 콤보 박스의 목록은 순서가 정해져 있고, 번호는 0부터 차례대로 시작한다. 콤보 박스 목록에서 제일 위에 있는 것이 0, 두 번째는 1, 세 번째는 2가 된다. 이와 같이 순서로 선택 종목의 번호Index를 알 수 있다.

다음은 앞에서 구현한 구조체를 가지고 콤보 박스와 어떻게 연동하는지 알아보자.

모든 종목 이름을 콤보 박스에 넣지만, 이 책의 중간부터는 모든 종목이 아닌 알고리즘으로 간추려진 종목만 콤보 박스 목록에 나타난다. 따라서 모든 종목의 데이터를 가지고 있는 'AllCompany' 구조체가 아닌 선택된 종목만을 가지고 있는 새로운 구조체가 필요하다. 이를 위해'SelectedCompany' 구조체를 'Stock.h' 파일에 추가로 구현한다. 'SelectedCompany' 구조체는 [코드 2-11]과 같이 선언한다. 구조체 안의 변수 'companies'는 포인터로, 'AllCompany' 구조체에서 'companies' 변수의 일부분을 가리킨다.

[코드 2-11] 선택된 종목들을 콤보 박스에서 사용하기 위해 생성된 'SelectedCompany' 구조체 (Stock.h)

```
##### 생략 #####
struct AllCompany {

int quantity;                    //전체 종목 개수
Company companies[MAX_COMPANY];  //2500개의 종목 구조체를 배열로 선언
```

```
};

struct SelectedCompany {
01      int quantity;
02      Company *companies[MAX_COMPANY];
};

class CStock
{
public:
        AllCompany allCompanies;
03      SelectedCompany selectedCompanies;
public:
        CStock(void);
        ~CStock(void);

        void Run();
04      void makeSelectedCompanyFromAllCompany();
        void ReadDataFromFile();
        void WriteDataToFile();
};
```

---

01 선택된 종목의 개수를 나타낸다.

02 포인터 변수인 'companies'는 'AllCompany' 구조체 안의 'companies'에서 선택된 것만 가리킨다. 'AllCompany'의 'companies'는 구조체 변수로, 모든 종목 데이터를 포함하므로 메모리에서 차지하는 크기가 크다. 'SelectedCompany' 구조체의 'companies'는 포인터 변수이므로 4바이트 크기의 배열이다.

03 다른 클래스에서 'CStock' 클래스를 생성하여 'SelectedCompany' 구조체로 접근하기 위해서 'CStock' 클래스 안에 'selectedCompanies' 변수를 선언한다.

04 'SelectedCompany'가 'AllCompany'의 모든 종목을 가리키기 위해서 'makeSelectedCompanyFromAllCompany()' 함수가 사용된다.

[코드 2-12]는 'Stock.cpp' 파일에 'makeSelectedCompanyFromAllCompany()'

함수를 구현하여 이전 장에 나온 'Run()' 함수를 실행한다.

[코드 2-12] 'allCompany'의 모든 종목을 가리키는 함수(Stock.cpp)

```
    void CStock::Run()
    {
        ReadDataFromFile();
        makeSelectedCompanyFromAllCompany();
    }

    void CStock::makeSelectedCompanyFromAllCompany()
    {
01      selectedCompanies.quantity = allCompanies.quantity;

        for ( int i=0; i<selectedCompanies.quantity; i++) {
02          selectedCompanies.companies[i] = &allCompanies.companies[i];
        }
    }
```

01 'selectedCompanies'는 'allCompanies'의 모든 종목을 가리키므로 총 개수는 같다.

02 'selectedCompanies'의 'companies' 변수는 'allCompanies'의 'companies' 주소를 가진다. 'companies'를 포인터로 사용하지 않으면 데이터를 복사하면서 메모리 낭비와 프로그램의 속도 저하가 발생한다.[05]

[코드 2-13]은 콤보 박스 목록에 종목 이름을 삽입하고 화면에 보이게 하는 작업이다. 여기서 'm_jongmok'은 클래스 변수이므로 해당 클래스에 포함된 함수들을 사용한다.[06]

05 물론 매우 빠르게 연산하므로 성능 저하를 직접 느끼지 못할 수 있지만, 예측할 수 있는 성능 저하는 줄이는 것이 좋다.

06 프로그래밍은 외우는 분야가 아니다. 그때마다 필요한 함수들을 찾아서 구현하므로 'm_jongmok'에서 사용하는 함수들이 어떻게 쓰이는지만 이해하자.

[코드 2-13] 콤보 박스 목록에 선택된 종목 이름 삽입(StockAnalysisDlg.cpp)

```
    void CStockAnalysisDlg::OnBnClickedBtnRun()
        {
            stock->Run();
01          m_jongmok.ResetContent();

            for (int i=0; i<stock->selectedCompanies.quantity; i++) {
02              m_jongmok.AddString(stock->selectedCompanies.companies[i]
    ->strName);
        }

03      m_jongmok.SetCurSel(0);
04      UpdateData(FALSE);
05      RedrawWindow();
    }
```

01 'Control' 변수인 'm_jongmok'의 'ResetContent()' 함수를 사용하여 콤보 박스를 초기화한다.

02 'AddString()' 함수를 사용하여 종목 이름을 하나씩 콤보 박스에 추가한다.

03 'SetCurSel(0)' 함수를 사용하여 콤보 박스의 0번째 항목을 현재 선택된 종목으로 설정한다.

04 'UpdateData()' 함수는 프로그램 화면의 데이터를 MFC 변수와 연동시킨다. 'UpdateData(TRUE)'면 프로그램 화면의 데이터를 MFC 변수에 삽입하고, 'UpdateData(FALSE)'면 MFC 변수의 값을 프로그램 화면에 보인다. 콤보 박스에 추가된 종목 이름은 'UpdateData(FALSE)'일 때 콤보 박스에 보인다.

05 'RedrawWindow()' 함수는 프로그램 화면에 그림을 다시 그려 준다. 여기서는 'UpdateData(FALSE)'에 의해서 변경된 데이터로 다시 그린다.[07]

콤보 박스는 주식 프로그램처럼 목록을 보여주는 프로그램에서 유용하게 쓰인다. 콤보 박스를 직접 만들어본 적이 있다면 '도구 상자'의 다른 항목들도 정보를 찾아서 쉽게 구현할 수 있을 것이다.

07 프로그램 화면에 데이터를 업데이트하는 'UpdateData()' 함수와 화면을 다시 그려주는 'RedrawWindow()' 함수는 MFC에서 자주 사용하는 중요한 함수들이다.

2.4 Chart 그리기

이번에는 [그림 2-10]처럼 MFC에서 차트Chart를 그리는 부분을 구현한다.

[그림 2-10] 주식 차트

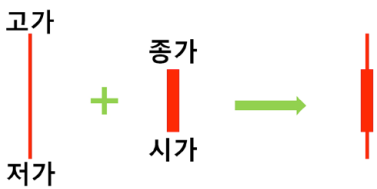
주식은 차트를 보면서 분석한다. 차트는 빨간색과 파란색 봉으로 그려지는데, 주가 변동에 따라 이 색이 정해진다. 프로그램 화면에 데이터가 상세히 나오면 좋지만, 프로그램 코드를 단순화시키기 위해 최소한의 데이터만을 화면에 보여주도록 구현하겠다.

이 부분을 보면 MFC에서 그림을 어떻게 그리는지를 이해할 수 있고, 구현된 코드가 많아서 많은 연산이 필요하더라도 상상하는 것 이상으로 컴퓨터는 빠르게 동작한다는 것을 알게 될 것이다. 컴퓨터의 두뇌인 CPU는 상당히 빠르게 동작하며, 메모리에 올라온 데이터도 빠르게 사용하는 편이다. 프로그램을 실행하면 가장 많은 시간이 걸리는 것이 파일 입출력 부분이다. 이제부터 구현하는 프로그램도 실행하면 파일로부터 데이터를 가져올 때 많은 시간이 걸린다. 하지만 데이터를 가지고

온 후 콤보 박스의 종목을 선택하면 선택된 종목의 차트는 즉시 그려진다.[08]

주식 차트Candle Chart는 MFC에서 선을 그리기만 하면 된다. 가장 많이 보는 주식봉은 [그림 2-11]과 같이 가늘고 긴 선과 두껍고 짧은 두 개의 선을 겹쳐서 그린다. 물론 사각형으로도 그릴 수 있으나, 선은 시작과 끝점의 위치만 알고 두께만 조절해주면 되므로 좀 더 수월하게 구현할 수 있다.

[그림 2-11] 두 개의 선으로 그린 주식봉



[그림 2-11]에서 가늘고 긴 선은 '고가'와 '저가'를, 짧고 굵은 선은 '시가'와 '종가'는 나타낸다. '종가'가 '시가'보다 높으면 빨간색(양봉)으로 그리고 반대이면 파란색(음봉)으로 그린다. 모든 데이터는 증권회사에서 가져올 수 있으므로 기본적인 차트는 데이터에 기반을 두어 그린다.

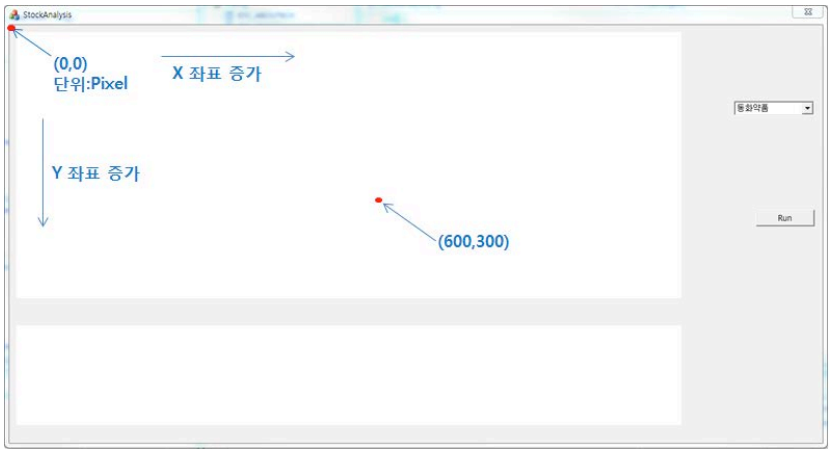
그림을 그릴 때는 프로그램 화면에서 그리고자 하는 위치를 프로그램에 정확히 알려주어야 한다. 위치는 픽셀Pixel 단위의 아주 작은 점의 좌표라 생각하면 된다.[09] [그림 2-12]와 같이 화면의 왼쪽 위 끝의 좌표는 (0,0)이고 오른쪽으로 갈수록 X 좌표가 증가하며, 아래로 갈수록 Y 좌표가 증가한다. 그래프는 이 좌표의 위치를 정확히 알고 선을 그린다.

---

08 사람은 눈으로 보고 뇌로 생각하며 손으로 매매 버튼을 누르지만, 컴퓨터는 데이터를 받는 즉시 0.1초도 안 돼서 판단이 끝나고 매매한다. 이것이 시스템 트레이딩의 매력 중 하나다. 그리고 주식을 하는 사람이라면 주식 매매에서 감정 조절이 안 된다는 것을 알 것이다. 알고리즘과 데이터로는 팔아야 하는데 다시 반등하리라는 기대 심리가 작용한다. 이것은 인간이기에 어쩔 수 없다. 그래서 주식을 하는 사람들은 좋은 알고리즘으로 프로그램을 만들고 싶어 하지만, 인간의 생각을 프로그램으로 만드는 것은 쉬운 작업이 아니다.

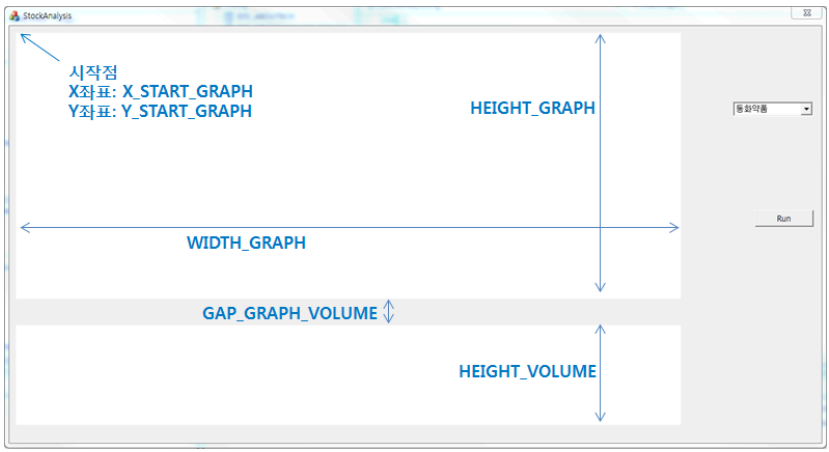
09 예를 들어, TV 해상도가 1920픽셀×1080픽셀이라면 화면에 가로로 1920개, 세로로 1080개의 픽셀이 있다.

[그림 2-12] 윈도우 화면에서 좌표 위치



이제는 프로그램의 관점에서 화면의 위치를 생각해 보자. 앞에서 프로그램을 개발할 때 숫자를 직접 넣는 것보다 상수를 사용하는 것이 편리하다고 설명했다. C++에서 상수는 'static const'를 사용하여 변수로 정의하는 방법이 있는데, 여기서는 단순히 '#define'으로 이름을 정의한다. [그림 2-13]에는 그림을 그리는 영역의 위치와 길이들의 이름이 정의되어 있다. 화면에서 하얀색 부분은 차트를 그리는 부분으로, 사각형을 그리고 그 안을 하얀색으로 채운다. 거의 모든 프로그램에서 사각형은 왼쪽 위의 시작점과 넓이(X 좌표 길이), 높이(Y 좌표 길이)를 가지고 그린다.

[그림 2-13] 프로그램 화면에 정의된 영역



[그림 2-13]에 정의된 이름들은 [코드 2-14]의 'Graph.h' 파일에서 '#define'으로 값을 정의한다. 이렇게 '#define'으로 정의된 상수들은 컴파일할 때 해당 값으로 변경된 후 컴파일이 진행된다. 어떤 책에서는 C++에서 '#define'을 사용하는 것보다 'static const'를 사용하는 것이 좋다고 설명하기도 하지만, 이 책에서는 코드의 구현을 간단히 하려는 목적이 있으므로 '#define'으로 정의한다. 참고로 다음 두 코드는 같은 내용이다.

```
#define X_START_GRAPH            10
static const int X_START_GRAPH = 10;
```

그래프를 그리는 데 필요한 'CGraph' 클래스는 [코드 2-14]와 [코드 2-15]에서 구현된다. 'CGraph'는 'CStock' 클래스의 값들을 이용하여 X, Y 좌표값을 계산하고, 이 X, Y 좌표값들을 사용하여 'StockAnalysisDlg.cpp'에서 화면에 그래프를 그린다. 즉, 'CGraph'는 화면에 차트를 그리기 위한 점들의 위치를 계산하는 클래스다.

[코드 2-14] 그래프 헤더(Graph.h)

```
#pragma once

#include "Stock.h"

#define X_START_GRAPH    10
#define Y_START_GRAPH    10

#define WIDTH_GRAPH  1000
#define HEIGHT_GRAPH 400

#define HEIGHT_VOLUME    150

#define GAP_GRAPH_VOLUME 40

#define WIDTH_LINE   1
#define WIDTH_THICK_LINE 3

struct Point {
    int X;
    int Y;
};

//-----------------------------------------------------------------
struct PointArray {
    int quantity;
    Point point[MAX_DATA];
};

//-----------------------------------------------------------------
struct PointData {
    PointArray startVal;     // 시가
    PointArray highVal;      // 고가
    PointArray lowVal;       // 저가
    PointArray lastVal;      // 종가
```

```
    PointArray volume;      //volume
};

//-----------------------------------------------------------------
class CGraph
{
    public:
        static PointData * GetPointData(Company* data);
        static void GetDataForChart(Company* data, PointData* ptData);
};
```

01 '#define'으로 차트를 그리는 영역을 정의한다.

02 차트를 그릴 때 봉의 두께를 정의한다. 가는 선은 두께가 1이며 굵은 선은 두께가 3이다.

03 하나의 점(X,Y)는 'Point' 구조체 안에서 정수형 X, Y 값으로 표현된다.

04 같은 특성의 점들을 배열로 저장하기 위해 'PointArray' 구조체를 만든다. 차트에 그려지는 모든 종가의 점Point은 'PointArray' 구조체의 'point[ ]' 배열에 차례대로 저장된다. 배열의 왼쪽은 최근 종가의 데이터고 오른쪽으로 갈수록 이전 데이터다.

05 'PointData' 구조체는 차트에 그려지는 모든 차트의 점을 저장한다. 차트에 그려지는 위치는 시가, 고가, 저가, 종가, 거래량의 점들로 각각 'PointArray' 구조체에 배열로 저장된다. 이전에는 실제 데이터들이 시간에 흐름에 따라 구조체로 만들어져 저장되었지만, 하나의 종목만을 쉽게 그리기 위해서 여기서는 데이터의 특성을 기준으로 나눈다.

06 'CGraph' 클래스로, 종목의 데이터를 받아서 점들의 위치만을 알려주면 되므로 속성Attribute은 정의되지 않고, 처리 함수만 정의한다.

07 'StockAnalysisDlg.cpp'에서 호출되는 'GetPointData'는 하나의 종목을 매개 변수로 받아서 이 종목의 좌표 데이터를 저장하는 'PointData' 구조체를 반환한다. 함수 앞에 'static'을 사용하므로 'CGraph' 클래스를 생성하지 않고 'CGraph::GetPointData( )'를 바로 호출할 수 있다.[10]

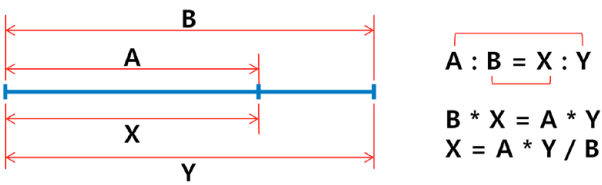
08 'GetDataForChart( )' 함수는 'CGraph'에서 이루어지는 모든 계산을 수행한다. void로 선언되어 반환하는 것은 없지만, 종목 데이터의 값을 계산하여 포인터로 넘겨진 'ptData'에 계산된 값을 넣어준다.[11]

10 'static'을 사용하는 이유는 C++의 클래스 특성보다는 C에서 함수를 직접 호출하는 것처럼 구현하기 위해서다.

11 이것은 참조에 의한 호출(Call-By-Reference)로 데이터를 전달하는 방법이다.

실제 데이터를 가지고 프로그램 화면에 그리려면 점의 좌표를 정확히 알려주어야 한다. 여기서 사용하는 방법은 중학교 수학 시간에 배운 z이다. [그림 2-14]는 비율 계산을 시각적으로 보여 준다. A와 B의 값이 같은 단위이고 X와 Y가 같은 단위일 때, 그림의 오른쪽과 같이 'A : B = X : Y'의 수식이 만들어진다. 이때 A, B, Y 세 값을 알고 있다면 X는 수식으로 계산할 수 있다. 만약 X가 프로그램 화면에서의 점의 위치라면 나머지 값들을 알고 있으므로 비율 계산으로 점의 위치 X를 알 수 있다.

[그림 2-14] 비율 계산



[코드 2-15]는 'CGraph'의 함수들을 구현한 것으로, 실제 데이터를 화면에 그려주기 위해 좌표를 계산한다. 함수의 계산이 끝나면 'PointData' 구조체를 반환하는데, 이 'PointData' 구조체 안에 차트를 그리기 위한 모든 점의 데이터가 저장된다. 구현된 함수 중에서 'GetPointData( )'는 'StockAnalysisDlg.cpp'에서 실행되고, 실제 점들 위치는 'GetDataForChart( )'에서 계산된다.

[코드 2-15] CGraph 구현 (Graph.cpp)

```
    #include "stdafx.h"
    #include "Graph.h"

    //-----------------------------------------------------------------
01  PointData* CGraph::GetPointData(Company* company)
    {
02      static PointData ptData;
03      GetDataForChart(company, &ptData);
```

```
04      return &ptData;
    }

    //---------------------------------------------------------------------
    void CGraph::GetDataForChart(Company* company, PointData* ptData)
    {
05      int i, maxVal, minVal, maxVol, interval;
06      maxVal = 0;
        int quantity = company->quantity;

07      for(i=0; i<quantity; i++)
            if (company->data[i].highVal > maxVal)
                maxVal = company->data[i].highVal;

08      minVal = maxVal;

09      for(i=0; i<quantity; i++)
            if (company->data[i].lowVal < minVal)
                minVal = company->data[i].lowVal;

        maxVol = 0;

        for(i=0; i<quantity; i++)
            if (company->data[i].vol > maxVol)
                maxVol = company->data[i].vol;

10      interval = (int)(WIDTH_GRAPH / quantity);
        int x_start, y_start;

11      x_start = X_START_GRAPH + WIDTH_GRAPH;
12      y_start = Y_START_GRAPH + HEIGHT_GRAPH;

13      ptData->startVal.quantity = company->quantity;
        for(i=0; i<ptData->startVal.quantity; i++) {
14          ptData->startVal.point[i].X = x_start - i * interval;
15          ptData->startVal.point[i].Y = y_start - (HEIGHT_GRAPH * (company-
```

```
>data[i].startVal - minVal)) / (maxVal - minVal);
        }

        ptData->highVal.quantity = company->quantity;
        for(i=0; i<ptData->highVal.quantity; i++) {
            ptData->highVal.point[i].X = x_start - i * interval;
            ptData->highVal.point[i].Y = y_start - (HEIGHT_GRAPH * (company-
>data[i].highVal - minVal)) / (maxVal - minVal);
        }

        ptData->lowVal.quantity = company->quantity;
        for(i=0; i<ptData->lowVal.quantity; i++) {
            ptData->lowVal.point[i].X = x_start - i * interval;
            ptData->lowVal.point[i].Y = y_start - (HEIGHT_GRAPH * (company-
>data[i].lowVal - minVal)) / (maxVal - minVal);
        }

        ptData->lastVal.quantity = company->quantity;
        for(i=0; i<ptData->lastVal.quantity; i++) {
            ptData->lastVal.point[i].X = x_start - i * interval;
            ptData->lastVal.point[i].Y = y_start - (HEIGHT_GRAPH * (company-
>data[i].lastVal - minVal)) / (maxVal - minVal);
        }

    //------------------- Volume Chart --------------------
        y_start = Y_START_GRAPH + HEIGHT_GRAPH + GAP_GRAPH_VOLUME +
HEIGHT_VOLUME;

        ptData->volume.quantity = company->quantity;
        for(i=0; i<ptData->volume.quantity; i++) {
            ptData->volume.point[i].X = x_start - i * interval;
            ptData->volume.point[i].Y = y_start  -  (HEIGHT_VOLUME * company-
>data[i].vol) / maxVol;
        }
```

```
}
```

01 한 종목 데이터인 'Company' 구조체를 매개 변수로 받아서 이 종목의 실제 데이터를 가지고 차트를 그리는 데 필요한 점들의 값을 계산한다. 이 값을 'PointData' 구조체에 저장하고 구조체를 마지막에 반환한다.

02 점들의 데이터를 저장하기 위한 'PointData' 구조체를 정의한다. 'static'을 사용한 것은 'ptData'라는 변수를 이 코드에서만 유일하게 사용하기 위해서다. 반환된 'ptData'는 프로그램이 종료될 때까지 메모리에서 공간을 차지한다.[12] 이후에 'GetPointData()' 함수가 다시 호출되어도 추가 메모리 할당 없이 이미 할당된 메모리의 데이터만 변경한다.

03 차트를 그리기 위한 점들은 'GetDataForChart( )' 함수에서 계산된다. 여기서 주의 깊게 볼 부분은 매개 변수 'company'와 '&ptData'다. 'GetPointData( )' 함수는 결과값을 반환하지만, 'GetDataForChart( )' 함수는 결과값을 매개 변수인 'ptData'에 저장하여 결과를 넘겨준다. 물론, 함수를 통일되게 같은 구조로 만들 수 있지만, 이런 방법으로 구현할 수도 있다. 두 가지 방법 모두 많이 사용되므로 알아두는 것이 좋다.[13]

04 'GetPointData( )'는 'PointData' 구조체의 메모리 주소값을 반환하므로 'ptData' 변수 앞에 '&' 문자를 사용한다. 02에서 'static'으로 선언된 변수 'ptData'는 메모리에 공간이 할당되었으므로 이 메모리의 주소값만 반환한다.

05 실제 데이터의 최대값과 최소값, 그리고 X-축의 간격Interval을 저장하기 위한 변수들을 선언한다.

06 최대값을 계산하기 위해서 제일 낮은 수로 초기화한다.

07 종목 데이터에서 가장 큰 값은 고가High Value이므로 최대값과 고가를 비교하여 최대값보다 고가가 크다면 고가를 최대값으로 가진다.

08 최소값을 계산하기 위해서는 가장 큰 수로 초기화한 후 작은 수를 찾아야 하므로 현재 가장 큰 수인 maxVal로 초기화한다.

09 종목 데이터에서 가장 작은 값은 저가Low Value이므로 최소값과 저가를 비교하여 최소값보다 저가가 작다면 저가를 최소값으로 가진다. 나중에 나오는 거래량Volume의 최대값인 maxVol을 찾는 방법도 이와 동일하다.

10 차트에서 봉 사이의 일정한 간격을 계산하여 'interval' 변수에 저장한다. 간격은 폭Width을 데이터의 개수로 나누면 알 수 있다.

11 증권회사로부터 받는 데이터는 최근 데이터를 먼저 받는다. 즉, 받은 데이터의 앞쪽은 최근 데이터고

12 'static'으로 선언된 변수는 사용을 시작할 때부터 프로그램이 종료될 때까지 데이터를 메모리에 저장하고 있다.

13 'GetDataForChart( )'를 'void' 함수가 아닌 'PointData'를 반환하게 구현할 수도 있다. 이때 'GetDataForChart( )'는 매개 변수로 포인터 변수를 받아서 참조에 의한 호출 방식으로 데이터 결과값을 전달한다.

뒤로 갈수록 오래전에 저장된 값이다. 실제 데이터 배열은 왼쪽이 최근 데이터지만, 차트에서는 오른쪽에 그려지는 그림이 최근 데이터이므로 오른쪽부터 차트를 그려야 한다. 이러한 이유로 그리는 영역의 가장 오른쪽을 시작 위치로 정한다.

12 차트에서 낮은 값은 Y 축의 아래쪽에 그려지고, 값이 클수록 위쪽에 그려진다. 따라서 시작 위치는 그리는 범위의 아래쪽에 나타난다. 주의할 점은 실제 Y 축의 값은 아래쪽이 크다는 것이다.[14]

13 주식 데이터의 개수와 점의 봉 개수는 같으므로 시가, 고가, 저가, 종가, 거래량의 점 개수는 주식 데이터의 개수로 설정한다.

14 X 축의 각 점은 오른쪽부터 차례대로 'interval' 만큼씩 왼쪽으로 옮겨진다.

15 [그림 2-14]의 비율 계산이 여기서 사용된다. 차트를 그리려면 Y 축 점의 위치를 알아야 하는데, 이것은 세 개의 값을 알면 된다. 즉, Y 축의 높이는 'HEIGHT_GRAPH'고, 실제 값의 최대 높이 차이는 ('maxVal' - 'minVal')이며, 최대 높이에 대한 데이터의 위치는 ('startVal' - 'minVal')이다. 점의 위치가 정해지면 시작점인 'y_start'에서 빼준다. 차트에서 아래쪽이 Y 축 값이 크므로 윗쪽에 그려지는 점은 'y_start'에서 점의 위치만큼 빼야 차트에서 점의 실제 위치가 나온다.

차트에 한 종목을 그리기 위해서는 해당 종목의 데이터를 가리키는 포인터 변수를 선언해야 한다. [코드 2-16]에서 'CStock' 클래스에 선언한 'ptrCompany'가 포인터 변수로, 'StockAnalysisDlg.cpp' 파일에서 ptrCompany 변수를 이용하여 한 종목에 대한 차트에 그린다.

[코드 2-16] 한 종목의 차트를 그리기 위한 포인터 변수 'ptrCompany' 선언(Stock.h)

```
class CStock
{
public:
      AllCompany allCompanies;
      SelectedCompany selectedCompanies;

      Company *ptrCompany;

public:
      CStock(void);
```

14 이에 관한 내용은 [그림 2-12]의 설명을 참고하기 바란다.

```
    ~CStock(void);

    void Run();
    void makeSelectedCompanyFromAllCompany();
    void ReadDataFromFile();
    void WriteDataToFile();
};
```

차트에 그리는 작업은 'StockAnalysisDlg.cpp' 파일에서 수행한다. [코드 2-17]에서 StockAnalysisDlg.h에 'flagPaint' 변수와 'DrawGraph( )' 함수를 선언한다. flagPaint 변수가 'true'면 그래프를 그리고, 'false'면 그래프를 그리지 않는다. 즉, flagPaint를 'true'로 선언하고 프로그램 화면을 새롭게 그리는 'RedrawWindow( )' 함수를 선언한 후 'DrawGraph( )' 함수를 실행하면 차트를 그린다.

[코드 2-17] 화면에 차트를 그리는 함수 'DrawGraph( )' 선언(StockAnalysisDlg.h)

```
class CStockAnalysisDlg : public CDialog
{
private:
    CStock *stock;
    bool flagPaint;

public:
    CComboBox m_jongmok;

##### 생략 #####

public:
    void DrawGraph();
    afx_msg void OnBnClickedBtnRun();
    afx_msg void OnCbnSelchangeComboJongmok();
```

```
};
```

[코드 2-18]은 'DrawGraph()'를 실행하기 위한 준비 작업이라고 볼 수 있다. 프로그램이 실행되면 가장 먼저 수행되는 'OnInitDialog()' 함수와 프로그램 화면을 새로 그리는 'OnPaint()' 함수의 추가 항목들을 보여준다. OnInitDialog()와 OnPaint() 함수는 비주얼 스튜디오에서 자동으로 생성된다. [코드 2-18]에는 DrawGraph() 함수를 제외한 나머지 추가 부분들을 구현한다.

[코드 2-18] 차트를 그리기 위한 한 종목 선택하기(StockAnalysisDlg.cpp)

```
01  BOOL CStockAnalysisDlg::OnInitDialog()          //---
    {
        ##### 생략 #####
        stock = new CStock();
02      flagPaint = false;

        return TRUE;
    }

    ##### 생략 #####

    //-------------------------------------------------------------------
03  void CStockAnalysisDlg::OnPaint()
    {
04      if (flagPaint) {
            DrawGraph();
        }
    ##### 생략 #####
    }

    ##### 생략 #####

    //-------------------------------------------------------------------
```

```
    void CStockAnalysisDlg::OnBnClickedBtnRun()
    {
        int i;

        stock->Run();
        m_jongmok.ResetContent();

        for (i=0; i<stock->selectedCcompanies.quantity; i++) {
            m_jongmok.AddString(stock->selectedCcompanies.companies[i]-
>strName);
        }

        m_jongmok.SetCurSel(0);
        UpdateData(FALSE);

05      if (i == 0)
            return;

06      stock->ptrCompany = stock->selectedCompanies.companies[0];
07      flagPaint = true;

        RedrawWindow();
    }

    //-------------------------------------------------------------------
08  void CStockAnalysisDlg::OnCbnSelchangeComboJongmok()
    {
09      int index = m_jongmok.GetCurSel();
10      stock->ptrCompany = stock->selectedCompanies.companies[index];
        RedrawWindow();
    }
```

---

01 프로그램이 실행될 때 시작되는 함수로, 함수 이름에 'Init'이라고 표현된 것처럼 초기화를 수행하는 함수다. 함수가 실행될 때 수행하고자 하는 코드들은 여기에 작성한다.

02 처음에는 차트를 그릴 필요가 없으므로 'flagPaint'는 'false'로 초기화한다.

03 화면이 새로 그려질 때마다 실행되는 함수로, 프로그램 안에서 'OnPaint( )' 함수를 강제로 실행하려면 'RedrawWindow( )' 함수를 호출한다.

04 'flagPaint' 변수가 'true'일 때에만 화면에 차트를 그리는 'DrawGraph( )' 함수를 실행한다. 프로그램이 실행되면 'OnPaint( )'함수가 실행되므로 처음에는 차트를 그리지 않기 위해서 'flagPaint' 변수를 사용한다.

05 'Run' 버튼을 누르면 'OnBnClickedBtnRun( )' 함수가 실행된다. 이때 변수 'i'가 0이라는 것은 선택된 종목의 개수가 하나도 없다는 것을 의미하므로 차트를 그리지 않고 함수를 빠져나간다.

06 콤보 박스에 종목이 있으면 목록의 가장 위에 있는 0번째 종목을 그리기 위해서 'ptrCompany'가 첫 번째 종목을 가리키게 설정한다.

07 차트를 그리기 위해서 'flagPaint'를 'true'로 설정하고 'RedrawWindow( )'를 호출하여 'OnPaint( )' 함수를 실행한다. 이때 'OnPaint( )' 함수 안에 있는 'DrawGraph( )' 함수가 실행되어 차트가 그려진다.

08 'OnCbnSelchangeComboJongmok( )' 함수는 콤보 박스 안에 여러 종목이 나열되어 있을 때 한 종목을 선택하면 실행된다.

09 콤보 박스에서 현재 선택된 종목의 'index'를 가져온다. 함수 이름에서 그 함수의 특성을 파악할 수 있는데, 'GetCurSel( )'이라는 이름에서 'Get Current Selected'를 유추할 수 있다.

10 06처럼 'ptrCompany' 변수가 현재 'index'의 종목을 가리키게 선언한다.

마지막으로 [코드 2-19]와 같이 차트를 그리는 함수 'DrawGraph( )'를 구현한다. MFC에서 차트를 그리려면 첫 번째는 사각형을 그리고, 두 번째는 선을 그리고, 마지막으로 화면에 글씨를 써준다

그림은 디바이스 컨텍스트DC, Device Context를 가져와서 그리는데, 여기서는 'CPaintDC' 클래스의 함수를 사용한다. 예를 들어, 색이 채워진 사각형을 그리려면 'FillRect( )' 함수를 사용하고, 선을 그리려면 'MoveTo( )'와 'LineTo( )' 함수를, 글자를 쓰려면 'TextOut( )' 함수를 사용한다.

이 프로그램에서 색은 'RGB(Red, Green, Blue)'로 나타낸다. [코드 2-19]에서 RGB(255,0,0)는 빨간색으로, Red가 255이고 Green과 Blue가 각각 0이라는 것을 나타낸다. 각 색에서 값의 범위는 0~255이다. 이것은 한 가지 색이 1바이트로 저장되고, 1바이트는 8비트이기 때문에 2^8=256이 되어 0~255 사이의 값으

로 표현된다. 모든 색이 최고값을 갖는 RGB(255,255,255)는 세 가지 색이 모두 섞여서 하얀색White이 되고, 모든 색이 하나도 섞이지 않은 RGB(0,0,0)은 검은색이 된다.[15] 여기서는 하얀색: RGB(255,255,255), 검은색: RGB(0,0,0), 빨간색: RGB(255,0,0), 파란색: RGB(0,0,255) 이렇게 네 가지 색을 사용한다.

[코드 2-19] 화면에 차트를 그리는 'DrawGraph( )' 함수(StockAnalysisDlg.cpp)

```
    void CStockAnalysisDlg::DrawGraph()
    {
        int i;
01      PointData *ptData= CGraph::GetPointData(stock->ptrCompany);
02      CPaintDC dc(this);
03      CBrush whiteBrush(RGB(255,255,255));
04      RECT rect = {X_START_GRAPH, Y_START_GRAPH, X_START_GRAPH+WIDTH_GRAPH,
    Y_START_GRAPH+HEIGHT_GRAPH};
05      dc.FillRect(&rect, &whiteBrush);
06      CPen bluePen, redPen, bluePenThick, redPenThick;
07      bluePen.CreatePen(PS_SOLID, WIDTH_LINE, RGB(0,0,255));
        redPen.CreatePen(PS_SOLID, WIDTH_LINE, RGB(255,0,0));
        bluePenThick.CreatePen(PS_SOLID, WIDTH_THICK_LINE, RGB(0,0,255));
        redPenThick.CreatePen(PS_SOLID, WIDTH_THICK_LINE, RGB(255,0,0));
08      for(i=0; i<stock->ptrCompany->quantity; i++) {
09          if (ptData->startVal.point[i].Y > ptData->lastVal.point[i].Y) {
                dc.SelectObject(redPen);
            }
            else {
                dc.SelectObject(bluePen);
            }
```

15 참고로 컴퓨터 모니터에 선이 연결되지 않으면 빛 데이터를 받지 못해서 검은색이 된다.

```
10          dc.MoveTo(ptData->highVal.point[i].X, ptData->highVal.point[i].Y);
11          dc.LineTo(ptData->lowVal.point[i].X, ptData->lowVal.point[i].Y);

12          if (ptData->startVal.point[i].Y > ptData->lastVal.point[i].Y) {
                dc.SelectObject(redPenThick);
            }
            else {
                dc.SelectObject(bluePenThick);
            }
            dc.MoveTo(ptData->startVal.point[i].X, ptData->startVal.point[i].Y);
            dc.LineTo(ptData->lastVal.point[i].X, ptData->lastVal.point[i].Y);
        }

        //------------------------- Volume Chart -------------------------
        int yOriginVolume = Y_START_GRAPH + HEIGHT_GRAPH + GAP_GRAPH_VOLUME +
    HEIGHT_VOLUME;

        RECT rect2 = {X_START_GRAPH, Y_START_GRAPH+HEIGHT_GRAPH+GAP_GRAPH_VOLUME,
    X_START_GRAPH+WIDTH_GRAPH, yOriginVolume};
13      dc.FillRect(&rect2, &whiteBrush);

        for(i=0; i<stock->ptrCompany->quantity-1; i++) {
            if (ptData->volume.point[i].Y < ptData->volume.point[i+1].Y) {
                dc.SelectObject(redPenThick);
            }
            else {
                dc.SelectObject(bluePenThick);
            }
            dc.MoveTo(ptData->volume.point[i].X, yOriginVolume);
            dc.LineTo(ptData->volume.point[i].X, ptData->volume.point[i].Y);

        }

        //------------------------ Write Text on DC ----------------------
        int maxVal, minVal, maxVol;
```

```
        maxVal = 0;

        int quantity = stock->ptrCompany->quantity;

14      for(i=0; i<quantity; i++)
            if (stock->ptrCompany->data[i].highVal > maxVal)
                maxVal = stock->ptrCompany->data[i].highVal;

        minVal = maxVal;

15      for(i=0; i<quantity; i++)
            if (stock->ptrCompany->data[i].lowVal < minVal)
                minVal = stock->ptrCompany->data[i].lowVal;

        maxVol = 0;

16      for(i=0; i<quantity; i++)
            if (stock->ptrCompany->data[i].vol > maxVol)
                maxVol = stock->ptrCompany->data[i].vol;

        CString str;
        str.Format("- %d원", maxVal);
17      dc.TextOut(WIDTH_GRAPH+13, 5, str, str.GetLength());

        str.Format("- %d원", minVal);
        dc.TextOut(WIDTH_GRAPH+13, HEIGHT_GRAPH, str, str.GetLength());

        str.Format("- %d", maxVol);
        dc.TextOut(WIDTH_GRAPH+13, HEIGHT_GRAPH+GAP_GRAPH_VOLUME, str,
    str.GetLength());
    }
```

---

01 종목 'ptrCompany'의 점 데이터 'ptData'를 얻는다. 'CGraph' 클래스의 'GetPointData( )' 함수가 사용된다.

02 'CPaintDC'를 가져온다. 차트를 그리기 위해 'CPaintDC' 클래스의 변수인 'dc'의 함수들을 사용한다.

03 'Pen'은 선을 그리기 위해 사용되고 'Brush'는 색을 칠하기 위해서 사용한다. 여기서는 사각형 안에 색을 칠하기 위해서 하얀색 'Brush'를 선언한다.

04 사각형의 위치를 선언한다. 사각형은 네 개의 값으로 그릴 수 있는데, 왼쪽 위 시작점의 X, Y 값과 오른쪽 아래 끝점의 X, Y 값을 정하면 된다.

05 'dc'의 'FillRect( )' 함수를 사용하여 하얀색 사각형을 그린다. 함수에는 사각형의 네 점과 'Brush'의 정보를 넣어준다. 여기서 그리는 사각형은 'Chart'의 봉이 그려지는 영역을 하얀색으로 만들기 위해서다.

06 선을 그리기 위해 'CPen' 클래스를 사용하고, 앞으로 사용할 색이 있는 펜들을 선언한다.

07 'CPen' 클래스의 'CreatePen( )' 함수를 사용하여 사용할 펜을 생성한다. 매개 변수로 선의 종류, 굵기, 색을 정해준다. 'PS_SOLID'는 실선이고, 'WIDTH_LINE'은 굵기로 'CGraph.h'에서 '1'로 정의된다. 색은 RGB로 정한다.

08 봉을 그린다. 종목의 데이터 개수만큼 반복하여 선을 하나씩 그린다.

09 시가보다 종가가 높으면 'redPen'을 사용하고, 시가보다 종가가 낮으면 'bluePen'을 사용한다. 시가보다 종가가 높은 것은 시가의 Y 값이 종가의 Y 값보다 크기 때문이다. 화면에서 아래쪽인 Y 값이 큰 값이다. 'dc'의 'SelectObject( )' 함수를 사용하여 'dc'의 펜을 정해준다. 여기서는 봉을 가늘고 긴 선으로 그린다.

10 'dc'의 'MoveTo( )' 함수를 사용하여 선의 시작점으로 펜의 위치를 옮긴다. 매개 변수는 시작점의 X, Y 값이다.

11 'MoveTo( )' 함수로 옮겨진 점부터 선을 그린다. 'LineTo( )' 함수는 선을 그리는 함수로, 매개 변수는 선 끝점의 X, Y 값이다.

12 봉에서 짧고 굵은 선을 그리기 위해 시가와 종가를 비교하여 두꺼운 펜의 색을 정한다.

13 거래량이 그려지는 프로그램의 아랫부분을 하얀색 사각형으로 그린다.

14 차트에 데이터의 최대값을 쓰기 위해 최대값을 고가로 알아낸다.

15 차트에 데이터의 최소값을 쓰기 위해 최소값을 저가로 알아낸다.

16 거래장의 최대값을 쓰기 위해 최대값을 거래량Volume으로 알아낸다.

17 글자를 화면에 쓰기 위해 'dc'의 'TextOut( )' 함수를 사용한다. 매개 변수로 글자가 출력될 위치의 X, Y 값과 글자인 'CString' 변수, 글자의 길이 정보를 넣어준다. 'GetLength( )' 함수는 글자의 길이를 가져온다.

MFC에서 그림을 그리기 위한 함수들은 사용법이 정해져 있다. 필요한 함수가 무엇인지를 찾아서 사용법을 아는 것이 좋다, 자주 사용하는 것을 자연스럽게 머릿속에 기억될 것이다. 프로그램을 만들면서 이미 구현된 내용을 사용해야 한다면, 인터넷을 통하여 정보를 얻기 바란다. 이 책에서는 많은 것을 다루지는 않지만, 직접

인터넷으로 필요한 정보를 얻어서 더 많은 것을 구현해 보기 바란다.

지금까지 주식 프로그램을 만들기 위한 준비 작업을 했다. MFC 프로그램에 대해서 배웠고, 차트를 그려 보고 해당 기능들을 다루어 보았다. MFC가 익숙하지 않다면, 지금까지 내용으로도 웬만한 MFC 프로그램은 혼자서 만들 수 있을 것이다.

다음 장에서는 주식 프로그램에서 차트를 분석하는 데 사용되는 이평선에 대하여 알아본다. 주식을 하는 사람 대부분이 보는 이평선에 대해서 기본적인 것을 익히고 프로그램을 어떻게 구현해야 하는지 알게 될 것이다.

3 | 이평선

주식 시장Stock Market은 주식을 상징하는 동물인 곰과 황소의 싸움으로 표현된다. 양손으로 내리치는 곰과 뿔로 들이받아서 쳐 올리는 황소의 싸움이 벌어지는데, 곰이 이기면 하락장이고 황소가 이기면 상승장이다. 그래서 뉴욕 월가Wall Street의 상징이 황소 동상이다.

주식 차트는 마치 곰과 황소의 격렬한 싸움처럼 항상 주가가 요동치면서 움직이므로 미래를 예측하기가 힘들다. 주식 매매를 하는 사람들은 순간의 움직임이 아닌 전체 장의 흐름을 보기 위해서 이평선을 이용한다. 이평선은 '이동평균선Moving Average Line'을 줄인 말로, 주식 매매할 때 참고하는 보조 지표다. 꺽인 선들의 잔가지들을 없애고 차트를 보려고 할 때 이평선을 이용한다. 주식의 기본이 이평선이라고 이해해도 될 만큼 주식 매매를 하는 모든 사람은 이평선을 가지고 주가를 분석한다. 이평선은 수시로 변하는 주가를 평균Average을 이용해서 완만한 선으로 만들어 현재의 추세를 분석하기 위해 사용된다. 주가의 변화는 바둑에서 경우의 수보다 훨씬 많다. 주식을 하는 사람들은 패턴Pattern이 없는 주식 시장에서 패턴을 찾기 위해서 여러 가지 보조 지표들을 만들어 사용하며, 이 중에서 가장 간단한 보조 지표가 이평선이다.

이평선은 평균을 만드는 개수에 따라 5평선, 20평선, 60평선, 120평선 등으로 구분한다. 5평선은 5개의 데이터로 평균을 만들고, 20평선은 20개의 데이터로 평균을 만든다. 평균을 만드는 개수가 많을수록 완만한 선이 되는데, 이것은 하나의 값이 평균에서 차지하는 비중이 더 작아지기 때문이다.

3.1 이평선 분석

이평선의 분석은 이평선의 원리만큼이나 단순한 편이다. 5평선이 20평선 위에 있는지, 아니면 아래에 있는지에 따라 분석이 달라진다. 또한, 5평선이 20평선을 넘어가는 순간을 '골든 크로스Golden Cross'라고 해서 올라가는 상승장이라 판단하여 중요하게 생각한다. 반대로 5평선이 20평선 아래로 떨어지는 순간을 '데드 크로스Dead Cross'라고 해서 장이 무너진다고 판단하기도 한다. 물론 이러한 개념은 5평선과 60평선으로도 비교할 수 있고, 평균을 만드는 데이터의 개수도 개인이 정해서 자신만의 이평선을 만들 수 있다. 일반적으로 보는 20평선 대신에 30평선 또는 50평선을 만들 수 있고, 현재의 주식 값과 이평선을 비교하여 분석할 수도 있다.

이평선은 지지선으로도 많이 분석한다. 주가가 이평선을 통과하기가 힘에 부치는 경우가 종종 있는데, 이것은 주가가 모든 사람의 마음이 반영되어 움직이기 때문이다. 즉, 모든 사람이 이평선을 보고 있으며, 이평선을 통과하기를 바라는 사람들과 이평선을 통과하지 못하리라 판단하는 사람들에 의해서 주가가 움직이기 때문이다.

시스템 트레이딩System Trading은 분봉으로 하는 것이 좋다. 필자가 분봉 데이터를 받아본 결과 많은 종목에서 분마다 매매가 이루어지지 않았으며 매매가 이루어졌어도 매매 가격의 변동이 없는 경우가 많았다. 그래서 이 책에서는 일봉을 가지고 검증을 한다. 실제 시스템 트레이딩은 주식 시장보다는 파생상품인 옵션Option에서 많이 하고 있으며, 필자도 그것이 맞다고 생각한다. 그러나 독자들은 옵션을 하지 않기를 권한다. 도박의 끝이 경마이듯 주식의 끝은 옵션이며 옵션에서 많은 사람이 실패를 맛본다.

이 책은 주식을 예제로 한 프로그램 책이므로 독자들이 누군가를 위해서 프로그램을 만들어 줄 수 있는 안내서이며, 현재 주식을 하는 사람 중에 프로그램 분석을 원하는 사람들을 위한 책이다. 앞으로 전개되는 주식 분석은 모든 종목의 일봉 데이터로 검증하며 특정 종목에 종속되지는 않았지만, 이 책을 이해한 독자들은 개인적으

로 프로그램의 추가 구현을 통해서 자신만의 알고리즘을 적용할 수 있을 것이다.

이 책에서 만드는 프로그램은 매매 시 증권사의 수수료 0.015%를 차감하며, 매도 시 추가로 증권거래세 0.3%를 차감한다. 또한, 매수 시에는 현재 가격보다 0.5% 높은 가격으로, 매도 시에는 현재 가격보다 0.5% 낮은 가격으로 거래한다. 이처럼 하는 이유는 실전과 좀 더 유사하게 매매를 하기 위해서다. 실제 시스템 트레이딩은 매매가 결정되면 해당 물량의 거래를 무조건 체결한다. 예를 들어, 팔아야 한다면 아래의 가격이 어떻게 되든 낮은 가격으로 던진다. 독자들은 이 책에서 시스템 트레이딩을 프로그램으로 어떻게 접근하는지에 중점을 두기 바란다.

3.2 이평선 구현

이평선은 증권 프로그램에서 일정 기간의 주가들의 평균값이다. 평균값을 구하는 식은 다음과 같다. 3평선은 3일치 주가를 더해서 3으로 나눈 것이고, 5평선은 5일치 주가를 더해서 5로, 10평선은 10일치 주가를 더해서 10으로 나눈다.

```
3평선      x = (a1 + a2 + a3) / 3
5평선      x = (a1 + a2 + a3 + a4 + a5) / 5
10평선     x = (a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9 + a10) / 10
```

주식 차트에서 이평선은 이처럼 평균값을 만들어서 그래프를 그려준다. 현재 값(주가)을 기준으로 이전 값들의 평균을 계산해서 값을 얻는다. 이 식에서 볼 수 있듯이 하나의 값(a1)은 3평선에서 x의 값에 1/3(33%) 정도의 비중을 차지하고, 5평선에서는 1/5(20%)의 영향을 주며, 10평선에서는 1/10(10%)을 차지한다. 즉, 이평선에서 평균을 만드는 값이 많이 포함될수록 현재 값이 이평선에 주는 영향은 점점 더 적어진다. 이평선에서 한 값(a1)의 비중이 큰 것은 현재 값이 이평선에 주는 영향이 큰 것이므로 3평선이 현재 추세를 가장 강하게 반영한다고 볼 수 있다. 따라

서 '골든 크로스'라고 부를 수 있는 3평선이 10평선을 돌파하여 위로 올라가면 장에서 상승하려는 힘이 크다고 분석한다. [그림 3-1]은 이번 장에서 구현하는 이평선을 프로그램에서 차트로 그린 화면이다.

[그림 3-1] 이평선

[코드 3-1]은 이평선의 구조체인 'MovementAverage' 구조체를 정의한 것으로, 주식 차트에서 주로 보는 5평선, 20평선, 60평선, 120평선을 구조체 안에 정의한다. 이평선 구조체는 한 종목인 'Company' 구조체에서 선언하여 사용한다.

[코드 3-1] 이평선 구조체(Stock.h)

```
struct MovementAverage {
      long avg5;            // 5 평선
      long avg20;           // 20 평선
      long avg60;           // 60 평선
      long avg120;          // 120 평선
};

struct Company {
```

```
    CString strJongMok, strName;      //종목 번호, 종목 명
    int quantity;                     //데이터 개수
    Data data[MAX_DATA];              //주가 데이터
    MovementAverage moveAverage[MAX_DATA]; //이평선 데이터
};
```

[코드 3-2]는 주식 데이터에서 종가로 이평선을 계산한다. 이 코드에서 5평선을 만드는 방법을 이해하면 나머지도 같은 방법으로 연산이 이루어지므로 쉽게 이해할 수 있다.

[코드 3-2] 이평선 계산(Stock.cpp)

```
    void CStock::makeMovementAverage()
    {
        int i, j, k;
        long sum;

        for(i=0; i<allCompanies.quantity; i++) {

01          Company *company = &allCompanies.companies[i];
            int quantity = company->quantity;

02          for(j=0; j<=quantity-5; j++) {
                sum = 0;
                for(k=j; k<j+5; k++)
03                  sum += company->data[k].lastVal;
04              company->moveAverage[j].avg5 = (long)(sum / 5);
            }

            for(j=0; j<=quantity-20; j++) {
                sum = 0;
                for(k=j; k<j+20; k++)
                    sum += company->data[k].lastVal;
```

```
                company->moveAverage[j].avg20 = (long)(sum / 20);
            }

            for(j=0; j<=quantity-60; j++) {
                sum = 0;
                for(k=j; k<j+60; k++)
                    sum += company->data[k].lastVal;
                company->moveAverage[j].avg60 = (long)(sum / 60);
            }

            for(j=0; j<=quantity-120; j++) {
                sum = 0;
                for(k=j; k<j+120; k++)
                    sum += company->data[k].lastVal;
                company->moveAverage[j].avg120 = (long)(sum / 120);
        }
    }
}
```

---

01 종목 중 하나를 포인터 변수인 'company'로 받아서 사용한다. 이때 해당 종목의 주소를 포인터 변수에 넣기 위해 '&' 문자를 사용한다.

02 5평선은 해당 종목 데이터의 마지막 4개에는 데이터를 넣지 않는다.[01] 데이터는 0번째가 최근 데이터이며 'j' 값이 커질수록 이전 데이터가 저장되어 있다.

03 'k'는 현재 값과 이전 4개 데이터의 값을 차례대로 'sum' 변수에 더한다. 이평선은 종가를 사용하여 만들어지므로 'lastVal' 값을 사용한다.

04 5개의 데이터는 'sum' 변수에 더해지고, 5로 나누어 평균값을 만든다.

[코드 3-3]은 이평선의 데이터를 이평선의 점 데이터로 변환하기 위해 Graph.h의 'PointData' 구조체에 변수들을 정의한다.

---

01 전체 데이터 중 가장 오래된 4개의 데이터는 5평선을 만들 때 계산을 위해서만 사용되므로 5평선의 개수는 전체 데이터 개수에서 4를 뺀다.

[코드 3-3] 화면에 그리기 위한 이평선 점 정의(Graph.h)

```
struct PointData {
      PointArray startVal;        //시가
      PointArray highVal;         //고가
      PointArray lowVal;          //저가
      PointArray lastVal;         //종가
      PointArray volume;          //거래량

      PointArray avg5;            //5평선
      PointArray avg20;           //20평선
      PointArray avg60;           //60평선
      PointArray avg120;          //120평선
};
```

주가 데이터를 화면에 그리려면 항상 점 데이터를 만들고 화면의 위치 값을 계산해야 한다. [코드 3-4]에서는 화면에 선을 그리기 위한 점의 위치를 계산한다.

[코드 3-4] 화면에 그리기 위한 이평선 점 계산(Graph.cpp)

```
      void CGraph::GetDataForChart(Company* company, PointData* ptData)
      {
      ##### 생략 #####

         //------------------- Movement Average --------------------
01       ptData->avg5.quantity = company->quantity - 5;
         for(i=0; i<ptData->avg5.quantity; i++) {
02          ptData->avg5.point[i].X = x_start - i * interval;
03          ptData->avg5.point[i].Y = y_start - (HEIGHT_GRAPH * (company-
      >moveAverage[i].avg5 - minVal)) / (maxVal - minVal);
         }

         ptData->avg20.quantity = company->quantity - 20;
```

```
    for(i=0; i<ptData->avg20.quantity; i++) {
        ptData->avg20.point[i].X = x_start - i * interval;
        ptData->avg20.point[i].Y = y_start - (HEIGHT_GRAPH * (company-
>moveAverage[i].avg20 - minVal)) / (maxVal - minVal);
    }

    ptData->avg60.quantity = company->quantity - 60;
    for(i=0; i<ptData->avg60.quantity; i++) {
        ptData->avg60.point[i].X = x_start - i * interval;
        ptData->avg60.point[i].Y = y_start - (HEIGHT_GRAPH * (company-
>moveAverage[i].avg60 - minVal)) / (maxVal - minVal);
    }

    ptData->avg120.quantity = company->quantity - 120;
    for(i=0; i<ptData->avg120.quantity; i++) {
        ptData->avg120.point[i].X = x_start - i * interval;
        ptData->avg120.point[i].Y = y_start - (HEIGHT_GRAPH * (company-
>moveAverage[i].avg120 - minVal)) / (maxVal - minVal);
    }

##### 생략 #####
}
```

---

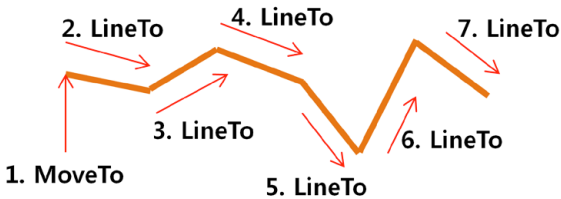
01 5평선의 개수는 전체 데이터 개수에서 4를 뺀다. 그러나 여기서는 5평선이라는 것을 강조하기 위해 4 대신 5를 뺀다.

02 5평선의 X 값은 주식 데이터의 X 값과 동일하다.

03 [코드 2-15]의 **15**와 같이 '[그림 2-14] 비율 계산'을 이용하여 5평선의 Y 값을 계산한다.

지금까지 만들어진 이평선의 데이터를 화면에 그리려면 StockAnalysisDlg.cpp 파일에 추가로 코드를 구현해야 한다. 이평선에서 점들을 연결하여 선을 그릴 때는 MFC의 MoveTo( )와 LineTo( ) 함수를 사용한다. [그림 3-2]와 같이 MoveTo( ) 함수를 사용하여 처음 시작점의 위치를 정해주고, LineTo( ) 함수를 반복해서 실행하면 연결된 선이 만들어진다.

[그림 3-2] Line 그리기



[코드 3-5]는 색을 정하고 MoveTo( )와 LineTo( ) 함수를 사용하여 선을 그려주는 코드이다. 5평선을 어떻게 그리는지 이해하면 나머지는 같은 방법을 사용하므로 쉽게 구현할 수 있다.

[코드 3-5] 화면에 이평선 그리기(StockAnalysisDlg.cpp)

```
void CStockAnalysisDlg::DrawGraph()
{
##### 생략 #####

    //----------------------- Movement Average -----------------------
    CPen avg5Pen, avg20Pen, avg60Pen, avg120Pen;

01  avg5Pen.CreatePen(PS_SOLID, 1, RGB(0,0,0));
    avg20Pen.CreatePen(PS_SOLID, 1, RGB(0,255,0));
    avg60Pen.CreatePen(PS_SOLID, 1, RGB(255,255,0));
    avg120Pen.CreatePen(PS_SOLID, 1, RGB(0,255,255));

02  dc.SelectObject(avg5Pen);
03  dc.MoveTo(ptData->avg5.point[0].X, ptData->avg5.point[0].Y);
    for(i=1; i<ptData->avg5.quantity; i++) {
04      dc.LineTo(ptData->avg5.point[i].X, ptData->avg5.point[i].Y);
    }

    dc.SelectObject(avg20Pen);
```

```
        dc.MoveTo(ptData->avg20.point[0].X, ptData->avg20.point[0].Y);
        for(i=1; i<ptData->avg20.quantity; i++) {
            dc.LineTo(ptData->avg20.point[i].X, ptData->avg20.point[i].Y);
        }

        dc.SelectObject(avg60Pen);
        dc.MoveTo(ptData->avg60.point[0].X, ptData->avg60.point[0].Y);
        for(i=1; i<ptData->avg60.quantity; i++) {
            dc.LineTo(ptData->avg60.point[i].X, ptData->avg60.point[i].Y);
        }

        dc.SelectObject(avg120Pen);
        dc.MoveTo(ptData->avg120.point[0].X, ptData->avg120.point[0].Y);
        for(i=1; i<ptData->avg120.quantity; i++) {
            dc.LineTo(ptData->avg120.point[i].X, ptData->avg120.point[i].Y);
        }

    ##### 생략 #####
    }
```

---

01 선을 정의한다. 'RGB(0,0,0)'은 검은색, 'PS_SOLID'는 실선을 의미한다. 즉, 5평선을 그리는 색연필인 'avg5Pen'은 굵기가 1인 검은색 실선이다.

02 'dc'에서 사용하기 위해 생성된 'avg5Pen'을 5평선을 그리기 바로 전에 선택한다.

03 5평선의 첫 번째 시작점으로 'MoveTo( )' 함수를 사용하여 위치를 옮긴다.

04 나머지 점들을 for 문에서 'LineTo( )' 함수를 사용하여 반복적으로 선을 그린다.

지금까지 이평선을 화면에 그리는 것을 구현하였다. 실제 주식 프로그램은 구현할 내용이 많고 복잡하지만, 쉽게 알 수 있도록 최대한 간략히 구현하였다. 코드가 다소 복잡하게 느껴질 수 있지만, 내려받은 소스 코드를 먼저 실행해 본 후 설명을 읽으면서 코드 분석을 해보기 바란다. 물론, 설명을 보지 않고 코드를 이해할 수 있다면 더 좋다.

3.3 골든 크로스 검증

5평선이 20평선을 돌파하여 위로 상승하는 것을 '골든 크로스'라고 한다. 이것은 상승하려는 힘이 현재 강해서 상승장을 이끈다고 분석한다. 반대로 5평선이 20평선을 돌파하여 아래로 하강하는 것을 '데드 크로스'라고 하며, 하락장이 강하다고 분석한다. 그렇다면 골든 크로스 이후에 정말로 주가는 계속 상승할까? 프로그램은 2,000개가 넘는 모든 종목에서 이러한 데이터를 단 몇 초 만에 계산하여 통계를 얻으므로 아주 요긴하게 사용될 수 있다.

이번에는 골든 크로스가 발생한 이후에 일정 기간이 지난 뒤 주식의 현재 종가가 올라가 있는지를 통계로 알아보자. 골든 크로스는 5평선이 20평선을 돌파했을 때와 5평선이 60평선을 돌파했을 때를 기준으로 하며, 1개월, 3개월, 6개월 이후에 주식 종가가 올랐는지를 확인한다. 실제 주식 프로그램은 이전의 주식 데이터를 가지고 시뮬레이션Simulation을 할 수 있어서 자신의 알고리즘을 검증하기에 좋다. 알고리즘이 간단하다면 여기서 설명하는 프로그램을 응용하여 자신의 알고리즘을 확인할 수 있다.

이번에 구현하는 코드는 생각보다 간단하다. 주식 데이터로 이평선을 만들어 놓은 상태이므로 이평선 값을 비교하여 결과를 도출한다. 골든 크로스는 바로 전 데이터에서 5평선이 20평선(또는 60평선) 아래에 있고, 현재 데이터에서 5평선이 20평선(또는 60평선) 위에 존재하는 것이 조건이다. 이 조건에 해당될 때 20일(5일 × 4주) 뒤 종가가 골든 크로스 때 종가보다 높으면 상승한 것이고, 낮다면 하락한 것으로 판단한다. 이와 같은 방법으로 3개월은 60일(5일 × 4주 × 3개월), 6개월은 120일(5일 × 4주 × 6개월)로 계산한다. 여기서는 일Day 데이터를 사용하므로 이와 같은 계산이 필요하다.

이번 장에서 구현하는 것은 골든 크로스를 분석하여 간단한 통계를 메시지 창에 출력하는 코드다. 이를 위해 골든 크로스를 분석하는 'analyzeGoldCross( )' 함수와

개별 종목에서 골든 크로스를 계산하기 위한 'analyzeCompany()' 함수를 추가한다.

[코드 3-6]은 통계를 위해서 골든 크로스 이후 종가를 저장하는 'DataGoldCross' 구조체로, 골든 크로스 때 종가를 저장하고 1개월, 3개월, 6개월 뒤의 종가도 저장한다. 골든 크로스 때 종가보다 1개월 뒤의 종가가 더 높은 종목은 상승 종목이라 판단하고 개수를 하나씩 증가시킨다. 이러한 방법으로 골든 크로스 이후에 상승한 종목의 개수와 하락한 종목의 개수를 알 수 있다.

[코드 3-6] 골든 크로스 이후 종가를 저장하는 구조체(Stock.h)

```
struct DataGoldCross {
      long valueAtGoldCross20;          //20평선과 골든 크로스 때 종가
      long valueAfter1Month20;          //20평선과 골든 크로스 1달 뒤 종가
      long valueAfter3Month20;          //20평선과 골든 크로스 3달 뒤 종가
      long valueAfter6Month20;          //20평선과 골든 크로스 6달 뒤 종가
      long valueAtGoldCross60;          //60평선과 골든 크로스 때 종가
      long valueAfter1Month60;          //60평선과 골든 크로스 1달 뒤 종가
      long valueAfter3Month60;          //60평선과 골든 크로스 3달 뒤 종가
      long valueAfter6Month60;          //60평선과 골든 크로스 6달 뒤 종가
};
```

[코드 3-7]은 모든 종목에서 골든 크로스가 발생한 종목을 찾아서 1개월, 3개월, 6개월 뒤 종가를 'DataGoldCross' 구조체에 저장한 후, 메시지 창에 통계를 출력한다. 코드 라인 수가 많지만, 출력할 데이터가 여러 개여서 반복되는 것일 뿐 내용은 간단하다.

[코드 3-7] 5평선의 골든 크로스 통계 만들기(Stock.cpp)

```
      void CStock::Run()
```

```
    {
        ReadDataFromFile();
        makeSelectedCompanyFromAllCompany();
        makeMovementAverage();

01      analyzeGoldCross();
    }

    //-------------------------------------------------------------------------
    void CStock::analyzeGoldCross()
    {
        int i;
02      DataGoldCross data[MAX_COMPANY];

        for(i=0; i<allCompanies.quantity; i++) {
03          Company *company = &allCompanies.companies[i];

04          analyzeCompany(company, &data[i]);
        }

05      int countWin1Month20 = 0; //20평선 골든 크로스 1개월 뒤 상승한 종목 개수
        int countWin3Month20 = 0;
        int countWin6Month20 = 0;
        int countWin1Month60 = 0; //60평선 골든 크로스 1개월 뒤 상승한 종목 개수
        int countWin3Month60 = 0;
        int countWin6Month60 = 0;
        int countLose1Month20 = 0; //20평선 골든 크로스 1개월 뒤 하락한 종목 개수
        int countLose3Month20 = 0;
        int countLose6Month20 = 0;
        int countLose1Month60 = 0; //60평선 골든 크로스 1개월 뒤 하락한 종목 개수
        int countLose3Month60 = 0;
        int countLose6Month60 = 0;

06      for(i=0; i<allCompanies.quantity; i++) {
            if(data[i].valueAfter1Month20 > data[i].valueAtGoldCross20)
```

```
{countWin1Month20 += 1;}
        if(data[i].valueAfter3Month20 > data[i].valueAtGoldCross20)
{countWin3Month20 += 1;}
        if(data[i].valueAfter6Month20 > data[i].valueAtGoldCross20)
{countWin6Month20 += 1;}
        if(data[i].valueAfter1Month60 > data[i].valueAtGoldCross60)
{countWin1Month60 += 1;}
        if(data[i].valueAfter3Month60 > data[i].valueAtGoldCross60)
{countWin3Month60 += 1;}
        if(data[i].valueAfter6Month60 > data[i].valueAtGoldCross60)
{countWin6Month60 += 1;}

        if(data[i].valueAfter1Month20 < data[i].valueAtGoldCross20)
{countLose1Month20 += 1;}
        if(data[i].valueAfter3Month20 < data[i].valueAtGoldCross20)
{countLose3Month20 += 1;}
        if(data[i].valueAfter6Month20 < data[i].valueAtGoldCross20)
{countLose6Month20 += 1;}
        if(data[i].valueAfter1Month60 < data[i].valueAtGoldCross60)
{countLose1Month60 += 1;}
        if(data[i].valueAfter3Month60 < data[i].valueAtGoldCross60)
{countLose3Month60 += 1;}
        if(data[i].valueAfter6Month60 < data[i].valueAtGoldCross60)
{countLose6Month60 += 1;}
    }

07      CString str;
    str.AppendFormat(_T("20평선 골든 크로스 1개월 뒤 상승한 종목 개수 : %d
\n"), countWin1Month20);
    str.AppendFormat(_T("20평선 골든 크로스 3개월 뒤 상승한 종목 개수 : %d
\n"), countWin3Month20);
    str.AppendFormat(_T("20평선 골든 크로스 6개월 뒤 상승한 종목 개수 : %d
\n"), countWin6Month20);
```

```
        str.AppendFormat(_T("60평선 골든 크로스 1개월 뒤 상승한 종목 개수 : %d
    \n"), countWin1Month60);
        str.AppendFormat(_T("60평선 골든 크로스 3개월 뒤 상승한 종목 개수 : %d
    \n"), countWin3Month60);
        str.AppendFormat(_T("60평선 골든 크로스 6개월 뒤 상승한 종목 개수 : %d
    \n"), countWin6Month60);
        str.AppendFormat(_T("20평선 골든 크로스 1개월 뒤 하락한 종목 개수 : %d
    \n"), countLose1Month20);
        str.AppendFormat(_T("20평선 골든 크로스 3개월 뒤 하락한 종목 개수 : %d
    \n"), countLose3Month20);
        str.AppendFormat(_T("20평선 골든 크로스 6개월 뒤 하락한 종목 개수 : %d
    \n"), countLose6Month20);
        str.AppendFormat(_T("60평선 골든 크로스 1개월 뒤 하락한 종목 개수 : %d
    \n"), countLose1Month60);
        str.AppendFormat(_T("60평선 골든 크로스 3개월 뒤 하락한 종목 개수 : %d
    \n"), countLose3Month60);
        str.AppendFormat(_T("60평선 골든 크로스 6개월 뒤 하락한 종목 개수 : %d
    \n"), countLose6Month60);
08      ::AfxMessageBox(str);

}

//-------------------------------------------------------------------------
    void CStock::analyzeCompany(Company *company, DataGoldCross *data) {

        int i;

09      data->valueAtGoldCross20 = -1;
        data->valueAfter1Month20 = -1;
        data->valueAfter3Month20 = -1;
        data->valueAfter6Month20 = -1;
        data->valueAtGoldCross60 = -1;
        data->valueAfter1Month60 = -1;
        data->valueAfter3Month60 = -1;
```

```
        data->valueAfter6Month60 = -1;

10      for(i=120; i<company->quantity-20; i++) {
11          if((company->moveAverage[i].avg5 > company->moveAverage[i].avg20)
&& (company->moveAverage[i+1].avg5 < company->moveAverage[i+1].avg20)) {
                data->valueAtGoldCross20 = company->data[i].lastVal;
                data->valueAfter1Month20 = company->data[i-20].lastVal;
                data->valueAfter3Month20 = company->data[i-60].lastVal;
                data->valueAfter6Month20 = company->data[i-120].lastVal;

                break;
            }
        }

12      for(i=120; i<company->quantity-60; i++) {
            if((company->moveAverage[i].avg5 > company->moveAverage[i].avg60)
    && (company->moveAverage[i+1].avg5 < company->moveAverage[i+1].avg60)) {
                data->valueAtGoldCross60 = company->data[i].lastVal;
                data->valueAfter1Month60 = company->data[i-20].lastVal;
                data->valueAfter3Month60 = company->data[i-60].lastVal;
                data->valueAfter6Month60 = company->data[i-120].lastVal;

                break;
            }
        }
    }
```

---

01 프로그램에서 'Run' 버튼을 누르면 파일에서 데이터를 읽고 이평선을 만들어 준 다음 골든 크로스를 분석하는 'analyzeGoldCross( )' 함수를 실행한다.

02 통계를 만들기 위해 종가들을 저장하는 'DataGoldCross' 구조체의 변수를 선언한다. 변수는 배열로 선언되고 모든 종목의 개수만큼 크기가 할당된다.

03 모든 종목의 개수만큼 for 문을 사용하여 반복한다. 'company'는 한 종목 데이터를 가리킨다.

04 'company'에 대한 골든 크로스를 분석하여 'data' 배열 중 하나에 결과를 저장한다. 'company'는 포인터 변수이므로 해당 종목의 주소값을 매개 변수로 전달하고, 'data[i]'는 '&'를 사용하여 포인터 변

수가 아닌데도 해당 데이터의 주소값을 전달한다. 즉, 두 개의 매개 변수는 모두 참조에 의한 호출[02]로 변수를 전달한다. 'analyzeCompany( )' 함수가 실행된 후에는 'data[i]'에 결과값이 저장된다.

05 통계를 만들기 위하여 종목 개수를 세는 변수[03]들을 선언한다.

06 통계 출력을 위하여 모든 종목에서 조건에 해당하는 것을 찾아 개수를 하나씩 증가시킨다. 예를 들어, 1개월 뒤 종가가 5평선이 20평선을 돌파한 골든 크로스 때의 종가보다 크다면 상승한 것이므로 'countWin1Month20' 변수에 1을 더한다.

07 메시지 창에 결과값을 출력하기 위하여 문자열 변수 'str'을 사용한다. 이후에 'AppendFormat( )' 메소드를 사용하여 통계 결과를 문자열에 계속 추가한다.

08 '::AfxMessageBox( )' 함수를 사용하여 결과를 메시지 창에 출력한다.

09 'DataGoldCross' 구조체의 변수들을 -1로 초기화한다. 골든 크로스가 발생하지 않았다면 변수는 -1을 유지할 것이고, 이후에도 변수값이 -1이라면 아무 데이터도 입력되지 않았다는 것을 뜻한다.[04]

10 for 문에서 'i'를 120부터 시작한 것은 6개월 이전의 데이터를 계산하기 위해서다. 종목 데이터에서 앞쪽에 있는 것이 최근의 데이터이므로 index는 120번째 이후의 데이터를 가지고 골든 크로스를 계산한다. 종목의 끝까지 for 문을 실행하지 않고 'i'에서 20을 뺀 것은 20평선 데이터에서 마지막 19개 데이터는 값이 존재하지 않기 때문이다.

11 5평선이 20평선에서 골든 크로스라는 것은 이전 값에서는 5평선이 20평선보다 낮고 현재 값에서 5평선이 20평선보다 높게 있다는 뜻이다. 이 조건을 만족할 때 골든 크로스의 종가와 20일, 60일, 120일 이후의 종가를 'DataGoldCross' 구조체의 변수에 넣어준다.

12 10과 마찬가지로 for 문의 'i'는 120(6개월)부터 시작하고 60평선을 다루므로 종료 조건에서 60을 빼준다.

[그림 3-3]은 실행 결과값이다.[05]

---

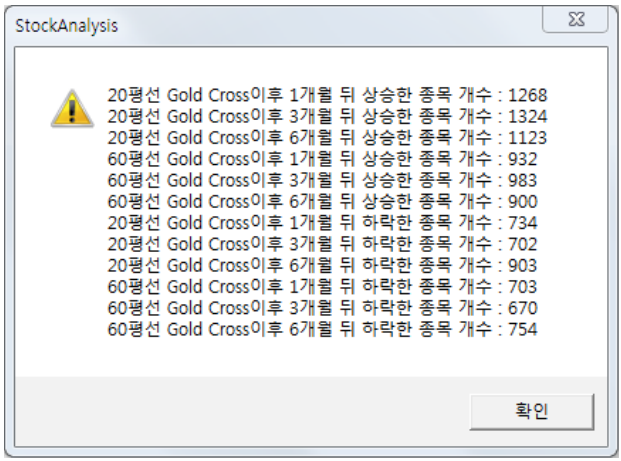
02 필자는 C++ 프로그램을 구현할 때 '값에 의한 호출(Call-by-value)' 방법보다 '참조에 의한 호출(Call-by-reference)' 방법을 주로 사용한다. 값에 의한 호출은 전달될 때 메모리 복사가 이루어지므로 전달되는 변수가 클 때 많은 양의 데이터 복사가 일어난다. 하지만 참조에 의한 호출은 메모리의 내용은 그대로 있고 메모리의 주소값 4바이트만 복사된다. 물론, 두 가지 호출 방법은 각각 장단점이 있다. 그러나 필자가 참조에 의한 호출을 좋아하는 이유는 연산으로 데이터가 바뀌는 것이 편할 때가 많기 때문이다. C/C++로 프로그래밍할 때는 항상 메모리 측면에서 생각하는 것이 좋다.

03 변수 이름은 유추하기 쉽도록 정한다. 예를 들어, 'countWin1Month20'은 5평선이 20평선(20)을 골든 크로스 후 1개월(1Month) 뒤에 주가가 상승하여 이긴(Win) 종목의 개수를 세는(count) 변수다. 나머지 변수도 이처럼 유추하면 된다. 예전에는 코드의 길이가 한 줄에 80개의 문자를 넘지 않던 시대가 있었다. 그때는 변수명을 매우 짧게 만들었지만, 지금은 변수명에 변수의 특징을 알 수 있게 모든 정보를 넣는 것이 좋다.

04 변수를 -1로 초기화하는 것은 종가가 항상 양수라서 필자가 그렇게 정한 것뿐이다.

05 책으로 프로그램을 설명하려면 프로그램이 최대한 간단해야 이해하기가 쉽다고 본다. 추가되는 코드들은 독자들이 계속 구현해 가면서 결과값을 출력하는 것도 하나의 방법일 수 있고, 통계로 살펴볼 것이 많다면 파일에 출력하는 것도 좋은 방법이다.

[그림 3-3] 5평선의 골든 크로스 통계 결과



결과를 보면 골든 크로스 이후 상승한 종목이 하락한 종목보다 많다. 그러나 여기서 사용한 데이터가 전체적으로 상승장일 때 데이터일 수 있고, 골든 크로스가 이루어져서 매수했어도 30% 이상은 하락할 수 있는 종목임에 유의해야 한다. 추세를 보고 장기 투자를 한다고 해도 골든 크로스 때 무조건 매수하는 것은 옳지 않을 수 있다. 독자들이 이번 장을 이해하였다면, 골든 크로스에서 매매했을 경우 1개월, 3개월, 6개월 뒤에 이익이 얼마나 나는지 알아보는 것도 재미있는 방법이다. 이 책에서는 골든 크로스 때 종가와 이후 종가의 크고 작음을 가지고 개수만을 카운트하였다. 하지만 빼기를 추가해서 모든 종목에서 종가의 차이를 계산하여 이익이 얼마나 났는지를 통계로 확인한다면 또 하나의 재미를 얻을 수 있을 것이다. 데드 크로스일 때의 통계도 쉽게 구현할 수 있음은 물론이다.

3.4 이평선 오실레이터

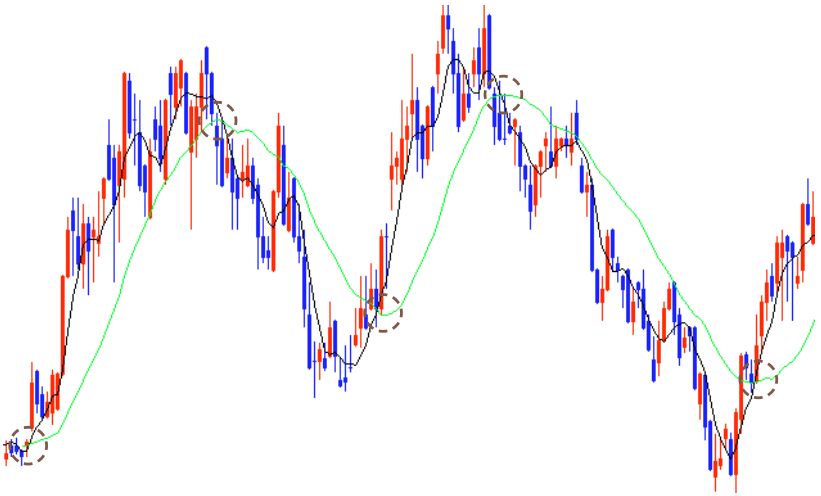
오실레이터Oscillator는 '진동하는 물건'이라는 뜻으로, 주식에서는 매매 시점을 알려주는 지표를 의미한다. 매수 시점과 매도 시점을 차트의 주가만을 가지고 계산하

여 알려주거나 자동매매를 하게 한다. 트레이딩 시스템을 만든다는 것은 아주 좋은 오실레이터를 만들어서 자동매매 프로그램을 만드는 것이라고 보면 된다. 매일같이 주식을 사고파는 데이트레이더Day Trader, 단타 투자자는 매매를 누구보다 빨리 잘하는 것이 중요하다. 그러나 아무리 빠르게 분석하고 판단해도 사람이기에 때문에 컴퓨터 매매보다는 한참 늦을 수밖에 없다. 이러한 관점에서 앞으로는 누가 더 좋은 프로그램을 가지고 있느냐가 주식 시장에서의 승패를 좌우할 것이다. 또한, 인간은 감정에 인해 올바른 판단을 못 할 때가 있지만, 컴퓨터는 감정이 없기에 냉철한 매매가 가능하다. 뉴욕 시장에서는 2000년대 중반부터 시스템 트레이딩이 활성화되었다. 오래전부터 만들어온 주식의 보조 지표들을 이용하여 분석하고, 수학자들은 새로운 알고리즘을 만들기 위해 계속 연구한다. 우리나라는 2000년대 말 증권회사에서 API를 제공하기 시작하면서부터 시스템 트레이딩을 하는 사람들이 조금씩 늘어가고 있다.

이평선은 주식을 매매하는 사람 대부분이 알고 있지만, 이평선에 오실레이터가 존재한다는 것은 많은 사람이 모르고 있다. 주식에서 골든 크로스와 데드 크로스는 알지만, 이것이 오실레이터가 된다는 것은 인지하지 못하는 것 같다.

[그림 3-4]는 5평선(검은색)과 20평선(초록색)에서 골든 크로스일 때 매수하고 데드 크로스일 때 매도할 수 있음을 보여준다. [그림 3-4]의 차트를 보고 이평선 오실레이터의 힘에 대해 놀랄 수도 있다. 원으로 표시된 곳에서만 매매가 이루어졌다면 괜찮은 수익이 날 수 있다. 이러한 차트에서는 이평선이 정확하게 들어맞는다. 온라인으로 홍보하는 곳에서 보여주는 사진은 이러한 차트가 대부분이다. 그렇지만 이런 차트는 많지 않다. 이평선은 후행성이라서 골든 크로스에서는 주가가 높이 올라간 곳에서 매수가 이루어지고 데드 크로스에서는 주가가 확실히 내려간 시점에서 매도한다. 물론, 드물게 나오는 이러한 차트에서는 어쨌든 이평선 오실레이터로 본 승률은 아주 높을 것이다.

[그림 3-4] CJ대한통운 일봉



이번 장에서는 가장 간단한 오실레이터인 이평선 오실레이터를 통해 트레이딩 시스템을 어떻게 만드는지를 살펴본다. 이 책에서 만드는 프로그램은 모든 종목에서 이평선 오실레이터로 매매하고, 재미있는 통계도 보게 될 것이다. 그리고 이평선 오실레이터가 후행성이어서 안 좋은 이유, 이평선 오실레이터를 왜 그대로 사용하지 않는지 알게 될 것이다. 이장에서 오실레이터가 무엇인지 가볍게 살펴보고, 다음 장에 소개될 MACD와 볼린저 밴드Bollinger Band의 오실레이터에 대해서 좀 더 깊이 있게 이해할 수 있기를 바란다.

이번 장에서 구현하는 이평선은 구현하는 사람이 이평선을 정할 수 있다. 즉, 5, 20, 60, 120평선만 가지고 골든 크로스와 데드 크로스를 확인하는 것이 아니라, 3평선과 10평선의 크로스 등 여러 가지 이평선으로 테스트할 수 있게 코드를 구현한다. 그러기 위해서는 두 개의 이평선 값을 쉽게 바꿀 수 있는 코드를 구현해야 한다. 가장 쉬운 방법은 [코드 3-8]에서와 같이 '#define'으로 값을 선언하는 것이다. 이렇게 코드를 만들고 난 후에는 '#define'에서 정의된 값만을 변경해서 새로

운 프로그램으로 바꿀 수 있다. 물론, 프로그램 화면에서 이평선의 값을 바꾸어서 테스트할 수도 있으나, 프로그램을 간단히 구현하기 위해 '#define'을 사용한다. [코드 3-8]에서는 두 개의 이평선을 가지고 자동매매를 하기 위해 특정 값들을 정의한다.

[코드 3-8] 이평선 및 매매 시 차감률 정의(Stock.h)

```
01   #define MOVE_AVERAGE_1      5
02   #define MOVE_AVERAGE_2      20

03   #define CHARGE_BUY          0.00015
04   #define CHARGE_SELL         0.00315
05   #define CHARGE_TRADE        0.005

     struct MovementAverage {
06         long avg1;
07         long avg2;
     };

     struct Company {
           CString strJongMok, strName;
           int quantity;
           Data data[MAX_DATA];
08         MovementAverage moveAverage[MAX_DATA];
     };
```

01 두 개의 이평선 중 첫 번째는 5평선을 사용하기 위하여 5로 'MOVE_AVERAGE_1'을 정의한다.

02 두 번째 이평선으로 20평선을 사용하기 위하여 20으로 'MOVE_AVERAGE_2'를 정의한다. 이후에 다른 이평선으로 프로그램을 변경하려면 여기서 정의한 두 개의 이평선 값들을 변경한다.

03 매수 시 증권회사에 0.015%의 수수료를 내야 하므로 주가에 0.00015를 곱하기 위한 'CHARGE_BUY'를 정의한다.

04 매도 시 증권회사에 세금을 포함한 수수료로 0.315%를 내야 하므로 주가에 0.00315를 곱하기 위한 'CHARGE_SELL'을 정의한다.

05 자동매매는 매매 시 무조건 거래가 이루어져야 한다. 매수할 때는 더 높은 가격에 주문을 넣고, 매도할 때는 낮은 가격에 주문을 넣는다. 매매가 이루어지는 가격은 매수 시에는 0.5% 높은 가격에 체결된다고 가정하고, 매도 시에는 0.5% 낮은 가격에 체결된다고 가정한다. 즉, 매매할 때 손실을 0.5%로 산정하고, 주가에 0.005를 곱하기 위하여 'CHARGE_TRADE'를 정의한다.

06 이평선은 'MovementAverage' 구조체에 저장되는데, 두 개의 이평선 중 첫 번째 이평선 값은 'avg1'에 저장된다.

07 두 번째 이평선 값은 계산된 후 'avg2'에 저장된다.

08 한 종목의 데이터가 저장되는 'Company' 구조체는 'moveAverage' 배열을 사용하여 이평선 데이터에 접근한다.

[코드 3-8]은 5평선과 20평선으로 이평선 오실레이터를 검증하기 위하여 MOVE_AVERAGE를 5와 20으로 정의하였다. 다양한 이평선으로 검증하려면 여기서 정의한 5와 20을 변경하여 컴파일한다. 예를 들어, 3평선과 10평선으로 검증하기 위해서는 'MOVE_AVERAGE_1'과 'MOVE_AVERAGE_2'를 3과 10으로 변경한다. 일반적으로 '#define'에서 정의된 값을 변경하는 것만으로 쉽게 알고리즘을 바꿀 수 있으며, 이후의 코드에서는 'MOVE_AVERAGE_1'과 'MOVE_AVERAGE_2' 이름만을 사용하여 프로그램을 구현한다.

[코드 3-9]에서는 이평선을 계산하고 오실레이터에 따라 매매한다. 'makeMovementAverage()' 함수는 이전에 구현하였으므로 특별히 신경 쓰지 않아도 되지만, 'simulateTrade()' 함수는 주의 깊게 봐야 한다. 이 함수에서 가상매매가 구현되는데, 중요한 변수 중의 하나가 'isBought'다. 이름에서 알 수 있듯이 매매 여부를 확인한다. 'isBought' 변수가 'false'이면, 사기 위한 조건을 계속 검증하고, 매수가 이루어지면 'isBought' 변수를 'true'로 변경하여 매도를 위한 조건을 계속 확인한다.

가장 최근 데이터까지 검증이 이루어진 후에는 마지막에 무조건 매도하게 구현한다. 시뮬레이션이므로 마지막에는 모든 거래를 중지하고 어떠한 종목도 보유하지 않기 위해서다. 이 책에서는 일봉 데이터로 프로그램을 만들어서 설명하지만, 실제

시스템 트레이딩에서는 분봉으로 하루 동안 모든 거래가 이루어지고 주식장이 끝났을 때는 모든 보유 주식을 파는 것이 좋다. 하루가 지나면 장이 어떻게 변할지 모르므로 주식을 사 놓는 것은 단순히 운에 맡기는 것이라서 시스템 트레이딩에는 적절치 않다고 생각한다.

새로운 장이 시작되면 처음부터 새롭게 시작하는 것이 좋다. 또한, 장이 시작되자마자 바로 매매에 들어가는 것이 아니라 계속 모니터링을 하다가 일정 시간이 지난 후부터 매매 알고리즘을 가동하는 것이 좋다. 물론, 시스템 트레이딩을 모두가 이 방법으로 한다면 문제가 발생할 수 있다. 장이 끝날 때 많은 사람이 물건을 던지면 항상 하락장이 발생하므로 이것도 맞지 않을 수 있다. 단지 필자가 시스템 트레이딩을 한다면 장이 끝날 때는 주식을 보유하지 않겠다는 하나의 의견으로 보기 바란다. 여기서는 시뮬레이션이므로 마지막에는 무조건 매수한 것은 매도하는 조건을 넣는다.

[코드 3-9] 두 개의 이평선 크로스 시 매매 구현 및 결과 출력(Stock.cpp)

```
    void CStock::Run()
    {
        ReadDataFromFile();
        makeMovementAverage();
01      simulateTrade();
    }

    void CStock::makeMovementAverage()
    {
        int i, j, k;
        long temp;

        for(i=0; i<allCompanies.quantity; i++)
        {
            Company *company = &allCompanies.companies[i];
```

```
            int quantity = company->quantity;

02          for(j=0; j<=quantity-MOVE_AVERAGE_1; j++)
            {
                temp = 0;
                for(k=j; k<j+MOVE_AVERAGE_1; k++)
                    temp += company->data[k].lastVal;
                company->moveAverage[j].avg1 = (long)(temp / MOVE_AVERAGE_1);
            }

            for(j=0; j<=quantity-MOVE_AVERAGE_2; j++)
            {
                temp = 0;
                for(k=j; k<j+MOVE_AVERAGE_2; k++)
                    temp += company->data[k].lastVal;
                company->moveAverage[j].avg2 = (long)(temp / MOVE_AVERAGE_2);
            }
        }
    }

03  void CStock::simulateTrade()
    {
04      int balance = 100000000;
05      int prevBalance;
06      int countWin = 0;
07      int countLose = 0;
08      int charge;
09      int chargeTradeCenter = 0;

10      selectedCompanies.quantity = 0;

        int i, j;

        for(i=0; i<allCompanies.quantity; i++) {
            Company *company = &allCompanies.companies[i];
```

```
        int quantity = company->quantity;

11      prevBalance = balance;
12      int isBought = false;

        for(j=quantity-MOVE_AVERAGE_2-1; j>0; j--) {
13          if(!isBought) {
14              if(company->moveAverage[j].avg1 > company->
    moveAverage[j].avg2 && company->moveAverage[j+1].avg1 <= company->
    moveAverage[j+1].avg2) {
15                  charge = (int)(company->data[j].lastVal * CHARGE_BUY);
                    balance -= charge;
16                  chargeTradeCenter += charge;
17                  charge = (int)(company->data[j].lastVal * CHARGE_TRADE);
                    balance -= charge;
18                  balance -= company->data[j].lastVal;
19                  isBought = true;
                }
20          }else{
21              if(company->moveAverage[j].avg1 < company->
    moveAverage[j].avg2 && company->moveAverage[j+1].avg1 >= company->
    moveAverage[j+1].avg2) {
22                  charge = (int)(company->data[j].lastVal * CHARGE_SELL);
                    balance -= charge;
                    chargeTradeCenter += charge;
23                  charge = (int)(company->data[j].lastVal * CHARGE_TRADE);
24                  balance += company->data[j].lastVal;
25                  isBought = false;
                }
            }
        }
26      if(isBouht) {
            charge = (int)(company->data[0].lastVal * CHARGE_SELL);
```

```
                balance -= charge;
                chargeTradeCenter += charge;
                charge = (int)(company->data[0].lastVal * CHARGE_TRADE);
                balance -= charge;
                balance += company->data[0].lastVal;
                isBought = false;
            }

27          if(balance > prevBalance) {
                countWin += 1;
28              selectedCompanies.companies[selectedCompanies.quantity] = company;
                selectedCompanies.quantity += 1;
29          }else if(balance < prevBalance) {
                countLose += 1;
            }
        }

        CString str;
30      str.AppendFormat(_T("Win 종목 개수: %d \n"), countWin);
        str.AppendFormat(_T("Lose 종목 개수: %d \n"), countLose);
        str.AppendFormat(_T("Balance : %d \n"), balance);
        str.AppendFormat(_T("수수료: %d \n"), chargeTradeCenter);
        ::AfxMessageBox(str);
    }
```

---

1 프로그램 화면에서 Run 버튼을 누르면 'CStock::Run( )' 함수가 실행된다. 파일로부터 데이터를 읽어 'makeMovementAverage( )' 함수에서 이평선이 계산되고, 'simulateTrade( )' 함수에서 가상 매매 시뮬레이션이 수행된다. 자동으로 매매가 이루어지고 데이터에 대한 결과를 메시지 창에 보여준다.

2 'makeMovementAverage( )' 함수에서 이평선을 계산할 때 이평선 값은 숫자를 직접 쓰는 것이 아니라, 헤더 파일인 'Stock.h'에 정의된 'MOVE_AVERAGE_1'과 'MOVE_AVERAGE_2' 상수를 사용한다. 주가 데이터는 앞쪽이 최근 데이터고 뒤로 갈수록 오래된 데이터이므로 배열 뒷쪽의 일정 부분은 이평선 데이터를 넣을 수가 없다. 따라서 'j <= quantity - MOVE_AVERAGE_1'과 같이 전체 데이터 개수 'quantity'에서 'MOVE_AVERAGE_1'의 상수를 빼서 계산한다.

3 'simulateTrade( )' 함수는 매매 시뮬레이션을 실행한다. 진행하면서 매매 조건을 항상 확인하고 조건이 성립하면 매매를 수행한다. 그리고 마지막에 통계를 메시지 창에 출력한다.

4 계좌에 1억이 있다고 가정하고, 잔액 변수인 'balance'에 100000000을 선언한다.

5 'prevBalance'는 이전 잔액을 기억하기 위한 변수다.[06]

6 통계를 내기 위한 변수로 매매에서 수익이 있는 종목이 발생하면 'countWin' 값이 1 증가한다.

7 통계를 내기 위한 변수로, 매매에서 손실이 있는 종목이 발생하면 'countLose' 값이 1 증가한다.

8 'charge' 변수는 매매 시 수수료를 계산하기 위해서 사용된다.

9 매매 시 증권사에 주는 수수료를 위한 변수로, 0으로 초기화하고 매매가 이루어질 때마다 'charge'(수수료)가 더해진다.

10 종목을 선택하여 콤보 박스에서 볼 수 있도록 'selectedCompanies' 변수를 사용한다. 27에서 수익이 난 종목만을 콤보 박스에 보여주기 위해 'selectedCompanies' 변수에 해당 종목을 추가한다.

11 한 종목에서 시뮬레이션을 시작하기 전에 'balance'를 'prevBalance'에 저장한다. 해당 종목의 매매가 끝난 후 'balance'가 'prevBalance'보다 크다면 수익이 난 것이고, 작다면 손실이 난 것이다.

12 주식을 매수했는지 확인하는 'isBought' 변수는 시작 시 'false'로 초기화된다.

13 'isBought'가 'false'면 주식을 사지 않은 것이므로, 매수 조건을 확인한다.

14 (14부터 19까지는 매수 부분이다.) 이평선이 골든 크로스를 확인하여 조건이 성립되면 매수한다. 골든 크로스는 첫 번째 이평선(5평선)이 두 번째 이평선(20평선) 아래에 있다가 현재는 위에 존재할 때를 말하며, 여기서는 'j+1'번째가 바로 전 이평선 데이터고, 'j'가 현재의 이평선 데이터다.

15 수수료는 현재 종가에 헤더 파일에서 정의된 'CHARGE_BUY(=0.00015)'를 곱한 값으로, 이 값은 'balance'에서 차감한다.

16 매매 수수료는 증권회사에 주는 것이므로 'chargeTradeCenter'에 'charge'가 더해진다. 프로그램 마지막 부분에서 'chargeTradeCenter' 값은 통계로 출력된다.

17 매수는 현재 값보다 비싼 값에서 이루어진다. 여기서는 'CARGE_TRADE'를 0.5%로 정하였으므로 만약 종가가 100,000원이면 100,500원에 매수하여 'balance'에서 500원을 추가로 빼야 한다.

18 매수는 주식을 산 것이므로 'balance'에서 현재 종가를 뺀다. 즉, 매수는 'balance'에서 종가를 빼고, 매도는 'balance'에 종가를 더하면 시뮬레이션이 이루어진다.

19 매수하였으므로 주식 구매를 확인하기 위해 'isBought'는 'true'로 설정되고, 이후부터는 20과 같이 매도 조건을 확인한다.

20 (20부터 24까지는 매도 부분이다.) 'isBought'가 'true'일 때 수행되는 조건으로 매도 조건(21 ~ 24)을 확인하고 매도한다.

---

06 보통 변수 이름을 만들 때 '이전의' 의미인 previous의 약어로 앞에 'prev'를 붙인다.

21 이평선이 데드 크로스가 났을 때를 확인하여 조건이 성립되면 매도한다. 데드 크로스는 첫 번째 이평선(5평선)이 두 번째 이평선(20평선) 위에 있다가 현재는 아래에 존재할 때로, 여기서는 'j+1'번째가 바로 전 이평선 데이터고, 'j'가 현재 이평선 데이터다.

22 수수료는 'CHARGE_SELL(=0.00315)'를 곱한 값으로, 이 값은 'balance'에서 차감하고 증권회사 수수료에는 더해진다.

23 매도는 현재 값보다 낮은 값에서 이루어진다. 여기서는 0.5% 낮은 가격에 매매하므로 만약 종가가 100,000원이면 99,500원에 매도하여 'balance'에서 500원을 추가로 빼야 한다.

24 매도하였으므로 현재 종가를 'balance'에 더한다.

25 매도하였으므로 주식이 없는 상태로 만들기 위해 'isBought'는 'false'가 되고, 이후부터 매수 조건 알고리즘인 14를 수행한다.

26 자동매매 후 마지막에 주식을 사 놓은 상태라면 무조건 매도를 실행한다[07].

27 한 종목의 시뮬레이션이 끝난 후에 매매 시작 전 잔액(prevBalance)보다 현재 잔액(balance)이 많다면 이긴 것이므로 통계를 위한 변수인 'countWin'이 1 증가한다.

28 프로그램이 종료된 후 콤보 박스에 이긴 종목만 보여주기 위하여 'selectedCompanies'에 해당 종목을 추가한다. 손해를 본 종목만 보고 싶다면 이 부분을 29 아래에 넣어준다.

29 매매 시작 전 잔액(prevBalance)보다 현재 잔액(balance)이 적다면 손실이 발생한 것이므로 통계를 위한 변수인 'countLose'가 1 증가한다.

30 모든 종목에 대한 시뮬레이션이 끝난 후 메시지 창에 그 결과를 보여준다.

지금까지 구현한 부분이 이번 장에서 가장 중요한 부분이었다. 이후에 구현하는 내용은 화면에 그래프를 그려주기 위한 것으로, 이전 장에서 구현한 것과 많은 부분이 비슷하다. 차트를 그리기 위해서 'Graph.cpp'에서 점의 위치를 계산하고, 'StockAnalysis Dlg.cpp'는 선을 그린다.

[코드 3-10] 화면에 그리기 위한 이평선 점 배열 정의(Graph.h)

```
struct PointData {
    PointArray startVal;
    PointArray highVal;
```

07 앞에서 설명한 것처럼 여기서 구현하는 프로그램은 시뮬레이션이므로 마지막에는 어떠한 종목도 구매 상태로 남겨 두지 않는다. 실제 시스템 트레이딩은 분봉을 가지고 진행된다. 다음날 장 초반에 주가가 급격히 움직이면서 이루어지는 변동을 기대한다는 것은 단순히 운에 맡기는 것이므로 장이 끝난 후에는 어떠한 종목도 보유하지 않는 것이 좋다고 생각한다.

```
        PointArray lowVal;
        PointArray lastVal;
        PointArray volume;

01      PointArray avg1;
        PointArray avg2;
    };
```

01 'PointArray'라는 점 배열을 선언하여 첫 번째 이평선은 'avg1'에, 두 번째 이평선은 'avg2'에 저장한다. 여기에 저장되는 데이터는 화면에 그래프를 그리기 위한 점의 위치를 나타낸다.

[코드 3-11] 화면에 그리기 위한 이평선 점 계산(Graph.cpp)

```
    void CGraph::GetDataForChart(Company* company, PointData* ptData)
    {
        ### 생략 ###

        //------------------- Movement Average --------------------
01      ptData->avg1.quantity = company->quantity - MOVE_AVERAGE_1;
        for(i=0; i<ptData->avg1.quantity; i++) {
            ptData->avg1.point[i].X = x_start - i * interval;
02          ptData->avg1.point[i].Y = y_start - (HEIGHT_GRAPH * (company->
    moveAverage[i].avg1 - minVal)) / (maxVal - minVal);
        }

        ptData->avg2.quantity = company->quantity - MOVE_AVERAGE_2;
        for(i=0; i<ptData->avg2.quantity; i++) {
            ptData->avg2.point[i].X = x_start - i * interval;
            ptData->avg2.point[i].Y = y_start - (HEIGHT_GRAPH * (company->
    moveAverage[i].avg2 - minVal)) / (maxVal - minVal);
        }
    }
```

01 이평선의 개수는 종목의 데이터 개수에서 평균을 만들기 위한 수를 빼준다. 예를 들어, 종목의 데이터 개수가 250이라면 5평선의 데이터 개수는 245(= 250 - 5)가 된다.[08]

02 모든 점의 위치는 [그림 2-14]에서 설명한 '비율 계산'을 이용해서 화면에서의 점의 위치를 계산한다.

[코드 3-12] 화면에 이평선 그리기(StockAnalysisDlg.cpp)

```
void CStockAnalysisDlg::DrawGraph()
{
    ### 생략 ###

  //------------------- Movement Average --------------------
    CPen avg1Pen, avg2Pen;

01  avg1Pen.CreatePen(PS_SOLID, 1, RGB(0,0,0));
    avg2Pen.CreatePen(PS_SOLID, 1, RGB(0,255,0));

    dc.SelectObject(avg1Pen);
02  dc.MoveTo(ptData->avg1.point[0].X, ptData->avg1.point[0].Y);
    for(i=1; i<ptData->avg1.quantity; i++) {
        dc.LineTo(ptData->avg1.point[i].X, ptData->avg1.point[i].Y);
    }

    dc.SelectObject(avg2Pen);
    dc.MoveTo(ptData->avg2.point[0].X, ptData->avg2.point[0].Y);
    for(i=1; i<ptData->avg2.quantity; i++) {
        dc.LineTo(ptData->avg2.point[i].X, ptData->avg2.point[i].Y);
    }
}
```

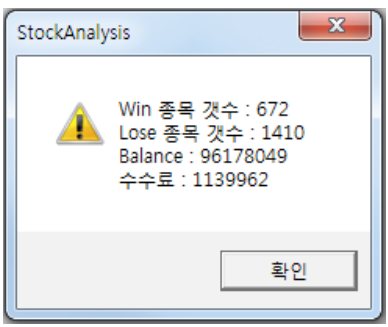
01 선을 그리기 위한 펜을 정의한다. 첫 번째 이평선은 'avg1Pen'으로 검은색 실선이고, 두 번째 이평선은 'avg2Pen'으로 녹색 실선이다.

02 'MoveTo( )' 함수를 이용하여 좌표를 첫 번째 점으로 이동하고, 'LineTo( )' 함수로 나머지 점들에 선을 그린다.

08 엄밀히 따지면 246으로 계산해도 되지만, 프로그램을 이해하기 쉽게 5를 뺀다.

프로그램을 실행하면 [그림 3-5]와 같은 결과가 나온다. 5평선과 20평선의 크로스에서 매매한 결과로, 손실이 난 종목이 이긴 종목보다 두 배 이상 많다. 모든 종목에서 한 번씩 매매되었고 약 400만 원 정도의 손실이 났으며 증권회사에 내는 수수료는 약 100만 원이다. 모든 종목의 매매 시점을 보고 싶다면 결과를 파일로 출력한다.[09]

[그림 3-5] 5평선과 20평선의 크로스에서 매매한 결과



앞에서 첫 번째 이평선과 두 번째 이평선은 'Stock.h' 파일에서 정의하였다. [코드 3-13]과 같이 두 개의 이평선을 3평선과 10평선으로 다시 정의하여 프로그램을 실행시키면 새로운 결과가 나온다. 3평선과 10평선으로 이평선 매매를 하면 5평선과 20평선으로 비교한 것보다 주가 변동에 더 민감하게 움직이므로 매매는 더 빈번히 일어난다. 이 프로그램을 실행하면 [그림 3-6]처럼 결과가 나온다. 5평선과 20평선으로 검증했을 때보다 수익률은 낮고, 손실이 난 종목이 수익이 난 종목의 4배나 된다. 또한, 1억 원에서 약 1000만 원의 손실이 발생하고, 수수료는 200만 원이 넘는다. 수수료가 이전 결과보다 2배 이상 많다는 것은 3평선과 10평선의 크로스 매매가 2배 이상 이루어졌음을 의미한다.

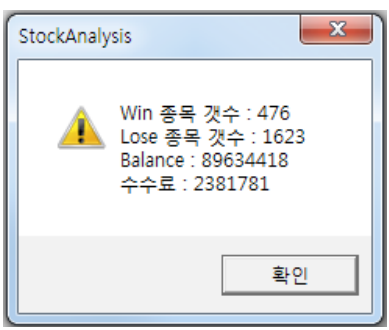
09 자신이 원하는 데이터를 짧은 시간 내에 볼 수 있어서 주식 프로그램에 대한 재미를 느낄 수 있을 것이다.

'#define'으로 상수를 정의하여 프로그램을 만들면 숫자 변경만으로 새로운 프로그램이 만들어진다. 변경될 수 있는 값들은 이처럼 '#define'을 사용하여 프로그램을 만드는 것이 좋다.

[코드 3-13] 3평선과 10평선 매매 정의하기(Stock.h)

```
#define MOVE_AVERAGE_1      3
#define MOVE_AVERAGE_2      10

#define CHARGE_BUY          0.00015
#define CHARGE_SELL         0.00315
#define CHARGE_TRADE        0.005
```

[그림 3-6] 3평선과 10평선의 크로스에서 매매한 결과

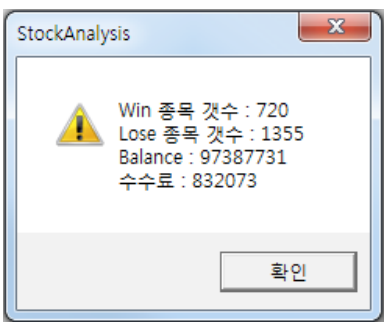


[코드 3-14]는 5평선과 30평선의 이평선 매매를 위하여 'Stock.h'에서 값을 변경한 것으로, 결과는 [그림 3-7]처럼 나왔다. 수수료가 약 80만 원 정도이므로 이전 검증보다 매매는 많이 줄어들었다. 수익이 난 종목이 이전 검증보다 많지만, 시뮬레이션 결과는 약 300만 원 정도 손해다. 이평선만 가지고 매매하면 수익이 나지 않는다는 것을 데이터를 통해 확인할 수 있다.

[코드 3-14] 5평선과 30평선 매매 정의하기(Stock.h)

```
#define MOVE_AVERAGE_1      5
#define MOVE_AVERAGE_2      30

#define CHARGE_BUY          0.00015
#define CHARGE_SELL         0.00315
#define CHARGE_TRADE        0.005
```

[그림 3-7] 5평선과 30평선의 크로스에서 매매한 결과



이 책에서는 최대한 프로그램을 간단하게 만들기 위해 많은 종류의 통계 데이터를 만들지 않았다. 여기까지 이해한 독자는 매매 횟수를 확인할 수 있고, 매매가 이루어진 시점에 선을 그어 매매 포인트를 좀 더 명확히 볼 방법을 구현하는 것도 좋을 것이다. 또한, 이긴 종목들의 종가 대비 수익률을 계산할 수 있으며, 이익이 최대인 종목도 확인할 수 있다. 주식 프로그래밍을 하는 순간 독자는 원하는 결과 데이터를 짧은 시간에 확인할 수 있는 것이 얼마나 편리하고 유용한 지를 알게 될 것이다. 그뿐만 아니라 컴퓨터가 얼마나 빨리 연산을 수행하는 지도 체감할 것이다.

최근에 지인으로부터 외국에서 시스템 트레이딩을 하는 사람들이 한국 시장에 많이 들어왔다는 이야기를 들었다. 초기에는 한국의 옵션 시장이 미국보다 많이 취약

했었는데, 현재는 프로그램 매매로 수익이 나기가 많이 힘들어졌다고 한다. 또한, 전에는 프로그램만을 사용하여 매매하였으나, 현재는 프로그램으로 신호를 주면 최종 결정은 사람이 하는 방법으로 바뀌는 중이라고 한다.

필자가 생각한 것보다 꽤 많은 사람이 시스템 트레이딩에 참여하고 있고, 프로그래밍하는 사람들의 상당수가 주식 시장에 참여하고 있다. 주식 프로그램에 관심이 있는 독자는 데이터로 충분히 시뮬레이션을 해보기 바란다. 필자는 현재도 5년 동안의 데이터를 가지고 시뮬레이션을 하고 있다. 주식 투자를 하려는 목적은 아니고, 이 책을 집필하면서 좀 더 깊이 이해하고자 함이다. 개인적으로 컴퓨터 게임보다 더 재미있다. 컴퓨터 게임은 나의 고귀한 시간을 버리는 느낌이지만, 주제를 가지고 프로그래밍을 하면 그 과정에서 많이 배우고 더 생산적이기 때문이다.

2013년 말 '한맥투자증권'이 파산했다. 단 1분 동안에 4만 건이 넘는 매매로, 약 460억 원의 손실이 발생하였다. 시스템 트레이딩이 잘못 만들어졌을 때의 피해를 보여 준 것으로, 단 한 번의 주문 실수로 회사가 무너질 수 있음을 단적으로 보여준 예다.

프로그램 매매는 잘 만들어지면 좋지만, 논리적 오류가 있을 때는 문제점을 찾기도 힘들 뿐만 아니라 모든 것을 한 번에 잃을 수도 있다. '한맥투자증권'의 경우는 프로그램에 잘못된 정보를 입력하여 발생하였다고는 하나 어쩌면 잘못된 정보를 입력했을 때 오류 메시지로 이를 알려주는 프로그램을 구현했다면 어땠을까?

4 | MACD

주식 시장에는 많은 수학자가 참여하고 있다. 보조 지표들을 공부할수록 수학적인 측면에서 깊이 있게 연구되었다는 점을 느낄 수 있다. MACD는 'Moving Average Convergence & Divergence'의 약자로 해석하면 '이동평균의 수렴과 발산'이다. 주식의 이평선은 수렴과 발산을 반복하는 특성이 있어서 이것을 이용하여 MACD가 만들어졌다.

4.1 MACD 분석

MACD를 이해하려면 먼저 지수이동평균을 알아야 한다. 지수이동평균EMA: Exponential Moving Average은 현재 주가에 지수 가중치를 두고 이평선을 만드는 것이라고 보면 된다. 즉, 현재의 주가 변동에 비중을 얼마나 두느냐에 따라 기존의 이동평균선에 변화를 줄 수 있다.

다음은 지수이동평균을 구하는 식이다.

EMA(0) = Value(0)

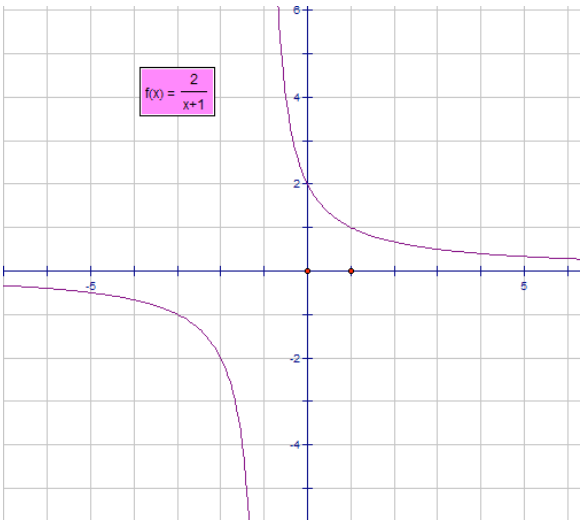
EMA(n) = Value(n) × K + EMA(n − 1) × (1 − K) (if n>1)

EMA(n): n번째 지수이동평균
Value(n): n번째 주가
K = 2/(이동평균 기간 + 1)

식을 보면 다소 어렵게 느껴질 수 있는데, 여기서 중요한 것은 K 값이다. 예를 들어, 1일 지수이동평균일 때 K는 1(= 2/(1 + 1))이므로 EMA(n) = Value(n)이 성립된다. 따라서 1일 지수이동평균은 현재의 주가(Value(n))가 되고, 이것은 1평선과 같다. 만약 2일 지수이동평균이면, K는 2/3가 되므로 현재의 주가 Value(n)는

2/3만큼 지수이동평균에 영향을 준다. 이처럼 x일 지수이동평균은 현재 주가를 2/(x + 1)만큼의 비중으로 반영한다. 이것을 그래프로 그리면 다음과 같다.

[그림 4-1] f(x) = 2/(x + 1) 그래프



[그림 4-1]은 x>0일 때 지수함수의 특성을 가진다. 지수이동평균은 이렇게 현재의 주가가 지수함수를 바탕으로 이동평균선에 반영된다고 보면 된다. 이전 장에서 공부한 이평선과 비교하면 이평선은 이동평균기간의 주가를 더하여 해당 기간으로 나누므로 현재 주가는 항상 일정 비율의 비중을 차지하고 이동평균 기간이 커질수록 이평선에 미치는 영향이 적어진다. 하지만 지수이동평균은 이동평균기간이 매우 커도 이평선과 비교하면 무척 낮은 비율이다.

수학적인 관점에서 지수이동평균이 지수함수임은 다음 식으로 알 수 있다.[01]

01 이 식이 다소 어렵게 느껴질 수 있으나, 단지 지수이동평균이 지수함수의 특징을 갖는다는 것을 설명하려고 수학식으로 표현한 것이므로 깊게 이해하지 않아도 된다.

$$
\begin{aligned}
EMA(n) &= k \times V(n) + (1-k)EMA(n-1) \\
&= k \times V(n) + (1-k)(k \times V(n-1)+(1-k)EMA(n-2)) \\
&= k \times V(n) + k(1-k)V(n-1) + (1-k)^2EMA(n-2) \\
&= k \times V(n) + k(1-k)V(n-1) + k(1-k)^2V(n-2) + (1-k)^3EMA(n-3) \\
&= k \times V(n) + k(1-k)V(n-1) + k(1-k)^2V(n-2) + \cdots + k(1-k)^{n-2}V(2) + k(1-k)^{n-1}V(1)
\end{aligned}
$$

이 식에서 V(n)은 n번째 주가인 Value(n)을 간략히 표현한 것이다.

지수함수는 $f(x) = a^x$ 식으로 표현되는데 이 식에서 n이 지수로 표현되므로 EMA(n)은 지수함수가 된다.

MACD는 이평선과 원리가 비슷하다. 5평선과 20평선의 크로스에서 매매 시점을 알 수 있듯이 MACD도 단기 지수이동평균선이 장기 지수이동평균선과 크로스했을 때를 중요하게 생각한다. 즉, 이평선에서 평균값을 이용하여 이평선이 만들어졌듯이 MACD에서는 지수이동평균 공식을 이용하여 이동평균선이 만들어져서 사용된다. MACD 값을 구하는 식은 다음과 같다.

MACD = 단기 지수이동평균 - 장기 지수이동평균

이 식에서 MACD가 양수이면 단기 지수이동평균이 장기 지수이동평균 위에 존재하고, 음수면 아래에 존재한다. 예를 들어, 이평선에서 5평선과 20평선의 크로스를 골든 크로스와 데드 크로스를 구분하였듯이 5일의 단기 지수이동평균과 20일의 장기 지수이동평균을 비교하여 MACD를 생각하면 된다.

MACD는 매매 시점을 알기 위해서 시그널Signal이라는 것을 추가한다. MACD 시그널은 앞의 식에서 나온 MACD 값을 다시 지수이동평균식에 이용하여 계산된다. 다음 식은 MACD 시그널을 구하는 공식으로 MACD 오실레이터라고 할 수 있다.

Signal = n일의 MACD 지수이동평균

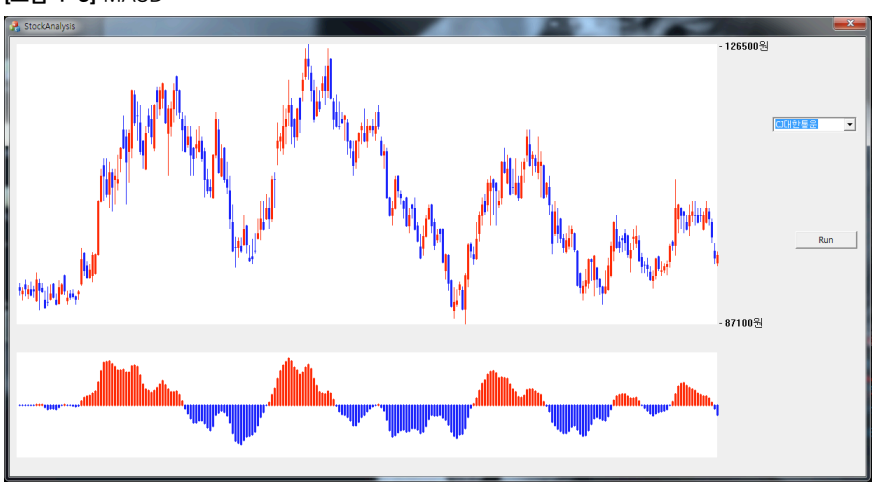
일반적으로 증권회사에서 제공하는 MACD의 지수이동평균기간은 다음과 같다.

MACD = 12일 지수이동평균 - 26일 지수이동평균
Signal = MACD의 9일 지수이동평균

프로그램으로 직접 구현하면 지수이동평균 기간을 바꾸어 다양한 결과들을 만들 수 있다. 프로그램을 직접 구현한다는 것은 주식 시장에서 큰 힘을 가진다. 일반 사람들은 증권회사에서 주는 내용만 보고 따라갈 수밖에 없지만, 프로그램을 직접 구현하면 주식 데이터를 여러 가지 방법으로 응용할 수 있기 때문이다. 이평선에서 일반적으로 사용하는 5, 20, 60, 120평선 이외의 이평선을 우리가 만들 수 있듯이 MACD에서도 12, 26, 9 이외의 다양한 값들로 오실레이터에 변화를 줄 수 있다.

지금까지의 내용은 수학적인 부분이 많아서 MACD가 무엇인지 쉽게 이해되지 않을 수 있다. MACD 오실레이터인 MACD 시그널을 먼저 살펴보자.

[그림 4-3] MACD



[그림 4-3]의 아랫부분은 MACD 오실레이터를 그린 것이다. 여기서 붉은색 구간은 상승 추세고 파란색 구간에서는 하락함을 알 수 있다. 이전에 구현한 이평선 오

실레이터와 마찬가지로 MACD 오실레이터는 파란색에서 붉은색으로 바뀔 때 매수하고, 붉은색에서 파란색으로 바뀔 때 매도한다.

4.2 MACD 구현

지금까지의 내용을 이해한 독자는 MACD를 쉽게 구현할 수 있을 것이다. MACD의 알고리즘을 하나의 함수에서만 구현하면 된다. 먼저 MACD 구조체를 만들고, 계산하여 결과값을 구조체 안에 저장한 후 화면에 그린다. 프로그래밍하다 보면 난수를 생성하기 위하여 씨앗이 되는 시드Seed 값을 넣어주게 된다. 그래서 MACD에서도 지수이동평균 기간의 이름에 'Seed'를 넣어 단기 이동평균은 'MACD_SEED_EMA_1'로, 장기 이동평균은 'MACD_SEED_EMA_2'라는 이름을 사용한다. 또한, 시그널을 위한 지수이동평균 기간은 상수로 MACD_SEED_SIGNAL을 사용하며, 이것은 Stock.h에 정의한다. [코드 4-1]은 MACD를 위한 시드와 구조체의 정의다.

[코드 4-1] MACD 시드 및 구조체 정의하기(Stock.h)

```
01  #define MACD_SEED_EMA_1     12
02  #define MACD_SEED_EMA_2     26
03  #define MACD_SEED_SIGNAL    9

    struct MACD {
04      long ema1;
05      long ema2;
06      int macd;
07      int signal;
    };

    struct Company {
        CString strJongMok, strName;
        int quantity;
```

```
        Data data[MAX_DATA];
08      MACD macd[MAX_DATA];
    };
```

01 단기 지수이동평균 기간을 정의한다.
02 장기 지수이동평균기 간을 정의한다.
03 시그널 지수이동평균 기간을 정의한다.
04 MACD 연산 후, 단기 지수이동평균 값을 저장한다.
05 MACD 연산 후, 장기 지수이동평균 값을 저장한다.
06 MACD 연산 후, MACD 값을 저장한다.
07 MACD 연산 후, 시그널 값을 저장한다.
08 MACD 배열로 데이터 개수만큼의 크기를 갖는다.

[코드 4-2]는 MACD 알고리즘을 'MakeMACD()' 함수에 구현하였다. 상수 'K'를 정의하고 단기 지수이동평균과 장기 지수이동평균을 계산한 후, MACD와 시그널 값을 구하는 연산을 차례대로 구현하였다.

[코드 4-2] MACD 계산(Stock.cpp)

```
    void CStock::Run()
    {
        ReadDataFromFile();
        makeSelectedCompanyFromAllCompany();
        MakeMACD();
    }

    ##### 생략 #####

    void CStock::MakeMACD()
    {
        int i, j;

01      float K1 = (float)(2.0 / (MACD_SEED_EMA_1 + 1));
```

```
        float K2 = (float)(2.0 / (MACD_SEED_EMA_2 + 1));
        float K3 = (float)(2.0 / (MACD_SEED_SIGNAL + 1));

        for(i=0; i< allCompanies.quantity; i++) {

            Company *company = &allCompanies.companies[i];
            int quantity = company->quantity;

            if(company->quantity > 0) {
02              company->macd[company->quantity-1].ema1 = company->data[company->
    quantity-1].lastVal;
                company->macd[company->quantity-1].ema2 = company->data[company->
    quantity-1].lastVal;
                company->macd[company->quantity-1].macd = 0;
                company->macd[company->quantity-1].signal = 0;

03              for(j=company->quantity-2; j>=0; j--) {
04                  company->macd[j].ema1 = (long)((K1 * company->
    data[j].lastVal) + ((1.0 - K1) * company->macd[j+1].ema1));
05                  company->macd[j].ema2 = (long)((K2 * company->
    data[j].lastVal) + ((1.0 - K2) * company->macd[j+1].ema2));
06                  company->macd[j].macd = company->macd[j].ema1  - company->
    macd[j].ema2;
07                  company->macd[j].signal = (int)((K3 * company->macd[j].macd)
    + ((1.0 - K3) * company->macd[j+1].signal));
                }
            }
        }
    }
```

---

01 상수 'K1', 'K2', 'K3'의 값을 정의한다. [코드 4-1]의 정의에 따라 K1=2/(12 +1)이고 K2=2/(26+1)이며 K3=2/(9+1)이 된다.

02 시작값을 정의한다. 데이터는 뒤로 갈수록 오래된 값이므로 가장 마지막 값으로 'ema1', 'ema2', 'macd', 'signal'을 정의한다.

03 가장 오래된 데이터의 바로 다음 데이터부터 차례대로 MACD연산이 이루어진다. 데이터의 앞쪽이 가장 최근 데이터이므로 'j'가 1씩 감소하게 구현한다.

04 단기 지수이동평균을 계산한다. 지수이동평균식은 'EMA(n) = k × V(n) + (1−k)EMA(n−1)'이며, 식에 맞게 값을 대입한다.

05 장기 지수이동평균을 계산한다.

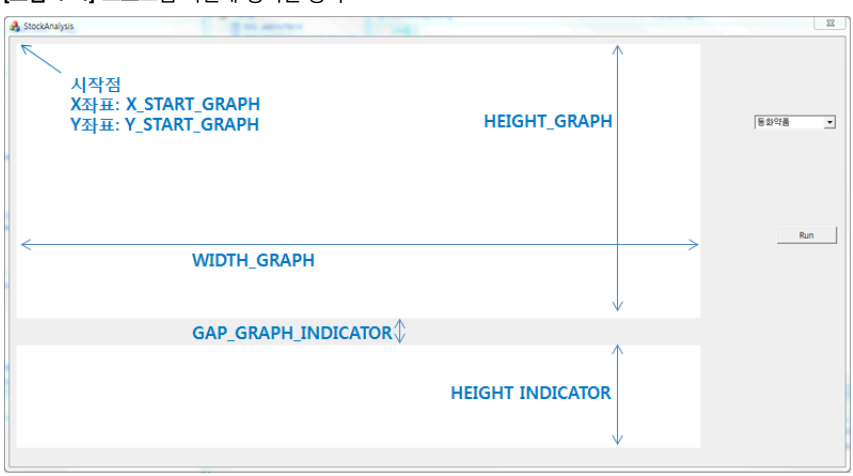
06 macd 값은 단기 지수이동평균에서 장기 지수이동평균을 뺀 것이다.

07 macd 값으로 지수이동평균을 계산하여 시그널 값을 얻는다.

'makeMACD()' 함수에서 모든 MACD의 데이터가 계산되어 구조체에 저장된다. 이번 장에서 가장 중요한 부분이지만, 아주 간단하게 구현된 것을 알 수 있다. 실제 프로그램의 핵심 부분은 간단하지만, MFC 프로그램에서는 화면에 그리기 위한 추가 작업이 많이 발생한다. 이후부터는 화면에 MACD를 그리기 위한 작업이다.

Graph.h 파일에 MACD를 그리는 영역으로 'HEIGHT_INDICATOR'가 정의된다. Graph.h에 정의된 영역을 화면에 표시하면 [그림 4-4]와 같다. [코드 4-3]에서는 MACD를 그리기 위해 'macdOscillator'를 'PointArray'로 선언한다.

[그림 4-4] 프로그램 화면에 정의된 영역



[코드 4-3] 화면에 MACD를 그리기 위한 점의 배열(Graph.h)

```
#define X_START_GRAPH            10
#define Y_START_GRAPH            10

#define WIDTH_GRAPH              1000
#define HEIGHT_GRAPH             400

#define HEIGHT_INDICATOR         150

#define GAP_GRAPH_INDICATOR      40

#define WIDTH_LINE               1
#define WIDTH_THICK_LINE         3

struct PointData {
     PointArray startVal;
     PointArray highVal;
     PointArray lowVal;
     PointArray lastVal;
     PointArray volume;

01   PointArray macdOscillator;
};
```

01 MACD의 시그널 데이터를 화면에 그리기 위해 계산된 점의 위치는 'macdOscillator'라는 'PointArray'에 저장된다.

[코드 4-4]는 MACD의 시그널 데이터를 화면의 점으로 바꾸어주는 연산이다. 시그널 값은 양수와 음수로 이루어져 있고, 화면의 오실레이터 영역 중간에 한 줄로(가로로) 0 값을 그려야 하므로 이것이 Y 축의 원점이란 의미에서 'originY'라는 변수를 사용한다. 또한, 시그널의 최대값을 알아내어 화면 밖으로 점이 나가지 않게 하려고 'maxY'와 'maxOscillator' 변수를 사용한다. 이 원점의 값인 0을 기준으로 위쪽은 붉은색 그래프가 그려지고, 아래쪽은 파란색 그래프가 그려진다.

[코드 4-4] 화면에 MACD를 그리기 위한 위치 계산하기(Graph.cpp)

```
void CGraph::GetDataForChart(Company* company, PointData* ptData)
{
    ##### 생략 #####
    x_start = X_START_GRAPH + WIDTH_GRAPH;
01  int originY = Y_START_GRAPH + HEIGHT_GRAPH + GAP_GRAPH_INDICATOR +
(int)(HEIGHT_INDICATOR/2);
02  int maxY = (int)((HEIGHT_INDICATOR/2) * 0.9);

    int maxOscillator = 0;
03  for(i=0; i<company->quantity; i++) {
        if(abs(company->macd[i].signal) > maxOscillator)maxOscillator =
abs(company->macd[i].signal);
    }

    if(maxOscillator != 0) {
        ptData->macdOscillator.quantity = company->quantity;
        for(i=0; i<ptData->macdOscillator.quantity; i++) {
            ptData->macdOscillator.point[i].X = x_start - i * interval;
04          ptData->macdOscillator.point[i].Y = originY  - (int)((maxY *
company->macd[i].signal) / maxOscillator) ;
        }
    }
}
```

01 시그널은 양수와 음수의 값을 가진다. 화면 중간에 0 값을 그리기 위하여 Y 축에서 시그널이 0인 위치가 'originY'에 계산된다. 'HEIGHT_INDICATOR'가 높이이므로 이 값을 2로 나눈 것이 가운데 0선이라고 볼 수 있다.

02 MACD의 시그널은 'INDICATOR'의 90% 안에 그려진다. 영역에 꽉 차게 그리지 않으려고 'maxY'는 높이에 0.9를 곱한다.

03 MACD의 시그널 최대값을 'maxOscillator'에 저장한다.

04 비율 계산을 이용하여 점의 실제 위치를 계산한다([그림 2-14] 비율 계산 참조).

지금까지는 화면에 그리는 점의 위치를 계산했다. 이제 화면에 MACD를 그리는 부분을 살펴보자. [코드 4-5]는 MACD를 화면 아랫부분에 그려주는 코드다.

[코드 4-5] 화면에 MACD 오실레이터 그리기(StockAnalysisDlg.cpp)

```
void CStockAnalysisDlg::DrawGraph()
{

    ##### 생략 #####

    CBrush whiteBrush(RGB(255,255,255));
    RECT rect2 = {X_START_GRAPH,
        Y_START_GRAPH+HEIGHT_GRAPH+GAP_GRAPH_ INDICATOR,
        X_START_GRAPH+WIDTH_GRAPH,
        Y_START_GRAPH+HEIGHT_GRAPH+GAP_GRAPH_INDICATOR+HEIGHT_INDICATOR};
01  dc.FillRect(&rect2, &whiteBrush);

02  int originY = Y_START_GRAPH + HEIGHT_GRAPH + GAP_GRAPH_INDICATOR +
(int)(HEIGHT_INDICATOR/2);

03  for(i=0; i<stock->ptrCompany->quantity; i++) {
        if(ptData->macdOscillator.point[i].Y < originY) {
            dc.SelectObject(redPenThick);
        }
        else{
            dc.SelectObject(bluePenThick);
        }
        dc.MoveTo(ptData->macdOscillator.point[i].X, originY);
        dc.LineTo(ptData->macdOscillator.point[i].X, ptData->
macdOscillator.point[i].Y);
    }
}
```

01 MACD를 그리기 위한 영역을 하얀색 사각형으로 만든다.

02 MACD 시그널은 양수와 음수의 값으로 이루어져 있으며, 시그널을 그리는 영역의 가운데를 0의 값으로 정한다. 'originY'는 Y 축에 대한 시그널의 원점이라고 보면 된다.

03 MACD 시그널의 좌표를 화면에 그린다. 화면에서 Y 축 아래로 내려갈수록 Y 값이 크므로 originY(원점)보다 작으면 빨간색, 크면 파란색으로 선을 그린다.

MACD의 공식만 보면 좀 어렵다고 느낄 수 있다. 하지만 좀 더 깊이 있게 분석하면 이평선의 크로스 개념과 크게 다르지 않음을 알 수 있다. 예를 들어, 12평선에서 26평선을 뺀 값을 MACD라고 생각할 수 있다. 단지 이평선을 구하는 계산을 지수이동평균 식으로 계산한 것이라 보면 된다. 필자도 MACD 알고리즘을 처음 접했을 때 참 어렵다고 느꼈다. 프로그램을 만들 때도 그저 알고리즘에 맞게 코드를 구현하였다. 그러나 MACD를 설명하려고 조금 더 깊게 공부한 후부터는 이평선과 크게 다르지 않다는 것을 알게 되었다. 어쩌면 현재 추세를 잘 반영할 수 있는 새로운 이동평균선을 만들 수 있다면 새로운 보조 지표가 만들어질 수 있을 것이다.

4.3 MACD 오실레이터

MACD 오실레이터는 MACD 시그널을 의미한다. 이전 장에서 다룬 이평선의 예를 들면, 5평선이 20평선 위에 있을 때는 시그널이 양수이고, 5평선이 20평선 아래 있을 때는 시그널이 음수라고 해 보자. 시그널이 음수에서 양수로 바뀌면 골든 크로스가 이루어진 것이고, 시그널이 양수에서 음수로 바뀌면 데드 크로스가 발생한 것이다. MACD는 이평선 계산을 단지 지수이동평균 식을 이용하여 계산한 것이므로 두 개의 이동평균을 뺄셈 연산으로 결과값을 만들어서 한 번 더 지수이동평균으로 완만하게 만들었다고 보면 된다.

이제부터 다룰 내용은 이평선 오실레이터에서 설명한 것과 거의 같다. 단지 사고파는 조건만 MACD 시그널을 이용했을 뿐이다.

[코드 4-6]은 MACD 지수이동평균 기간을 정의한 것이다. 증권회사에서 단기 지수이동평균 기간을 12, 장기 지수이동평균 기간을 26, 시그널 지수이동평균 기간

을 9로 정하여 사용하므로 여기서도 12, 26, 9로 정의해서 오실레이터 검증한다.

[코드 4-6] MACD(12-26-9) 지수이동평균기간 및 매매 시 차감률 정의하기(Stock.h)

```
#define MACD_SEED_EMA_1          12
#define MACD_SEED_EMA_2          26
#define MACD_SEED_SIGNAL         9

#define CHARGE_BUY               0.00015
#define CHARGE_SELL              0.00315
#define CHARGE_TRADE             0.005
```

[코드 4-7]은 MACD 오실레이터 매매를 구현한 것으로, [코드 3-9]에서 이평선 오실레이터를 구현한 것과 코드가 같다. 단지 사고파는 조건에서만 MACD 시그널을 사용한다([코드 4-7]의 회색 부분). 이평선 오실레이터에서 코드를 자세히 설명하였으므로 여기서는 코드 설명을 생략하고 변화된 조건만 설명한다.

[코드 4-7] MACD 매매 구현 및 결과 출력하기(Stock.cpp)

```
void CStock::Run()
{
    ReadDataFromFile();
    makeMACD();
    simulateTrade();
}

void CStock::simulateTrade()
{
    int balance = 100000000;
    int prevBalance;
    int countWin = 0;
    int countLose = 0;
    int charge;
```

```
int chargeTradeCenter = 0;

selectedCompanies.quantity = 0;

int i, j;

for(i=0; i<allCompanies.quantity; i++) {
    Company *company = &allCompanies.companies[i];
    int quantity = company->quantity;

    prevBalance = balance;
    int isBought = false;

    for(j=quantity-2; j>0; j--) {
        if(!isBought) {
01          if(company->macd[j].signal > 0) {
                charge = (int)(company->data[j].lastVal * CHARGE_BUY);
                balance -= charge;
                chargeTradeCenter += charge;
                charge = (int)(company->data[j].lastVal * CHARGE_TRADE);
                balance -= charge;
                balance -= company->data[j].lastVal;
                isBought = true;
            }
        } else {
02          if(company->macd[j].signal < 0) {
                charge = (int)(company->data[j].lastVal * CHARGE_SELL);
                balance -= charge;
                chargeTradeCenter += charge;
                charge = (int)(company->data[j].lastVal * CHARGE_TRADE);
                balance -= charge;
                balance += company->data[j].lastVal;
                isBought = false;
            }
        }
```

```
            }
            if(isBought) {
                charge = (int)(company->data[0].lastVal * CHARGE_SELL);
                balance -= charge;
                chargeTradeCenter += charge;
                charge = (int)(company->data[0].lastVal * CHARGE_TRADE);
                balance -= charge;
                balance += company->data[0].lastVal;
                isBought = false;
            }

            if(balance > prevBalance) {
                countWin += 1;
                selectedCompanies.companies[selectedCompanies.quantity] = company;
                selectedCompanies.quantity += 1;
            } else if(balance < prevBalance) {
                countLose += 1;
            }
        }

        CString str;
        str.AppendFormat(_T("Win 종목 개수: %d \n"), countWin);
        str.AppendFormat(_T("Lose 종목 개수: %d \n"), countLose);
        str.AppendFormat(_T("Balance : %d \n"), balance);
        str.AppendFormat(_T("수수료: %d \n"), chargeTradeCenter);
        ::AfxMessageBox(str);
    }
```

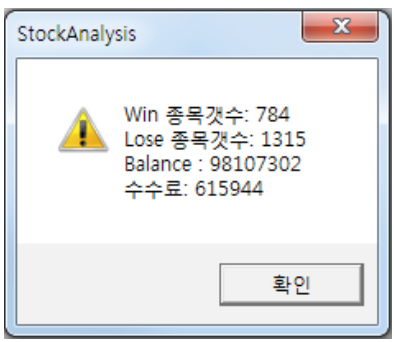
---

01 MACD 시그널이 양수일 때 매수한다. 시작 부분이므로 양수일 때만 조건으로 넣는다. 매수가 이루어지고 난 후에는 시그널이 항상 양수이므로 02와 같이 음수일 때는 매도한다.

02 MACD 시그널이 양수일 때 매도하고 이후에는 항상 양수여서 음수로 변했을 때는 데드 크로스가 발생하므로 바로 전 값이 양수인가를 조건에서 확인하지 않아도 된다. 따라서 매수 시점에서는 MACD 시그널이 음수인지만을 확인한다.

[그림 4-5]는 프로그램의 실행 결과를 보여주는데, MACD 자동매매에서 승률은 그렇게 높지 않다. 보조 지표 하나를 이해한 사람은 보조 지표에 대한 확신을 가진다. 문제는 두 번 지고 한 번 이겨도 자신이 사용하는 보조 지표가 매매 시점을 잘 보여준다고 착각할 수 있다는 것이다. 마치 도박에서 진 것은 잊고, 이긴 것만을 기억하며 자신이 도박을 잘한다고 착각하는 것과 비슷하다. 보조 지표는 단지 보조 지표일 뿐이고, 이것을 응용하여 자신만의 알고리즘을 만드는 것이 좋다.[02]

[그림 4-5] MACD(12-26-9) 매매 결과



MACD에서 변화를 줄 수 있는 것은 지수이동평균을 계산하는 기간이다. 프로그램으로 이 지수이동평균 기간을 마음대로 변경해서 여러 가지 결과를 만들어 보는 것이다. [코드 4-8]에서는 지수이동평균기간을 5, 10, 3으로 변경한다. 'Stock.h' 파일에서 숫자만 변경하여 컴파일하고 실행하면 된다.

[코드 4-8] MACD(5-10-3) 지수이동평균 기간 정의하기(Stock.h)

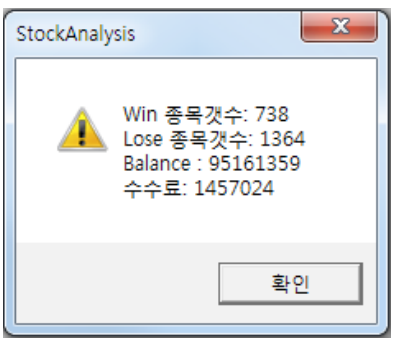
```
#define MACD_SEED_EMA_1          5
#define MACD_SEED_EMA_2          10
```

02 이기는 알고리즘을 공개할 수 있는 사람이 얼마나 될까? 공개된 보조 지표 중 승률이 괜찮은 것이 거의 없다. 매매 수수료의 매력을 아는 증권회사 입장에서는 많은 사람이 자주 주식 매매 하기를 바라는 마음에서 다양한 보조 지표들을 만들어 공유하는 것이 아닐까 하는 생각이다.

```
#define MACD_SEED_SIGNAL        3

#define CHARGE_BUY              0.00015
#define CHARGE_SELL             0.00315
#define CHARGE_TRADE            0.005
```

[그림 4-6]은 실행 결과를 보여준다.

[그림 4-6] MACD(5-10-3) 매매 결과

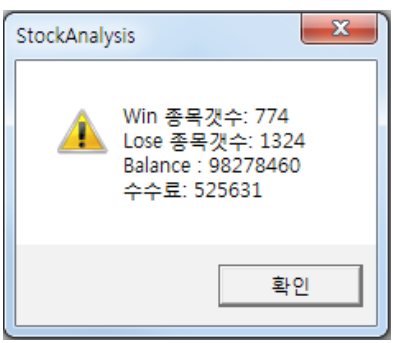


[그림 4-6] MACD(5-10-3) 결과와 [그림 4-5] MACD(12-26-9) 결과를 비교해 보자. MACD(5-10-3)에서 수수료가 훨씬 많이 발생한 것으로 보아 매매 횟수가 많았음을 알 수 있다. 승률은 좀 더 낮고, 손실도 많이 발생했다. 지수이동평균 기간이 좀 더 길어진다면 어떻게 될까? [그림 4-7]은 [코드 4-9]에서 지수이동평균 기간을 15, 30, 11로 변경하여 실행한 결과를 보여준다. 지수이동평균 기간이 길어졌다고 결과가 특별히 좋아지진 않았다. MACD(12-26-9)와 비교하여 MACD(15-30-11)의 수수료가 적으므로 매매 횟수는 줄어들었으나 승률은 좀 더 낮다. 손실은 적긴 하지만, 이것은 매매 횟수가 적어서 줄어든 것일 수 있다. 물론, 프로그램에서 매매 수수료나 세금에 대한 공제가 없다면 다른 결과가 나왔을 수 있다. 그러나 자동매매 시뮬레이션을 할 때는 최대한 안 좋은 조건을 대입하면서 프로그래밍하기를 권한다.

[코드 4-9] MACD(15-30-11) 지수이동평균기간 정의하기(Stock.h)

```
#define MACD_SEED_EMA_1          15
#define MACD_SEED_EMA_2          30
#define MACD_SEED_SIGNAL         11

#define CHARGE_BUY               0.00015
#define CHARGE_SELL              0.00315
#define CHARGE_TRADE             0.005
```

[그림 4-7] MACD(15-30-11) 매매 결과



모든 보조 지표는 후행성이라고 보면 되는데, 필자는 MACD가 이평선과 비슷하다고 생각한다. MACD를 이용해 분석하면서 시그널이 양수로 변할 때 매수하고 음수로 변할 때 매도하는 것이 일반적이지만, 프로그램을 만들 때는 매매 시점을 다양하게 변경할 수 있다. 예를 들어, 시그널이 계속 낮아지다가 높아질 때 매수하고 계속 높아지다가 낮아질 때 매도하는 것도 한 방법일 수 있으며, 프로그램으로 적용해서 검증 작업을 거치는 것도 한 방법이다. 보조 지표를 많은 사람이 보고 있지만, 보조 지표를 만드는 법을 알게 되면 좀 더 응용할 힘이 생긴다.

바둑을 배울 때 단수가 높아지면 복기를 한다. 자신이 둔 것을 그대로 기억하고 똑같이 두면 상대방을 분석할 수 있다. 주식에도 복기라는 것이 있다. 사람들은 하루

동안의 주식 변동을 보면서 이익이나 손해가 난 것을 그냥 돈의 개념으로만 본다. 돈을 벌었다고 기뻐하고 잃었다고 화를 낸다. 자신이 왜 사고팔았는지 분석하지 않는다. 주식에서도 복기가 필요하다. 왜 내가 사고팔았는지 이유를 분석하고 자기 생각이 맞는지 틀리는지를 분석해야 한다. 이겨서 기분 좋아 술 한잔 하고, 졌다고 화가 나서 술 한잔 하는 것이 아니라, 주식 시장에 대한 자기 생각을 검증해야 한다.

MACD라는 보조 지표를 이제 알게 된 사람들은 모든 부분에서 훌륭하게 활용할 수 있는 보조 지표라고 착각할 것이다. 그러나 공개된 보조 지표로는 이길 수 없으며 대부분 종목에 완벽히 적용되지 않는다. MACD도 하나의 보조 지표로 받아들이고 응용해야 한다. 하나의 보조 지표만을 사용해서는 안 된다. 보조 지표마다 적용되는 것이 있고 아닌 것이 있다. 여러 개의 보조 지표를 분석하여 적절히 응용해야 한다.

물론, 필자도 좋은 알고리즘을 만드는 것이 힘들다는 것을 많이 느낀다. 옵션에 MACD만을 적용했을 때, 30%는 이겼으나 70%는 손해가 발생하였다. 보조 지표들은 선행성이 없고 모두 후행성이다. 그러나 주식에서 기본이 되는 이평선, MACD, 그리고 볼린저 밴드를 이해하는 것이 주식 프로그램을 만드는 입문 과정이라고 감히 단언할 수 있다. 주식 프로그램을 만들고 나서 필자가 느낀 것은 주식을 하지 않더라도 프로그래머라면 주식 프로그램이 무엇인지, 프로그래머로서 어떻게 접근을 하는 것이 좋은지를 일반인보다 깊이 있게 알아야 한다. 만들어 준 것만을 보는 것이 아니라, 어떻게 만드는지를 아는 순간 깊이 있게 이해하게 된다.

차트만 보고 분석하는 것에는 한계가 있다. 주가에 변화를 주는 요인은 상당히 많은데, 변화를 줄 수 있는 정확한 정보를 일반인은 알 수가 없다. 일반인에게까지 정보가 왔을 때 흔히 말하는 정보지(?)는 거짓이거나 이미 죽은 정보일 수 있다. 프로그램을 보면 차트는 거짓말을 안 한다. 주식에 참여한 사람들의 심리가 만들어내는 차트를 보고 좀 더 알고리즘으로 접근하는 것이다.

4.4 MFC Window Message

MFC 프로그래밍을 할 때 꼭 알아야 할 것 중 하나가 윈도우 메시지Window Message다. 윈도우 메시지는 운영체제에서 보내는 메시지를 받아서 처리하는 것이다. 운영체제 관점에서 모든 프로그램은 개별 프로세스일 뿐이다. 키보드로 타이핑하는 것과 마우스 버튼을 클릭하는 모든 동작은 특정 프로세스에 메시지로 전달해야 한다.

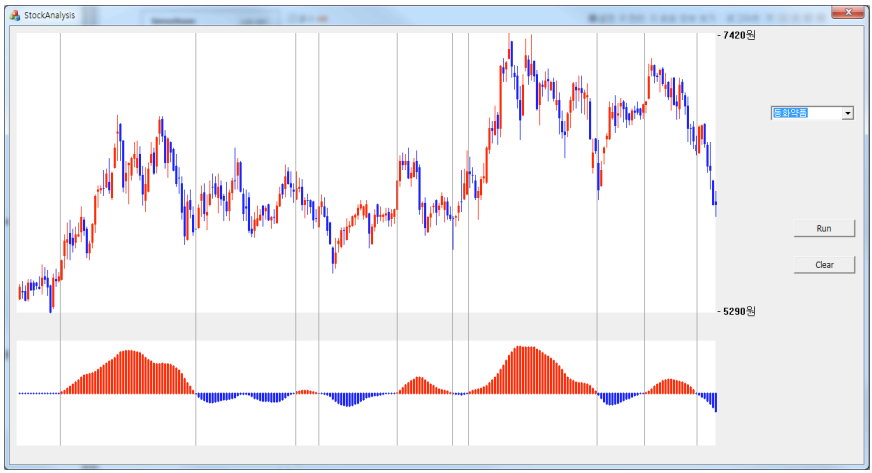
운영체제에는 각 프로세스를 위한 메시지 큐Message Queue라는 것이 있다. 각 이벤트Event를 메시지 큐에 넣고, 프로세스는 메시지 큐에 있는 것을 차례대로 꺼내어 처리한다. 컴퓨터의 속도가 워낙 빨라서 모든 동작이 동시에 이루어진다고 생각하지만, 마우스를 이동하는 것 자체도 운영체제는 정신 없이 처리하고 있다고 봐야 한다.

여기서는 운영체제에 대해서 깊게 이해할 필요는 없다. 마우스를 클릭할 때 프로그램에서 어떻게 처리하는지, 마우스 왼쪽 버튼 클릭 시 어떻게 코드로 구현하는지를 알고 이해하면 된다. 다른 윈도우 메시지도 이와 같은 방법으로 구현한다.

마우스 왼쪽 버튼을 눌렀을 때 세로 선 한 개를 그리는 동작을 구현해 보자. 차트를 보면서 MACD 시그널의 매매 시점에 마우스를 갖다 대고 버튼을 누르면 세로 선을 그리도록 한다. 차트를 눈으로 보는 것이 아니라 선을 이용해서 정확한 위치를 보기 위해서다.

[그림 4-8]은 마우스 버튼을 클릭했을 때 그려지는 회색 선들을 보여준다. 매매 시점 선 그리기는 프로그램에서 자동으로 만드는 것이 좋다. 매매가 이루어진 데이터의 인덱스를 저장하고 있다가 일괄적으로 선을 그려주면 되지만, 프로그램을 간단히 만드는 것이 좋으므로 이 책에서는 생략한다.

[그림 4-8] 마우스 왼쪽 버튼을 클릭했을 때 세로선 그리기



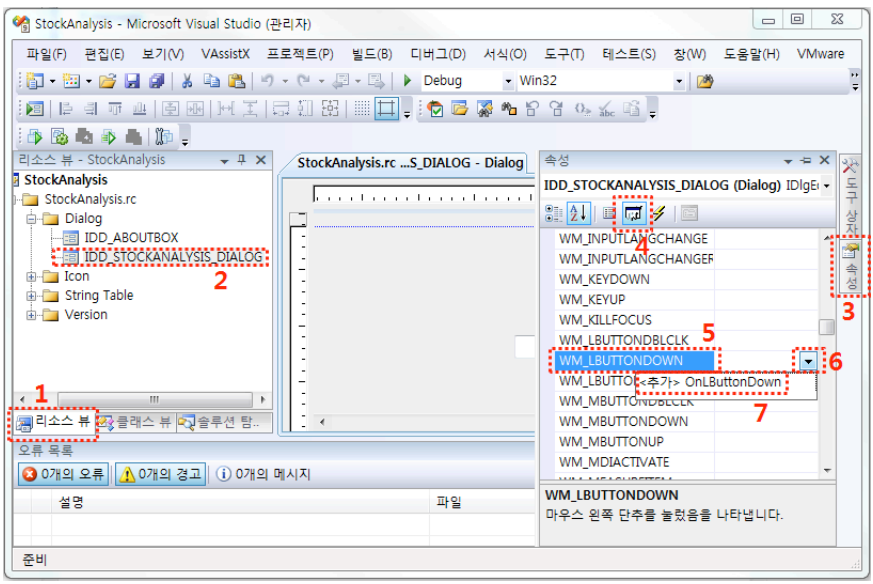
[그림 4-9]는 비주얼 스튜디오에서 윈도우 메시지를 처리하기 위한 함수를 만드는 방법을 보여준다. 먼저 프로그램 화면 페이지를 연다. [그림 4-9]와 같이 '리소스 뷰(1)'를 선택하고, 'IDD_STOCKANALYSIS_DIALOG(2)'를 선택하여 다이얼로그 페이지를 연다. '속성(3)' 탭을 선택하여 메뉴 아이콘 중에서 '메시지(4)'를 클릭하면 모든 메시지 리스트가 보인다. 메시지의 이름 앞에는 'WM_'이 붙어 있어서 윈도우 메시지의 약어임을 알 수 있다.

리스트에 있는 윈도우 메시지 중 'WM_LBUTTONDOWN'이 마우스 왼쪽 버튼을 눌렀을 때 발생되는 메시지다. 이름에서 알 수 있듯이 마우스 왼쪽 버튼(L^{eft}Button)이 아래로(Down) 눌렀을 때 전달되는 윈도우 메시지(WM_)를 뜻한다. 마찬가지로 마우스 오른쪽 버튼을 눌렀을 때 전달되는 메시지는 'WM_RBUTTONDOWN'이고, 더블 클릭했을 때 발생되는 메시지는 'WM_RBUTTONDBLCLK'(DBL^{Double}CLK^{Click})이다. 또한, 키보드를 눌렀을 때 메시지는 'WM_KEYDOWN'이고, 키보드를 눌렀을 때 나오는 메시지는 'WM_KEYUP'

다. 키보드를 누르면 운영체제에서 'WM_KEYDOWN'과 'WM_KEYUP' 메시지들을 프로그램으로 연속해서 보낸다.

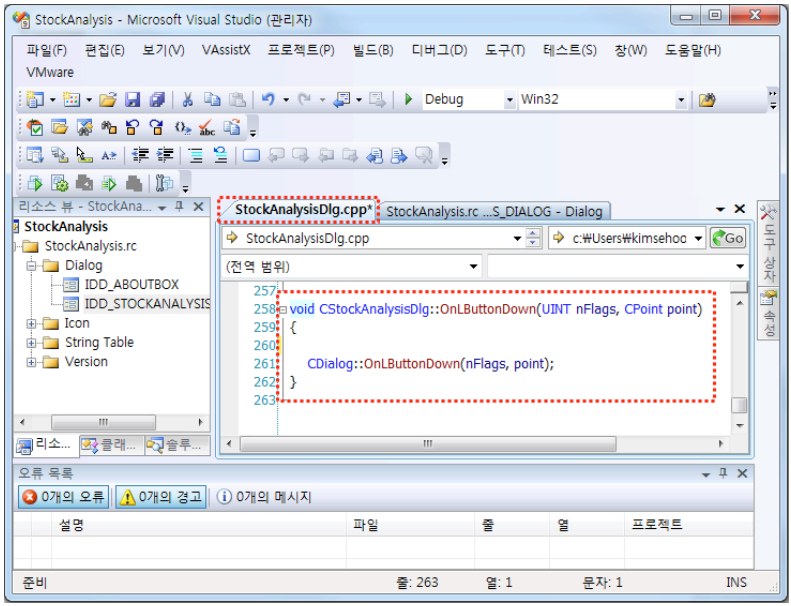
마우스 왼쪽 버튼을 눌렀을 때 메시지를 만들려면 'WM_LBUTTONDOWN(5)'을 선택한 후 오른쪽 리스트를 연다(6). '〈추가〉 OnLButtonDown(7)' 항목이 보이면 이것을 선택하여 새로운 메시지 이벤트를 추가한다.

[그림 4-9] 윈도우 메시지 만들기



[그림 4-10]은 메시지가 추가되고 난 후의 화면을 보여준다. 'StockAnalysisDlg.cpp' 파일에 'OnLButtonDown( )' 함수가 자동으로 추가되었다. 여기에 마우스 왼쪽 버튼을 눌렀을 때 수행해야 하는 코드를 추가한다.

[그림 4-10] 윈도우 메시지 선택 시 생성되는 코드



이때 'StockAnalysisDlg.h' 파일에는 'OnLButtonDown()' 함수가 선언되고, 'StockAnalysisDlg.cpp' 파일의 'Message Map'에는 'ON_WM_LBUTTONDOWN()' 함수가 자동으로 추가된다.

[StockAnalysisDlg.h]

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

[StockAnalysisDlg.cpp]

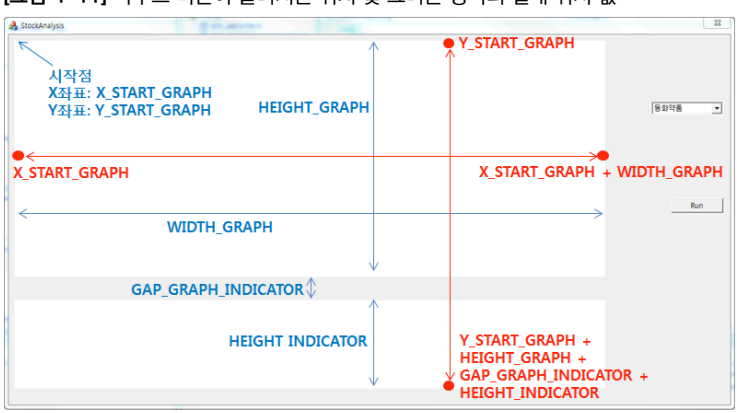
```
BEGIN_MESSAGE_MAP(CStockAnalysisDlg, CDialog)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
```

```
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_BTN_RUN, &CStockAnalysisDlg::OnBnClickedBtnRun)
    ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

마우스 왼쪽 버튼을 누르면 'OnLButtonDown(UINT nFlags, Cpoint point)' 함수가 실행된다. 여기서 중요한 것은 매개 변수로 받는 포인트 변수다. 포인트 변수는 프로그램 화면에서 버튼이 클릭되었을 때 마우스 위치의 X, Y 좌표를 알려주는 'CPoint' 구조체다. 즉, 'point.x'는 X 좌표, 'point.y'는 Y 좌표의 위치 값을 갖는다. 마우스의 위치를 알 수 있으므로 화면의 특정 위치에서 버튼이 눌리면 화면에 선을 그린다.

[그림 4-11]은 프로그램 화면에서 점의 절대 위치를 표시한다. 이를 코드로 구현하려면 마우스의 절대 위치 값을 알아야 한다. [코드 4-10]은 'OnLButtonDown( )' 함수를 구현한 것으로 차트와 MACD 시그널이 그려지는 영역에서 마우스 버튼이 눌리면 회색 세로 선을 그린다.

[그림 4-11] 마우스 버튼이 눌러지는 위치 및 그리는 영역의 절대 위치 값



[코드 4-10] 마우스 왼쪽 버튼 눌렀을 때 실행되는 코드(StockAnalysisDlg.h)

```
    void CStockAnalysisDlg::OnLButtonDown(UINT nFlags, CPoint point)
    {
        CClientDC dc(this);
01      CPen pen( PS_SOLID, 0, RGB( 150, 150, 150 ) );
02      dc.SelectObject( &pen );

03      if(point.x > X_START_GRAPH && point.x < X_START_GRAPH+WIDTH_GRAPH) {
04          if(point.y > Y_START_GRAPH && point.y < Y_START_GRAPH+HEIGHT_GRAPH
    + GAP_GRAPH_INDICATOR+HEIGHT_INDICATOR) {
05              dc.MoveTo(point.x, Y_START_GRAPH);
06              dc.LineTo(point.x, Y_START_GRAPH+HEIGHT_GRAPH +
    GAP_GRAPH_INDICATOR+HEIGHT_INDICATOR);
            }
        }

        CDialog::OnLButtonDown(nFlags, point);
    }
```

01 회색 선을 그리기 위하여 펜을 생성한다. 펜은 회색(RGB(150,150,150)) 실선으로 선언한다.

02 화면에 선을 그리기 위하여 dc에서 생성한 펜을 선택한다.

03 마우스 위치의 X 좌표인 'point.x'가 화면에서 차트의 안쪽에 있는지를 확인한다.

04 마우스 위치의 Y 좌표인 'point.y'가 그림이 그려지는 영역 안쪽에 있는지를 판단하여 선을 그려준다.

05 세로선의 위쪽 끝 위치로 펜을 옮긴다.

06 세로선의 아래쪽 끝에 선을 그려준다.

지금까지 마우스 버튼을 눌렀을 때 선을 그리는 것을 구현하였는데 이 과정에서 프로그램에서 임의로 그린 선을 지울 필요가 있다. 그러나 선을 지우개로 지우듯이 없앨 수는 없다. 프로그램 화면은 2차원이므로 프로그램에서 선을 지운다는 것은 선 없는 그림을 새로 그리는 것이다. 즉, 마우스 버튼을 누르기 전의 화면을 다시 그리면 된다.

프로그램에서 'Clear' 버튼을 만들어주고 'Clear' 버튼을 눌렀을 때 실행되는 코드를 [코드 4-11]에서 구현하였다. 여기서 'RedrawWindow()'를 실행하여 화면을 다시 그리면 된다. 버튼을 만드는 것은 책의 첫 부분에서 설명하였으므로 여기서는 생략한다.

[코드 4-11] Clear 버튼 선택 시 새로 그리기(StockAnalysisDlg.h)

```
void CStockAnalysisDlg::OnBnClickedBtnClear()
{
	flagPaint = true;
	RedrawWindow();
}
```

윈도우 메시지는 MFC 프로그램에서 매우 중요하다. 지금까지 구현한 것을 이해한 독자라면 웬만한 윈도우 프로그램을 쉽게 만들 수 있다.

누군가 알려주는 프로그램은 기억이 오래 남지 않는다. 시간이 많이 걸리더라도 정보를 직접 찾아 보는 것이 좋다. 필자는 스스로 인터넷 프로그래머라고 이야기하곤 한다. 2000년도에 프로그램을 배울 때 한 친구가 말해준 뒤로 거의 모든 정보는 구글링을 통해서 얻고 있다.

프로그램 언어는 사용법이 정해져 있다. 남들이 정해 놓은 것을 그대로 사용해야 하므로 어떻게 사용하는지를 알아야 한다. 다른 사람의 것을 베끼는 것이 아니라, 사용법을 알아야 만들 수 있는 분야다. 더하여 무엇을 만들지를 정하고 자신이 필요한 기능을 어느 프로그램에선가 사용했다면, 구현을 위한 정보들이 인터넷에 있을 확률은 아주 높다. 물론, 깊이 있는 프로그램을 하기 위해서는 알고리즘이 중요하므로 공개된 여러 가지 알고리즘을 직접 구현해 보기를 권한다.

5 | Bollinger Band

볼린저 밴드Bollinger Band는 존 볼린저John Bollinger가 만든 주식 보조 지표다. 처음 볼린저 밴드를 봤을 때 그래프가 무엇을 의미하는지 전혀 알 수가 없었다. 『볼린저 밴드 투자기법』(존 볼린저 지음, 이레미디어, 2010)이라는 책을 읽고 직접 프로그램으로 구현하고 나서야 비로소 이해되었다.

볼린저 밴드는 수학의 표준편차Standard Deviation를 사용하여 주식을 분석하는 것으로, 주가가 일반적인 패턴보다 갑자기 큰 폭으로 하락하여 정해진 구간 이하로 내려갔을 때 매수하고, 급격하게 상승하여 정해진 구간 위로 올라갔을 때 매도하는 방식이다. 주가가 급격히 변동한 다음에는 조정이 일어나는 데 이것을 이용한 매매로 볼 수 있다. 필자는 프로그램을 만들면서 주가가 박스권[01]에서 움직일 때 볼린저 밴드는 괜찮은 보조 지표가 될 수 있다고 느꼈다. 존 볼린저는 자신의 책에서 거래량이 반영된 MACD와 볼린저 밴드를 함께 사용하여 투자하는 것이 좋다고 말했다. 이것은 MACD와 볼린저 밴드의 특징이 서로 다르므로 두 개를 적절히 이용하는 것이 좋다는 의미로 해석할 수 있다.

이 책에서는 거래량을 다루지 않는다. 하지만 이에 관한 책이 나왔듯이 주식 시장에서 거래량은 상당히 중요한 요소다. 주가는 소수의 사람이 적은 거래량으로도 조작(?)할 수 있지만, 거래량은 신빙성 있는 데이터이기 때문이다. 항상 그런 것은 아니지만, 아래에서 받치는 물량과 위에서 누르는 물량의 차이로 주식이 어떻게 될지 예측할 수 있어서 거래량을 이용하여 주가를 보는 사람들도 상당히 많다. 예를 들어, 아래에서 주식을 사겠다는 거래량이 위에서 주식을 팔겠다는 거래량보다 많다면 위로 올라가려는 황소의 힘이 아래로 누르는 곰의 힘보다 세다고 판단하여 주가 상승을 예측할 수도 있다. 물론, 위에서 갑자기 던지는 큰손의 힘이 존재한다면 주가는 내려갈 수밖에 없지만, 거래량이 많다는 것은 다수의 사람이 관심이 있음을

01 주가의 파동이 일정한 가격 폭 안에서만 움직일 때 그 가격의 범위를 말한다. (출처: 표준국어대사전)

보여주는 것이다.

5.1 볼린저 밴드 분석

볼린저 밴드에서 사용하는 표준편차는 데이터들이 평균값에서 얼마나 떨어져 있는지를 알아보는 식이다. 데이터들이 평균에서 멀리 떨어져 있으면 표준편차는 커지고, 평균에 가깝게 있으면 표준편차는 작아진다. [그림 5-1]은 표준편차를 구하는 공식으로 표준편차는 항상 양수가 된다. 주어진 데이터의 평균을 구하고, 각 데이터에서 평균을 빼서 편차를 구한다. 이 편차를 제곱한 값들의 평균이 분산되고, 분산의 제곱근이 표준편차가 된다. 표준편차는 데이터들이 평균에 얼마나 몰려 있는지를 수치화한 것이다.

[그림 5-1] 표준편차 공식

$$\sigma = \sqrt{E((X - E(X))^2} = \sqrt{E(X^2) - (E(X))^2}$$

(σ = 표준편차, E(X) = 모든 X의 평균, $E(X^2)$ = 모든 X^2의 평균, X = 입력된 값)

표준편차를 이해하려면 직접 계산해 보는 것이 좋다. 다음 두 종류의 데이터를 가지고 표준편차를 계산해 보자. 첫 번째 데이터는 {2, 4, 4, 4, 5, 5, 7, 9}이고, 두 번째 데이터는 {1, 3, 3, 3, 5, 5, 9, 11}이다. 첫 번째 데이터는 최소값이 2, 최대값이 9이고, 두 번째 데이터는 최소값이 1, 최대값이 11로, 첫 번째 데이터보다 간격이 더 넓다.

[표 5-1] 표준편차 계산

데이터	2, 4, 4, 4, 5, 5, 7, 9	1, 3, 3, 3, 5, 5, 9, 11
평균	$\frac{2+4+4+4+5+5+7+9}{8} = 5$	$\frac{1+3+3+3+5+5+9+1}{8} = 5$
제곱	$(2-5)^2 = (-3)^2 = 9$ $(4-5)^2 = (-1)^2 = 1$ $(4-5)^2 = (-1)^2 = 1$ $(4-5)^2 = (-1)^2 = 1$ $(5-5)^2 = (0)^2 = 0$ $(5-5)^2 = (0)^2 = 0$ $(7-5)^2 = (2)^2 = 4$ $(9-5)^2 = (4)^2 = 16$	$(1-5)^2 = (-4)^2 = 16$ $(3-5)^2 = (-2)^2 = 4$ $(3-5)^2 = (-2)^2 = 4$ $(3-5)^2 = (-2)^2 = 4$ $(5-5)^2 = (0)^2 = 0$ $(5-5)^2 = (0)^2 = 0$ $(9-5)^2 = (4)^2 = 16$ $(11-5)^2 = (6)^2 = 36$
제곱평균	$\frac{9+1+1+1+0+0+4+16}{8} = 4$	$\frac{16+4+4+4+0+0+16+36}{8} = 10$
표준편차	$\sqrt{4} = 2$	$\sqrt{10} = 3.16$

두 데이터를 비교해 보면 평균은 5로 같지만, 평균에 가까이 분포한 첫 번째 데이터는 표준편차가 2가 나오고 평균으로부터 좀 더 넓게 분포한 두 번째 데이터는 표준편차가 3.16이 나온다.

볼린저 밴드는 이처럼 계산되는 표준편차를 이용한다. 주가의 변동이 크지 않은 부분에서는 볼린저 밴드의 범위가 작고, 주가의 변동이 큰 부분에서는 볼린저 밴드의 범위가 커진다. 볼린저 밴드를 계산할 때 사용하는 데이터의 개수는 프로그래머가 임의로 정할 수 있다. 일반적으로 현재 데이터를 포함한 20개의 이전 데이터를 가지고 계산한다. 물론, 이 수치는 프로그래머가 다양하게 변경하면서 검증 작업을 할 수 있다.

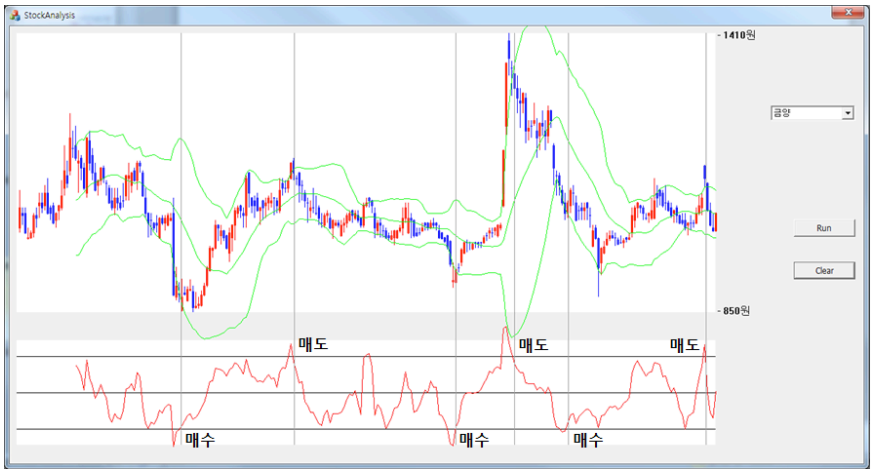
볼린저 밴드는 직접 보는 것이 이해하기 쉽다. [그림 5-2]를 살펴보자.

[그림 5-2] 볼린저 밴드와 표준편차

[그림 5-2]의 녹색 선이 볼린저 밴드를 나타낸다. 총 세 개의 선이 있는데 가운데 선은 20개 데이터의 평균값[02]이고, 위에 있는 선은 20평선에 표준편차를 더한 값, 가장 아래 있는 선은 20평선에서 표준편차를 뺀 값이다. 화면 아래에 표시된 붉은 선은 볼린저 밴드의 오실레이터를 그린 것이다. [그림 5-2]에서 1번으로 표시된 구간에서는 주가 변동이 크지 않아서 볼린저 밴드의 윗선과 아랫선의 범위가 넓지 않지만, 2번 구간에서는 주가 변동이 커지면서 볼린저 밴드의 폭이 상당히 넓어졌음을 알 수 있다.

02 볼린저 밴드의 평균값은 이평선의 20평선과 같은 개념으로 보면 된다.

[그림 5-3] 볼린저 밴드 매매 포인트



[그림 5-3]은 화면 아래에 그려진 볼린저 밴드의 오실레이터를 이용하여 매수와 매도를 구현한 것이다. 화면 아래에 가로로 그려진 세 개의 검은 선은 위에 그려진 세 개의 녹색 선을 일직선으로 평준화한 것이다. 위쪽에서 주가의 변화에 따라 다양하게 그려지는 그래프를 보기 쉽게 일직선으로 비율에 맞추어 일정한 간격으로 다시 그렸다.

볼린저 밴드 매매는 주가가 가장 아래 있는 선 이하로 내려갔을 때 사고, 가장 위에 있는 선 이상으로 올라갔을 때 파는 것이다. 즉, [그림 5-3]에 표시된 시점에 매수와 매도가 일어나면 훌륭한(?) 수익을 낼 수 있다. 물론 [그림 5-3]은 볼린저 밴드를 쉽게 이해할 수 있도록 가장 좋은 종목을 예로 든 것으로, 모든 종목에 적용했을 때는 수익이 나지 않을 때가 많았다.

주식의 보조 지표를 분석할 때는 알려진 방법으로만 분석하기보다는 응용하면 좋다. 예를 들어, 볼린저 밴드 오실레이터에서 가운데 검은 선은 20평선을 나타내므로 붉은 선이 가운데 검은 선 위에 있다면 주가가 20평선 위에 있는 것이고, 검은

선 아래에 있다면 주가가 20평선 아래에 있는 것이다.

매매 시점을 다양하게 바꾸어 검토해 보는 것도 좋다. 필자는 오실레이터가 아래의 선을 통과하여 내려가는 시점이 아니라 아래로 내려간 이후 다시 올라서 재통과하는 시점에 매수하였으며, 오실레이터가 위의 선을 통과한 후 올라갔다가 내려올 때 매도 시점을 잡았다. 이렇게 매매 시점을 잡았을 때 [그림 5-3]처럼 좋은 결과가 나왔다. 프로그램을 만드는 것은 다양한 방법으로 적용하고 단시간에 결과를 확인할 수 있다는 것이 가장 큰 장점이다.

5.2 볼린저 밴드 구현

볼린저 밴드에서 변경할 수 있는 값은 두 개다. 첫 번째가 평균을 만드는 개수이고, 두 번째가 표준편차의 폭을 늘리기 위하여 곱하는 값이다. 평균을 만드는 개수는 보통 20개를 사용한다[03]. 표준편차의 폭을 늘리는 값은 보통 2를 사용하는데, 표준편차에 2를 곱하여 볼린저 밴드에서 위와 아래의 두 선을 만들었다. 표준편차 폭의 수치를 다르게 하면 다른 결과가 나올 수 있다. 예를 들어, 2보다 작은 값을 사용하면 볼린저 밴드의 폭이 작아지므로 매매 신호가 더 자주 일어나고, 2보다 크면 아주 큰 폭의 변화가 생기지 않는 이상, 매매 신호는 거의 일어나지 않을 것이다. 증권회사의 HTS 프로그램에서도 변경할 수 있는 두 개의 값을 주로 20과 2로 사용한다. 어쩌면 볼린저 밴드에서 가장 이상적인 값일 수 있어서 여기서도 이 값들을 사용한다. 물론 다음 장의 오실레이터를 이용한 가상의 자동매매에서는 이 수치들을 변경하면서 프로그램으로 검증할 것이다.

이번 장에서 구현되는 코드도 이전 장과 비슷하다. 중요한 부분은 볼린저 밴드를 구현하는 'Stock.cpp' 파일의 'MakeBollinger( )' 함수다. 볼린저 밴드는 각각의 평균을 구하고 거기에 따른 표준편차를 계산하므로 이전 보조 지표보다 많은 연산

03 그래서 볼린저 밴드의 가운데 이평선을 20평선으로 그린다.

을 수행하지만, 빠르게 연산을 수행하여 체감 시간은 얼마 되지 않는다. [코드 5-1]에서는 볼린저 밴드를 만들기 위한 설정 값들과 구조체를 정의한다.

[코드 5-1] 볼린저 밴드 설정 값 및 구조체 정의하기(Stock.h)

```
01   #define BOLLINGER_MOVE_AVG        20
02   #define BOLLINGER_TIME_SD         2

03   struct Bollinger {
          long lineTop;
          long lineMiddle;
          long lineBottom;
          float oscillator;
     };

     struct Company {
          CString strJongMok, strName;
          int quantity;
          Data data[MAX_DATA];
04        Bollinger bollinger[MAX_DATA];
     };
```

01 볼린저 밴드의 가운데 이평선을 만드는 개수로, 평균과 표준편차 계산에서 사용된다.

02 표준편차의 폭을 조정하기 위해 표준편차에 곱해지는 값이다.

03 볼린저 밴드의 값을 저장하기 위한 구조체로, 'lineTop'은 맨 위의 선, 'lineMiddle'은 가운데 선, 'lineBottom'은 맨 아래의 선이며, 'oscillator'는 오실레이터를 화면 아랫부분에 그리기 위해서 계산된 값이다.

04 하나의 종목인 'Company'는 볼린저 밴드 구조체 변수인 'bollinger'를 가진다.

[코드 5-2]는 볼린저 밴드에서 가장 중요한 볼린저 밴드 수치를 구현한 것으로, 평균과 표준편차를 계산해서 볼린저 밴드 구조체 안에 저장한다. 표준편차 변수 이름을 'sd'로 정의한다[04].

04 표준편차는 영어로 Standard Deviation이다.

[코드 5-2] 볼린저 밴드 계산(Stock.cpp)

```
#include <math.h>

void CStock::Run()
{
    ReadDataFromFile();
    makeSelectedCompanyFromAllCompany();
01  MakeBollinger();
}

void CStock::MakeBollinger()
{
    int i, j, k;
    float sum;

    for(i=0; i<allCompanies.quantity; i++) {
        Company *company = &allCompanies.companies[i];
        int quantity = company->quantity;

02      for(j=company->quantity-BOLLINGER_MOVE_AVG; j>=0; j--) {
            sum = 0;
03          for(k=0; k<BOLLINGER_MOVE_AVG; k++) {
                sum += company->data[j+k].lastVal;
            }
            long average = (long)(sum / BOLLINGER_MOVE_AVG);

            sum = 0;
            for(k=0; k<BOLLINGER_MOVE_AVG; k++) {
04              sum += (float)(average - company->data[j+k].lastVal) *
(average - company->data[j+k].lastVal);
05              float sd = sqrt((float)sum / BOLLINGER_MOVE_AVG);

06              company->bollinger[j].lineMiddle = average;
07              company->bollinger[j].lineTop = company->
```

```
        bollinger[j].lineMiddle + (long)(BOLLINGER_TIME_SD * sd);
08                  company->bollinger[j].lineBottom = company->
        bollinger[j].lineMiddle  - (long)(BOLLINGER_TIME_SD * sd);
09                  company->bollinger[j].oscillator = (company->data[j].lastVal
         - company->bollinger[j].lineMiddle) / (BOLLINGER_TIME_SD * sd);
                }
            }
        }
    }
```

01 'Run' 버튼을 누르면 'Run( )' 함수 안의 'MakeBollinger( )' 함수가 실행되어 볼린저 밴드를 만든다.

02 볼린저 밴드는 이평선을 사용하므로 한 종목의 전체 데이터에서 처음 이평선을 만드는 데이터는 볼린저 밴드값을 갖지 않는다. 또한, 한 종목의 데이터는 'index'가 낮은 앞쪽 데이터가 가장 최근의 데이터이므로 'index'가 높은 오래된 데이터부터 계산하려고 'index j'를 하나씩 차례대로 차감한다.

03 'BOLLINGER_MOVE_AVG'가 20이므로 현재 값을 포함한 이전 값 20개로 평균을 계산하여 'average' 변수에 저장한다. 즉, 'average'는 20평선의 데이터다.

04 평균에서 개별 값들을 뺀 후 제곱하여 더한다.

05 제곱의 합을 개수인 'BOLLINGER_MOVE_AVG'로 나눈 후 제곱근을 구한다. 'sqrt( )'는 수학 함수로, 정의된 제곱근을 구한다. 코드 윗부분에 '#include <math.h>'를 선언하여 'math' 함수를 사용한다.

06 볼린저 밴드의 가운데 선인 'lineMiddle'은 이평선 값으로, 위에서 계산된 평균값 'average'를 넣는다.

07 볼린저 밴드 맨 위의 선인 'lineTop'은 이평선인 가운데 선에서 표준편차의 배수인 'BOOLINGER_TIME_SD'를 표준편차에 곱한 후 이평선에 더한다.

08 볼린저 밴드 맨 아래의 선인 'lineBottom'은 표준편차에 배수를 곱한 후 이평선에서 빼준다.

09 볼린저 밴드의 오실레이터는 종가('lastVal')에서 이평선('lineMiddle')을 빼주고 볼린저 밴드의 폭으로 나눠준다[05]. 즉, 종가가 볼린저 밴드의 맨 위의 선에 닿으면 1이고, 맨 아래의 선에 닿으면 -1이 되며, 맨 위의 선과 가운데 선 사이에 있으면 'oscillator'는 '0<oscillator<1'을 만족하며, 맨 위의 선보다 크면 1보다 큰 값을 가진다[06].

[코드 5-2]에서 오실레이터를 만드는 부분이 다소 어렵게 느껴질 수 있다. 일정한 기준으로 데이터를 가공하는 것은 주식 프로그램에서 아주 중요하다. 이 책에서 깊

05 2장에서 설명한 '비율 계산'을 적용한다.

06 볼린저 밴드의 오실레이터는 볼린저 밴드의 세 선을 기준으로 하여 주가 변동을 일정한 비율로 만들어 수치화한 것이다.

이 다루지는 않지만, 값이 다른 주식 데이터를 동일한 기준에서 종종 바라볼 필요가 있다. 주식 프로그램을 만드는 것은 프로그램만 잘하면 되는 것이 아니라 깊이 있는 분석을 할 줄 알아야 한다[07].

[코드 5-3]은 화면에 볼린저 밴드를 그리기 위한 점의 위치값을 가진 변수들을 'PointData' 구조체에 정의한다.

[코드 5-3] 화면에 볼린저 밴드를 그리기 위한 점 배열(Graph.h)

```
struct PointData {
        PointArray startVal;
        PointArray highVal;
        PointArray lowVal;
        PointArray lastVal;
        PointArray volume;

01      PointArray bollingerTop;
02      PointArray bollingerMiddle;
03      PointArray bollingerBottom;
04      PointArray bollingerOscillator;
};
```

01 볼린저 밴드 맨의 위쪽 선을 그리는 점 배열이다.
02 볼린저 밴드의 가운데 선을 그리는 점 배열이다.
03 볼린저 밴드의 아래쪽 선을 그리는 점 배열이다.
04 볼린저 밴드 오실레이터를 그리는 점 배열이다.

[코드 5-4]는 화면에 볼린저 밴드를 그리는 점의 위치를 계산한다. Stock.cpp 파일에서 만든 볼린저 밴드에서 비율 계산을 이용하여 화면의 위치값을 얻는다. 이평선과 MACD에서 구현한 방법과 비슷하므로 코드를 보고 쉽게 이해할 수 있을 것이다.

07 여담이지만 주식 분석을 하는 지인이 말하길 초보자가 주식을 알려면 하나의 종목을 1년 동안 관심 있게 지켜보는 것이 좋다고 한다. 주식을 전혀 안 해본 사람이라면 한 종목을 한 주만 사서 1년 동안 관찰하는 것도 주식을 이해하는 좋은 방법일 수 있다.

[코드 5-4] 화면에 볼린저 밴드를 그리기 위한 위치 계산(Graph.cpp)

```
    void CGraph::GetDataForChart(Company* company, PointData* ptData)
    {
        int i, maxVal, minVal, interval;
        maxVal = 0;
        int quantity = company->quantity;

01      for(i=0; i<quantity; i++)
            if(company->data[i].highVal > maxVal)
                maxVal = company->data[i].highVal;

        minVal = maxVal;

02      for(i=0; i<quantity; i++)
            if(company->data[i].lowVal < minVal)
                minVal = company->data[i].lowVal;

        ##### 생략 #####

03      interval = (int)(WIDTH_GRAPH / quantity);

04      int x_start = X_START_GRAPH + WIDTH_GRAPH;
05      int y_start = Y_START_GRAPH + HEIGHT_GRAPH;

        ptData->bollingerTop.quantity = quantity;
        ptData->bollingerMiddle.quantity = quantity;
        ptData->bollingerBottom.quantity = quantity;
        ptData->bollingerOscillator.quantity = quantity;

        for(i=0; i<quantity; i++) {
            ptData->bollingerTop.point[i].X = x_start - i * interval;
06          ptData->bollingerTop.point[i].Y = y_start  - (HEIGHT_GRAPH *
    (company->bollinger[i].lineTop - minVal)) / (maxVal - minVal);
            ptData->bollingerMiddle.point[i].X = x_start - i * interval;
07          ptData->bollingerMiddle.point[i].Y = y_start  - (HEIGHT_GRAPH *
    (company->bollinger[i].lineMiddle - minVal)) / (maxVal - minVal);
            ptData->bollingerBottom.point[i].X = x_start - i * interval;
```

```
08          ptData->bollingerBottom.point[i].Y = y_start  -  (HEIGHT_GRAPH *
    (company->bollinger[i].lineBottom - minVal)) / (maxVal - minVal);
        }

09      int originY = Y_START_GRAPH + HEIGHT_GRAPH + GAP_GRAPH_INDICATOR +
    (int)(HEIGHT_INDICATOR/2);
        float maxY = (int)((HEIGHT_INDICATOR/2) * 0.7);

        for(i=0; i<quantity; i++) {
10          ptData->bollingerOscillator.point[i].X = x_start - i * interval;
            ptData->bollingerOscillator.point[i].Y = originY - (int)(company->
    bollinger[i].oscillator * maxY);
        }
    }
```

---

01 차트에서 점의 위치를 나중에 계산하기 위하여 주가의 최고값을 얻는다.

02 차트에서 점의 위치를 나중에 계산하기 위하여 주가의 최저값을 얻는다.

03 데이터의 X 축 간격을 얻는다.

04 인덱스가 작은 앞부분이 최근 데이터가 있는 배열이고, 인덱스가 커질수록 오래 전 데이터이므로 인덱스가 증가하는 방향으로 그리기 위해서는 화면의 오른쪽부터 왼쪽으로 그려야 한다. 따라서 X 축의 시작점은 오른쪽 끝이 된다.

05 화면의 Y 축이 아래로 내려갈수록 점의 Y 값이 증가한다. 주가는 화면 아래쪽이 낮고 위쪽이 높으므로 Y의 시작점은 차트 화면의 아래가 된다.

06 2장에서 설명한 '비율 계산'을 이용하여 볼린저 밴드의 위 선을 계산한다.

07 '비율 계산'을 이용하여 볼린저 밴드의 가운데 선을 계산한다.

08 '비율 계산'을 이용하여 볼린저 밴드의 아래쪽 선을 계산한다.

09 볼린저 밴드의 오실레이터는 화면 아래에 그리고, 그리는 부분의 가운데가 0의 값을 가진다. 따라서 'HEIGHT_INDICATOR'를 2로 나눈 위치가 원점이 된다.

10 볼린저 밴드의 오실레이터를 그리기 위하여 1의 위치를 'HEIGHT_INDICATOR'의 원점에서 시작하여 70%의 위치에 정한다. 오실레이터는 1보다 클 수 있으므로 1의 위치를 중간 정도에 설정한다.[08]

11 '비율 계산'을 이용하여 볼린저 밴드의 오실레이터 위치를 계산한다.

---

08 상용 프로그램이라면 오실레이터의 최대값과 최소값을 계산하여 선의 위치를 정해야 하지만, 여기서는 간단한 코드 구현을 위하여 이 부분은 생략하였다.

화면에 그리기 위한 점들의 위치가 계산되었으므로 이대로 화면에 그리기만 하면 된다. [코드 5-5]는 이전 장과 마찬가지로 점들을 이용하여 화면에 선을 그리는 방법으로 구현하였다.

[코드 5-5] 화면에 볼린저 밴드와 오실레이터 그리기(StockAnalysisDlg.cpp)

```
    void CStockAnalysisDlg::DrawGraph()
    {
    ##### 생략 #####

        CPen greenPen;
01      greenPen.CreatePen(PS_SOLID, WIDTH_LINE, RGB(0,255,0));
        dc.SelectObject(greenPen);

02      dc.MoveTo(ptData->bollingerTop.point[0].X, ptData->
    bollingerTop.point[0].Y);
        for(i=1; i<stock->ptrCompany->quantity-BOLLINGER_MOVE_AVG; i++) {
            dc.LineTo(ptData->bollingerTop.point[i].X, ptData->
    bollingerTop.point[i].Y);
        }
03      dc.MoveTo(ptData->bollingerMiddle.point[0].X, ptData->
    bollingerMiddle.point[0].Y);
        for(i=1; i<stock->ptrCompany->quantity-BOLLINGER_MOVE_AVG; i++) {
            dc.LineTo(ptData->bollingerMiddle.point[i].X, ptData->
    bollingerMiddle.point[i].Y);
        }
04      dc.MoveTo(ptData->bollingerBottom.point[0].X, ptData->
    bollingerBottom.point[0].Y);
        for(i=1; i<stock->ptrCompany->quantity-BOLLINGER_MOVE_AVG; i++) {
            dc.LineTo(ptData->bollingerBottom.point[i].X, ptData->
    bollingerBottom.point[i].Y);
        }
```

```
05      RECT rect2 = {X_START_GRAPH,
            Y_START_GRAPH+HEIGHT_GRAPH+GAP_GRAPH_INDICATOR,
            X_START_GRAPH+WIDTH_GRAPH,
            Y_START_GRAPH+HEIGHT_GRAPH+GAP_GRAPH_INDICATOR+HEIGHT_INDICATOR};
        dc.FillRect(&rect2, &whiteBrush);

06      int originY = Y_START_GRAPH + HEIGHT_GRAPH + GAP_GRAPH_INDICATOR +
    (int)(HEIGHT_INDICATOR/2);

        CPen blackPen;
07      blackPen.CreatePen(PS_SOLID, WIDTH_LINE, RGB(0,0,0));
        dc.SelectObject(blackPen);

08      dc.MoveTo(X_START_GRAPH,originY);
        dc.LineTo(X_START_GRAPH+WIDTH_GRAPH,originY);

09      int lineUp = Y_START_GRAPH+HEIGHT_GRAPH+GAP_GRAPH_INDICATOR+
    (int)(HEIGHT_INDICATOR/2) - (int)((HEIGHT_INDICATOR/2) * 0.7);
10      int lineDown = Y_START_GRAPH+HEIGHT_GRAPH+GAP_GRAPH_INDICATOR+
    (int)(HEIGHT_INDICATOR/2) + (int)((HEIGHT_INDICATOR/2) * 0.7);

11      dc.MoveTo(X_START_GRAPH,lineUp);
        dc.LineTo(X_START_GRAPH+WIDTH_GRAPH,lineUp);
12      dc.MoveTo(X_START_GRAPH,lineDown);
        dc.LineTo(X_START_GRAPH+WIDTH_GRAPH,lineDown);

        dc.SelectObject(redPen);
13      dc.MoveTo(ptData->bollingerOscillator.point[0].X, ptData->
    bollingerOscillator.point[0].Y);
        for(i=1; i<stock->ptrCompany->quantity-BOLLINGER_MOVE_AVG; i++) {
            dc.LineTo(ptData->bollingerOscillator.point[i].X, ptData->
    bollingerOscillator.point[i].Y);
        }
    };
```

01 볼린저 밴드의 녹색 선을 그리기 위하여 실선의 녹색 펜을 생성한다.
02 볼린저 밴드의 위쪽 선을 그리고, 첫 번째 점으로 펜의 위치를 정하여 연속해서 선을 그린다.
03 볼린저 밴드의 가운데 선을 그린다.
04 볼린저 밴드의 아래쪽 선을 그린다.
05 볼린저 밴드의 오실레이터를 그리기 위한 영역을 하얀 사각형으로 그린다.
06 오실레이터의 가운데는 원점이 되고, 원점의 Y 축을 'originY'로 선언한다.
07 오실레이터에 수평으로 세 개의 선을 그리기 위해서 검은색 펜을 생성한다.
08 오실레이터의 가운데 선(원점)을 그린다.
09 1의 값을 갖는 오실레이터 위쪽 검은 선을 높이의 70%에서 그리기 위하여 Y 축 위치를 'lineUp'으로 선언한다.
10 -1의 값을 갖는 오실레이터 아래쪽 검은 선을 높이의 70%에서 그리기 위하여 Y 축 위치를 'lineDown'으로 선언한다.
11 오실레이터의 위쪽 검은색 직선을 그려준다.
12 오실레이터의 아래쪽 검은색 직선을 그려준다.
13 'Graph.cpp' 파일에서 계산된 오실레이터를 그리고, 첫 번째 점에 위치를 정하여 연속해서 선을 그린다.

주식 차트를 분석해 보하면 조정 구간에서는 큰 변동이 없다. 볼린저 밴드는 이런 구간에서 폭을 줄여주고, 변동이 큰 구간에서는 폭이 상당히 커진다. 즉, 폭이 좁은 구간과 폭이 큰 구간의 기준을 다르게 정하여 주식의 움직임을 관찰한다. 프로그램으로 주식을 분석하다 보면 주가가 항상 크게 움직이면 매매 시점을 정하기가 좀 더 수월할 것 같다는 생각이 든다. 예를 들어, 주식이 항상 큰 폭으로 움직인다면 프로그램에서 주식 데이터를 가져오는 주기를 1초 이하로 정하여 주가가 한 방향으로 움직이기 시작할 때 누구보다 빨리 매매하면 좋을 것 같다. 하지만 주식의 변동 폭은 항상 달라서 기준을 정하기가 상당히 어렵다. 볼린저 밴드는 최근 주가의 움직임을 벗어나는 큰 폭의 움직임을 감지하기에 좋으며 매매 시점을 어디에 둘 것인지를 다양하게 정할 수 있다. 또한, 볼린저 밴드의 가운데 선은 이평선이므로 이평선을 구현하면서 보았듯이 골든 크로스와 데드 크로스를 볼린저 밴드에서도 한 번 더 생각해 볼 필요가 있다.

5.3 볼린저 밴드 오실레이터

이평선과 MACD에서 2,000개가 넘는 종목을 가지고 오실레이터를 검증하였다. 볼린저 밴드 오실레이터도 이전에 구현한 것과 방법은 같으며 매매 조건만 볼린저 밴드 오실레이터로 변한다. 볼린저 밴드에서도 수치를 변경하여 결과를 확인한다. 변경되는 수치는 평균을 계산하기 위한 'BOLLINGER_MOVE_AVE'와 표준편차의 크기 변경을 위해 곱해지는 'BOLLINGER_TIME_SD'다. 'BOLLINGER_MOCE_AVG'의 값이 커지면 볼린저 밴드의 가운데 선인 이평선이 좀 더 완만해지고 더 많은 데이터를 가지고 표준편차가 계산된다. 'BOLLINGER_TIME_SD'값이 커지면 볼린저 밴드의 위와 아래 두 선의 거리는 크게 벌어진다. 여기서는 그래프를 보여주지는 않지만, 제공하는 예제 프로그램이나 직접 구현한 프로그램으로 수치 변화에 따른 볼린저 밴드의 차이를 확인해 보기 바란다.

[코드 5-6]은 볼린저 밴드의 수치인 'BOLLINGER_MOVE_AVG'를 20으로, 'BOLLINGER_TIME_SD'를 2로 정의한다.

[코드 5-6] 볼린저 밴드(20-2) 이동평균과 표준편차 배수 정의(Stock.h)

```
#define BOLLINGER_MOVE_AVG      20
#define BOLLINGER_TIME_SD       2

#define CHARGE_BUY              0.00015
#define CHARGE_SELL             0.00310
#define CHARGE_TRADE            0.005
```

[코드 5-7]은 볼린저 밴드의 오실레이터 매매를 구현한 것으로, [코드 3-9]와 거의 같고, 사고파는 조건에서만 볼린저 밴드 오실레이터를 사용한다. 이평선 오실레이터에서 코드를 설명하였으므로 여기서는 바뀐 조건만을 설명한다.

[코드 5-7]에서는 바로 이전 오실레이터가 -1 이하이고 현재 오실레이터가 -1보다 크면 매수하고, 바로 이전 오실레이터가 1 이상이고 현재 오실레이터가 1보다 작으면 매도한다.

[코드 5-7] 볼린저 밴드 매매 구현 및 결과 출력(Stock.cpp)

```
void CStock::Run()
{
    ReadDataFromFile();
    MakeBollinger();
    simulateTrade();
    }

void CStock::simulateTrade()
{
    int balance = 100000000;
    int prevBalance;
    int countWin = 0;
    int countLose = 0;
    int charge;
    int chargeTradeCenter = 0;

    selectedCompanies.quantity = 0;

    int i, j;

    for(i=0; i<allCompanies.quantity; i++) {
        Company *company = &allCompanies.companies[i];
        int quantity = company->quantity;

        prevBalance = balance;
        int isBought = false;

        for(j=quantity-BOLLINGER_MOVE_AVG-1; j>0; j--) {
```

```
                if(!isBought) {
01                  if(company->bollinger[j+1].oscillator <= -1 && company->
    bollinger[j].oscillator > -1) {
                        charge = (int)(company->data[j].lastVal * CHARGE_BUY);
                        balance -= charge;
                        chargeTradeCenter += charge;
                        charge = (int)(company->data[j].lastVal * CHARGE_TRADE);
                        balance -= charge;
                        balance -= company->data[j].lastVal;
                        isBought = true;
                    }
                } else {
02                  if(company->bollinger[j+1].oscillator >= 1 && company->
    bollinger[j].oscillator < 1) {
                        charge = (int)(company->data[j].lastVal * CHARGE_SELL);
                        balance -= charge;
                        chargeTradeCenter += charge;
                        charge = (int)(company->data[j].lastVal * CHARGE_TRADE);
                        balance -= charge;
                        balance += company->data[j].lastVal;
                        isBought = false;
                    }
                }
            }
            if(isBought) {
                charge = (int)(company->data[0].lastVal * CHARGE_SELL);
                balance -= charge;
                chargeTradeCenter += charge;
                charge = (int)(company->data[0].lastVal * CHARGE_TRADE);
                balance -= charge;
                balance += company->data[0].lastVal;
                isBought = false;
```

```
            }

            if(balance > prevBalance) {
                countWin += 1;
                selectedCompanies.companies[selectedCompanies.quantity] =
company;
                selectedCompanies.quantity += 1;
            } else if(balance < prevBalance) {
                countLose += 1;
            }
        }

        CString str;
        str.AppendFormat(_T("Win 종목개수: %d \n"), countWin);
        str.AppendFormat(_T("Lose 종목개수: %d \n"), countLose);
        str.AppendFormat(_T("Balance : %d \n"), balance);
        str.AppendFormat(_T("수수료: %d \n"), chargeTradeCenter);
        ::AfxMessageBox(str);
    }
```

---

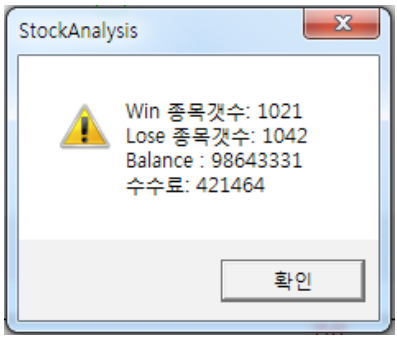
01 바로 전 오실레이터가 -1 이하이고, 현재 오실레이터가 -1보다 클 때 매수한다. 배열의 앞쪽이 최근 날짜고 뒤로 갈수록 이전 날짜이므로 'j'가 현재 데이터고 'j+1'이 바로 전 데이터다.

02 바로 전 오실레이터가 1 이상이고 현재 오실레이터가 1보다 작을 때 매도한다.

볼린저 밴드에서는 매매 시점이 명확히 어디라는 말은 없다. 단지 필자가 프로그래밍하면서 위 또는 아래로 통과한 후 다시 통과할 때가 적절한 매매 타이밍이라고 생각했다. 물론 가운데 선을 매매 시점의 기준으로 정할 수도 있다. 주식 프로그래밍을 할 때는 여러 방법을 시도해 보면서 직접 검증하고, 프로그램 화면을 보면서 생각의 다각화를 꾀할 필요가 있다.

[그림 5-4]는 매매 결과로, 이전에 검증했던 이평선이나 MACD보다는 승률이 높다. 물론 수수료와 매매 시 발생하는 손실 때문에 수익은 좋지 않다.

[그림 5-4] 볼린저 밴드(20-2) 매매 결과



볼린저 밴드(20-2)를 모든 종목에 적용했을 때 승률이 거의 50%였다. 주식은 확률 게임인데 적용했을 때 확률이 50%라면 승률을 더 올릴 수 있는 여지는 있다고 본다. 어쩌면 승률이 60%만 되어도 자동매매에 적용할 가치는 있을 수 있다. 물론, 이겼을 때 이익이 손실보다 크다는 조건이 필요하다. 또한, 아래쪽 선을 통과할 때가 매수 시점이라고는 하지만, 아직 가운데 선 아래에 있다면 하락 중인 장이므로 위험할 수도 있다.

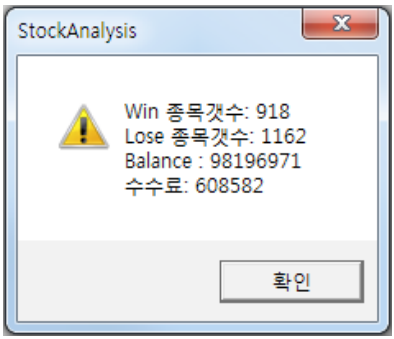
[코드 5-8]은 'BOLLINGER_MOVE_AVG'를 10으로 설정해서 평균과 표준편차를 계산하는 개수를 작게 잡는다.

[코드 5-8] 볼린저 밴드(10-2) 이동평균과 표준편차 배수 정의하기(Stock.h)

```
#define BOLLINGER_MOVE_AVG      10
#define BOLLINGER_TIME_SD       2

#define CHARGE_BUY              0.00015
#define CHARGE_SELL             0.00310
#define CHARGE_TRADE            0.005
```

[그림 5-5]는 매매 결과로, 수수료가 증가한 것으로 보아 매매 횟수는 많지만, 승률은 더 낮다.

[그림 5-5] 볼린저 밴드(10-2) 매매 결과



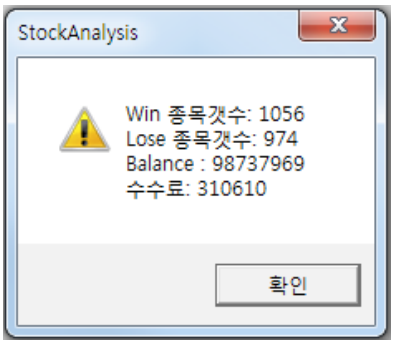
[코드 5-9]에서는 'BOLLINGER_MOVE_AVG'를 30으로 설정한다.

[코드 5-9] 볼린저 밴드(30-2) 이동평균과 표준편차 배수 정의하기(Stock.h)

```
#define BOLLINGER_MOVE_AVG      30
#define BOLLINGER_TIME_SD       2

#define CHARGE_BUY              0.00015
#define CHARGE_SELL             0.00310
#define CHARGE_TRADE            0.005
```

매매 결과는 [그림 5-6]과 같다. 재미있는 것은 수수료가 가장 낮으므로 매매가 이전보다 줄었지만, 이긴 종목 개수는 앞의 두 결과보다 많다. 10평선보다는 20평선이 결과가 좋으며 20평선보다는 30평선이 더 좋은 결과를 보여준다. 매매 횟수가 줄어들어도 수익이 좀 더 높다면 더 좋지 않을까 싶다.

[그림 5-6] 볼린저 밴드(30-2) 매매 결과



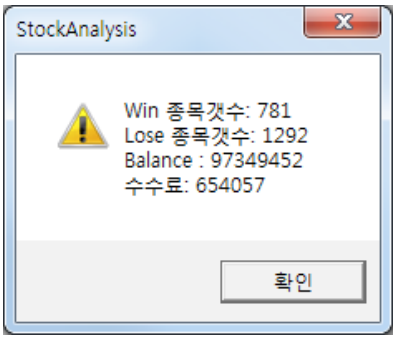
이번에는 표준편차의 크기에 변화를 주고 검증해 보자. [코드 5-10]에서 'BOLLINGER_MOVE_AVG'는 20으로 변하지 않으며, 'BOLLINGER_TIME_SD'를 1.5로 바꾸어 표준편차의 폭을 줄인다.

[코드 5-10] 볼린저 밴드(20-1.5) 이동평균과 표준편차 배수 정의하기(Stock.h)

```
#define BOLLINGER_MOVE_AVG      20
#define BOLLINGER_TIME_SD       1.5

#define CHARGE_BUY              0.00015
#define CHARGE_SELL             0.00310
#define CHARGE_TRADE            0.005
```

[그림 5-7]의 매매 결과를 보면, 표준편차의 폭을 줄여서 매매가 상당히 빈번하지만, 수익률은 가장 낮다. 코드에서는 이긴 종목만을 화면에 그려주는데, 직접 앞의 코드에서 'selectedCompanies'의 위치를 수정했을 때 어떠한 상황이 나오는지도 분석해 보기를 바란다.

[그림 5-7] 볼린저 밴드(20-1.5) 매매 결과



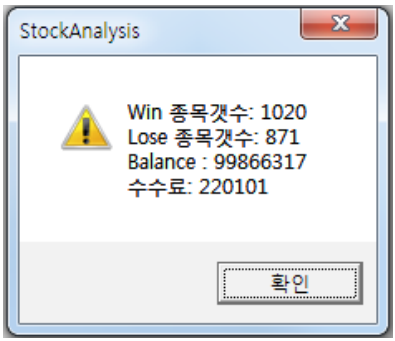
[코드 5-11]은 'BOLLINGER_TIME_SD'의 값을 2.5로 높여서 표준편차의 폭을 길게 만든다. 결과적으로 오실레이터가 1이나 -1을 통과하는 횟수는 줄어든다.

[코드 5-11] 볼린저 밴드(20-2.5) 이동평균과 표준편차 배수 정의하기(Stock.h)

```
#define BOLLINGER_MOVE_AVG      20
#define BOLLINGER_TIME_SD       2.5

#define CHARGE_BUY              0.00015
#define CHARGE_SELL             0.00310
#define CHARGE_TRADE            0.005
```

[그림 5-8]의 매매 결과를 보면 매매 횟수가 현저히 줄고, 매매가 이루어지지 않은 종목도 꽤 있다. 이긴 종목이 1,020개고 진 종목이 871개이므로 2,130개의 전체 종목 중 239개 종목에서는 매매가 전혀 이루어지지 않았다.

[그림 5-8] 볼린저 밴드(20-2.5) 매매 결과



볼린저 밴드의 매매 시점은 다양하게 정할 수 있다. -1, 0, 1의 모든 선을 기준으로 상향 돌파했을 때 매수하고, 하향 돌파를 했을 때 매도하는 방식을 취할 수도 있다. 실제로 차트에서는 빈번하게 이루어지곤 한다. 이 책은 최대한 이해하기 쉽도록 프로그램을 간단하게 만들었지만, 각자 여러 가지 방법으로 검증해 보기를 권한다. 자동매매는 매매 시점이 가장 중요하다고 보는데, 항상 이기는 매매 시점을 만들 수는 없지만, 승률을 높이는 프로그램을 만드는 것은 가능할 것이다. 의사가 병을 70% 이상 판별할 수 있다면 명의라는 말이 있는데, 주식 프로그램도 70% 이상의 승률이면 멋진 프로그램이 아닐까 하고 생각한다.

볼린저 밴드를 계속 보고 있으면 여러 가지가 보인다. 그중 하나가 아래쪽 선을 통과한 후 주가가 올라갔어도 가운데 선 아래에 계속 머무른다면 장은 계속 하락 중이라는 것이다. 물론 위쪽 선도 가운데 선 위에 있다면 계속 상승장일 수 있다. 계속 하락장임에도 볼린저 밴드만을 믿고 매매하는 것은 상당히 위험할 수가 있다. MACD와 같이 보면서 나름대로 분석해야 한다. 사람은 자신이 생각하는 것에 확신을 할 때가 있다. 주식이 어떻게 변할 것이라는 생각은 본인이 기대하는 바람이지 꼭 그렇게 움직이지 않을 때가 많다. 자신이 정한 가격 이하로 내려가면 마음이 아프더라도 팔고 다시 올라왔을 때 사는 것도 좋은 방법이다.

창업할 때 준비 없이 돈만을 가지고 시작하는 대부분의 사람들은 망할 수 있다. 창업의 조건 중에는 첫째가 아이템, 둘째가 사람, 셋째가 자본이라고도 한다. 주식을 한다는 것은 개인사업자등록을 하는 것과 비슷하다. 함께 가야 할 사람은 필요 없을 수 있지만, 아이템은 확실히 준비되어야 한다. 주식에서 아이템은 정보가 아닌 지식이다. 사람들은 정보를 가지고 주식을 한다고 생각하지만, 정보의 생명은 시간이다. 아무리 좋은 정보도 늦게 받으면 위험한 정보가 되고, 또한 직접 얻은 것이 아닌 정보를 믿고 주식을 한다는 것은 언젠가는 모든 걸 잃을 준비가 된 사람일 것이다.

주식에서 아이템은 정보를 판단하는 지식일 수 있으며, 주식이라는 사업을 하기 위해서 처음부터 매매에 달려들 것이 아니라, 공부를 충분히 하는 것이 좋으리라 생각한다. 주식을 잘해서 돈을 번다는 사람들이 있지만, 대부분은 사람들을 현혹하여 세력화하려는 경향이 있다. 또한, 돈을 받고 정보를 준다는 것은 같은 정보를 더 많은 돈을 주는 사람에게 더 빨리 정보를 준다는 의미다. 특정 종목을 지칭하여 사라고 하는 사람을 따라가는 사람들은 주식을 하지 말라고 말하고 싶다. 열심히 일하면 인정받았던 옛날에는 성실이 최고였지만, 이제는 잘해야 하는 세상이 되었다. 주식 시장에 오래 있었기에 자신은 도사라고 생각하는 사람들이 많지만, 감으로 오래 매매한 것과 공부하여 준비한 것에는 분명히 큰 차이가 있다. 필자는 주식매매는 많이 안 해봤지만, 지금도 준비하고 있고 먼 훗날 가능성이 있을 때만 달려들 것이다. 물론, 데이터를 가지고 시뮬레이션도 충분히 돌려볼 것이다.

지금까지의 내용을 이해했다면 2,000개가 넘는 종목의 데이터를 단 몇 초 만에 읽어와 많은 연산을 수행하고, 단시간에 결과를 알려주는 프로그램에 놀랐을 것이다. 이것이 프로그램 검증 방법이며, 앞으로의 매매는 프로그램 싸움으로 점차 발전할 것이다. 필자가 생각하는 프로그래머는 시간당 얼마를 받고 일하는 사람이 아니라, 좋은 프로그램을 만들어 그것을 가지고 돈을 벌게 하는 사람이다. 더 좋은 프로그램을 만들려고 노력하고 다른 프로그램들과 항상 경쟁하면서 발전해야 할 것이다.

6 | 알고리즘 추가하기

프로그램을 만들 때는 주제에 대해서 깊이 이해한 후 구현을 해야 한다. 이 책의 주제인 주식 프로그램도 마찬가지로, 프로그래머는 주식을 일반 사람들보다 좀 더 깊이 있게 이해해야만 한다. 이 책에서 완성된 프로그램을 만들어주지는 않으나 내용을 이해한 독자라면 어렵지 않게 추가로 구현할 수 있을 것이다.

[그림 6-1]의 봉 차트와 이평선, 그리고 거래량을 어떻게 구현하는지는 2장과 3장에서 설명하였다. 그리고 MACD와 볼린저 밴드는 4장과 5장에서 설명하였다. [그림 6-1] 아랫부분에 있는 여러 항목은 MFC 도구에서 쉽게 생성해서 구현할 수 있다. 이번 장에서는 화면 오른쪽에 알고리즘Algorithm 항목들을 만들고, 알고리즘을 추가하여 자신이 원하는 검색 방법을 만들어 본다.

[그림 6-1] 주간 데이터 프로그램[01]

이전 장까지는 시스템 트레이딩 쪽에 초점을 맞췄다면, 이번에는 주식 데이터만을 가지고 프로그래머가 원하는 검색 조건에 맞는 종목의 리스트를 보여주는 프로그램을 구현한다. 예를 들어, 하한가로 내려간 종목과 상한가를 치고 있는 종목, 한 달 동안의 거래량이 평균보다 5배 이상 많은 종목 등을 프로그램으로 어떻게 찾는지를 알아본다. 이를 다양한 방법으로 응용해 보기 바란다. 이 책은 프로그램을 어떻게 만드는지에 초점을 맞추었으므로 만드는 방법(How-To)을 이해할 수 있기를 바란다.

01 [그림 6-1]은 몇 년 전에 필자에게 주식을 가르쳐주신 박준근 님께 만들어 드린 프로그램으로 본인에게 허락을 받고 사진을 게시한다.

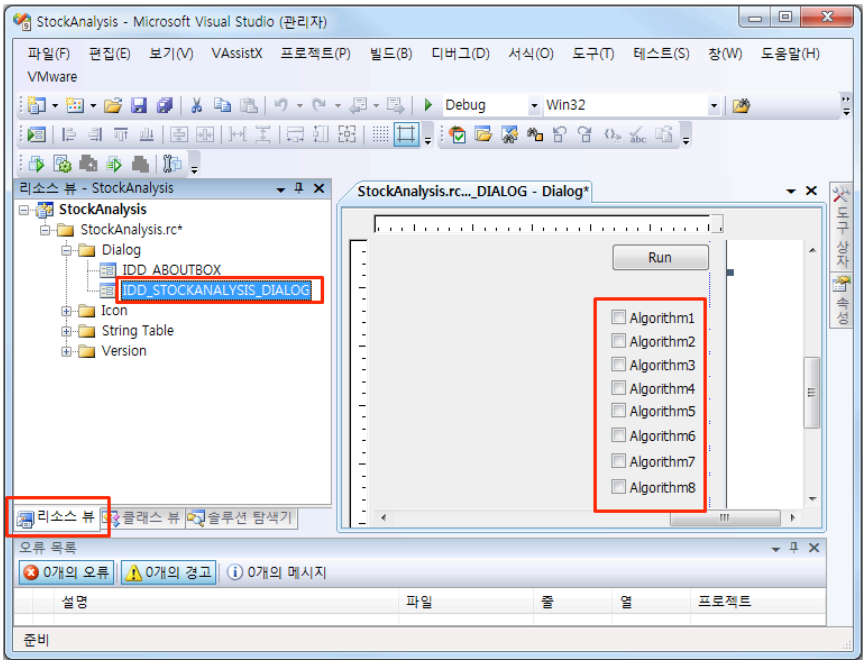
6.1 MFC – Check Box와 Flag 사용

이번 장에 나오는 내용은 3장의 이평선 코드에 추가로 구현한다. MFC의 체크 박스를 만들어서 검색 조건을 선택하고 종목을 찾는데, 하나 이상의 체크 박스를 선택하여 중복 검색도 가능하게 구현한다. [그림 6-2]처럼 오른쪽 아래에 알고리즘 체크 박스를 추가하면 하나의 알고리즘은 하나의 검색 조건을 가진다. 체크 박스를 이용하여 프로그램을 만들면 프로그램의 소스 코드를 건드리지 않고 편하게 프로그램을 사용할 수 있다.

[그림 6-2] 8개의 알고리즘 체크 박스 추가하기

[그림 6-3]은 체크 박스를 추가하는 방법을 보여준다. 왼쪽 아래의 '리소스 뷰' 탭에서 'IDD_STOCKANALYSIS_DIALOG'를 선택하여 프로그램 화면을 보여주는 다이얼로그를 연다. 그다음 도구 상자에서 체크 박스를 선택하여 화면에 다수의 체크 박스를 만든다. 그리고 각각의 체크 박스의 'ID'와 'Caption'에 고유한 이름을 부여한다.

[그림 6-3] 다이얼로그 파일에서 체크 박스 추가하기

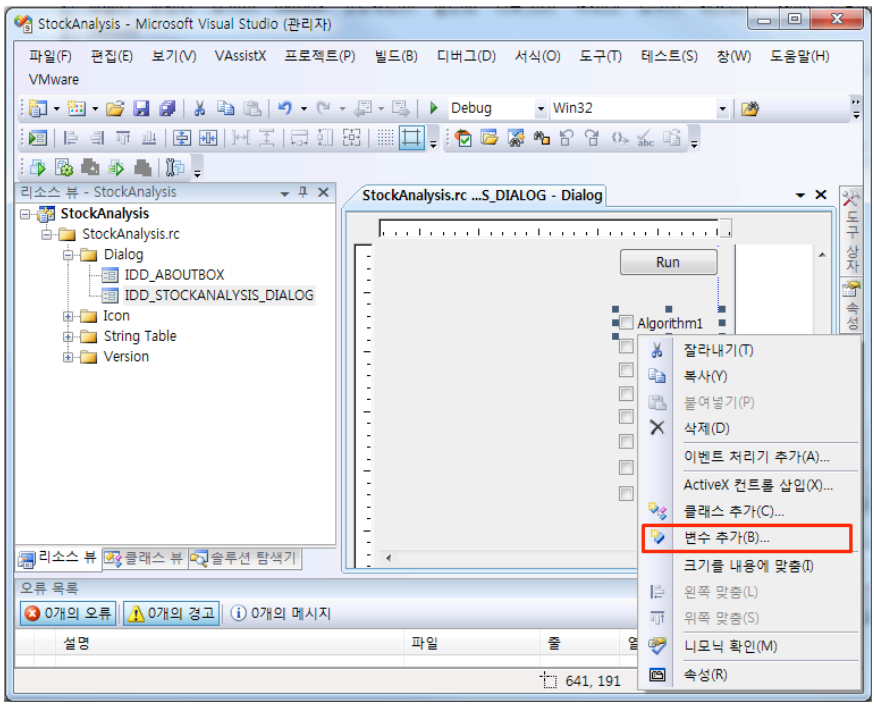


여기서는 첫 번째 체크 박스의 'ID'는 'IDC_CHECK_ALGORITHM1'으로, 'Caption'은 'Algorithm1'으로 정한다. 나머지 체크 박스들도 이처럼 정하고 숫자를 변경한다. 1장의 '버튼 만들기'에서 설명한 방법과 같으므로 쉽게 구현할 수 있을 것이다. 다만, 1장에서는 버튼을 클릭했을 때 이벤트 처리를 하기 위해 'OnClick()' 함수를 만들지만, 여기서는 체크 박스가 선택되거나 선택이 안 된 상태로만 표시되며 프로그램에서는 어떠한 동작도 처리하지 않는 점이 다르다. 따라서 체크 박스를 더블 클릭하여 이벤트 함수를 만드는 것이 아니라 체크 박스가 선택이 되었는지만 변수를 사용하여 확인한다.

[그림 6-4]는 체크 박스에 변수를 추가하는 방법을 보여준다. 체크 박스 위에 마우

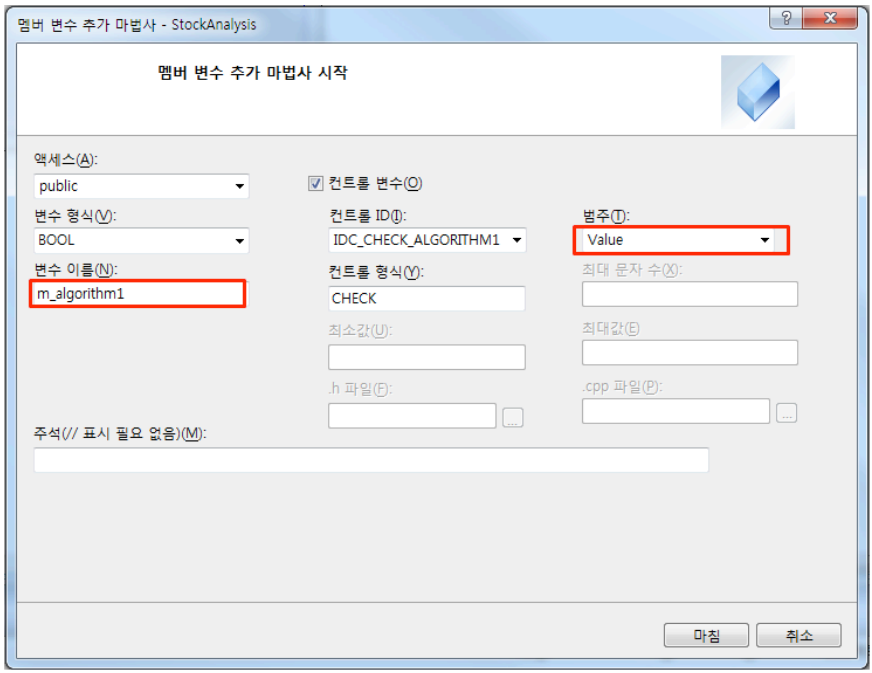
스 대고 오른쪽 버튼을 누르면 항목들이 보인다. 이 중에서 '변수 추가(B)' 항목을 선택한다.

[그림 6-4] 체크 박스의 변수 추가하기



'변수 추가'를 선택하면 [그림 6-5]와 같은 '멤버 변수 추가 마법사' 창이 뜬다. 오른쪽 '범주(T)' 항목은 'Value'를 선택하고 왼쪽의 '변수 이름(N)'에는 해당 체크 박스가 선택되었는지 확인할 때 사용할 변수 이름을 넣는다. 여기서는 멤버Member를 의미하는 'm_'을 앞에 넣어 'm_algorithm1'으로 첫 번째 체크 박스와 연동될 변수 이름을 짓는다. 모든 체크 박스에 연동될 변수를 만들어 준다. 이렇게 생성된 변수들은 자동으로 프로그램 코드가 생성된다.

[그림 6-5] 체크 박스의 'Value' 변수 추가하기



[코드 6-1]과 [코드 6-2]는 비주얼 스튜디오에서 자동으로 생성한 변수들을 보여준다. 이처럼 마법사를 사용하면 자동으로 코드가 생성되는 것은 윈도우 프로그램의 장점 중 하나다.

[코드 6-1]에서 자동으로 추가된 체크 박스 변수들은 마법사에서 생성할 때 'BOOL' 형으로 선택된다. 체크 박스가 선택되면 변수는 'TRUE' 값을 갖고, 체크 박스가 선택되지 않으면 변수는 'FALSE' 값을 가진다.

[코드 6-1] 헤더 파일에 자동으로 추가된 체크 박스 변수(StockAnalysisDlg.h)

```
class CStockAnalysisDlg : public CDialog
{
      ##### 생략 #####
      public:
            void DrawGraph();
            afx_msg void OnBnClickedBtnRun();
            afx_msg void OnCbnSelchangeComboJongmok();
            BOOL m_algorithm1;
            BOOL m_algorithm2;
            BOOL m_algorithm3;
            BOOL m_algorithm4;
            BOOL m_algorithm5;
            BOOL m_algorithm6;
            BOOL m_algorithm7;
            BOOL m_algorithm8;
};
```

[코드 6-2]는 'StockAnalysisDlg.cpp' 파일에 자동으로 생성된 체크 박스 변수 관련 코드들로, 'm_algorithm1(FALSE)'는 변수를 'FALSE'로 초기화한 것이다. 모든 변수가 'FALSE'로 초기화되어 있으므로 프로그램이 처음 실행될 때는 체크 박스가 모두 선택되어 있지 않는다. 만약 'm_algorithm1(TRUE)'처럼 'FALSE'에서 'TRUE'로 바꾸면 첫 번째 체크 박스인 'Algorithm1'은 프로그램이 시작될 때 체크 박스가 선택된 것으로 표시된다. [코드 6-2]의 뒷부분은 체크 박스와 멤버 변수들을 서로 연결하는 내용이다.

[코드 6-2] 자동으로 추가된 체크 박스 변수들(StockAnalysisDlg.cpp)

```
CStockAnalysisDlg::CStockAnalysisDlg(CWnd* pParent /*=NULL*/)
```

```
    : CDialog(CStockAnalysisDlg::IDD, pParent)
    , m_algorithm1(FALSE)
    , m_algorithm2(FALSE)
    , m_algorithm3(FALSE)
    , m_algorithm4(FALSE)
    , m_algorithm5(FALSE)
    , m_algorithm6(FALSE)
    , m_algorithm7(FALSE)
    , m_algorithm8(FALSE)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CStockAnalysisDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Control(pDX, IDC_COMBO_JONGMOK, m_jongmok);
    DDX_Check(pDX, IDC_CHECK_ALGORITHM1, m_algorithm1);
    DDX_Check(pDX, IDC_CHECK_ALGORITHM2, m_algorithm2);
    DDX_Check(pDX, IDC_CHECK_ALGORITHM3, m_algorithm3);
    DDX_Check(pDX, IDC_CHECK_ALGORITHM4, m_algorithm4);
    DDX_Check(pDX, IDC_CHECK_ALGORITHM5, m_algorithm5);
    DDX_Check(pDX, IDC_CHECK_ALGORITHM6, m_algorithm6);
    DDX_Check(pDX, IDC_CHECK_ALGORITHM7, m_algorithm7);
    DDX_Check(pDX, IDC_CHECK_ALGORITHM8, m_algorithm8);
}
```

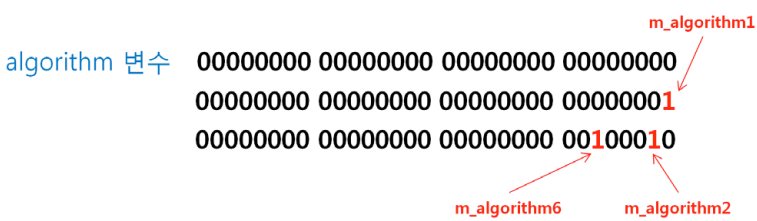
앞의 코드에서 체크 박스가 선택되었는지는 멤버 변수인 8개의 'm_algorithm'으로 확인할 수 있다. 그렇다면 이 변수들을 연산이 이루어지는 'Stock.cpp' 파일에 어떻게 전달해야 할까?

8개를 한번에 전달하는 것은 더 많은 체크 박스를 추가로 만들고, 코드도 굉장히

복잡해지므로 효과적인 방법이 아니다. 이를 개선하려면 하나의 변수를 만들어 새로 생성된 변수 안에 모든 체크 박스의 신호를 넣는 방법을 사용한다. 예를 들어, 32비트의 정수형 변수를 만들어서 한 비트가 체크 박스 한 개의 신호를 나타내면 된다.

[그림 6-6]은 'algorithm'이라는 변수를 보여준다. 'algorithm'은 32개의 비트로 이루어져 있고, 한 비트는 0 또는 1의 값을 가진다. 따라서 총 32개의 체크 박스 변수의 데이터를 저장할 수 있다. [코드 6-1]의 체크 박스 변수를 위해서는 오른쪽 8비트만 사용하면 되고, 오른쪽 끝부터 차례대로 순서를 정한다. 첫 번째 줄의 32비트는 모두 0이므로 이것은 8개의 체크 박스들이 모두 FALSE라는 것을 의미한다. 두 번째 줄에서는 오른쪽 끝 비트만 1인데 이것은 첫 번째 체크 박스 변수인 'm_algorithm1'만 'TRUE'고 나머지는 'FALSE'임을 나타낸다. 세 번째 줄에서는 오른쪽에서 두 번째와 여섯 번째의 비트들이 1이다. 이것은 두 번째와 여섯 번째 체크 박스만 선택되어 'TRUE'라는 것을 의미한다.

[그림 6-6] 체크 박스 변수들의 값을 모두 포함한 알고리즘 변수



[그림 6-6]을 체크 박스 변수들의 위치에 따른 데이터로 보면 다음과 같다.

```
m_algorithm1: 00000000 00000000 00000000 00000001
m_algorithm2: 00000000 00000000 00000000 00000010
m_algorithm3: 00000000 00000000 00000000 00000100
```

```
m_algorithm4: 00000000 00000000 00000000 00001000
m_algorithm5: 00000000 00000000 00000000 00010000
m_algorithm6: 00000000 00000000 00000000 00100000
m_algorithm7: 00000000 00000000 00000000 01000000
m_algorithm8: 00000000 00000000 00000000 10000000
```

코드에서 이 값들을 어떻게 표현할 수 있을까? 2진수를 16진수로 변환해서 표현하면 된다. 숫자 앞에 '0x'라는 문자가 있으면 C++에서는 16진수로 인식해서 4비트의 2진수는 한 디지트Digit의 16진수가 된다. 즉, 32비트는 8(= 32/4)개의 16진수로 표현되므로 앞의 체크 박스 변수의 위치는 다음과 같이 변환된다.

```
m_algorithm1: 0x00000001        (0001(2) = 0x1)
m_algorithm2: 0x00000002        (0010(2) = 0x2)
m_algorithm3: 0x00000004        (0100(2) = 0x4)
m_algorithm4: 0x00000008        (1000(2) = 0x8)
m_algorithm5: 0x00000010
m_algorithm6: 0x00000020
m_algorithm7: 0x00000040
m_algorithm8: 0x00000080
```

한 비트를 true와 false의 신호로 인식하여 저장 공간에 넣는 방법에 대해서 자세히 설명하였다. 이 방법은 특히 네트워크 프로그래밍할 때 많이 사용된다. 타 기기와의 통신에서 네트워크 속도에 한계가 있다면 전송되는 데이터의 크기를 줄이는 것이 아주 중요하다.

[코드 6-3]은 체크 박스의 선택에 따른 조건 검색 작업 중 'Stock.h' 파일에서 구현한 코드로, 어떤 체크 박스가 선택되었는지를 확인하기 위하여 플래그Flag를 사용한다. 'FLAG_ALGORITHM1'은 첫 번째 체크 박스와 관련된 플래그다.

[코드 6-3] 플래그 정의하기(Stock.h)

```
01    #define FLAG_ALGORITHM1          0x00000001
      #define FLAG_ALGORITHM2          0x00000002
      #define FLAG_ALGORITHM3          0x00000004
      #define FLAG_ALGORITHM4          0x00000008
      #define FLAG_ALGORITHM5          0x00000010
      #define FLAG_ALGORITHM6          0x00000020
      #define FLAG_ALGORITHM7          0x00000040
      #define FLAG_ALGORITHM8          0x00000080

      class CStock
      {
            public:
                  AllCompany allCompanies;
                  SelectedCompany selectedCompanies;
                  Company *ptrCompany;

            public:
                  CStock(void);
                  ~CStock(void);

02                void Run(unsigned int algorithm);
                  void makeSelectedCompanyFromAllCompany();
                  void makeMovementAverage();
03                bool condition1(Company *company);
                  bool condition2(Company *company);
                  bool condition3(Company *company);
                  bool condition4(Company *company);
                  bool condition5(Company *company);
                  bool condition6(Company *company);
                  bool condition7(Company *company);
                  bool condition8(Company *company);
                  void ReadDataFromFile();
```

```
        void WriteDataToFile();
};
```

01 체크 박스의 선택 여부를 확인하기 위하여 선언한다. 'FLAG_ALGORITHM1'은 첫 번째 체크 박스를 확인하기 위한 것이고, 'FLAG_ALGORITHM5'는 다섯 번째 체크 박스를 확인하기 위하여 사용된다.

02 모든 체크 박스의 선택 여무는 'algorithm'이라는 변수를 통해서 전달되며, 'algorithm' 변수는 'unsigned int'형으로 선언되어 32비트의 값을 갖는다.

03 체크 박스 순서에 따라 'condition()이라는 함수가 실행된다. 첫 번째 체크 박스가 선택되면 'condition1()' 함수가 실행되고, 두 번째 체크 박스가 선택되면 'condition2()' 함수가 실행된다. 이처럼 8개의 체크 박스를 위하여 8개의 'condition()' 함수가 실행되고, 'condition()' 함수는 true 또는 false를 반환한다.

[코드 6-4]는 조건 검색 함수를 구현한 것으로, 이것을 이해하면 추가로 체크 박스 구현하기가 쉽다. 특히, 추가 조건 검색은 'condition()' 함수 안에서만 구현되므로 프로그램을 아주 편리하게 사용할 수 있다. [코드 6-4]에는 첫 번째 'condition()' 함수인 'condition1()' 함수만 구현되어 있다. 이 함수는 현재 종가가 10만 원 이상인 종목만 검색하여 화면에 보여준다. 이렇게 'condition()' 함수들을 나누어서 만드는 이유는 조건 검색을 구현할 때 구분을 쉽게 하기 위해서이다. 예를 들어, 프로그램을 실행시킨 상태에서 체크 박스를 선택하면 조건 검색이 쉽게 변경될 수 있는데, 체크 박스 1과 2를 선택하고 조건 검색을 한 후 1과 3을 선택하고 조건 검색을 하면 다른 결과를 얻을 수 있다. 즉, 개별적으로 만들어진 조건 검색들을 다양하게 선택하여 조건 검색 결과들을 보는데 편리하다.

[코드 6-4] 조건 검색 함수 구현(Stock.cpp)

```
void CStock::Run(unsigned int algorithm)
{
    bool flag;
    selectedCompanies.quantity = 0;

    ReadDataFromFile();
```

```
        makeMovementAverage();

01      for( int i=0; i<allCompanies.quantity; i++) {
02          Company *company = &allCompanies.companies[i];

            flag = true;

04          if(FLAG_ALGORITHM1 & algorithm)
05              if(!condition1(company))-
                    flag = false;
06          if(flag)
                if(FLAG_ALGORITHM2 & algorithm)
                    if(!condition2(company))
                        flag = false;
                if(flag)
                    if(FLAG_ALGORITHM3 & algorithm)
                        if(!condition3(company))
                            flag = false;
                if(flag)
                    if(FLAG_ALGORITHM4 & algorithm)
                        if(!condition4(company))
                            flag = false;
                if(flag)
                    if(FLAG_ALGORITHM5 & algorithm)
                        if(!condition5(company))
                            flag = false;
                if(flag)
                    if(FLAG_ALGORITHM6 & algorithm)
                        if(!condition6(company))
                            flag = false;
                if(flag)
                    if(FLAG_ALGORITHM7 & algorithm)
                        if(!condition7(company))
                            flag = false;
```

```
            if(flag)
                if(FLAG_ALGORITHM8 & algorithm)
                    if(!condition8(company))
                        flag = false;

07          if(flag) {
                selectedCompanies.companies[selectedCompanies.quantity] = company;
                selectedCompanies.quantity++;
            }
        }
    }

08  bool CStock::condition1(Company *company)
    {
09      if(company->data[0].lastVal > 100000)
            return true;

        return false;
    }

    bool CStock::condition2(Company *company)
    {
        return false;
    }

    bool CStock::condition3(Company *company)
    {
        return false;
    }

    bool CStock::condition4(Company *company)
    {
        return false;
    }
```

```
bool CStock::condition5(Company *company)
{
    return false;
}

bool CStock::condition6(Company *company)
{
    return false;
}

bool CStock::condition7(Company *company)
{
    return false;
}

bool CStock::condition8(Company *company)
{
    return false;
}
```

---

01 모든 종목 데이터를 저장하고 있는 'allCompanies'에서 한 종목씩 검색하기 위하여 for 문을 사용한다.

02 'allCompany'에서 순서대로 한 종목을 'company' 변수로 가리키게 한다.

03 'flag' 변수는 종목이 조건 검색에서 포함되는지를 확인하기 위해 사용한다. 처음에는 'flag'를 'true'로 설정하고, 해당 조건 검색에 포함되지 않는다면 연산 중간에 'false'로 변한다. 조건 검색에 포함되어 마지막까지 'true' 값을 가지면 해당 종목은 화면에 보이도록 'selectedCompanies'에 포함된다.

04 'algorithm'에서 첫 번째 체크 박스가 선택되었는지를 확인한다. '&' 비트 연산자를 사용하여 'FLAG_ALGORITHM1'의 1로 표시된 위치의 비트가 1인지를 확인한다. 예를 들어, 'algorithm'이 '00000000 00000000 00000000 00001001'의 값을 갖고 있다면 ''FLAG_ALGORITHM1'은 '0x00000001'이 된다. 오른쪽 마지막 비트가 1이므로 true 값을 가진다. 이러한 방법으로 'algorithm'은 'FLAG_ALGORITHM1'과 'FLAG_ALGORITHM4'의 '&' 연산에서 'true'가 되고, 나머지는 'false' 값을 가진다.

05 'flag'는 처음에 'true'로 설정되므로 조건 검색 함수인 'condition1()'은 'false'를 반환하고, 'flag'를 'false'로 설정하여 결과에서 배제한다.

06 다수의 체크 박스들이 선택되어 조건 검색이 여러 번 수행되어야 하지만, 앞선 조건 검색에서 'flag' 값이 'false'가 되면 이후의 조건 검색은 수행할 필요가 없다. 'flag' 값이 'true'라는 것은 이전에 수행된 조건 검색에 해당되거나 조건 검색이 수행되지 않았을 때이다.

07 모든 조건 검색이 이루어지고 난 후에도 'flag' 값이 'true'면 조건에 검색된 것이므로 해당 종목을 화면에 그리기 위하여 'selectedCompanies' 변수에 추가한다.

08 'condition()' 함수들은 매개 변수로 포인터 변수를 갖는데, 이 변수는 데이터를 받는 것이 아니라 저장된 데이터의 메모리 주소만을 전달받는다.[02]

09 종목 데이터를 저장한 'data' 배열에서 0번째 데이터가 가장 최근 데이터이므로 'data[0].lastVal'는 가장 최근 종가가 된다. 종가가 10만 원 이상이면 'true'를 반환하고 아니면 'false'를 반환한다.

[코드 6-4]에 추가 조건을 만드는 방법은 간단하다. 확인하고 싶은 체크 박스에 해당하는 'condition()' 함수에 자신만의 조건을 코드로 추가 구현하면 된다. 여기서 10만 원 이상인 종가만을 검색한 것은 가장 간단한 예를 보이기 위해서다. 앞에서 설명한 구조체에 저장된 데이터를 이해했다면 추가 구현은 매우 쉽다.

[코드 6-5]는 프로그램 화면의 Run 버튼을 클릭했을 때 수행되는 'StockAnalysisDlg.h' 파일의 'OnBnClickedRun()' 함수를 구현한 것으로, 체크 박스 구현의 마지막 부분이다. 여기서 중요한 것은 '|' 비트 연산이다.[03]

[코드 6-5] Run 버튼 클릭 시 동작 함수(StockAnalysisDlg.cpp)

```
    void CStockAnalysisDlg::OnBnClickedBtnRun()
    {
        int i;

01      UpdateData(TRUE);

02      unsigned int algorithm = 0;

        if(m_algorithm1)
```

02 포인터에 관한 자세한 내용은 『소수와 RSA 알고리즘으로 배우는 Big Number 연산』(한빛미디어, 2013)을 참고하기 바란다.

03 '|'은 'OR' 연산으로 C 언어 입문책 또는 『소수와 RSA 알고리즘으로 배우는 Big Number 연산』(한빛미디어, 2013)의 부록 'Big Number 연산에 필요한 C 기초'에 자세히 나와 있다. 'How-to Series'는 전작에서 설명한 내용은 이후 책에서는 다시 설명하지는 않으므로 전작을 참고하기 바란다.

```
03          algorithm |= FLAG_ALGORITHM1;
        if(m_algorithm2)
            algorithm |= FLAG_ALGORITHM2;
        if(m_algorithm3)
            algorithm |= FLAG_ALGORITHM3;
        if(m_algorithm4)
            algorithm |= FLAG_ALGORITHM4;
        if(m_algorithm5)
            algorithm |= FLAG_ALGORITHM5;
        if(m_algorithm6)
            algorithm |= FLAG_ALGORITHM6;
        if(m_algorithm7)
            algorithm |= FLAG_ALGORITHM7;
        if(m_algorithm8)
            algorithm |= FLAG_ALGORITHM8;

04      stock->Run(algorithm);
05      m_jongmok.ResetContent();

        for(i=0; i<stock->selectedCcompanies.quantity; i++) {
            m_jongmok.AddString(stock->selectedCompanies.companies[i]->strName);
        }

        m_jongmok.SetCurSel(0);

        UpdateData(FALSE);

        if(i == 0)
            return;

        stock->ptrCompany = stock->selectedCompanies.companies[0];

        flagPaint = true;

        RedrawWindow();
    }
```

01 화면의 체크 박스가 선택되었다 하더라도 멤버 변수인 'm_algorithm'은 변화가 없다. 하지만 'UpdateData(TRUE)'를 하면 화면에 표시된 데이터들을 기준으로 모든 멤버 변수의 값이 업데이트된다. 반대로 'UpdateData(FALSE)'를 실행하면 멤버 변수의 값을 기준으로 화면의 데이터가 변한다.

02 모든 체크 박스의 데이터를 저장하는 변수인 'algorithm'을 0으로 초기화한다. 'algorithm'은 'unsigned int'로 선언되어 음수를 표시하는 비트가 없고, 32 비트로 이루어져 있다.

03 'm_algorithm1'이 'true'면 'algorithm'의 제일 오른쪽 비트를 1로 바꿔준다. 'algorithm |= FLAG_ALGORITHM1'은 'algorithm = algorithm | FLAG_ALGORITHM1'과 같은 의미이며, '|'은 OR 비트 연산이다. 같은 방법으로 나머지 체크 박스에 대하여 'algorithm' 변수를 설정한다.

04 'stock'의 'Run( )' 함수를 실행하고, 매개 변수로 'algorithm'을 넣어준다. 프로그램의 가장 중요한 연산들은 'stock.cpp'에서 수행된다.

05 프로그램의 결과를 가지고 콤보 박스에 조건 검색 결과를 넣어준다[04].

이번 장에서 중요한 부분은 모두 구현되었다. 체크 박스를 사용하여 프로그램을 검색할 수 있는 틀을 만들었다. 이전 장들을 모두 이해했다면 자신만의 조건 검색을 만들어서 'condition( )' 함수에 추가하면 된다. 장이 끝난 후 매일 새로운 종목을 찾아야 한다면 이번 장이 아주 요긴할 것이다. 이 책에서는 설명하지 않지만, 실시간으로 데이터를 가져오는 것까지 구현한다면 장 중간에도 계속해서 자신이 원하는 종목을 찾을 수 있다. 예를 들어, 현재 종가가 5% 이상 상승한 종목들을 찾아서 거래량 등을 확인하는 것이다. 5% 이상 상승하고, 아래서 받치는 거래량(황소의 힘)이 위에서 누르는 거래량(곰의 힘)보다 월등히 많다면 추가로 상승할 힘이 있다고 판단할 수 있다. 이렇게 자신만의 알고리즘으로 나름대로 전략을 짜는 것도 재미있을 수 있다.

[그림 6-7]은 지금까지 구현한 프로그램의 결과를 보여준다. 오른쪽의 콤보 박스에 나오는 모든 종목은 종가가 10만 원 이상이다.

04 이후 코드는 2장의 콤보 박스에서 설명했으므로 2장을 참고하기 바란다.

[그림 6-7] 현재 종가가 10만 원 이상인 종목 검색 결과

핵심적인 부분은 모두 구현되었다. 이후 내용은 'condition()' 함수에 몇 가지 조건 검색들은 넣는 방법을 설명한다. 간단한 예를 들어 설명하고, 여러 가지 조건 검색을 만든다고 해도 코드가 길지는 않다.

데이터가 저장된 구조체 부분을 머릿속에서 그리고 있어야 자신만의 코드를 쉽게 추가할 수 있다는 점을 강조하고 싶다. 이 책을 읽는 독자라면 이 책에서 제공한 프로그램을 그대로 따라 하지 말고, 자신만의 프로그램으로 처음부터 만드는 것을 추천한다. 처음에는 쉽지 않겠지만, 구현을 계속하다 보면 잘 만들 수 있게 될 것이다.

6.2 상/하한가 종목 찾기

이번에는 상한가와 하한가 종목을 찾는 검색 조건을 구현한다. 프로그램의 틀은 이전에 모두 구현하였으므로 여기서는 'condition2()' 함수에 상한가 종목을 찾는 조건을 구현하고, 'condition3()' 함수에 하한가 종목을 찾는 조건만 구현한다. [코드 6-6]은 현재 종가가 상한가 10% 안에 존재하는 종목을 검색하는 코드다.

[코드 6-6] 현재 종가가 상한가 10% 안에 존재하는 종목 검색(Stock.cpp)

```
bool CStock::condition2(Company *company)
{
    long maxVal, minVal;

    maxVal = 0;
01  for(int i=0; i<company->quantity; i++) {
        if(maxVal < company->data[i].highVal)
            maxVal = company->data[i].highVal;
    }
    minVal = maxVal;
02  for(int i=0; i<company->quantity; i++) {
        if(minVal > company->data[i].lowVal)
            minVal = company->data[i].lowVal;
    }

03  long compareVal = maxVal - (long)((maxVal - minVal) * 0.1);

04  if(company->data[0].lastVal > compareVal)
        return true;

    return false;
}
```

01 해당 종목의 고가인 'highVal'과 비교하여 한 종목에서 최고값을 찾아 'maxVal'에 저장한다.
02 해당 종목의 저가인 'lowVal'를 비교하여 한 종목에서 최저값을 찾아 'minVal'에 저장한다.
03 기준이 되는 'compareVal'은 최고값에서 10% 아래의 값을 취한다.
04 현재의 종가인 'data[0].lastVal'이 기준인 'compareVal'보다 크다면 상한가 10% 안에 존재하는 종목이므로 'true'를 반환한다.

[그림 6-8]은 결과 화면으로, 종가가 상한가를 치는 종목들만 검색되어 오른쪽 콤보 박스에 추가되었다. 조건 검색을 하면 자신이 원하는 종목을 쉽게 찾을 수 있는다. 하지만 2,000개가 넘는 주식 종목을 모두 직접 확인하는 것은 불가능하므로 이

렇게 프로그램을 사용하여 간단히 해결할 수 있다. 이것이 프로그램의 힘이다.

[그림 6-8] 현재 종가가 상한가 10% 안에 존재하는 종목 검색 결과

[코드 6-7]은 현재 종가가 하한가 10% 안에 존재하는 종목을 검색하는 코드로, [코드 6-6]과 많은 부분이 같다. 단, 기준이 되는 'compareVal'의 위치는 최소값보다 10% 위에 존재한다.

[코드 6-7] 현재 종가가 하한가 10% 안에 존재하는 종목 검색(Stock.cpp)

```
bool CStock::condition3(Company *company)
{
	long maxVal, minVal;

	maxVal = 0;
	for(int i=0; i<company->quantity; i++) {
		if(maxVal < company->data[i].highVal)
			maxVal = company->data[i].highVal;
	}
	minVal = maxVal;
```

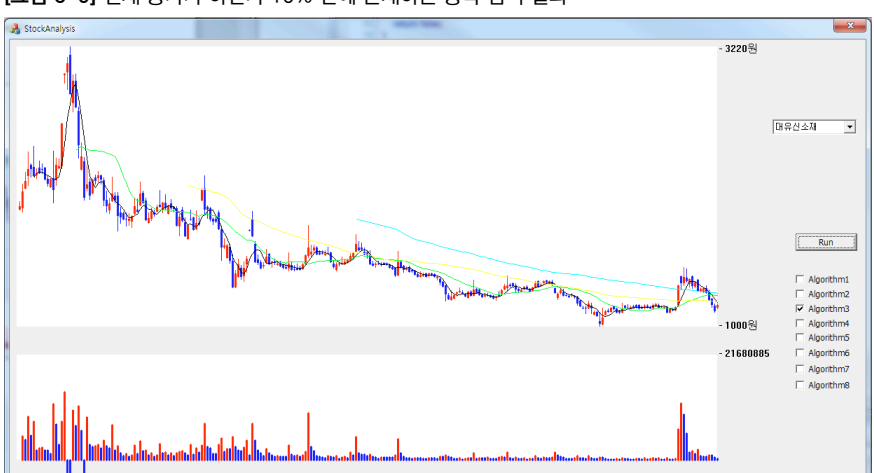
```
            for(int i=0; i<company->quantity; i++) {
                  if(minVal > company->data[i].lowVal)
                        minVal = company->data[i].lowVal;
            }

01          long compareVal = minVal + (long)((maxVal - minVal) * 0.1);

02          if(company->data[0].lastVal < compareVal)
                  return true;

            return false;
      }
```

01 기준이 되는 'compareVal'는 최소값에서 10% 위의 값을 취한다.

02 현재 종가인 'data[0].lastVal'가 기준인 'compareVal'보다 작다면 하한가 10% 안에 존재하는 종목이므로 'true'를 반환한다.

[그림 6-9]는 실행 결과를 보여준다. 오른쪽 콤보 박스에 추가된 종목들은 하한가 10% 안에 존재한다.

[그림 6-9] 현재 종가가 하한가 10% 안에 존재하는 종목 검색 결과



쉬운 예로 상한가와 하한가를 들었다. 경험에 의하면 하한가를 치는 종목에서도 재미있는 일이 연출된다. 분명히 상장 폐지될 종목임에도 확대해서 보니 100원까지 내려갔던 종목이 1,000원 이상까지 올라갔다. 계산상으로 100원에 산 사람은 1000%의 수익이 날수도 있지만, 상장 폐지 직전에도 거래되는 것이 참 신기했다.[05]

6.3 거래량 많은 종목 찾기

이번 장에서는 거래량이 많은 종목을 검색한다. 검색 조건은 이전 10일 동안의 거래량 평균보다 현재의 거래량이 5배 이상 많은 종목이다. [코드 6-8]은 거래량에 대한 검색으로, 네 번째 체크 박스에 해당하는 'condition4()' 함수에 구현한다.

[코드 6-8] 현재 거래량이 이전 10일 동안의 거래량 평균보다 5배 이상인 종목 검색(Stock.cpp)

```
    bool CStock::condition4(Company *company)
    {
        long sum = 0;
01      for(int i=1; i<=10; i++) {
            sum += company->data[i].vol;
        }
        long average = (long)(sum / 10);
02      if(company->data[0].vol > (5 * average))
            return true;
        return false;
    }
```

1 'data[0]'은 가장 최근 데이터이므로 'data[1]'부터 'data[10]'까지의 평균을 구한다. 계산된 평균은 'average'에 저장한다.

2 변수 'average'에 5를 곱하고, 이 값보다 현재의 거래량인 'data[0].vol'이 크면 'true'를 반환한다.

05 어쩌면 증권가 정보지의 힘(?)이 아닐까 한다. 주식을 잘한다는 사람들의 경향은 주로 추천 종목을 말할 때 자신이 산 종목을 말하는 경우가 많다. 다른 사람들을 통해 더 올라가게 하려는 마음도 있을 수 있지만, 한편으로는 털고 나오고 싶은 마음도 있을 것이다. 돈 때문에 죽는 사람들이 있는 세상인데 남의 돈보다는 내 돈이 가장 귀한 돈이라고 생각하는 것 같다.

[그림 6-10] 현재 거래량이 이전 10일 동안의 거래량 평균보다 5배 이상인 종목 검색 결과

[그림 6-10]을 보면 오른쪽 콤보 박스에는 현재 거래량이 많은 종목이 포함된다. 거래량이 많다는 것은 많은 사람이 관심이 있다는 것이므로 주식 시장에서는 중요한 의미다. 주식 데이터 중에서 가장 신빙성 있는 것은 거래량이다. 급격한 거래량 뒤에는 주식이 요동칠 확률이 높다.

6.4 이평선 비교

이번에는 이평선으로 분석하는 것을 구현한다. 체크 박스를 두 개 선택했을 때 두 조건을 동시에 만족하는 종목을 검색한다. 'condition5()' 함수는 5평선이 20평선 위에 있는 종목을 검색하고, 'condition6()' 함수는 20평선이 60평선 위에 있는 조건을 검색한다. 프로그램에서는 다섯 번째 체크 박스와 여섯 번째 체크 박스가 동시에 선택되어야 한다. [코드 6-9]에서 이 내용을 'condition5()'와 'condition6()' 함수에 구현한다.

[코드 6-9] 5평선이 20평선 위에 있고, 20평선이 60평선 위에 있는 종목 검색(Stock.cpp)

```
bool CStock::condition5(Company *company)
{
01      if(company->moveAverage.avg5[0] > company->moveAverage.avg20[0])
            return true;
        return false;
}

bool CStock::condition6(Company *company)
{
02      if(company->moveAverage.avg20[0] > company->moveAverage.avg60[0])
            return true;
        return false;
}
```

01 현재 5평선이 20평선 위에 있을 때 'true'를 반환한다.

02 현재 20평선이 60평선 위에 있을 때 'true'를 한다.

[그림 6-11] 5평선이 20평선 위에 있고, 20평선이 60평선 위에 있는 종목 검색 결과

[그림 6-11]은 이평선 비교 결과를 보여준다. 배포한 프로그램 소스 코드를 실행시켜보면 코드를 좀 더 쉽게 이해할 수 있다.

주식 프로그램을 만드는 것은 주식을 하는 사람에게는 날개를 단 것과 마찬가지다. 이것은 자신이 생각하는 것을 실전에 바로 적용할 때 발생할 수 있는 위험 요소를 줄인다. 또한, 여기서 구현한 것보다 더 많은 체크 박스를 만들어서 간단한 알고리즘 조건 검색을 설정하고, 몇 개의 알고리즘을 다양하게 선택하여 종목을 찾으면 주식 프로그램을 만드는 것이 매우 재미있을 것이다.

필자는 1년에 한 권씩 'How-To Series'의 개별 주제를 책으로 집필할 예정이다. 두 번째 주제로 주식 프로그램을 다룬 이유는 독자들의 관심 때문이다. 첫 번째 책인 『소수와 RSA 알고리즘으로 배우는 Big Number 연산』(한빛미디어, 2013)[06]은 많은 사람이 관심 있는 분야는 아니지만, 앞으로 나올 책들을 이해하는 데 참고가 될 것이다. 'How-To Series'를 읽고 모두 이해하는 독자라면 웬만한 프로그램은 쉽게 만들 수 있는 실력이 생길 것이라고 확신한다.

프로그램은 자기가 만들고 싶은 것을 무작정 시작하고 만드는 것이 좋다. 물론 주제를 정하기가 어렵겠지만, 프로그래머라면 재미로라도 자신만의 프로그램을 하나 이상 만들 필요가 있다. 앞으로 계속되는 필자의 책들은 주제만 다를 뿐 시리즈의 일부분이므로 이전 책에서 설명한 내용은 다시 설명하지 않을 계획이다. 주식 때문에 이 책을 읽는 독자는 다른 책을 볼 필요가 없지만, 순수한 프로그래밍만을 원하는 독자들은 시리즈의 다른 책들도 읽어보기를 권한다.

06 http://goo.gl/xAwrhR

APPENDIX

A 증권회사 API – Xing

우리나라에서 가장 좋은 주식 시스템을 보유하고 있는 곳은 증권거래소일 것이다. 증권거래소는 주식의 모든 거래를 정확히 체결하여 데이터를 실시간으로 증권사들에 제공해야 하고, 매우 짧은 시간 동안 무수히 많은 거래가 이루어진다. 이것을 모두 처리하기 위해서는 하드웨어도 중요하지만, 소프트웨어를 설계하는 사람도 아주 중요하다고 생각한다. 증권거래소에서는 주식 관련 데이터를 일반인에게는 제공하지 않는다. 일반인에게 데이터를 제공한다면 정상 매매도 디도스DDos 공격처럼 트래픽Traffic이 매우 증가할 것이다.

2000년대 후반부터는 증권사에서 API를 제공하여 일반인도 주식 데이터를 가져다가 프로그램을 만들어서 매매할 수 있다. 그럼 API는 무엇일까? Application Programming Interface의 약어로, 응용 프로그램을 만들 때 사용할 수 있는 인터페이스라는 뜻이다. 증권사에서 만들어 놓은 함수를 사용하여 데이터를 가져오는데, 이때 증권사에서 만들어 놓은 함수들의 집합을 증권 API라고 볼 수 있다. 즉, 대신증권에서 만들어 놓은 함수들의 집합은 '대신 API'고 이베스트투자증권에서 만든 함수들의 집합은 '이베스트투자 API'가 된다. 증권거래소도 API를 가지고 있다. 증권사는 증권거래소의 API를 사용하여 데이터를 가져오고, 일반인은 증권사의 API를 사용하여 데이터를 가져온다. 증권사의 HTS나 MTS도 API를 사용하여 만들어졌다.

API란 말은 프로그램의 전 분야에서 많이 사용한다. 예를 들어, 카카오톡에서도 다른 프로그램과 카카오톡을 연동할 수 있는 API를 제공한다. 일반인들이 카카오에

서 제공하는 함수를 가지고 메신저 프로그램을 만들어 카카오톡 친구들과 대화할 수 있다. 주식을 하는 사람 대다수는 컴퓨터 프로그램에 관심은 있으나 접해보지 못한 사람들이 참 많다. 이런 사람들과 얘기할 때 증권사 API에 대하여 이야기하면 상당히 관심 있어 할 것이다.

증권사에서 제공하는 HTS(Home Trading System) 프로그램은 증권사 API를 이용하여 데이터를 주고받는다. API를 공개한다는 것은 HTS에서 사용하는 API를 일반인들도 사용할 수 있다는 것이므로 자신만의 HTS 프로그램을 만들 수 있다는 의미다. HTS에서 주식의 매매 가격을 입력하고 매수와 매도 주문을 하는데, 개인 프로그램은 API로 현재 가격을 실시간으로 알 수 있으므로 버튼만 누르면 자동으로 주문을 넣게 만들 수 있다. 또한, HTS에서 주로 사용하는 화면은 몇 개에 불과하므로 개인 HTS에서 필요한 것들만을 한 화면에 구현한다면, 증권사 HTS를 실행시킬 필요도 없어진다. 그리고 자신만의 확실한 알고리즘이 있다면 자동매매 시스템을 만들어서 운용할 수도 있다.

많은 증권사에서 공개한 몇 가지 API를 사용해 본 결과 개인적으로 이베스트투자증권의 API가 가장 괜찮았고, MFC 프로그램을 만들기에도 가장 좋았다. 자동매매 프로그램을 구현할 때 매매를 자주 하면 증권사에 내는 수수료도 중요한 고려사항인데, 이베스트투자증권은 온라인 증권사여서 다른 증권사보다 수수료가 저렴하다. 또한, API에 대한 자료도 풍부하므로 C++를 사용하여 주식 프로그램을 만든다면 이베스트투자증권 API인 Xing을 사용하는 것이 좋다.

2014년 봄, 이베스트투자증권에서 Xing API로 만든 프로그램을 등록하는 이벤트가 진행되어 필자는 두 개의 프로그램을 등록하였다. 첫 번째가 이 책에서 사용하는 데이터 파일을 만드는 프로그램이고, 두 번째는 파생상품인 옵션(Option)에서 볼린저 밴드와 MACD를 실시간으로 그려주는 프로그램이다. 주식이 아닌 옵션을 선택한 이유는 주식보다 옵션이 매매가 활발하여 역동적인 차트를 볼 수 있기 때문이

다. 이 책에서는 하나의 완성된 프로그램을 만들지는 않았지만, 부록에서 제공하는 프로그램을 분석할 수 있게 충분히 설명하였다. 증권사에 참여한 프로그램의 골격도 이 책에서 설명한 내용과 구조를 비슷하게 만들었다.

이베스트투자증권의 Xing API를 사용하려면 회원 가입을 하고 API 사용 등록을 한다. 이베스트투자증권에서 진행하는 API 교육을 들으면 Xing API 사용법을 배울 수 있고, 초급과 전문가 과정을 모두 들으면 좋다. API를 적용할 때 'DevCenter'라는 프로그램이 아주 유용하다. API를 통하여 주고받는 데이터를 'DevCenter'에서 보내주어 API를 적용하는 시간이 현저히 줄어든다. 개인적으로 'DevCenter'가 없었다면 API를 적용하는 것이 불가능했다고 생각한다. 이 책을 빌려 'DevCenter'를 만든 이베스트투자증권의 진은정 과장님께 감사드린다.

이 책에서 다루지는 않았지만, 윈도우 프로그램에서는 메시지가 중요하다. 윈도우에서는 화면에서 사용자에 의해 발생하는 이벤트를 제외한 모든 이벤트가 메시지의 전달이라고 볼 수 있다. 윈도우 프로그램은 메시지를 받아서 특정 함수를 실행하는데 이때 실행되는 함수가 콜백Callback 함수다. 이 함수들은 프로그램이 실행될 때 실행되는 것이 아니라, OS나 특정 메시지에 의해서 실행된다. API에서 콜백 함수는 매우 중요하게 사용된다. 증권사에 데이터를 요청한 후에 데이터를 받는다면 프로그램에서 만들어 놓은 콜백 함수가 실행되며, 증권사로부터 받은 데이터의 가공은 콜백 함수에서 처리한다.

[코드 AA-1]과 [코드 AA-2]는 Xing을 사용할 때 데이터를 받는 콜백 함수의 사용법을 보여준다. Xing의 헤더 파일인 'IXingAPI.h'에 정의된 메시지인 'XM_RECEIVE_DATA'와 'XM_RECEIVE_REAL_DATA'를 사용자 메시지로 받아서 'OnXMReceiveData' 함수와 'OnXMReceiveRealData' 함수를 실행하도록 정의한다. Xing API는 데이터를 받으면 'XM_RECEIVE_DATA' 메시지를 프로그램의 메시지 큐에 넣고, 프로그램은 이 메시지를 받아서 'OnXMReceiveData' 함수

를 실행한다. 이때 받은 데이터는 매개 변수인 'wParam'과 'lParam'으로 접근할 수 있다. 'wParam'은 데이터의 형식을 나타내는 숫자고, 'lParam'은 받은 데이터가 저장된 메모리의 첫 번째 주소를 가진다. 'wParam'은 받은 데이터가 요청 데이터인지, 시스템 에러인지 등의 정보를 파악하고, 'lParam'은 메모리에 대한 포인터로 받은 데이터에 접근할 수 있다.[01]

Xing API를 이용하여 데이터를 요청하는 방법은 두 가지가 있다. 첫 번째는 데이터를 요청하고 한 번만 데이터를 받는 'Receive Data' 방법이고, 두 번째는 데이터를 요청하면 실시간으로 데이터를 계속해서 받는 'Receive Real Data' 방법이다. 처음에 프로그램을 만들 때는 'Receive Data'로 데이터 전체를 받고, 필요한 데이터는 실시간으로 받는다. 이 책에 수록되는 두 개의 프로그램 중 첫 번째는 'Receive Data'만을 사용하였고, 두 번째는 'Receive Data'와 'Receive Real Data'를 모두 사용하여 구현하였다.

[코드 AA-1] 콜백 함수 - OnXMReceiveData와 OnXMReceiveRealData

```
    class CRealTimeChartDlg : public CDialog
    {
        protected:
01          afx_msgLRESULT OnXMReceiveData( WPARAM wParam, LPARAM lParam );
02          afx_msgLRESULT OnXMReceiveRealData( WPARAM wParam, LPARAM lParam );
    }
```

01 'Receive Data'를 요청한 후(Request Data), 한 번만 실행되는 'OnXMReceiveData' 함수를 선언한다. 예외적으로 연속된 데이터가 있으면 여러 번 실행될 수 있다.

02 'Receive Real Data'를 요청한 후(Request Real Data), 데이터를 받을 때마다 실시간으로 실행되는 'OnXMReceiveRealData' 함수를 선언한다.

메시지를 받았을 때 함수를 실행할 수 있도록 'ON_MESSAGE'를 이용하여 메시

01 포인터에 관한 자세한 내용은 『소수와 RSA 알고리즘으로 배우는 Big Number 연산』(한빛미디어, 2013)을 참고하기 바란다.

지와 함수를 연결한다. [코드 AA-2]는 이때 실행되는 콜백 함수들의 선언부다.

[코드 AA-2] 콜백 함수 – OnXMReceiveData와 OnXMReceiveRealData

```
    BEGIN_MESSAGE_MAP(CRealTimeChartDlg, CDialog)
01      ON_MESSAGE( WM_USER + XM_RECEIVE_DATA, OnXMReceiveData )
02      ON_MESSAGE( WM_USER + XM_RECEIVE_REAL_DATA, OnXMReceiveRealData )
    END_MESSAGE_MAP()

    ##### 생략 #####

03  LRESULT CRealTimeChartDlg::OnXMReceiveData( WPARAM wParam, LPARAM lParam )
    {
        ##### 생략 #####
    }

04  LRESULT CRealTimeChartDlg::OnXMReceiveRealData( WPARAM wParam, LPARAM lParam )
    {
        ##### 생략 #####
    }
```

01 사용자 메시지인 'WM_USER'는 MFC에서 기본값이 정해져 있다. 사용자 메시지에 'XingAPI.h'에서 정의한 'XM_RECEIVE_DATA' 값을 더하여 새로운 사용자 메시지가 만들어진다. 즉, 'WM_USER'의 값인 '0x0400'과 'XM_RECEIVE_DATA'의 값인 '3'이 더해져서 새로운 사용자 메시지인 '0x0403'이 만들어지고, 이 메시지를 받았을 때 'OnXMReceiveData' 함수가 실행된다.

02 실시간 데이터를 받을 때 사용하는 메시지인 'XM_RECEIVE_REAL_DATA'와 이 메시지를 받았을 때 실행되는 'OnXMReceiveRealData' 함수를 'ON_MESSAGE'로 연결한다.

03 데이터 요청 시에만 실행되는 함수인 'OnXMReceiveData' 함수는 여기서 구현한다.

04 실시간 데이터를 받을 때마다 실행되는 함수인 'OnXMReceiveRealData' 함수는 여기서 구현한다.

프로그램을 구현하면서 필요한 데이터는 XingAPI에서 정의한 특정 함수를 사용하여 받아야 한다. 'Receive Data' 요청은 'XingAPI.Request()' 함수를 사용하고, 'Receive Real Data' 요청은 'XingAPI.AdviseRealData()' 함수를 사용한다. 이

두 함수 안에 데이터의 요청 값들을 넣어서 전송하는데 이때 구조체[02]를 사용한다. 구조체에는 요청하는 데이터의 속성값들을 넣는다. 구조체는 Xing API에서 제공하는 헤더 파일에 정의되어 있고, 이 구조체 안에 정의된 변수들의 값을 넣어준다.

[코드 AA-3]은 Xing API에서 정의한 요청 구조체 중 하나고, [코드 AA-4]는 이 구조체를 사용하여 증권사 서버에 데이터를 요청하는 방법의 예다. 여기서 설명하는 't8413' 구조체는 증권사에서 일반인이 사용할 수 있게 만들어 놓은 것이다.

[코드 AA-3] 주식 차트(일/주/월)를 요청하기 위해 사용되는 t8413InBlock 구조체(t8413.h, 이베스트투자증권 제공 파일)

```
01  typedef struct _t8413InBlock
    {
02      char shcode[6];       char _shcode;
03      char gubun[1];        char _gubun;
04      char qrycnt[4];       char _qrycnt;
05      char sdate[8];        char _sdate ;
06      char edate[8];        char _edate ;
07      char cts_date[8];     char _cts_date;
08      char comp_yn[1];      char _comp_yn ;
09  } t8413InBlock, *LPt8413InBlock;
```

01 'struct _t8413InBlock'은 구조체의 정의로, 앞에 'typedef'가 사용된 것은 09에서 't8413InBlock'과 '*LPt8413InBlock'으로 재정의하기 위해서다.

02 증권사의 API를 통하여 주고받는 데이터는 모두 문자열이며 이 문자열의 크기는 정해져 있다. 'shcode'는 종목 코드를 나타내고 6자리 문자열의 크기를 가진다. 예를 들어, '동화약품' 종목 코드는 '000020'로 6자리이며, 이 종목 코드를 'shcode'에 넣어준다. '_shcode'는 종목 코드의 속성값으로 1바이트 크기다. 't8413'의 모든 변수 뒤에 속성값이 있는데, 데이터를 요청할 때는 사용되지 않는다.

03 'gubun'은 1바이트 크기로, 요청하는 데이터의 주기 구분을 나타낸다. 예를 들어, 'gubun'에 2를 넣으면 '일' 데이터를 요청하는 것이고, 3을 넣으면 '주' 데이터, 4를 넣으면 '월' 데이터를 요청하는 것이다.

02 구조체에 관한 자세한 내용은 『소수와 RSA 알고리즘으로 배우는 Big Number 연산』(한빛미디어, 2013)을 참고하기 바란다.

04 'qrycnt'에는 요청 건수를 넣는데, 4바이트 크기의 문자열이므로 여기서 사용하는 250개 데이터를 가져오려면 '0250'과 같이 앞에 0을 넣어서 4자리의 문자열로 만들어야 한다.

05 'sdate'에는 요청하는 데이터의 시작 날짜를 넣는다. 예를 들어, 2013년 4월 18일은 '20130418'과 같이 8자리의 문자열로 만들어 입력한다.

06 'edate'에는 요청하는 데이터의 종료 날짜를 넣는다. 시작 날짜와 같은 형식이다.

07 'cts_date'에는 연속 일자를 넣는다. 연속적으로 데이터를 받을 때 사용한다.

08 'comp_yn'은 압축 여부를 입력한다. 'Y'면 압축된 데이터를 받는 것이고, 'N'이면 압축되지 않은 데이터를 받는다. 경험상 압축을 푸는데 시간이 소요되므로 항상 'N'을 선택하였다.

09 앞에서 'typedef'을 이용하여 구조체의 이름을 't8413InBlock'과 포인터 변수인 'LPt8413InBlock'으로 재정의하였다. 이 구조체를 사용하려면 변수를 't8413InBlock'과 'LPt8413InBlock'를 사용하여 선언한다.

이베스트투자증권에서 제공하는 'DevCenter'에는 모든 구조체에 대한 내용이 간략하게 잘 기록되어 있다. [코드 AA-4]는 앞에서 선언한 구조체에 어떻게 값을 넣어 사용하는지 보여준다. 구조체 변수를 선언하고, 구조체 안을 모두 비운 후 필요한 데이터 값들을 넣어서 요청한다. 이 책에서 사용하는 주식 데이터를 가져오기 위해서는 't8413'을 사용되는데, 't8413InBlock'과 't8413OutBlock'을 이용하여 요청한다. 증권사에 'InBlock'으로 요청하고, 'OutBlock'을 통하여 데이터를 받는다. 증권사 입장에서 요청은 안으로 들어오는 것이므로 'In'을 사용하고, 데이터를 주는 것은 밖으로 내보내는 것이므로 'Out'이라는 이름을 붙였다.

[코드 AA-4] 주식 데이터를 요청하는 Request 사용

```
01  t8413InBlock     pckInBlock;
02  FillMemory( &pckInBlock, sizeof( pckInBlock ), ' ' );
03  memcpy( pckInBlock.shcode, "000020", sizeof( pckInBlock.shcode));
04  memcpy( pckInBlock.gubun, "2", sizeof( pckInBlock.gubun));
05  memcpy( pckInBlock.qrycnt, "0250", sizeof( pckInBlock.qrycnt));
06  memcpy( pckInBlock.edate, "20140418", sizeof( pckInBlock.edate));
07  memcpy( pckInBlock.comp_yn, "N", sizeof( pckInBlock.comp_yn));
08  iXingAPI.Request(GetSafeHwnd(), "t8413", &pckInBlock, sizeof(pckInBlock),
```

```
            0," ",-1);
```

01 주식 데이터를 요청하는 구조체 't8413InBlock'의 변수인 'pckInBlock'을 선언한다. 변수의 'pck'는 패킷Packet의 약어다[03].

02 'FillMemory' 함수는 포인터가 가리키는 메모리에 정해진 크기만큼 특정 문자로 채우는 함수다. 여기서는 '&pckInBlock'이 메모리의 주소를 가리키고, 'sizeof(pckInBlock)'으로 구조체의 크기만큼 공간으로 채운다.

03 종목 코드인 'shcode'에 '동화약품'의 코드 '000020'을 넣어준다. 메모리 복사Memory Copy를 뜻하는 'memcpy'는 C 프로그래밍에서 많이 쓰이는 함수 중 하나다.

04 주기 구분에는 '2'를 넣어서 일 데이터를 받는다.

05 데이터 요청 개수는 네 자리의 문자열이므로 250개의 데이터를 요청하면 앞에 0을 넣어서 4자리를 맞춘다.

06 종료 날짜를 넣어준다.

07 압축하지 않은 데이터를 받도록 'N'으로 설정한다.

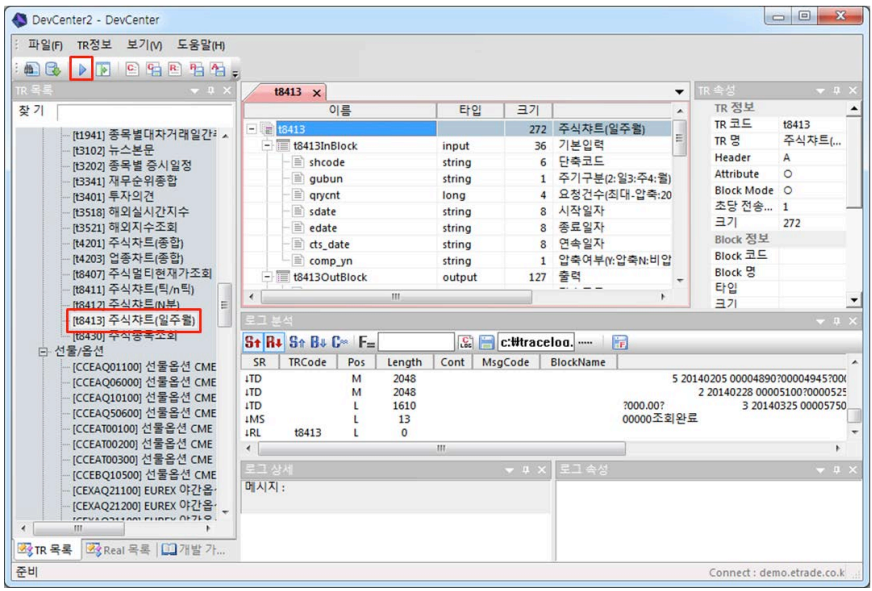
08 'Request' 함수를 사용하여 't8413'을 요청한다. 'GetSafeHwnd( )'는 현재 프로그램 화면의 윈도우 핸들Window Handle을 가져오는 MFC 함수다. 윈도우 프로그램은 화면에 보이는 윈도우마다 고유한 값이 있어서 각각의 윈도우는 정해진 고유의 값으로 접근할 수 있다. 'Request' 함수에 해당 프로그램 화면의 고유값인 윈도우 핸들을 알려주면 Xing API는 데이터를 받아서 해당 윈도우에 메시지 'XM_RECEIVE_DATA'를 사용하여 데이터를 가져가도록 알려준다. 즉, 윈도우 핸들을 가지고 해당 윈도우의 메시지 큐에 데이터를 받았다는 메시지를 넣어준다. 't8413'을 매개 변수로 데이터를 요청하고, 요청 구조체인 'pckInBlock'을 함수 안에 넣어준다. 구조체의 크기는 'sizeof(pckInBlock)'으로 명시한다[04].

Xing API는 'DevCenter'로 테스트할 수 있다. [그림 AA-1]은 'DevCenter'의 화면으로, 't8413'을 선택한 후 Play 버튼을 누르면 [그림 AA-2]처럼 TRTransaction 창이 열린다. 'In Block' 영역에 값을 입력하고 '조회' 버튼을 누르면 'Out Block' 영역에 데이터를 출력하여 주고받는 데이터를 테스트할 수 있다.

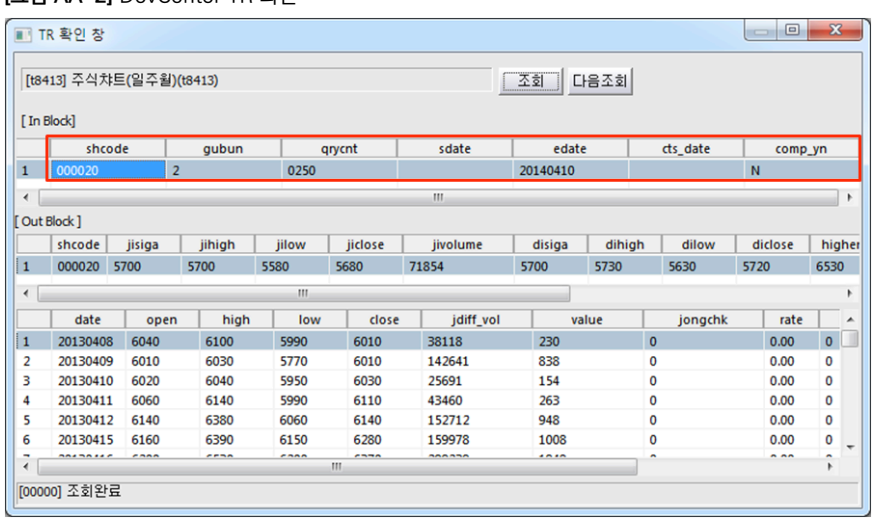
03 변수 이름은 프로그래머 마음대로 정할 수 있다.

04 C 프로그래밍에서는 포인터가 상당히 중요하다. 포인터는 주소값과 그 크기를 알려주어 메모리에 직접 접근해서 사용할 수 있게 해준다.

[그림 AA-1] DevCenter 메인 화면



[그림 AA-2] DevCenter TR 화면



증권사 API를 사용하려면 데이터를 받는 데 많은 시간을 할애해야 한다. 필자는 API를 이용한 프로그램을 만들면서 80% 이상의 시간을 API를 이해하는데 보내야 했다. API는 단시간에 습득할 수 있는 것이 아니다. 또한, 다른 사람이 사용하는 프로그램을 만들어야 하므로 프로그래머는 API를 완벽히 이해해야 한다[05].

API를 사용하는 것은 MFC에서 DLL[Dynamic Linked Library]을 사용하는 것이다. DLL 파일들은 윈도우에서 C++로 만든 클래스들의 집합이다. 윈도우 프로그래밍을 할 때 MFC는 DLL 파일들을 사용하지만, C#이나 비주얼 베이직, 엑셀 등은 COM[Component Object Model]을 사용한다. COM은 DLL 기반으로 윈도우 환경에서 여러 프로그램에 편리하게 사용할 수 있게 만든 것으로 DLL보다 속도가 느린 것이 단점이다. 주식 시장이 활발하게 움직일 때는 주가 변동이 아주 빠른데, 자동매매 프로그램을 만든다면 속도도 중요한 요소 중의 하나라고 생각한다. 다른 프로그래밍 언어와 비교하면 MFC는 배우는 데 다소 시간이 걸릴 수 있지만, 윈도우 환경에서 주식 프로그램을 만들 때는 MFC가 최고의 선택이라고 감히 말하고 싶다.

B Xing - 주식데이터 가져오기

주식의 모든 종목을 분석할 때 가장 중요한 것은 주식 데이터를 얻는 것이다. 여기서는 이 책에서 사용하는 주식 데이터를 증권회사로부터 가져오는 프로그램을 설명한다[06].

주식 데이터는 일, 주, 월별로 가져오는데, 이 데이터를 프로그램과 연동시키면 된다. 이 책의 내용을 이해했다면 분이나 틱[Tick] 데이터를 이용하여 단타 매매를 위한

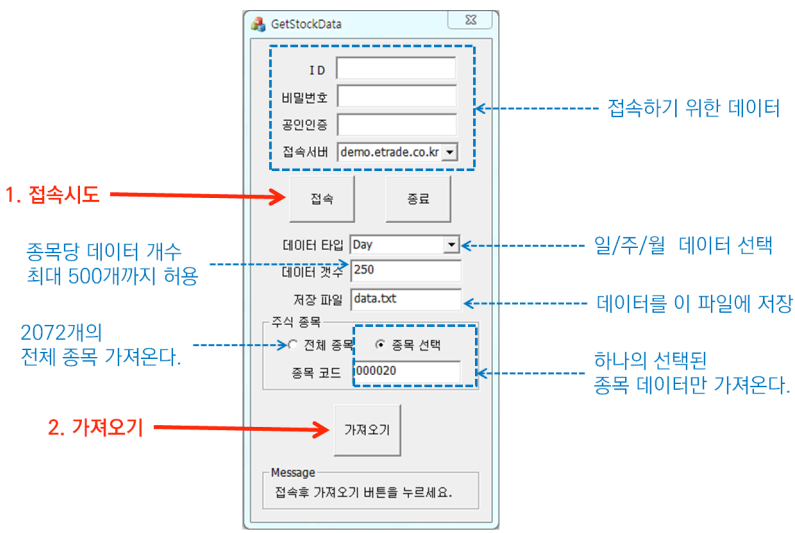
05 이베스트투자증권은 'DevCenter'를 제공하는 등 API에 많은 공을 들이고 있는 것 같다. 그 덕분에 개발하는 데 많은 도움이 되고 있다.

06 여기서 설명하는 프로그램은 이베스트투자증권의 이벤트에 등록한 프로그램으로, 등록한 프로그램의 소유권은 이베스트투자증권에 귀속되지만, 책으로 공개해도 괜찮다는 답변을 받아서 이 책에 수록한다. 당부하고 싶은 것은 평일 9시부터 15시까지 장중에는 데이터를 받지 않기를 바란다. 프로그램을 실행하여 데이터를 받으면 API를 제공해 주는 이베스트투자증권의 서버에 과도한 부하를 유발할 수 있고, 결국 안 좋은 영향을 미칠 수 있기 때문이다.

알고리즘 분석에도 사용할 수 있다.

[그림 AB-1]은 프로그램의 실행 화면이다. 프로그램을 실행하려면 접속 ID와 비밀번호를 입력하고 '접속' 버튼(1)과 '가져오기' 버튼(2)을 누른다. 종목 요청은 초당 요청 제한 수가 있어서 2,000개가 넘는 전체 종목을 받는 데 약 40분 정도가 걸린다[07].

[그림 AB-1] 주식 데이터를 가져오는 프로그램(GetStockData)



증권사에서 데이터를 가져오려면 증권사 서버에 자신의 계정으로 접속하고 데이터를 요청해서 받으면 된다. 이 책의 프로그램은 데이터를 한 종목씩 가져올 수도 있고, 전체를 가져올 수도 있다. 모든 데이터를 가지고 온 후 마지막에 저장 파일에 기록한다. 기록되는 형식은 이 책에서 사용하는 데이터 파일 형식이고, 가져온 데이터 파일을 프로그램에서 사용한다.

07 주식 종목은 1초에 한 번만 요청할 수 있다.

여기서는 Xing API와 관련된 중요한 코드 내용만을 설명한다. 프로그램 소스 코드에는 주석을 달았으므로 나머지 내용도 쉽게 이해할 수 있다.

[코드 AB-1]은 Xing API를 사용할 때 가장 먼저 수행하는 Xing API 초기화와 서버 접속 및 로그인 부분이다[08].

[코드 AB-1] Xing API 초기화와 접속 및 로그인(GetStockDataDlg.cpp)

```
// iXingAPI는 전체 프로그램에서 사용하는 Global 변수이므로 ‘ stdafx.cpp ’에 선언한다.
IXingAPI iXingAPI;

//-------------------------- 1. xingAPI 초기화--------------------------
if( iXingAPI.Init() == FALSE ) {
     CString strPath;
     int nLen = (int)GetModuleFileName( AfxGetInstanceHandle(),
     strPath.GetBuffer( MAX_PATH ), MAX_PATH );
     strPath.ReleaseBuffer( nLen );
     int nFind = strPath.ReverseFind( '\\' );
     if( nFind > 0 ) strPath = strPath.Left( nFind );
     if( iXingAPI.Init( strPath ) == FALSE ) {
          ::AfxMessageBox(_T("xingAPI DLL을Load 할수없습니다."));
     }
}

//-------------------------- 2. 서버 접속--------------------------
BOOL bResult;
if(m_server.GetCurSel() == 0) {
// 서버 콤보 박스에서 실서버가 선택되었다면 실서버에 접속
     bResult = iXingAPI.Connect( GetSafeHwnd(),"hts.etrade.co.kr",20001,WM_US
ER,3000,512 );
} else {
```

08 이베스트투자증권에서 제공하는 예제 코드로 서경자 님이 제공해 주셨다 필자가 Xing API로 MFC 프로그램을 만들 때 이 코드가 없었다면 시작이 매우 힘들었을 것이다. 이 책을 빌려 깔끔한 소스 코드를 제공해 주신 서경자 님께 감사드린다.

```
	bResult = iXingAPI.Connect( GetSafeHwnd(),"demo.etrade.co.kr",20001,WM_U
SER,3000,512 );
}
if( bResult == FALSE ) {// 접속 실패 처리
	CString strMsg = iXingAPI.GetErrorMessage( iXingAPI.GetLastError());
	::AfxMessageBox(_T("서버접속실패: ") + strMsg);
}

//------------------------- 3. 로그인 -------------------------
if( FALSE == iXingAPI.Login( GetSafeHwnd(), m_id, m_password, m_password2,
m_server.GetCurSel(), TRUE ) ) {// 로그인
	CString strMsg = iXingAPI.GetErrorMessage( iXingAPI.GetLastError() );
	::AfxMessageBox(_T("로그인실패: ") + strMsg);
}
```

[코드 AB-1]은 API가 제공하는 내용을 그대로 따라 해야 하는 부분이므로 설명을 생략한다. MFC에서 Xing API를 사용하려면 무조건 [코드 AB-1]과 같이 구현한 후에 원하는 프로그램을 만들어야 한다.

[코드 AB-2]는 '가져오기' 버튼을 눌렀을 때 실행되는 함수인 'OnBnClickedBtn Getdata' 함수를 구현한 부분이다. 전체 종목을 가져오려면 Xing API의 't8430'을 요청하고, 한 종목을 가져오려면 't8407'을 요청한다.

[코드 AB-2] '가져오기' 버튼을 눌렀을 때 호출되는 함수 - OnBnClickedBtnGetdata (GetStockDataDlg.cpp)

```
	void CGetStockDataDlg::OnBnClickedBtnGetdata()
	{
	##### 생략 #####

01	if(!m_radio) {
```

```
            t8430InBlock  pckInBlock;
            FillMemory( &pckInBlock, sizeof( pckInBlock ), ' ' );
            memcpy( pckInBlock.gubun, "0", sizeof( pckInBlock.gubun));
            EntryOrderError( iXingAPI.Request(GetSafeHwnd(), "t8430",
    &pckInBlock, sizeof(pckInBlock), 0, " ", -1) );
02      } else {
03          stock->allCompany.quantity = 1;

            t8407InBlock  pckInBlock;
            FillMemory( &pckInBlock, sizeof( pckInBlock ), ' ' );
            memcpy( pckInBlock.nrec, "001", sizeof( pckInBlock.nrec));
            memcpy( pckInBlock.shcode, m_jongmokcode, sizeof(
    pckInBlock.shcode));
            EntryOrderError( iXingAPI.Request(GetSafeHwnd(), "t8407",
    &pckInBlock, sizeof(pckInBlock), 0, " ", -1) );
        }
    }
```

---

01 'Radio'는 '전체 종목' 또는 한 종목만을 받는 '종목 선택' 항목이다. 만약 'm_radio'가 0이면 전체 종목을 받기 위해 't8430'을 요청한다.

02 '종목 선택' 항목이므로 종목 코드를 가지고 한 종목을 받기 위해 't8407'을 요청한다.

03 한 종목만 받으므로 'CStock' 클래스의 변수인 'stock' 종목의 총 개수를 1로 설정한다.

[코드 AB-2]에서 증권사 서버에 요청하여 데이터를 받으면, Xing API는 'XM_RECEIVE_DATA' 메시지를 메시지 큐에 넣고, [코드 AB-3]의 'OnXMReceiveData' 함수가 실행된다. 콜백 함수인 'OnXMReceiveData' 함수는 받은 데이터가 어떤 요청으로 받았는지를 판단한 후 TR 코드에 맞는 처리를 수행한다.

't8430'를 요청받으면 'stock' 구조체 안에 주식의 모든 종목 이름과 코드 번호를 저장하고, 한 종목 데이터를 받기 위하여 't8407'을 다시 요청한다. 't8407'를 요

청받으면 데이터에서 내용을 가져와서 해당 종목의 구조체에 저장한 후 1초 뒤[09]에 다음 종목 데이터를 가져오기 위하여 다시 't8407'을 요청한다. 부록에 수록된 소스 코드에는 주석 처리가 되어 있으므로 프로그램의 구조를 이해한다면 코드 분석은 쉽다. [코드 AB-3]에서는 중요한 부분만을 설명한다.

[코드 AB-3] 데이터를 받은 후 호출되는 콜백 함수 – OnXMReceiveData(GetStockDataDlg.cpp)

```
    LRESULT CGetStockDataDlg::OnXMReceiveData( WPARAM wParam, LPARAM lParam )
    {
        int i;
        Company *company;

        if( wParam == REQUEST_DATA ) {
            LPRECV_PACKET lpRecvPacket = (LPRECV_PACKET)lParam;

            //--------------------------- t8407 ---------------------------
01          if(strcmp( lpRecvPacket->szTrCode, "t8407") == 0) {

            ##### 생략 #####

            //--------------------------- t8430 ---------------------------
02      } else if(strcmp( lpRecvPacket->szTrCode, "t8430") == 0) {
            LPt8430OutBlock pOutBlock;

03          stock->allCompany.quantity = (int)( lpRecvPacket->nDataLength
    / sizeof(t8430OutBlock));

            for(i=0; i<stock->allCompany.quantity; i++) {
04              pOutBlock = (LPt8430OutBlock)(lpRecvPacket->lpData+(sizeof
    (t8430OutBlock)*i));

05              company = &stock->allCompany.company[i];
```

09 앞에서 설명한 것처럼 't8407'은 1초에 한 번만 요청할 수 있다.

```
06              CString hname((char*)pOutBlock->hname, sizeof(pOutBlock->hname));
                company->strName = hname;

07              CString shcode((char*)pOutBlock->shcode, sizeof(pOutBlock->shcode));
                company->strJongMok = shcode;
            }

08          flagNext = false;
09          company = &stock->allCompany.company[indexCompany];

10          t8413InBlock  pckInBlock;
            FillMemory(&pckInBlock, sizeof(pckInBlock ), ' ' );
            memcpy(pckInBlock.shcode, company->strJongMok, sizeof
    (pckInBlock.shcode));
            memcpy(pckInBlock.gubun, dataType, sizeof(pckInBlock.gubun));
            memcpy(pckInBlock.qrycnt, quantityData, sizeof(pckInBlock.qrycnt));
            memcpy(pckInBlock.edate, date, sizeof(pckInBlock.edate));
            memcpy(pckInBlock.comp_yn, "N", sizeof(pckInBlock.comp_yn));
11          EntryOrderError(iXingAPI.Request(GetSafeHwnd(), "t8413",
    &pckInBlock, sizeof(pckInBlock), 0, " ", -1));

            //--------------------------- t8413 ---------------------------
12      } else if(!strcmp( lpRecvPacket->szTrCode, "t8413" ) ) {
            LPt8413OutBlock1 pOutBlock1;

13          if(!flagNext) {
                LPt8413OutBlock pOutBlock = (LPt8413OutBlock)lpRecvPacket->lpData;
                CString shcode((char*)pOutBlock->shcode, sizeof(pOutBlock->shcode));

                for(i=indexCompany; i>=0; i--) {
                    if(!shcode.Compare(stock->allCompany.company[i].strJongMok)){
                        stock->allCompany.company[i].keyToRecv =
    CString((char*)lpRecvPacket->szContKey, sizeof(lpRecvPacket->szContKey));
                        break;
                    }
```

```
                }
14              flagNext = true;
15          } else {
                for(i=indexCompany; i>=0; i--) {
                    CString szContKey((char*)lpRecvPacket->szContKey,
sizeof(lpRecvPacket->szContKey));
                    if(!szContKey.Compare(stock->allCompany.company[i].keyToRecv)) {
                        company = &stock->allCompany.company[i];
                        break;
                    }
                }
16              int quantity = (int)(lpRecvPacket->nDataLength /
    sizeof(t8413OutBlock1));
                for(i=0; i<quantity; i++) {
17                  pOutBlock1 = (LPt8413OutBlock1)(lpRecvPacket->lpData +
    (sizeof(t8413OutBlock1) *i));
18                  company->quantity = quantity;

                    CString dateData( (char*)pOutBlock1->date, sizeof(pOutBlock1->
    date) );
                    company->data[i].date = dateData;

                    company->data[i].startVal=changeStringToLong((char*)pOutBlock1->
    open,8);
                    company->data[i].highVal=changeStringToLong((char*)pOutBlock1->
    high,8);
                    company->data[i].lowVal = changeStringToLong((char*)pOutBlock1->
    low, 8);
                    company->data[i].lastVal=changeStringToLong((char*)pOutBlock1->
    close,8);
                    company->data[i].vol=changeStringToLong((char*)pOutBlock1->
    jdiff_vol,12);
```

```
            }
19              Sleep(1000);

                indexCompany += 1;
20              if(indexCompany >= stock->allCompany.quantity) {
                    StoreDataInFile();
                    return 0L;
                }

                flagNext = false;
                company = &stock->allCompany.company[indexCompany];

21              t8413InBlock  pckInBlock;
                FillMemory(&pckInBlock, sizeof(pckInBlock ), ' ');
                memcpy(pckInBlock.shcode, company->strJongMok,
    sizeof(pckInBlock.shcode));
                memcpy(pckInBlock.gubun, dataType, sizeof(pckInBlock.gubun));
                memcpy(pckInBlock.qrycnt, quantityData, sizeof(
    pckInBlock.qrycnt));
                memcpy(pckInBlock.edate, date, sizeof( pckInBlock.edate));
                memcpy(pckInBlock.comp_yn, "N", sizeof( pckInBlock.comp_yn));
                EntryOrderError(iXingAPI.Request(GetSafeHwnd(), "t8413",
&pckInBlock, sizeof(pckInBlock), 0, " ", -1));
            }
        }
22      } else if(wParam == MESSAGE_DATA) {// 메시지를 받음
            LPMSG_PACKET pMsg = (LPMSG_PACKET)lParam;
            CString strMsg((char*)pMsg->lpszMessageData, pMsg->nMsgLength);
            if(!strMsg.Compare(_T("해당자료가 없습니다."))) {
                m_message.Format(_T("종목 코드가잘못되었습니다."));
                UpdateData(FALSE);
            }
            iXingAPI.ReleaseMessageData( lParam );
```

```
    } else if( wParam == SYSTEM_ERROR_DATA ) {// System Error를 받음
        LPMSG_PACKET pMsg = (LPMSG_PACKET)lParam;
        CString strMsg( (char*)pMsg->lpszMessageData, pMsg->nMsgLength );
        ::AfxMessageBox(strMsg);
        iXingAPI.ReleaseMessageData( lParam );
    } else if( wParam == RELEASE_DATA ) {// Release Data를 받음
        iXingAPI.ReleaseRequestData( (int)lParam );
    }

    return 0L;
}
```

01 't8407'은 한 종목을 가져올 때 종목 코드에 따른 종목 이름을 가져오기 위해서 수행한다.

02 't8430'은 전체 종목을 가져와서 종목 코드와 종목 이름을 'stock' 구조체에 넣어준 다음 첫 번째 종목의 데이터를 요청한다. 'lpRecvPacket->szTrCode'는 요청한 'TR'의 코드값을 가지고 있고, 이것으로 어떤 'TR'의 데이터를 받았는지를 확인하여 'TR'에 맞는 처리를 수행한다.

03 'lpRecvPacket'은 매개 변수로 받은 'lParam'으로, 받은 데이터의 메모리 주소값을 가지고 있다. 메모리의 처음 주소를 알고 있으므로 't8430'의 데이터의 형식에 맞추어 데이터를 가져온다. 'lpRecvPacket->nDataLength'는 받은 데이터 블록의 전체 크기 값을 가지고 있다. 이 값을 블록 크기로 나누어서 블록 개수를 얻는다. 블록 개수가 종목의 전체 개수다.

04 받은 데이터의 메모리에서 각 블록에 차례대로 접근한다. 메모리 개수와 블록 크기를 알고 있으므로 읽을 메모리로 직접 접근할 수 있다.

05 'stock' 클래스에서 한 종목의 구조체를 차례대로 가져온다.

06 종목 이름을 메모리에서 가져와서 구조체에 넣어준다.

07 종목 코드를 메모리에서 가져와서 구조체에 넣어준다.

08 한 종목에 대한 전체 데이터를 가져올 때 한번에 모든 데이터를 받는 것이 아니라 두 번에 걸쳐서 나누어 받는다. 첫 번째와 두 번째 받은 데이터를 구분하려고 'flagNext'가 사용되고, 'false'로 초기화한다.

09 'indexCompany'는 종목을 차례대로 가져오는 데 사용되는 숫자로, 0으로 초기화한다[10]. 즉, 첫 번째 종목의 데이터를 요청하기 위하여 첫 번째 종목의 구조체를 선택한다.

10 't8413'은 한 종목의 데이터를 요청하는 데 사용한다. 'InBlock'에 요청하기 위한 속성값들을 넣어주고 'Request' 함수를 사용하여 데이터를 요청한다.

10 이 값은 여기서 처음 사용되므로 0이 된다.

11 'EntryOrderError' 함수는 만약 요청 중에 오류가 발생했을 때 그 오류를 출력하기 위하여 사용되는 함수로, 'iXingAPI.Request'가 이 요청을 수행한다.

12 't8413'은 한 종목에 대한 데이터를 받을 때 수행되고, 두 번에 걸쳐 데이터를 받는다. 두 번째 받는 데이터에 필요한 종목의 데이터가 있으므로 'flagNext'로 첫 번째와 두 번째 데이터를 구분한다.

13 'flagNext'가 'false'면 '!flagNext'는 'true'가 되므로 여기서는 첫 번째로 받은 데이터일 때 수행된다. 해당 종목의 구조체에 연속 키를 나타내는 'szContKey'를 넣는다. 두 번째 받는 데이터에서는 이 연속 키를 확인하여 같은 연속 키를 가지고 있는 종목의 구조체에 두 번째 데이터를 넣는다.

14 첫 번째 데이터를 받은 후에는 'flagNext'를 'true'로 설정하면 이후에 받는 데이터는 두 번째 데이터로 처리된다.

15 두 번째 받는 데이터를 처리한다. 종목 구조체에서 연속 키가 같은 종목을 선택하여 데이터를 넣는다.

16 03과 마찬가지로 전체 블록 크기를 한 블록 크기로 나누어서 받은 데이터의 개수를 얻는다.

17 04와 마찬가지로 블록을 차례대로 접근하여 가리킨다.

18 한 블록에서 주식의 날짜/시가/고가/저가/종가/거래량 데이터를 가져와서 구조체에 넣어준다. 'changeStringToLong'은 숫자 문자열을 'long'형으로 바꾸어주는 함수로, 'stdafx.cpp' 파일에 구현되었다.

19 't8413'은 1초에 1번만 데이터를 요청할 수 있는 'TR'이므로 'Sleep(1000)'을 사용하여 1초의 지연을 준다.

20 만약 마지막 데이터를 받았다면 구조체에 저장된 데이터들을 파일에 저장하고 더는 데이터를 요청하지 않고 종료한다.

21 다음 데이터를 차례대로 요청하려면 't8413' 데이터를 받은 후 다음 종목 데이터를 다시 요청한다.

22 콜백 함수에서 매개 변수 'wParam'를 이용하여 받은 데이터의 종류를 알 수 있다. 'MESSAGE_DATA'는 증권사 서버로부터 메시지를 받았다는 뜻이고, 'SYSTEM_ERROR_DATA'는 시스템 에러 데이터를 받았다는 뜻이므로 데이터의 종류에 따라 적절한 코드 처리를 수행한다.

'GetStockData' 프로그램은 주식이 움직이는 동안은 실행할 필요가 없으므로 장중에는 전체 데이터를 받지 않는 것이 좋다[11]. 증권사 API를 사용하려면 API를 이해하는데 상당한 시간을 투자해야 한다. 데이터는 한 번에 받을 수도 있고, 여러 번에 걸쳐서 받을 수도 있다. 여러 번에 걸쳐서 받으면 콜백 함수가 여러 번 실행되므로 순서를 정하여 몇 번째로 받은 데이터인지를 확인해야 한다. 필자는 API 프로그램을 하면서 받은 데이터를 분석하는데 많은 시간을 할애했다. 때로는 한 문제를

11 · 많은 사람이 프로그램을 동시에 실행하면 증권사 서버에 과도한 트래픽을 줄 수 있다.

가지고 하루 이상 소비한 적도 있다. 어떤 프로그램을 누군가 만들었다면 나도 만들 수 있다 생각하고 끈기 있게 문제를 해결하는 자세가 필요하다. 프로그래머는 누구보다 많이 공부해야 하고, 항상 새로운 것을 습득해야 한다. 또한, 이것을 즐길 줄 알아야 한다.

C Xing - 실시간 그래프 그리기

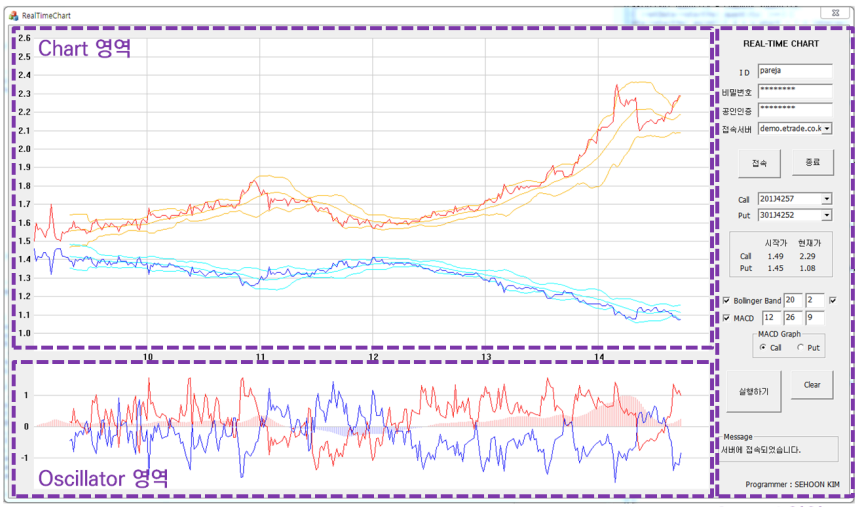
실시간 그래프를 그릴 때는 활발하게 움직이는 데이터를 가지고 그리는 것이 좋다. 그래서 가장 활발하게 움직이는 주식의 파생상품인 옵션 데이터를 가지고 실시간 그래프를 그리는 프로그램을 만들었다. 물론 여기서 그래프를 그리는 방법은 이 책에 수록된 내용에 기반을 두고 있으므로 Option.h와 Option.cpp 파일에 있는 내용을 어렵지 않게 이해할 수 있을 것이다. 앞에서 구현한 Stock.h와 Stock.cpp 파일을 Option.h와 Option.cpp로 바꾸어 주었을 뿐 대부분 구조는 비슷하다.

옵션은 주식보다 위험성이 크다. 우리나라 주식은 상한가와 하한가가 존재해서 하루 동안 주가가 15% 이상 움직일 수 없지만, 옵션은 장이 크게 움직일 때는 몇 배 이상의 가격으로 움직이고, 만기 이후에는 투자금이 0원이 될 수도 있다. 주식은 항상 오르지만은 않는데 개인이 공매도할 수 없는 우리나라에서는 주식이 항상 올라야만 이익을 얻을 수 있다. 하지만 주식을 좀 더 들여다보면 주가가 내려갈 때도 좋아하는 투자가들은 많다.

옵션은 'Call'과 'Put'이라는 두 종목이 항상 힘겨루기한다. KOSPI200을 가지고 이루어지는 옵션은 KOSPI200이 오르면 'Call'을 산 사람이 이익이고, 반대로 내리면 'Put'을 산 사람이 이익이다. 이론적으로는 주식이 오르고 내리는 것에 상관없이 언제나 이익을 낼 수 있는 시스템이지만, 실전에서 이익을 보는 사람은 많지 않다. 누군가 잃어야 따는 사람이 있는 전형적인 제로섬Zero-sum 게임이기 때문이다. 옵션은 우리나라에서 많이 투자하는 상품으로, 뉴욕시장보다 훨씬 큰 금액이

움직이고 있다.

[그림 AC-1] 실시간 차트의 변화를 그려주는 RealTimeChart



[그림 AC-1]은 옵션의 실시간 차트를 그려주는 'RealTimeChart' 프로그램의 실행 화면이다. 장이 열리는 9시부터 15시까지의 분봉을 한 화면에 그린다. 이베스트투자증권 계좌를 가지고 API 사용을 신청하면 실시간 차트를 볼 수 있다. 이 차트에서 마지막 분봉 2초마다 가격 변동이 있으면 그래프가 미세하게 변화되는 것을 알 수 있다.

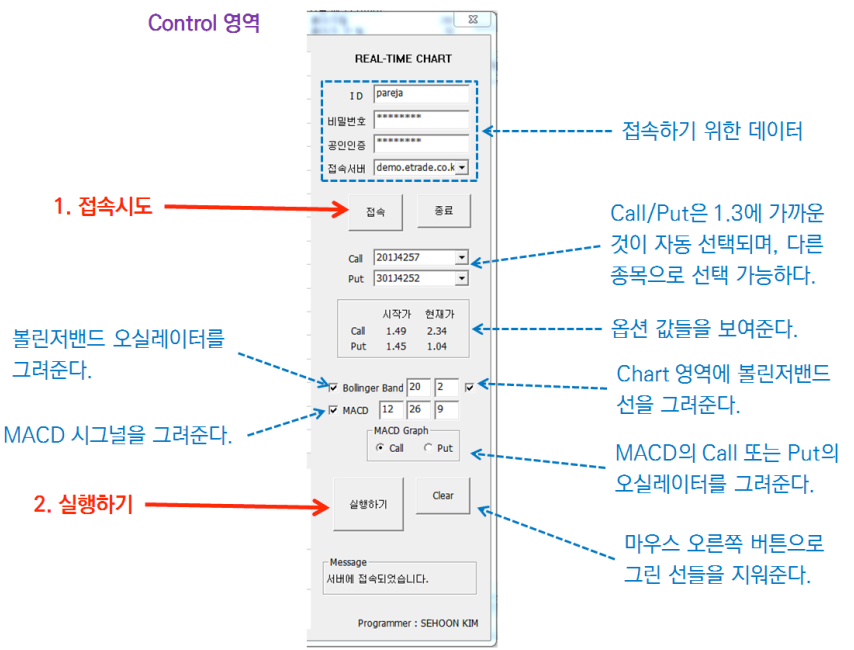
차트는 분봉의 종가만을 선으로 그려주며, 마지막 분봉이 현재가를 반영하여 움직인다. 프로그램은 2초마다 움직이지만, 5분마다 분 데이터를 받아와서 다시 그린다. 프로그램을 이해하려면 실행한 후, 최소 몇 분 동안은 차트를 관찰할 것을 권한다.

옵션은 'Call'과 'Put'에 19개의 행사가가 있는데, 가격이 1.3(13만 원) 근처에 있는 종목이 가장 활발하게 움직이는 편이고, 많은 종목이 상대적으로 거래가 많

지 않다. 그래서, 'RealTimeChart' 프로그램은 1.3 근처에서 시작하는 'Call'과 'Put'을 자동으로 선택하여 이 두 개만을 보는 것으로 옵션 시장 흐름을 볼 수 있다고 생각한다.

[그림 AC-2]는 차트 화면 오른쪽에 있는 'Control' 영역을 자세히 보여준다. 프로그램을 시작하려면 접속 정보를 입력한 후 '접속' 버튼을 누르고(1), 잠시 대기하다가 '실행하기' 버튼을 누른다(2). 프로그램의 사용법은 쉽게 익힐 수 있다. 제공되는 소스 코드를 응용하여 자신만의 프로그램을 만들기 바란다.

[그림 AC-2] RealTimeChart의 Control 영역



'RealTimeChart' 프로그램은 상업적으로 판매되는 프로그램은 아니지만, 하나의 완성된 MFC 프로그램이다. 소스 코드에 간단한 주석이 되어 있어서 여기서는

자세한 설명을 생략한다. 프로그램의 많은 부분이 차트를 그리기 위해서 구현되었다. 소스 코드를 이해하려면 코드를 자세히 보는 것보다는 처음에는 함수 위주로 보는 것이 좋다. 즉, 무엇을 하는 함수인지를 알고 확장해서 보는 것이 좋다. 그러고 나서 구현하는 코드가 있으면 필요한 부분을 응용하기를 바란다. 다른 사람의 프로그램을 응용한다는 것은 'Copy & Paste'가 아닌 자신만의 새로운 코드를 만드는 것이다. 또한, 코드의 변수 이름까지 자신이 알아보기 쉽게 변화를 주는 것도 좋다.

프로그래밍하는 사람들은 주식을 많이 모르고, 주식을 하는 사람은 프로그램을 만들기 힘들다. MFC 프로그래밍을 쉽게 배울 수 있도록 이 책을 집필했지만, 주식도 이해하고 배울 수 있다. 주식을 하는 많은 사람이 차트를 보지만, 차트의 특성을 깊이 있게 알지 못하는 편이다. 그러나 보조 지표를 어떻게 만드는지 알게 되면 보조 지표의 장단점을 파악할 수 있어서 좀 더 깊이 있는 분석이 가능해진다. 이 책이 주식 프로그램의 많은 것을 알려주지는 않지만, 시작하는 방법은 알려준다고 생각한다. 머지않아 많은 사람이 자동매매 프로그램으로 주식을 할 것이다. 우리가 무슨 일을 하든 컴퓨터는 항상 수익을 내려고 끊임없이 일할 것이다. 주식 프로그램의 궁극적인 목적은 좋은 자동매매 시스템을 만드는 것이다. 2020년 전에 주식 시장에서 프로그램 싸움이 보편화하리라 생각한다. 그것은 알고리즘 싸움이므로 좋은 프로그램을 만드는데 이 책이 도움되기를 바란다.

집필을 마치며

'How-To Series'는 프로그램을 어떻게 만들 것인지를 함께 이해해가는 시리즈 형태로, 재미있는 주제로 독자의 프로그래밍 실력 향상을 돕는 목적이 있다. 프로그램은 단시간에 배울 수 있는 영역이 아니라서 꾸준한 시행착오를 겪으면서 향상된다. 재미있는 주제로 프로그램을 하나씩 만들어 가면 자신도 모르게 실력이 향상될 것이다. 이 책의 코드를 그저 실행만 하는 사람과 이 책을 참고로 자신만의 프로그램을 만드는 사람의 차이는 크게 벌어질 것이다.

이 책에 나오는 코드는 부분적으로 구현하였다. 즉, 완성된 코드가 아니다. 독자가 스스로 하나의 프로그램에 이 책에서 구현된 모든 내용을 집어넣기를 바란다. 또한, 자신만의 프로그램을 만들려면 기능적인 부분을 많이 구현하는 것이 좋다. 예를 들어, 이 책에서는 마우스 버튼을 누르면 선을 그리게 구현했지만, 자동매매가 이루어졌을 때 매수와 매도를 자동으로 다른 색으로 그리게 만들 수 있을 것이다. 'How-To Series'는 완성된 프로그램을 보여 주지는 않지만, 프로그램을 만들 수 있는 기본적이고 다양한 방법들을 알려준다.

주식이라는 주제로 프로그램 책을 집필하면서 필자도 주식에 좀 더 깊이 있게 접근을 할 수 있었다. 필자가 책을 쓰는 이유 중의 하나는 지식을 기록하고 싶어서다. 내가 아는 지식은 내가 죽으면 없어지지만, 기록된 지식은 다른 사람에게 전달할 수 있다. 또한, 학생을 가르치면서 교사 또한 배우듯, 주식 프로그램 책을 집필하면서 주식에 대해서 좀 더 다양한 내용을 알 수 있었다.

필자의 주변에도 주식하는 사람들이 있지만, 대부분 이익을 보지 못했다. 수시로 주식 시세를 확인해야 하므로 직장생활을 하면서 주식을 하는 것은 안 좋은 일이라고 생각한다. 일도 안 되고 주식도 제대로 할 수 없는 상황이 올 수 있다. 가능할

지는 모르지만, 시스템 트레이딩은 컴퓨터만 켜놓고 있으면 컴퓨터가 돈을 벌어오는 시스템이 되어야 한다. 내가 낮잠을 자도 컴퓨터는 열심히 일하게 하는 것이 프로그래머가 할 일이다. 물론, 이상적인 이야기일 수 있지만, 먼 훗날 누군가는 그런 시스템을 만들 것이라 필자는 생각한다.

이 책의 독자에게 당부하고 싶은 것은 주식 데이터로 자신의 알고리즘을 검증하고 주식을 하라는 것이다. 주식한다는 것은 회사를 하나 차렸다고 볼 수 있다. 개인사업자가 되어 투자하고 돈을 벌려면, 충분한 준비가 선행되어야 하지 않을까? 주식시장에는 위험한 믿음이 있는데, 자신이 선택한 종목이 언젠가는 오를 것이라고 굳게 믿는 것이다. 아주 위험한 생각일 수 있다. 또 한 가지는 다른 사람으로부터 추천을 받아서 주식을 하는 사람이다. 추천 종목을 50%만 맞추어도 잘 맞히는 것이다. 주식은 오르기도 하고 떨어지기도 한다. 필자는 남에게 종목을 말하지 않는다. 혹여나 그 종목이 떨어지면 어찌 감당할까?

이 책의 주제는 주식이지만, 프로그래머인 필자의 관점에서는 MFC 프로그램을 어떻게 만드는지를 설명한 책이다. 'How-To Series'의 세 번째 주제는 이미지 인식인 OCR을 다룰 예정으로 C# 언어로 개발할까 생각했지만, 마이크로소프트에 국한된 언어는 좋지 않을 것 같아 MFC의 C++로 구현하고자 한다. 물론, 이 책에서 MFC에 대한 기본적인 내용은 설명하였으므로 중복 설명은 하지는 않는다. 첫 번째 책인 『소수와 RSA 알고리즘으로 배우는 Big Number 연산』(한빛미디어, 2013)에도 프로그램을 만드는 필자의 노하우가 설명되어 있지만, 다른 책에서 다시 다루지는 않는다. 책을 사지 않는 독자라도 필자의 개인 홈페이지[12] '집필' 게시판에서 첫 책의 초안을 볼 수 있다.

두 번째 책을 출판할 수 있게 해준 한빛미디어와 정지연 과장님께 감사드린다. 마지막으로 끝까지 읽어준 독자께도 감사드린다.

12 http://kimsehoon.com