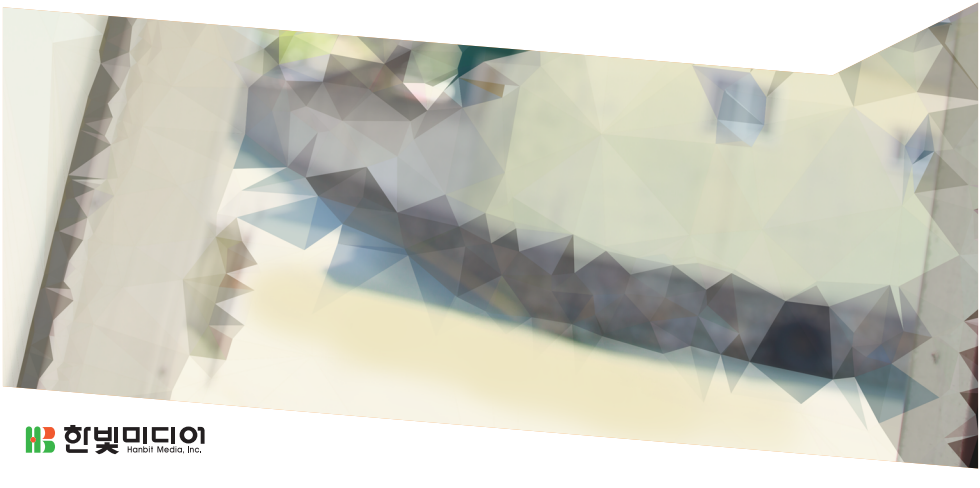
Hanbit RealTime 112

OCR 프로그래밍

이미지 인식

김세훈 지음



한빛미디어 Hanbit Media, Inc.

OCR 프로그래밍 이미지 인식

김세훈 지음

한빛미디어 Hanbit Media, Inc.

표지 사진 **김세훈**

이 책의 표지는 저자 김세훈 님이 보내 주신 풍경사진을 담았습니다.
리얼타임은 독자의 시선을 담은 풍경사진을 책 표지로 보여주고자 합니다.

사진 보내기 **ebookwriter@hanbit.co.kr**

OCR 프로그래밍 이미지 인식

초판발행 2015년 8월 21일

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 978-89-6848-773-6 15000 / **정가** 13,000원

총괄 배용석 / **책임편집** 김창수 / **기획·편집** 정지연 / **교정** 이미연
디자인 표지/내지 여동일, 조판 최송실
마케팅 박상용 / **영업** 김형진, 김진불, 조유미

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

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**

"내가 지금 뭐 하고 있지?" 2010년 40살이 되어 불현듯 나를 돌아봤다. 열심히 살려고 부단히 노력했던 것 같은데, 회사와 집만을 오가며 답답한 삶을 살고 있음을 느꼈다. 그래서 다짐했다. "이제부터 즐기면서 살자." 그런데 삶은 녹록하지가 않았다. "되는 일보다 안 되는 일이 더 많다. 그것이 인생이다."라고 하시는 아버지의 말씀도 진리처럼 느껴졌다. 몇 년이 흐른 후 내가 변해 있음을 느꼈다. 무엇이 나를 변화하게 했을까? 그건 지금까지 5년 넘게 매일같이 인생에 대한 글을 쓴 일이었다. 청소년기에 가치관이 정립된다고 학교에서는 가르치지만, 나는 죽을 때까지 가치관을 정립하며 죽을 때도 깨우치지 못한 진리들이 많을 것이다. 좋은 말씀을 써 오라는 숙제를 받은 초등학생 조카에게 필자는 말했다. "일기 쓰면 인생이 변한다." 하루의 일과가 아닌, 인생의 진지한 글들을 써 보세요. 시간이 한참 흐른 후 자신이 변해 있음을 발견하게 됩니다.

저자는 현재 '지식소프트' 1인 회사에서 부장으로 재직 중이다.

언어를 배울 때 사전을 찾아본 적이 있을 것이다. 사전에서 원하는 단어를 찾는 방법은 간단하다. 정렬된 순서에 의해서 찾고자 하는 단어가 일치하면 찾은 것이다. 인식 프로그램의 가장 대표적인 것이 전자사전인데, 전자사전에서 단어를 찾는 것도 일반 사전에서 단어를 찾는 것과 마찬가지다. 데이터베이스가 구축되어 있고, 찾고자 하는 단어를 찾으면 된다. 모든 인식 프로그램이 이 원리를 따르고 있다. 그렇다면 이미지나 소리의 인식은 무엇이 다를까? 프로그램에서 새로운 데이터 타입을 만들고 사전처럼 기존에 만들어진 데이터와 비교할 때 정확한 일치가 어려우므로 확률을 사용한다는 점이다. 즉, 일치하는 확률이 높은 것을 결과값으로 결정하며, 얼마나 정확한 결과값을 도출하느냐가 인식률의 높고 낮음을 나타낸다.

문자 인식에 대한 기술은 오래전에 나왔으며, 이제는 소리 인식으로 넘어갔다. 그러나 기술 공개는 이루어지지 않고 있다. 이 책에서 설명하는 내용은 필자가 생각해낸 방법이므로 이 방법만이 정답은 아니다. 하지만 개인적으로 이 방법이 문자 인식OCR을 하는 최적의 방법이라 생각한다.

데이터 타입이 무엇인지는 'How-to Series' 1편인 『소수와 RSA 알고리즘으로 배우는 Big Number 연산 - 구조체와 자료구조의 이해』(한빛미디어, 2013)에서 설명하였으며, 프로그래밍의 시작은 이 데이터 타입을 이해하는 것이라고 본다. 이 책에서도 문자 이미지를 담기 위한 데이터 타입을 가장 먼저 다룰 것이다. 여기서 다루는 새로운 데이터 타입을 이해하면 왜 한글 인식이 영문 인식보다 어려운지를 알게 될 것이다.

그다음으로는 문자 이미지를 어떻게 데이터 타입에 넣는지를 다룬다. 이는 문자

인식에서 가장 어려운 부분이다. 하나의 문자 이미지가 문자 데이터 타입에 담기면 기존에 만들어 놓은 기본 데이터 타입과 비교하면 된다. 전체 이미지에서 하나의 문자 이미지만을 가져오는 방법은 이미지마다 다르게 적용될 수 있기에 이미지를 분석하는 기술을 요구한다.

이미지 인식의 마지막은 이미지 변형Image Transform이다. 이미지 변형은 포토샵 같은 프로그램에서 주로 사용하는데, 간단한 이미지 변형으로 인식률을 어떻게 높이는지 알게 될 것이다. 이미지 변형에서는 수학적인 내용이 포함되나 중학교 수준의 삼각함수를 이해하고 있는 독자라면 쉽게 이해할 수 있다.

'How-to Series'는 프로그램 언어를 설명하는 책이 아니며, 프로그램을 어떻게 만드는지를 독자와 함께 공부하려는 목적으로 만들었다. 이 책을 가장 효과적으로 보려면 전체를 한 번 빨리 훑고 직접 프로그램을 만들면서 막히는 부분은 이 책을 보면서 풀어나가는 것이 좋다.

'How-to Series'는 1년에 한 가지 주제로 펴내고 있다. 책마다 개별 주제로 만들어지지만 먼저 출간된 책에서 설명한 내용을 이후의 책에서는 생략하는 경우가 있다. 시리즈 3편인 이 책에서 구현하는 프로그램은 MFC를 사용하는데, MFC는 『MFC 프로그래밍 : 주식 분석 프로그램 만들기』(한빛미디어, 2014)에서 상세히 설명하였기 때문에 이 책에서는 자세한 설명을 생략하였다. 예를 들어, '도구 상자'를 사용하여 버튼을 만든다거나 클래스를 추가하는 방법 등이 생략되었다. MFC에 대해 충분히 이해하고 있지 않다면 『MFC 프로그래밍 : 주식 분석 프로그램 만들기』를 참조하면서 보기를 권한다.

세 번째 책인 『OCR 프로그래밍: 이미지 인식』은 힘들 때 항상 함께해 준 나의 친구 '박흥준'에게 바친다. 또한, 나의 사랑하는 조카 '신효인'에게 선물로 준다. 나의 영원한 모델인 효인이의 개인 사진사로 활동하면서 조금 더 사진에 대해 관심을 갖게 되었고, 이미지 인식에 쉽게 접근할 수 있었다고 생각한다.

대상 독자 및 참고사항

이 책은 C++를 알고 있는 초·중급자를 대상으로 Windows 환경에서 MFC로 OCR 프로그램을 만드는 방법을 설명합니다. OCR을 주제로 처음부터 단계적으로 프로그램을 만들어 갑니다. 사진에서 문자를 인식하는 OCR 기술은 다양한 프로그램으로 응용될 수 있는 기술입니다. 이 책에서는 Windows 환경에서 구현하였지만, 동일한 원리로 스마트폰 애플리케이션에서 응용할 수 있습니다.

이 책에 수록된 코드들은 Windows에서 생성하여 실행해야 하며 Visual Studio 2008에서 구현되었습니다.

예제 테스트 환경

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

예제 파일은 다음에서 다운로드할 수 있습니다.

- http://www.hanbit.co.kr/exam/2773

'How-to Series'는 방법론적인 부분들에 초점을 맞추어 만들어진 책입니다. 일방적인 지식의 전달이라기보다는 "이러한 방법도 있구나"하고 이해하는 것이 중요합니다.

차례

chapter 1

OCR이란 무엇인가?

인식은 인공지능AI의 한 분야라고 볼 수 있다. 그 시작은 사전과 같은 간단한 것이나 궁극적으로 추구하는 것은 사람 간의 의사소통일 것이다. 지리적인 문제와 다양한 언어는 인간의 의사소통을 단절시켰지만, 인공지능은 기계를 통한 연결로 다른 지역이나 다른 나라 사람과의 의사소통을 원활히 하기 위해 발전해 왔다.

그런 면에서 음성 인식은 많이 발전한 듯 보이지만 아직 시작 단계다. 그렇다면 이미지 인식은 어떨까? 이 책에서 다루려는 이미지 인식은 문자에 국한되고 일반적인 사물을 인식하는 것은 현재로써는 어렵다. 하지만 2차원적 이미지 인식도 음성 인식과 별반 다르지 않다. 그런데도 음성 인식보다 이미지 인식이 먼저 발전한 이유는 무엇 때문일까? 일정한 틀 안에 이미지를 담기가 더 쉬웠기 때문이다. 문자를 나누기가 조금 더 쉬웠고 프로그램 언어로 표현하는 방법이 용이했다.

1999년에 처음으로 문자 인식 펜을 본 적이 있다. 종이에 인쇄된 알파벳을 따라 그으면 단어의 뜻이 펜 표면에 뜨고, 스캔한 글이 인식되어 워드 파일로 만들어졌다. 이후 문자 인식OCR, Optical Character Recognition은 더 발전하고 많이 보편화되었다. 예를 들어, 주차장에서 카메라로 자동차 번호판을 읽는 것도 문자 인식이다.

이 책의 마지막에 자동차 번호판 인식을 다루지만, 일반 카메라로 찍은 사진이기에 실제 환경과는 다른 방법으로 인식할 것이다. 시중의 자동차 번호판 인식은 일반 카메라가 아닌 특수 카메라로 찍기 때문에 문자 부분과 바탕 부분이 명확히 흑백으로 구분되어 인식하기가 훨씬 용이할 수 있다.

이 책을 통해 OCR을 이해하고 응용한다면 일정한 형태의 문자를 인식하는 방법을 쉽게 찾을 수 있을 것이다.

1.1 사진에서 문자 이미지 가져오기

사진에서 문자를 인식하기 위해 가장 먼저 해야 할 일은 문자 이미지 부분을 알아내는 것이다. 이 책에서는 이를 파싱한다고 부를 것이다. 이제부터 이미지 파싱이라고 하면 이미지에서 문자 부분을 떼어 낸다고 이해하면 된다. 문자 이미지는 사각형으로 따낼 것이며 2차원(평면)의 사진에서는 색으로만 구분하기 때문에 정해진 문자 색을 기준으로 문자 이미지를 인식할 것이다. 또한, 문자 색으로 지정된 특정 색 이외의 것은 모두 바탕으로 생각하고 무시하면 된다. 즉, 문자 색이 검은색이라면 사진에서 검은색 이외의 부분은 모두 흰색이고, 사진 한 장에는 검은색과 흰색만이 존재한다고 보면 된다. 즉, 사진을 두 가지 색으로만 나누어 문자와 바탕으로만 생각한다. 흰색 종이 위에 다양한 색의 문자들이 있다면, 이때는 흰색을 제외한 나머지 색을 검은색으로 바꾸어 두 가지 색으로만 이루어진 이미지를 만들면 된다.

파싱의 기본은 문자와 바탕이라는 두 가지 색으로만 이루어진 이미지를 만드는 것이다. 하지만 실제 프로그래밍에서는 색을 바꾸는 작업은 하지 않을 것이다. 이 작업도 CPU의 많은 작업을 필요로 하기 때문에 프로그램의 속도를 늦출 뿐이다. 다음 그림은 이미지에서 문자를 인식하기 위해 어떻게 변환 작업을 하는지를 보여준다.

[그림 1-1]은 문자 색을 기준으로 나머지를 바탕으로 만들었고, [그림 1-2]는 바탕 이외의 색을 모두 문자로 판단하여 변환하였다. 이때 중요한 것은 변환한 후의 이미지는 검은색 문자와 흰색 바탕으로만 이루어졌다는 것이다. 무엇을 문자로 인식하느냐에 따라 파싱의 처음 작업은 다양하게 이루어질 수 있다.

그림 1-1 문자 색 기준의 이미지 변환

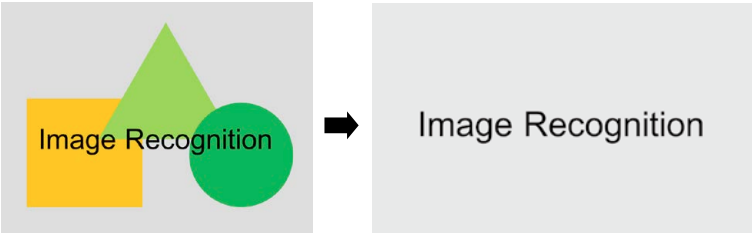
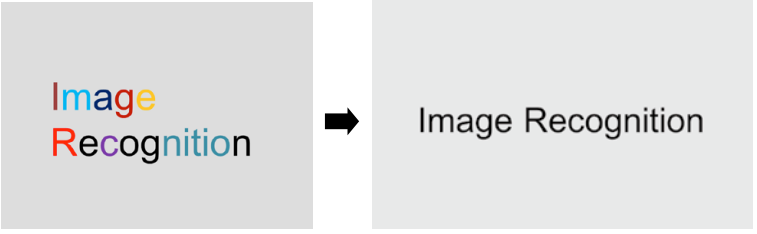


그림 1-2 바탕색 기준의 이미지 변환



앞에서와 같이 인식할 문자가 명확하다면 파싱 작업을 하여 개별 알파벳으로 나누면 된다. 'Image Recognition' 단어는 16개의 알파벳으로 나누면 되는데 나눈 문자는 [그림 1-3]과 같이 개별 이미지에 꽉 차게 보일 것이다. 즉, 검은색 부분을 기준으로 나누어 사각형의 테두리에는 항상 검은색이 존재하게 된다.

문자를 하나씩 나누는 작업은 이미지마다의 특성을 분석해야 한다. 실제 이미지는 기울어져 있을 수도 있고, 사진 속의 이미지라면 사진을 찍는 각도에 따라 이미지가 틀어져 있을 수도 있다. 하지만 이 책에서는 쉽게 이해하기 위해 깨끗한 이미지를 예로 사용하여 쉬운 파싱을 할 것이다. 문자 인식에서 가장 어려운 부분이 파싱인데, 다행히도 문자와 문자 사이는 겹친 부분이 없기에 파싱이 용이하다. 각각의 문자가 겹쳐 있다면 인식이 불가능할 수도 있다.

[그림 1-3]과 같이 하나씩 나누어진 문자 이미지는 인식을 위해 크기가 같은 이미지로 변환해야 한다. 모든 문자는 비교할 수 있는 일정한 틀 안에 담아야 하기 때문이다.

그림 1-3 하나의 문자로 나누기

[그림 1-4]는 가로, 세로의 크기가 같은 사각형 안에 각 알파벳을 담은 것이다. 하나의 문자는 늘려질 수도 있고 축소될 수도 있다. I[아이]는 사각형 안에 꽉 차게 된다. 그렇다면 소문자 l[엘]은 어떻게 될까? 이 책에서는 문자 형식을 Arial Type으로 진행을 하기 때문에 대문자 I[아이]와 소문자 l[엘]은 같은 이미지로 나타난다. 이러한 경우에는 문자가 처음 시작하는 문자인지를 판단해야 하며 프로그램으로 이 부분을 해결해야 한다.

그림 1-4 크기가 같은 문자 만들기

문자 인식 프로그램에서는 [그림 1-4]와 같이 이미지 자체를 일정한 크기로 변환하는 작업을 하지 않는다. 이러한 작업은 속도에 영향을 주는 불필요한 작업이므로 문자 인식 프로그램에서는 이미지의 변환 없이 바로 연산이 가능한 데이터 타입에 넣어 준다. 파싱을 소개하면서 쉽게 이해하기 위해 앞에서와 같이 시각적으로 접근하였다. 지금까지의 내용을 이해하였다면 다음 내용인 데이터 타입에 하나의 문자를 넣는 작업을 쉽게 이해할 수 있을 것이다.

1.2 연산이 가능한 데이터 만들기

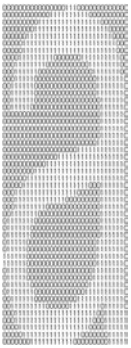
이미지 안의 문자는 단지 그림일 뿐이다. 프로그래밍하기 위해서는 문자 이미지를 데이터 형태로 가지고 있어야 한다. 우리는 파싱을 통하여 문자 이미지를 검은색과 흰색의 두 가지 색으로만 표현하였다. 여기에 더하여 문자의 크기도 일정하게 만들었다. 그렇다면 이것을 어떻게 새로운 데이터 타입 안에 담을 수 있을까?

컴퓨터에서 가장 작은 단위는 bit이며, 하나의 bit는 0과 1의 값으로 이루어져 있

다. 문자의 검은색을 1로 바꾸고 바탕의 흰색을 0으로 바꾼 후 bit를 사각형 안에 넣는다면 앞에서 파싱한 문자 이미지를 표현할 수 있다. 즉, [그림 1-5]처럼 'a'라는 이미지를 1,536개의 bit로 표현하는 것이 가능하다(가로 32개×세로 48개 = 1,536개).

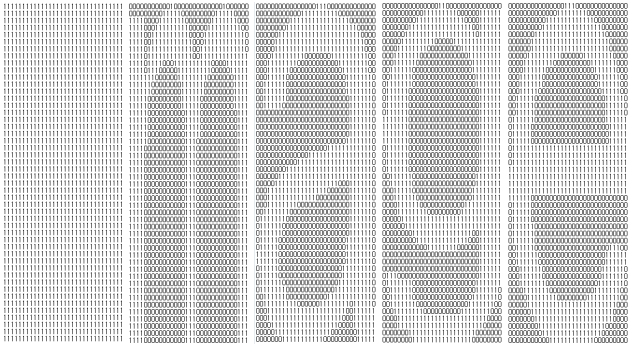
[그림 1-5]를 보면 'a'라는 것을 쉽게 알 수 있다. 0과 1만을 가지고도 시각적으로 쉽게 어떤 문자인지를 알 수 있다.

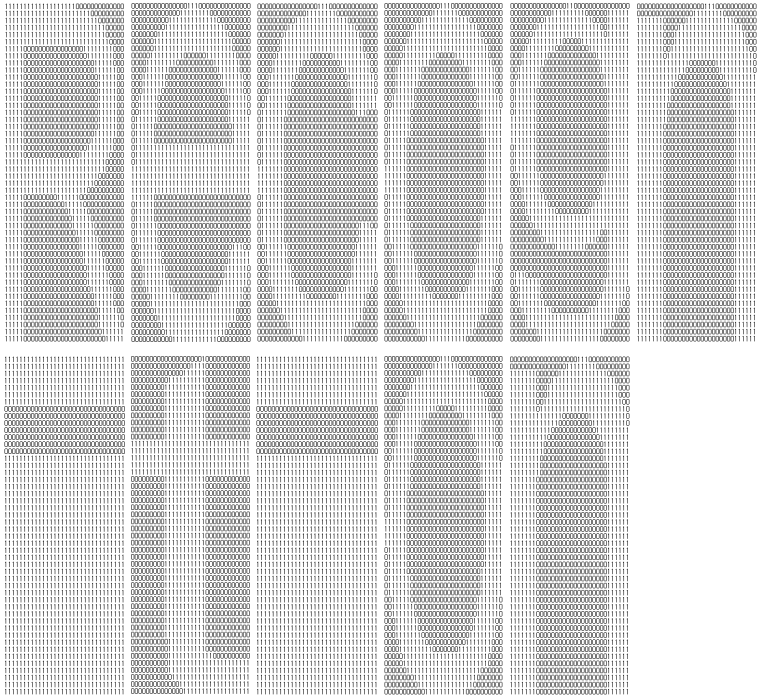
그림 1-5 0과 1로 표현된 문자 'a'



이와 같은 방법으로 'Image Recognition'이란 단어를 변환한다면 [그림 1-6]과 같이 나타날 것이다.

그림 1-6 0과 1로 표현된 'Image Recognition'





이미지의 크기를 32×48의 크기 안에 담은 것은 이유가 있는데, 프로그램 언어에서 정수형Integer 데이터 타입의 크기가 32bits이기 때문이다. 즉, 48개의 정수Integer를 하나의 새로운 데이터 타입으로 만들면 우리는 한 문자를 담는 그릇을 만든 것이다. 연산이 가능한 데이터 타입을 만드는 것은 프로그래머의 선택이며, 데이터 타입의 크기를 크게도 작게도 만들 수 있다.

이 책에서 다루게 될 데이터 타입은 두 가지다. 하나는 정수에 기반한 데이터 타입이며, 다른 하나는 8×12bits의 크기를 위한 문자형Character 데이터 타입에 기반한 데이터 타입이다. 앞으로 작은 문자 이미지에는 작은 그릇을 사용할 것이고, 큰 문자 이미지에는 큰 그릇을 사용할 것이다. 작은 이미지를 큰 데이터 타입에 넣는 것은 무의미하고 연산을 위한 속도도 더 느려지기 때문이다.

지금까지의 내용이 이미지 인식의 핵심이라 할 수 있다. 어느 정도 프로그래밍을

하는 독자는 이미지 인식에 어떻게 접근해야 하는지 이해하였을 것이다. 이해가 안 되었어도 앞으로 다루는 프로그래밍을 통하여 이미지 인식에 대하여 조금 더 깊이 있게 이해할 수 있을 것이다.

1.3 확률을 이용한 인식

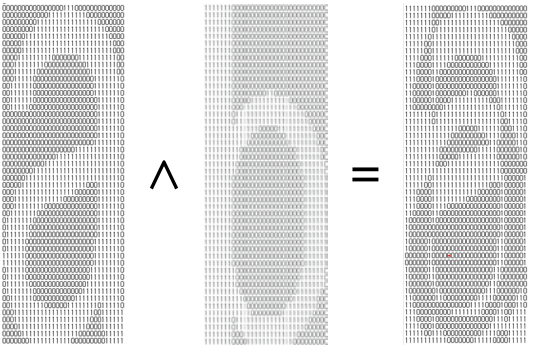
이미지 인식은 비교할 대상이 있어야 한다. 여기서는 비교를 위해 만들어진 이미지를 기본 이미지Standard Image라고 부르겠다. 이 책에서는 영문과 숫자만 인식하는데, 총 62개의 기본 이미지를 만들어서 사용한다(영문 소문자 26개+대문자 26개+숫자 10개). 인식할 이미지는 62개의 기본 이미지와 차례대로 비교하며 가장 근접한 이미지를 채택하여 결과값으로 결정한다. 그렇다면 연산할 수 있게 만든 데이터로 어떻게 bit가 같은지 비교하는지 알아보자.

프로그래밍 언어의 연산자 중 친근하지는 않지만 암호 프로그래밍 등을 하면 가장 많이 사용하는 것이 XOR(Exclusive OR, ^) 연산자다. 이 책에서도 XOR 연산자를 많이 사용할 것이다. XOR 연산자의 연산 결과는 다음과 같은데, 연산되는 bit가 같으면 0을, 다르면 1을 결과로 나타낸다.

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

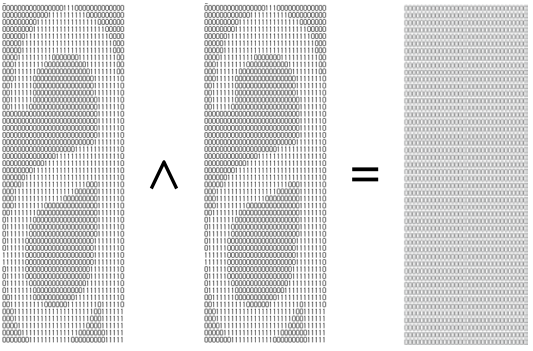
그렇다면 우리가 앞에서 연산할 수 있게 만든 데이터 타입과 어떻게 비교하면 될까? 기본 이미지 데이터Standard Image Data와 비교하기 위한 이미지 데이터Image Data를 위치가 같은 bit끼리 XOR 연산하면 된다. [그림 1-7]은 'a'와 'b' 두 개의 이미지 데이터를 비교한 것이다.

그림 1-7 a ^ b



서로 다른 두 개의 이미지를 비교하면 0과 1의 불특정 이미지 데이터로 나타난다. 하지만 두 개의 이미지가 동일하다면 XOR 연산자로 나타나는 이미지는 모두 0으로만 나타난다. [그림 1-8]은 'a'와 'a'를 XOR 연산자로 나타낸 것인데, 모두 0으로 나타나게 된다.

그림 1-8 a ^ a

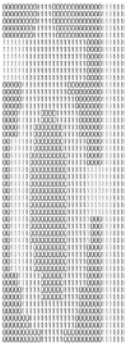


확률을 이용한 인식은 어떻게 이루어질까? 동일한 bit는 0으로 표현되기에 XOR 연산의 결과값을 가지고, 1,536개의 bit 중에 0의 숫자가 많은 것을 결과값으로 나타내면 된다(실제 이미지 인식에서는 모두 0으로 나오지 않는다). 하지만 연산을 통하여 0이 가장 많이 나온 기본 데이터를 결과로 출력하면 된다. 인식률이 높다는 것은 확률을 이용한 결과값이 올바른 결과값을 많이 도출한다는 것이다.

1.4 영문이 한글보다 인식률이 높은 이유

이 책에서 다루는 문자 인식은 영문과 숫자만을 대상으로 한다. 영문 인식은 62개의 기본 데이터를 만들면 되지만, 한글 인식은 만 개 이상의 기본 데이터를 만들어야 한다. 그런데 만 개 이상의 기본 데이터를 만들기도 어렵고 인식률도 떨어진다. 그렇다면 왜 한글 인식률이 높지 않은지 이미지 데이터를 가지고 알아보자.

그림 1-9 한글 이미지 데이터(허)



이 책에서 다루는 이미지 데이터는 가로 32개와 세로 48개 bit를 사용하는데, 이것으로 표현할 수 있는 총 가짓수는 1,536개다(가로 32개×세로 48개=1,536개). 이 이미지 데이터로 만 개가 넘는 한글을 표현하는 것은 불가능하다. 물론 우리는 더 큰 이미지 데이터를 만들어 사용할 수 있다. 하지만 만 개의 한글을 위한 기본 이미지 데이터를 만드는 것도 상당히 어려운 작업이며, 한 문자를 인식하기 위해서 만 개의 데이터와 항상 비교해야 한다면 연산 속도가 매우 느려질 것이다. 또한, 정확히 써진 한글만을 인식하는 환경이 아닌 곳에서는 이미지 변형을 수행하여 인식률을 높여야 하는데, 이미지 변형으로 인하여 오히려 다른 문자로 인식할 확률이 커질 수 있다.

chapter 2

기본 이미지 데이터 만들기

TV나 모니터를 사면 해상도를 확인하는데, HDTV의 해상도는 일반적으로 '1920×1080'과 같은 형식으로 표현한다. 이것은 화면의 가로에 1,920개, 세로에 1,080개의 픽셀(색을 나타내는 한 점)이 있는 것을 말한다. 즉, TV의 화면은 2백만 개가 넘는 점(1,920×1,080=2,073,600개의 픽셀)으로 이루어져 있고, 각 점이 하나의 색을 표현하여 화면을 구성한다.

그렇다면 한 점은 몇 가지의 색으로 이루어졌을까? 초등학교 미술 시간에 여러 가지 색의 물감을 계속 섞으면 점점 검은색(감산혼합)으로 변하는 것을 본 적이 있을 것이다. 그러나 TV와 같이 빛으로 표현되는 색은 색이 혼합될수록 점점 흰색(가산혼합)으로 변하게 된다. 물감의 3원색은 '빨/파/노'이며 빨간색, 파란색, 노란색을 섞으면 검은색이 되는 데 반하여, 빛의 색은 '빨/파/녹'이며 빨간색, 파란색, 녹색을 섞으면 흰색이 된다. 화면에서 하나의 점인 픽셀은 '빨강, 녹색, 파랑'이라는 세 개의 색을 가진 데이터로 색을 표현하는데 이것을 RGB[Red, Green, Blue]라고 한다.

RGB는 1-byte의 Red, 1-byte의 Green, 1-byte의 Blue를 가지고 있다. 1-byte는 8-bits로 이루어졌기 때문에 표현되는 RGB의 색은 각각 256(=2^8)개의 가짓수를 가진다. 즉, 하나의 픽셀은 256개의 Red, 256개의 Green, 256개의 Blue로 표현되기 때문에 16,777,216(=256×256×256)개의 색을 표현할 수 있으며, 프로그램에서는 이것을 Real Color라고 표현한다. 1-byte의 값은 0부터 시작하여 255까지의 값을 가진다. 예를 들어, 빨간색은 RGB(255,0,0)이라고 표현하며, Red는 255로 최고값이고 Green과 Blue는 0의 값을 가

진다. 즉, 빨간색은 Red에 다른 색을 섞지 않은 것이다. 마찬가지로 녹색은 RGB(0,255,0)이고, 파란색은 RGB(0,0,255)이다. 모든 색이 다 섞이는 RGB(255,255,255)는 흰색이 되고, 하나도 안 섞인 RGB(0,0,0)은 검은색이 된다. 회색은 RGB의 세 값이 같은 경우이며, RGB(50,50,50)은 검은색에 가까운 회색, RGB(200,200,200)은 흰색에 가까운 회색으로 나타난다. 즉, 회색의 값으로만 표현하게 되면 흑백 화면을 보는 것과 같고, 세 가지 색의 값이 모두 같기 때문에 한 값만을 가지고 표현할 수 있다. 즉, RGB(50,50,50)은 50이라는 값만을 가지고 표현할 수 있기 때문에 RGB의 3-byte 대신에 50이라는 값만을 갖는 1-byte 크기만 필요하며, 이렇게 보면 Real Color를 표현하기 위한 데이터를 흑백으로 표현하면 3분의 1로 데이터 크기를 줄일 수 있다.

RGB에 관련해서는 Google에서 'RGB 색상표'를 검색하면 쉽게 이해할 수 있다. [그림 2-1]은 'RGB 색상표'의 처음 일부분이다. [그림 2-1]에서 오른쪽 위의 흰색은 'FFFFFF'로 표현되며 이것은 흰색을 16진수로 표현한 것이다. 이처럼 프로그램을 만들면서 많이 다루는 것이 1-byte를 16진수로 표현하는 것이다. 예를 들어, C/C++에서는 '0x'를 숫자 앞에 쓰면 16진수로 표현한 것이다. 16진수는 16개의 값으로 표현하는데, 0부터 9까지의 10개 숫자와 A부터 F까지의 6개 숫자를 가지고 표현한다. 16개의 값은 다음과 같다.

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
```

이를 0과 1로 표현하는 2진수로 표현하면 다음과 같다. 괄호 안의 값이 2진수로 표현한 것이다.

```
0(0000), 1(0001), 2(0010), 3(0011), 4(0100), 5(0101), 6(0110), 7(0111), 8(1000),
9(1001), A(1010), B(1011), C(1100), D(1101), E(1110), F(1111)
```

하나의 색의 값을 갖는 1-Byte는 두 개의 16진수로 표현한다. 예를 들면 '00101110'의 값을 갖는 1-byte는 앞쪽의 '0010'와 뒤쪽의 '1110'으로 '2E'와 같이 표현한다. [그림 2-1]를 보면 노란색은 RGB(255,255,0)으로 표현한다. 이것을 16진수로 표현하면, Red의 'FF'와 Green의 'FF', 그리고 Blue의 '00'을 함께 써서 'FFFF00'이며, 보통 RGB를 나타내는 의미에서 '#' 기호를 앞에 써서 '#FFFF00'으로 표현한다.

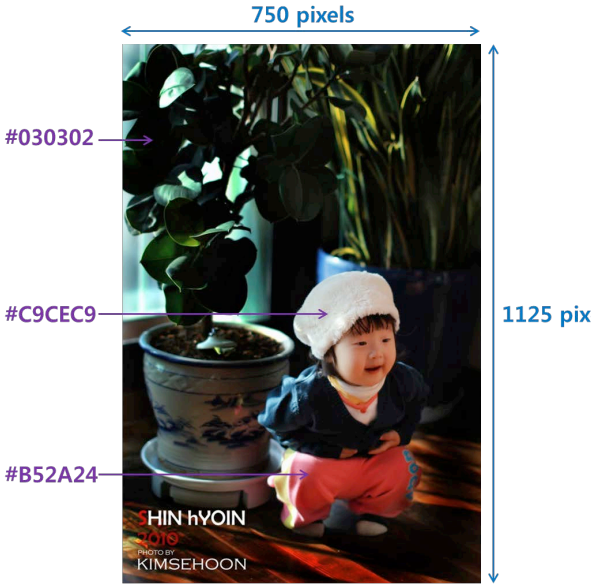
그림 2-1 RGB 색상표

000000 R - 000 G - 000 B - 000	333333 R - 051 G - 051 B - 051	666666 R - 102 G - 102 B - 102	999999 R - 153 G - 153 B - 153	CCCCCC R - 204 G - 204 B - 204	FFFFFF R - 255 G - 255 B - 255
000033 R - 000 G - 000 B - 051	333300 R - 051 G - 051 B - 000	666600 R - 102 G - 102 B - 000	999900 R - 153 G - 153 B - 000	CCCC00 R - 204 G - 204 B - 000	FFFF00 R - 255 G - 255 B - 000
000066 R - 000 G - 000 B - 102	333366 R - 051 G - 051 B - 102	666633 R - 102 G - 102 B - 051	999933 R - 153 G - 153 B - 051	CCCC33 R - 204 G - 204 B - 051	FFFF33 R - 255 G - 255 B - 051
000099 R - 000 G - 000 B - 153	333399 R - 051 G - 051 B - 153	666699 R - 102 G - 102 B - 153	999966 R - 153 G - 153 B - 102	CCCC66 R - 204 G - 204 B - 102	FFFF66 R - 255 G - 255 B - 102
0000CC R - 000 G - 000 B - 204	3333CC R - 051 G - 051 B - 204	6666CC R - 102 G - 102 B - 204	9999CC R - 153 G - 153 B - 204	CCCC99 R - 204 G - 204 B - 153	FFFF99 R - 255 G - 255 B - 153
0000FF R - 000 G - 000 B - 255	3333FF R - 051 G - 051 B - 255	6666FF R - 102 G - 102 B - 255	9999FF R - 153 G - 153 B - 255	CCCCFF R - 204 G - 204 B - 255	FFFFCC R - 255 G - 255 B - 204

디지털 카메라로 찍은 사진을 모니터로 볼 때 색상은 RGB로 표현하며 사진의 사이즈는 픽셀로 정해진다. [그림 2-2]는 사진의 크기와 색상을 보여 준다. 필자의 조카인 신효인 사진의 크기는 가로 750개, 세로 1,125개의 픽셀로 이루어져 있다. 또한, 사진 왼쪽에는 해당 부분의 RGB 값을 확인할 수 있다(화면의 RGB 값을 알 수 있는 프로그램은 인터넷에서 쉽게 얻을 수 있다). 어두운색은 RGB 값이 작고 밝은 색은 RGB가 크며, 분홍색은 Red 값이 크고 나머지 값이 작음을 알 수 있다. 모든 사진

은 이와 같이 가로와 세로 그리고 각 픽셀의 데이터만 알면 된다.

그림 2-2 사진의 크기와 색상

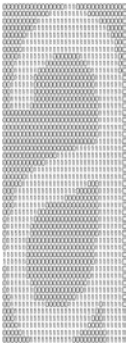


이 책에서 구현하는 OCR은 사진을 읽어서 RGB의 값으로만 되어 있는 문자를 얻는 것이다. 이는 사진의 크기와 각각의 픽셀 정보를 가지고 구현되며, 프로그램에서 인식을 위해 사용하는 새로운 데이터 타입도 크기와 데이터 값을 정해서 구현하면 된다. 사진을 편집하는 많은 프로그램에서는 사진의 크기, RGB 정보만을 이용한다. 예를 들어, 사진의 크기를 줄여 주는 프로그램은 가로와 세로의 값을 변경하는 것이고, 사진에 낙관을 찍는 것은 특정 위치의 RGB 데이터를 낙관의 값으로 바꾸어 주는 것이며, 흑백 사진을 만든다는 것은 RGB 세 값의 평균으로 Red, Green, Blue의 값을 바꾸어 주는 것이다. 이처럼 사진의 데이터에 대한 이해만 하고 있다면 사진 편집 프로그램에도 쉽게 접근할 수 있다.

2.1 새로운 데이터 구조체

사진은 픽셀의 개수로 크기가 정해지고 픽셀의 RGB 값으로 색이 정해진다. 1장에서 설명한 것과 같이 인식하려는 각각의 문자도 크기와 색을 정하면 된다. 인식하려는 문자는 [그림 2-3]과 같이 가로는 32개의 픽셀, 세로는 48개의 픽셀로 이루어져 있다. 색을 나타내는 점의 값은 0과 1만으로 표현하면 되는데, 이것은 프로그래머가 정하면 된다. 즉, 가로와 세로를 더 크게 만들 수도 있으며, 이 책에서는 문자는 1이고 바탕은 0으로 하였지만 반대의 값으로 정하여도 상관없다. 이 책에서 모든 문자 데이터 이미지는 '32×48'의 크기로 프로그램을 구현하고 하나의 픽셀은 하나의 bit로 표현할 것이다.

그림 2-3 알파벳 'a'의 데이터 이미지



필자의 첫 번째 책인 『소수와 RSA 알고리즘으로 배우는 Big Number 연산』[01] (한빛미디어, 2013)에서 구조체를 비중 있게 다루었는데, 이 책에서도 데이터를 문자의 값과 데이터 이미지 정보를 저장하는 데 구조체를 사용한다. [코드 2-1]은 이 책에서 사용하는 새로운 데이터 타입의 구조체를 정의한 것이며, 모든 기본 문자Standard Letter의 이미지는 `Letter` 구조체에 저장된다. `Letter` 구조체의 멤버 변수는 `value`와 `image`인데, 'a'라는 이미지의 데이터가 저장된다면 `value`에는 'a' 문자가 저장되고, `image`에는 [그림 2-3]의 데이터 이미지가 저장된다.

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

image 변수는 'unsigned int'로 정의하였다. int 데이터 타입은 4-bytes의 크기를 가지고 1-byte는 8-bits로 이루어졌기 때문에 32-bits의 크기임을 알 수 있다. 즉, 가로로 32개의 bit를 표현할 수 있으며, image[48]과 같이 48개 크기의 배열로 표현하여 세로의 크기를 48로 정하였다.[02]

[코드 2-1] Letter 구조체

```
struct Letter {
    char value;
    unsigned int image[48];
};
```

앞으로 사용하는 모든 데이터 이미지는 'unsigned int image[48]'과 같은 형태로 저장되어 연산이 이루어지게 된다. 'int' 형이 아닌 'unsigned int'로 데이터 타입을 정한 것은 bit를 이동시키는 Shift 연산자인 '〉〉'을 가지고 연산이 이루어졌을 때 왼쪽에 항상 '0'을 채우기 위해서다[03].

이미지 인식에서는 비교하기 위한 기본 데이터를 미리 만들어 놓아야 한다. 이 책에서는 알파벳 소문자 26개, 대문자 26개, 숫자 10개의 이미지만을 인식하려고 한다. 특수문자도 모두 만드는 것이 좋지만 쉽게 설명하기 위해 62개의 데이터만을 이용하겠다.

[코드 2-2]는 기본 데이터 구조체인 Standard 구조체를 정의한 것으로, 크기가 62인 배열 변수 letter만을 멤버 변수로 가진다.

[코드 2-2] Standard 구조체

```
struct Standard {
    Letter letter[62];
};
```

02 필자가 데이터 이미지를 '32×48'로 정한 것은 저장하기 위한 값을 간단히 하기 위해서이며, 대부분 문자 이미지는 가로보다 세로가 더 크기 때문이다.

03 Shift 연산자는 『소수와 RSA 알고리즘으로 배우는 Big Number 연산』 'APPENDIX A의 비트 연산자' 부분에 자세히 설명되어 있으므로 여기서는 생략한다.

프로그램을 만들다 보면 'Letter letter[62];'와 같이 구조체를 배열로 선언하는 경우가 많다. 구조체가 새로 추가된 데이터 타입이라는 것을 이해한다면 데이터 타입의 배열화는 당연한 것으로 받아들일 수 있다. 더 나아가 클래스도 구조체에서 발전된 것이기 때문에 클래스도 배열로 만드는 것이 가능하다.

2.2 기본 이미지 데이터

이 책에서 다루는 문자체Font는 Arial을 기준으로 하며, 모든 기본 데이터는 Arial 문자체로 만든다. 기본 데이터를 만들기 위해 [그림 2-4]와 같이 62개의 문자 이미지를 포토샵 프로그램으로 'standardimage.jpg' 파일을 만들어 프로그램에서 사용한다. 기본 데이터를 만드는 과정을 이해하면 OCR의 기본 개념은 대부분 알 수 있다. 그중 하나의 문자를 이미지에서 분리하고 Standard 구조체에 넣는 과정을 이해하는 것이 중요하다. 즉, 다음의 이미지에서 문자를 순서대로 분리해 내어서 순서대로 62개의 데이터 이미지에 넣어 주면 기본 데이터를 만들 수 있다.

그림 2-4 Arial 알파벳과 숫자 이미지(standardimage.jpg)

이 책의 프로그램은 윈도우 프로그램으로 만들며, MFC를 사용한다. 프로그램의 선택은 프로그래머가 편한 것을 선택하면 된다.[04] 프로그램을 만들 줄 아는 사람이라면 C#이나 Java로도 구현할 수 있다. 실제 프로그램은 사진 파일을 읽어 와서 사진의 픽셀 값을 읽을 수 있으면 구현할 수 있기 때문에 어떠한 언어로 개발하여도 상관없다.

[그림 2-5]는 standardimage.jpg 파일을 읽어서 화면에 보여 주고 사진의 크기 정보를 출력하는 프로그램 화면이다. 이미지를 클릭하면 마우스가 가리키는 점 Pixel의 사진상의 위치와 RGB 정보가 오른쪽에 나타난다.

그림 2-5 이미지를 출력하고 픽셀 정보를 얻는 프로그램

이 프로그램에서 중요한 것은 이미지 정보를 저장하는 CImage 클래스와 이미지 클래스에서 사진의 정보(위치와 RGB 값)를 얻는 방법을 이해하는 것이다. CImage

04 필자가 MFC를 선택한 이유는 두 번째 책인 『MFC 프로그래밍 : 주식 분석 프로그램 만들기』(한빛미디어, 2014)에서 MFC의 기본적인 부분을 설명해서 추가적인 설명 없이 프로그램만을 구현하기 위해서다. 또한, 평소 필자는 C++로 프로그래밍하는 것을 좋아하고 윈도우 프로그램이 사용자가 편하게 사용할 수 있다고 생각하기 때문이다.

클래스의 주요 사용법은 다음과 같다.

1. CImage 정의하기(예: CImage image;)
2. 사진의 정보를 CImage로 저장하기(예: image.Load(" 파일이름 ");)
3. 사진의 크기 정보 가져오기(예: image.GetWidth(); image.GetHeight();)
4. 특정 위치의 픽셀 색 정보 가져오기(예: COLORREF rgb = image.GetPixel(x, y);)
5. 특정 위치의 픽셀 색 변경하기(예: image.SetPixel(x, y, rgb);)
6. RGB의 특정 색 값 얻기(예: int red = GetRValue(rgb); int blue = GetBValue (rgb);)

이미지 인식은 이 6가지 방법만 알면 프로그램을 구현할 수 있다. 여기서 사용하는 정보를 얻기 위한 함수들은 MFC에서 규정한 것이므로 그대로 따라야 한다. 나머지 코드는 순수한 C++ 언어로 구현할 것이다. [코드 2-3]과 [코드 2-4]는 [그림 2-5] 프로그램 구현의 주요 내용이다.[05] 프로그램은 대화상자 기반으로 만들었으며, Visual Studio에서 프로젝트 생성 시 많은 파일이 생성되는데 프로그램의 구현은 'ImageRecognitionDlg.h'와 'ImageRecognitionDlg.cpp'에서만 이루어진다.

[코드 2-3] CImage 클래스 변수 정의(ImageRecognitionDlg.h)

```
class CImageRecognitionDlg : public CDialog
{
public:
    CImage image;                    //---①
    CString m_inputfile;             //---②
    CString m_size;
    CString m_point;
    CString m_rgb;
    // 생략
};
```

05 MFC에서 버튼 만드는 방법과 마우스 버튼을 눌렀을 때 동작하는 Window Message 구현 방법은 『MFC 프로그래밍 : 주식 분석 프로그램 만들기』(한빛미디어, 2014)에서 자세히 설명하였기에 여기서는 생략한다

① CImage 클래스 변수인 image 변수를 선언하였으며, 사진 파일의 모든 정보는 image 변수에 저장되어 사진 데이터를 가져오거나 사진의 정보 변경이 가능하다.

② 프로그램 화면의 정보를 가져오거나 계산된 값을 화면에 보여 주기 위해 사용하는 멤버 변수다. 멤버 변수를 만들려면 직접 코드를 입력해도 가능하지만, Visual Studio에서 자동으로 만드는 것도 가능하다.[06] 변수의 특징은 파일 이름에서 알 수 있는데, m_inputfile은 입력파일 이름을 가져오기 위한 변수이고, m_size는 파일의 크기, m_point는 마우스의 위치, m_rgb는 마우스가 가리키는 점의 RGB 값을 출력하기 위해 사용하는 변수다.

[코드 2-4]는 사용자가 [실행] 버튼을 눌렀을 때 실행되는 함수인 OnBnClicked BtnRun( ) 함수를 구현하였으며, 마우스 왼쪽 버튼이 눌렀을 때 실행되는 함수는 'OnLButtonDown(UINT nFlags, CPoint point)'다. 마우스를 눌렀을 때 중요한 것이 마우스 위치 정보인데, 매개변수로 받는 점 클래스인 'CPoint point'로 점의 위치를 알 수 있다. [실행] 버튼이 선택되면 프로그램은 화면의 정보를 가져와서 사진 파일을 CImage 클래스에 저장하고 화면에 이미지를 출력한다. 사진을 CImage 클래스의 변수에 담기 위해 CImage의 멤버 함수인 Load( )를 실행한다. Windows 프로그램에서 사용하는 이미지는 모두가 'Bitmap'이라는 압축되지 않은 파일 형식을 사용하는데, 사진은 압축율이 좋은 JPEG 파일 형식을 사용한다(확장자는 jpg). Load( ) 함수는 압축된 JPG 사진 파일을 압축되지 않은 Bitmap으로 변환하여 CImage 클래스에 담는 작업이다. MFC에서는 사진을 화면에 출력하는 데 DC[Device Context]를 사용하며 사진 이미지를 DC에 넣어서 그려주면 된다(화면에 그림을 그리기 위해서는 DC를 항상 사용해야 한다).

[코드 2-4] 실행 버튼과 마우스 왼쪽 버튼 클릭 시 동작 구현(ImageRecognitionDlg.cpp)

```
void CImageRecognitionDlg::OnBnClickedBtnRun()
{
    UpdateData(TRUE);                                    //--①

    if (image != NULL)                                   //--②
```

06 이는 『MFC 프로그래밍 : 주식 분석 프로그램 만들기』의 '6.1 MFC - Check Box와 Flag 사용'에서 자세히 설명하였다.

```
        image.Destroy();

    HRESULT hResult = image.Load(m_inputfile);                          //--③

    if (FAILED(hResult))                                                //--④
    {
        CString strError = TEXT("ERROR : ");
        strError += m_inputfile + TEXT(" 파일을열수가없습니다.");
        ::AfxMessageBox(strError);
        return;
    }

    m_size.Format(_T("Size: %d x %d"), image.GetWidth(), image.GetHeight()); //--⑤
    UpdateData(FALSE);

    CClientDC dc(this);                                                 //--⑥
    image.BitBlt(dc.m_hDC, 0, 0);                                       //--⑦
}

void CImageRecognitionDlg::OnLButtonDown(UINT nFlags, CPoint point)     //--⑧
{
    if (image == NULL)                                                  //--⑨
        return;

    if (point.x < image.GetWidth() && point.y < image.GetHeight()) {    //--⑩

        COLORREF rgb = image.GetPixel(point.x,point.y);                 //--⑪

        m_point.Format(_T("Point(%d,%d)"), point.x, point.y);           //--⑫
        m_rgb.Format(_T("RGB(%d,%d,%d)"), GetRValue(rgb), GetGValue(rgb),
GetBValue(rgb));                                                        //--⑬

        UpdateData(FALSE);
    }

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

① [실행] 버튼을 클릭하면 멤버 변수가 화면의 정보를 가져와야 한다. 'UpdateData(TRUE);'는 화면의 정보를 멤버 변수에 넣게 되고, 'UpdateData(FALSE);'는 멤버 변수의 값들을 화면에 출력하는 동작을 한다. 즉, m_inputfile 멤버 변수는 화면의 사진 파일 이름을 업데이트한다.

② image 변수에 사진의 정보를 넣을 때 image 변수는 아무런 정보를 가지고 있지 않아야 한다. 그래서 image 변수가 NULL이 아니면 이전 image는 정보를 지워 주어야 한다.

이 부분이 코드에서 구현되지 않으면 [실행] 버튼을 두 번 이상 눌렀을 때 프로그램에서 Error가 발생한다.

③ CImage 클래스의 Load("파일이름") 함수를 사용하여 사진 파일의 정보를 Bitmap으로 변환한 후 image 변수에 넣는다.

④ 사진 파일 이름이 잘못 입력되거나 해당 사진 파일이 없을 때 Load() 함수는 Error 값을 반환한다. hResult 변수가 Error 값을 갖게 되면 Error 메시지를 출력한다. Error가 발생했을 때는 마지막에 'return;'을 실행하여 이후의 코드를 수행하지 않아야 한다. 'OnBnClickedBtnRun()' 함수는 void로 선언되어서 반환값이 없지만, 이런 void 함수의 중간 종료는 값을 반환하지 않고 'return;'으로 행위를 중지할 수 있음을 알아둘 필요가 있다. 생각보다 많이 사용하는 방법이다.

⑤ image.GetWidth()는 이미지의 가로 픽셀 개수를 반환하며, image.GetHeight()는 세로 픽셀 개수를 반환한다. m_size에 사진의 크기를 출력하기 위해 m_size 내용을 변경하며, 'UpdateData(FALSE);' 함수를 사용하여 화면에 사진 크기를 출력한다.

⑥ 사진의 출력을 위해 DC를 선언한다. 'CClientDC dc(this);'는 현재 윈도우의 DC를 선언하는 것이다. 여기서 this는 현재 프로그램 화면Window의 객체다.

⑦ CImage 클래스의 BitBlt() 함수를 사용하여 image 변수의 이미지를 현재 화면에 그려 준다.

⑧ Visual Studio에서 Window Message 함수를 선언하면 마우스 왼쪽 버튼을 눌렀을 때 실행되는 함수를 자동으로 만들어 준다. 이 함수에서 중요한 것은 매개변수인 point 변수인데, point.x는 현재 화면에서 마우스의 x축의 위치값을 가지고 있고, point.y는 y축의 값을 가지고 있다.

⑨ 프로그램을 실행하고 나서 [실행] 버튼을 선택하지 않았다면 image 변수는 어떠한 정보도 가지고 있지 않기 때문에 함수를 종료한다. 이 부분이 구현되지 않으면 프로그램을 실행하고 나서 화면에서 마우스 왼쪽 버튼을 눌렀을 때 프로그램 Error가 발생한다.

⑩ 마우스의 위치가 사진 안에 있을 때만 점Point의 위치와 RGB 값을 출력한다.

⑪ CImage 클래스의 GetPixel() 함수는 마우스가 가리키는 점Point의 RGB 값을 반환한다. 'COLORREF rgb'는 RGB의 값을 저장한다. COLORREF 데이터 타입은 실제 4-byte의 크기를 갖는데 오른쪽부터 Red, Green, Blue의 값을 저장하며 가장 왼쪽에는 0x00으로 채워진다. 예를 들어, 빨간색이라면 rgb 변수는 '000000FF'의 값을 가지게 되고, 녹색이라면 '0000FF00'이고, 마지막으로 파란색이라면 '00FF0000'의 값을 가지고 있다. COLORREF의 저장 형태는 Microsoft에서 정한 것이기 때문에 그대로 따르면 되고,

저장값까지 기억할 정도로 중요하지는 않다. 단지 COLORREF에서 각각의 색은 1-byte에 저장된다는 것만 알면 된다.

⑫ 점의 위치를 화면에 출력하기 위해 m_point 멤버 변수의 내용을 변경한다.

⑬ 점의 RGB 값을 화면에 출력하기 위해 m_rgb 멤버 변수의 내용을 변경한다. GetRValue( )는 rgb의 Red 값을 정수로 반환하며, GetGValue( )는 Green 값을 반환하고, GetBValue( )는 Blue 값을 정수로 반환한다. 물론, 정수로 변경된 각각의 색 값은 0~255 범위의 값이다.

사진 파일을 CImage 클래스에 넣은 이후, 사진의 데이터를 가져오는 것을 설명하였다. OCR에서 필요한 정보는 사진의 크기와 모든 픽셀의 정보이므로 그것만 가져오면 된다. 여기에서 조금 더 생각할 것은 픽셀의 정보에 투명도가 있다는 것이다. PNG 형태의 사진 파일은 RGB 정보와 함께 투명도 값을 저장한다. 투명도를 이용하는 대표적인 프로그램이 포토샵인데, 다수의 사진을 레이어로 설정하여 각각의 레이어는 크기, 모양, RGB, 투명도 등의 정보를 가지고 편집할 수 있게 만들었다. 투명도에 대한 개념을 잘 알고 활용할 수 있다면 포토샵 같은 프로그램을 우리나라 프로그래머들도 만들 수 있다. 물론, 오랜 시간을 통하여 수정 보완된 프로그램인 만큼 그 완성도를 따라가기 힘들 수 있지만 불가능하지는 않다.

2.3 문자 이미지 추출하기

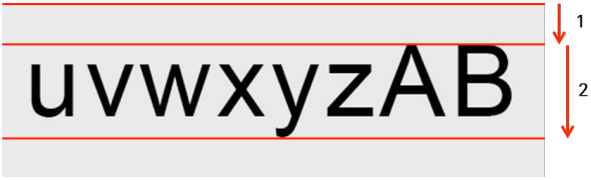
기본 이미지에서 문자를 추출할 때 가장 먼저 해야 할 작업은 이미지에서 각 문자를 하나씩 개별 이미지로 나누는 작업이다. 앞에서 언급한 바와 같이, 필자는 이 작업을 '이미지 파싱Image Parsing'이라고 부르려고 한다. 이미지 파싱은 이미지에 따라서 다양한 방법이 사용될 수 있다. 이번 장에서는 문서처럼 저장된 이미지를 다루기 때문에 조금 더 수월하게 이미지 파싱 작업을 할 수 있다. 스캔한 문서가 있다면 이번 장에서 다루는 방법을 이용하면 된다.

문서의 이미지 파싱은 먼저 라인Line별로 파싱Parsing을 진행하고, 이후에 각각의 라

인에서 하나의 문자Character 이미지를 차례대로 얻으면 된다. 기본 데이터를 얻기 위해 만든 기본 이미지에서 문자는 검은색이고 바탕은 흰색에 가까운 회색인데, 검은색을 기준으로 이미지 픽셀의 위치를 정하면 된다. 즉, 이미지의 위에서부터 가로 한 줄의 픽셀의 색을 검사하고 모든 픽셀에 검은색이 없다면 문자의 라인이 시작된 것이 아니므로 해당 픽셀의 가로줄은 바탕이 된다. 이후에 바로 아래 픽셀의 가로줄을 차례대로 검사하며 픽셀에 검은색이 포함되었다면 라인이 시작된 것으로 인식하면 된다.

[그림 2-6]은 라인의 세로축에 대한 시작과 끝을 알아내는 방법이다. 위에서부터 순차적으로 가로줄에 있는 모든 픽셀들의 색을 검사하여 검은색이 나오면 라인의 시작임을 알게 되고, 라인이 시작되는 위치를 알게 된 이후에는 픽셀의 가로줄에 검은색이 없을 때까지 진행하여 라인의 세로축 끝을 알 수 있다. 즉, 픽셀의 가로줄이나 세로줄을 검사할 때, 문자의 색인 검은색 픽셀이 나타나면 문자가 포함된 영역으로 인식하고, 검은색 픽셀이 하나도 없다면 바탕의 픽셀 줄이라고 판단한다. 무수히 많은 픽셀의 색을 검사하여 비교하는 과정은 컴퓨터이기에 가능한 작업이며 프로그램을 만들어서 구현하면 생각보다 아주 빠르게 처리된다.

그림 2-6 문서에서 라인의 세로축을 알아내는 방법



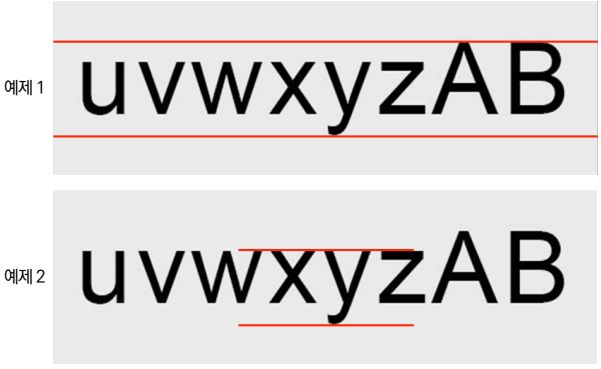
1. 픽셀 가로줄에서 검은 픽셀이 나올 때까지 진행하며 라인의 세로축 시작이 된다.
2. 픽셀 가로줄에서 검은 픽셀이 없을 때까지 진행하며 라인의 세로축 끝이 된다.

파싱을 필자는 3단계로 구분하였다. 1단계는 라인을 구분하는 것이며, 2단계는 개별 문자로 나누는 단계이며, 마지막 3단계는 하나의 문자에 대한 정확한 영역을 계산하는 단계다. 즉, [그림 2-7]과 [그림 2-8]은 1단계를 이미지로 설명하고 있으며, [그림 2-9]는 2단계인 개별 문자로 나누는 이미지를 보여 주고, [그림

2-10]은 1단계와 2단계를 거쳤지만 아직은 완전하지 않은 문자의 이미지 영역에 대한 정확한 이미지 영역의 계산을 보여 준다. 이번 장에서는 기본 데이터를 얻는 방법을 보여 주며, 이후의 모든 이미지 파싱 작업도 지금 설명하는 방법을 따른다고 이해하면 된다.

[그림 2-6]에서는 문서 이미지에서 픽셀의 가로줄 전체를 가지고 라인의 세로축을 얻는 방법을 설명했다면, [그림 2-7]에서는 픽셀의 전체 가로줄이 아닌 중간의 일부분에 대한 픽셀 검사로도 라인을 알 수 있음을 보여 준다. [그림 2-7]에서 '예제 1'은 전체 가로줄을 비교하는 것이고, '예제 2'는 중간의 일부 가로줄을 비교하는 것이다. 라인에 대한 세로축(Y축)의 위치에 대한 차이가 있지만, 두 가지 모두 라인을 알아내는 방법으로 사용할 수 있다. 필자는 '예제 2'의 방법을 사용하는데, 픽셀의 모든 가로줄을 비교하는 것이 훨씬 더 많은 연산을 수행할 수밖에 없으므로 속도 측면에서 '예제 2'의 방법이 유리하기 때문이다. 또한, 이후에 파싱의 마지막 작업(3단계)에서 개별 문자에 대한 정확한 범위를 얻기 때문에 1단계에서 가로줄의 일부분만을 가지고 문서의 라인을 계산하는 것도 문제가 되지 않는다.

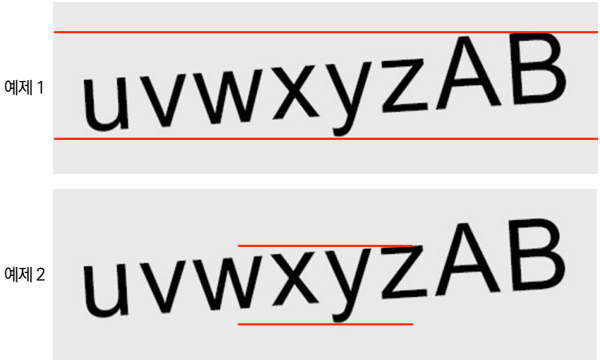
그림 2-7 1단계 - 라인 파싱



[그림 2-8]은 픽셀의 가로줄 중간 일부분만을 가지고 문서 라인을 확인해야 하는 다른 이유를 보여 준다. 스캐너로 얻은 문서 이미지는 거의 다 조금씩 기울어져 있다. 그런데 [그림 2-8]의 '예제 1'과 같이 전체 가로줄을 기준으로 비교하면 라인

구분이 불가능할 수도 있다. '예제 1'과 같이 라인의 범위가 넓어질 수 있는데, 기울기가 너무 심하게 기울어져 있다면, 위나 아래에 존재하는 다른 라인의 범위를 포함할 수 있기 때문이다. 물론, 문서가 아주 많이 기울어져 있다면 라인의 구분도 힘들 뿐만 아니라 인식하는 문자도 많이 변형되어 인식이 불가능할 수 있다. 텍스트 문서를 스캔할 일이 있으면 가능한 한 기울어짐을 최소화하는 것이 좋다.

그림 2-8 1단계 - 기울어진 문서의 라인 파싱

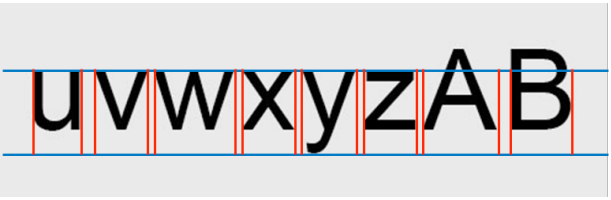


문서 이미지에서 픽셀의 가로줄을 따라 라인의 세로축(Y축)을 구분하였다면 마찬가지 방법으로 픽셀의 세로줄을 기준으로 개별 문자의 가로축(X축)을 알 수 있다. [그림 2-9]는 파싱의 2단계인 개별 문자의 구분을 그림으로 나타낸 것이다. 비교하는 픽셀의 세로줄 범위는 1단계에서 알게 된 라인의 시작과 끝의 Y축 값이다. 문자가 서로 붙어 있다면 개별 문자로 구분하는 것이 어렵다. 하지만 다행히도 대부분의 문서에는 문자의 간격이 존재하기에 2단계에서 픽셀의 세로줄에 검은색 픽셀이 있다면 문자 영역이고, 검은색 픽셀이 없다면 바탕 영역으로 인식하여 구분하면 된다. 1단계에서 픽셀 줄의 비교가 위에서 아래로 진행되었다면, 2단계에서는 픽셀의 비교는 왼쪽에서 오른쪽으로 진행된다.

이미지 파싱의 1단계와 2단계를 수행하면 하나의 문자에 대한 사각형 영역이 만들어진다. 이 사각형 영역의 X축 값은 그대로 사용해도 괜찮지만, Y축 값은 정확히 문자의 영역을 나타내지 않는다. [그림 2-9]의 이미지를 보면 'u', 'v', 'w', 'x'

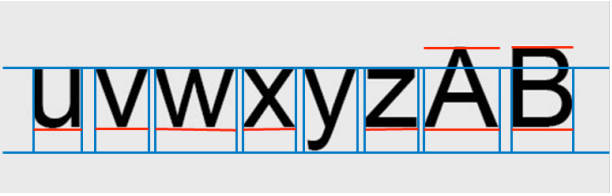
영역은 아랫부분에 바탕의 이미지를 포함하고 있으며, 'A'와 'B' 영역은 아랫부분에 바탕의 이미지를 포함하고 있을 뿐만 아니라 윗부분에도 문자를 정상적으로 포함하지 않고 있다. 문자 인식은 하나의 문자에 대한 정확한 사각형을 아는 것이 중요하기 때문에 사각형 영역을 더 정확히 수정해 줄 필요가 있다.

그림 2-9 2단계 - Letter 파싱



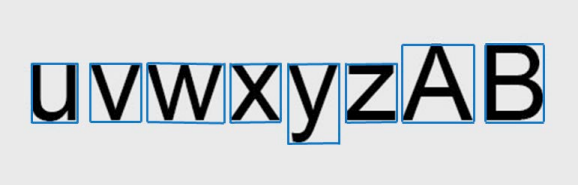
마지막 단계인 3단계에서 사각형 영역이 정확히 각각의 문자만을 포함하도록 조정해 준다. [그림 2-10]은 사각형의 Y축을 위아래로 조정하여 정확히 문자의 사각형 영역을 만들어 주는 방법을 보여 준다. 즉, X축의 간격이 정해졌기 때문에 X축 간격의 범위 안에서 픽셀의 가로줄을 위아래로 검사하며, 바탕과 문자의 경계선을 다시 검사하게 된다. [그림 2-10]에서 'u' 문자의 윗부분은 가로줄을 위아래로 움직여도 수정이 없어 경계선이 확정된 것이며, 아랫부분은 가로줄을 위로 움직여서 새로운 문자 영역의 Y축 값을 얻게 된 것이다. 마찬가지로, 'A' 영역은 윗부분의 가로줄을 위로 올려서 새로운 픽셀의 가로줄 영역을 찾았다. 이와 같은 작업은 모든 문자의 위와 아래의 영역에서 진행되었다.

그림 2-10 3단계 - Letter 세로 영역 조정



[그림 2-11]은 이미지 파싱의 모든 단계를 거쳐 최종적으로 만들어지는 영역이다. 이제 각각의 문자에 대한 모든 영역을 알게 되었다.

그림 2-11 이미지 파싱의 1, 2, 3단계 작업 후 이미지 영역



이미지에 대한 사각형의 영역을 데이터로 갖기 위해서는 두 가지 방법이 있다. 첫 번째 방법은 왼쪽 위의 시작점 (x, y) 값을 알고 넓이Width와 높이Height를 가지고 사각형 데이터를 저장하는 방법이고, 두 번째 방법은 왼쪽 위의 시작점 (x1, y1)의 값과 오른쪽 아래의 끝점 (x2, y2)의 값을 데이터로 갖는 방법이다. 두 가지 방법 모두 사각형의 위치와 크기를 정확히 알 수 있다.

그러나 이 프로그램에서는 두 번째 방법인 시작점과 끝점의 위치값으로 프로그램을 구현하는 것이 좋다. 원본 이미지의 점의 위치만을 가지고 프로그래밍해야 조금 더 간결한 코드가 나올 수 있기 때문이다. 첫 번째 방법으로 넓이와 높이 데이터를 가지고 픽셀의 가로줄과 세로줄을 검사하게 되면 가로줄이나 세로줄의 시작점은 알 수 있지만, 끝점은 다시 계산하여 정확한 위치를 알아내야 하는 불편함이 있다.

필자는 구조체를 좋아하여 이번 프로그램에서도 구조체를 사용하는데, [코드 2-5]는 사각형의 구조체를 정의한 것이다.[07]

[코드 2-5] 사각형 구조체(OCR.h)

```
struct Point {              //—①
    int x;
    int y;
};

struct Rect {               //—②
```

07 앞으로도 필자가 C/C++로 프로그램을 만든다면 구조체를 사용하지 않는 프로그램은 없을 것이다. 필자가 만들어 나가는 'How-to Series'의 첫 주제로 'Big Number 연산'을 선택한 것은 구조체를 확실히 이해하려는 목적도 있었다.

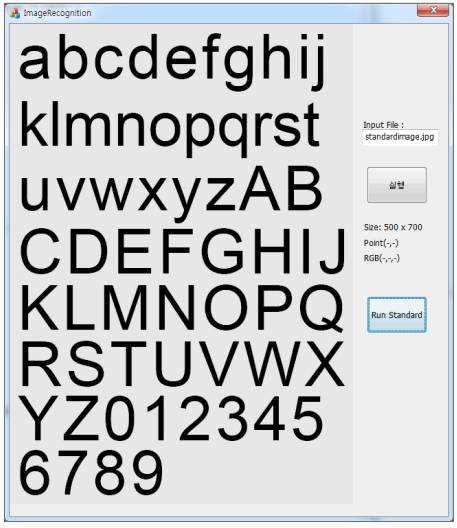
```
    Point start;
    Point end;
};
```

① 평면에서 점Point의 위치는 X축과 Y축의 값으로 알 수 있으며, Point 구조체는 (x, y) 데이터를 가진다.

② 사각형은 시작점과 끝점의 데이터로 표현할 수 있어 두 개의 점 데이터를 가진다.

이번 장에서 구현하는 프로그램은 이전 장에서 구현한 프로그램에 [Run Standard] 버튼을 추가하고, 버튼을 선택했을 때 기본 이미지를 사진 파일로 출력하는 프로그램이다. 이미지 파싱의 결과를 사진 파일로 만들어서 파싱이 정확히 되었는지를 확인하는 것이 중요하다. 물론 프로그램이 완성되었을 때는 이번 장에서 구현하는 함수를 사용하지는 않지만, 실제 프로그램 개발에서는 많은 단위Unit 테스트가 필요하다. 여기서의 이미지 파싱은 이후의 모든 이미지 파싱의 기본 개념이다. [그림 2-12]는 구현할 프로그램의 실행화면이며 [Run Standard] 버튼이 만들어진 것을 확인할 수 있다.

그림 2-12 프로그램 실행 화면(Run Standard 버튼)



[코드 2-6]은 [Run Standard] 버튼을 선택했을 때 실행되는 함수로, 이전 장에서 [실행] 버튼을 선택했을 때 구현되는 코드와 동일하다. 마지막에 OCR을 위해 추가된 COCR 클래스의 함수를 실행하여 입력된 사진의 이미지 파싱을 진행한다. 앞으로 COCR 클래스에 프로그램의 이미지 인식에 관련된 거의 모든 기능을 구현한다.[08]

[코드 2-6] Run Standard 버튼 선택 시 호출하는 함수(ImageRecognition.cpp)

```
void CImageRecognitionDlg::OnBnClickedBtnRunstandard()        //---①
{
    UpdateData(TRUE);

    if (image != NULL)
        image.Destroy();

    HRESULT hResult = image.Load(m_inputfile);

    if (FAILED(hResult))
    {
        CString strError = TEXT("ERROR : ");
        strError += m_inputfile + TEXT(" 파일을열수가없습니다.");
        ::AfxMessageBox(strError);
        return;
    }

    m_size.Format(_T("Size: %d x %d"), image.GetWidth(), image.GetHeight());
    UpdateData(FALSE);

    CClientDC dc(this);
    image.BitBlt(dc.m_hDC, 0, 0);

    //------------------------- OCR -------------------------
    ocr = new COCR();                                  //---②
    ocr->CreateStandard(&image);                       //---③
}
```

① [Run Standard] 버튼을 선택하면 실행되는 함수다. 프로그램 실행 후 [Run Standard] 버튼만을 누를 수 있기 때문에 이전 장에서 구현한 [실행] 버튼의 기능을 다시 사용하였으며, 마지막에 2와 3의 두 줄이 추가되었다.

08 MFC에서 클래스 추가 등을 포함한 사용법은 『MFC 프로그래밍 : 주식 분석 프로그램 만들기』(한빛미디어, 2014)에서 설명하였으므로 여기서는 생략한다.

② 이미지 인식을 위해 추가된 COCR 클래스를 생성한다.

③ COCR 클래스에서 구현된 CreateStandard( ) 함수를 실행하여 기본 이미지 데이터를 만들게 된다. 이전 장에서는 이미지 파싱만을 구현하게 되며, 이후에 나머지 추가적인 부분이 구현된다. 매개변수로 &를 사용하여 image의 주소값을 전달한다. 이처럼 'Call By Reference'를 사용하는 이유는 크기가 큰 이미지를 복사하여 여러 개를 만들 필요 없이 처음 생성된 하나의 이미지만을 사용하는 것이 속도와 메모리 사용에 있어서 더 효과적이기 때문이다.

Visual Studio에서 클래스를 추가하면 클래스를 의미하는 C가 클래스 이름 앞에 써지므로 OCR 클래스는 COCR로 만들어진다. 앞으로 구현하는 거의 모든 기능은 COCR 클래스에서 함수Method로 만들어지며, [코드 2-7]에서 정의된 구조체를 이해하는 것이 가장 중요하다.

[코드 2-7]에서 가장 중요한 구조체는 Letter 구조체로, 이미지 데이터를 포함하고 있다. 이번 장에서 설명하는 이미지 파싱에서는 사용하지 않지만, 이미지 파싱 이후에 각각의 문자 이미지를 연산이 가능한 데이터로 만들어 저장하는 것이 Letter 구조체다. 이후의 구조체는 Letter 구조체를 사용하여 데이터를 담기 위해 사용하는 구조체다. Standard는 기본 데이터를 담기 위한 구조체, Data는 Letter 구조체와 이미지에서의 사각형의 위치값을 저장하기 위한 구조체, AllData는 이미지 인식을 위한 모든 데이터를 저장하기 위한 구조체다.

COCR 클래스에서는 멤버 변수를 확인할 필요가 있다. image 멤버 변수는 인식하려는 사진 파일의 이미지 클래스를 가리키는 포인터, standard 멤버 변수는 이미지 비교를 위한 기본 이미지를 저장한 변수, allData 멤버 변수는 이미지 파싱을 진행한 후 모든 이미지 데이터를 저장하기 위한 변수다. data 멤버 변수는 이미지 인식을 진행할 때 하나의 이미지 데이터를 가리키는 포인터 변수로, 모든 이미지 데이터를 하나씩 차례대로 가리키면서 이미지 인식을 진행하기 위해 사용한다.

프로그램을 구현할 때 어떠한 구조체와 클래스를 정의하는 것이 좋은지 프로그램의 시작 단계에서 충분한 시간을 가지고 생각하는 것이 좋다. 처음에 정의한 구조

체와 클래스가 올바르다면 이후 프로그램은 쉽게 구현할 수 있으나 프로그램이 구현된 후에 구조체나 클래스를 변경하려면 많은 수정 작업을 해야 하기 때문이다.

[코드 2-7] 데이터 저장을 위한 구조체와 OCR 클래스 정의(OCR.h)

```
#pragma once
#define MAX_COUNT_STANDARD       80                //---①

#define MAX_COUNT_DATA           2000
#define RANGE_OF_COLOR_TO_CHECK      100           //---②

struct Point {                                     //---③
    int x;
    int y;
};

struct Rect {                                      //---④
    Point start;
    Point end;
};

struct Letter {                                    //---⑤
    char value;
    unsigned int image[48];
};

struct Standard {                                  //---⑥
    int count;
    Letter letter[62];
};

struct Data {                                      //---⑦
    Letter letter;
    Rect rect;
};

struct AllData {                                   //---⑧
    int count;
    Data data[MAX_COUNT_DATA];
};

class COCR                                         //---⑨
{
private:
    CImage *image;                                 //---⑩
```

```
    Standard standard;                              //---⑪
    AllData allData;                                //---⑫
    Data *data;                                     //---⑬

    int colorToCheck;                               //---⑭

public:
    COCR(void);
    ~COCR(void);

    void CreateStandard(CImage *image);             //---⑮
    void ParsingStepFirst();                        //---⑯
    void ParsingStepSecond(int yTop, int yBottom); //---⑰
    void ParsingStepThird(Rect *rect);              //---⑱
    void PrintImageToFile(int fileNo, Rect *rect); //---⑲
};
```

① 상수 변수 MAX_COUNT_STANDARD를 80으로 정의하였으며, 기본 이미지 데이터의 최대 개수는 80개를 넘지 않는다. 여기서 구현하는 프로그램은 62개의 기본 데이터를 만들게 되는데, 이후에 기본 이미지의 개수를 증가시킬 수 있기 때문에 이와 같이 상수 변수를 사용하여 최대 개수를 정의한다. 또한, 상수 변수 MAX_COUNT_DATA를 2000으로 정의하였다. 이는 인식을 위한 사진 이미지에서 최대 문자 수는 2000개를 넘지 않는다는 가정하에 최대값 2000으로 정의한 것이다. 예를 들어, A4 용지에 25개의 라인이 있고, 한 라인에는 80개의 문자가 있다고 가정하면 2,000개의 문자가 된다.

② 상수 변수 RANGE_OF_COLOR_TO_CHECK는 문자 색을 인식하기 위한 값을 나타낸다. 색을 나타내는 RGB에서 검은색은 RGB(0,0,0)이고, 흰색은 RGB(255,255,255)이므로 문자 색의 인식을 위해 기준이 되는 값을 중심으로 충분한 범위를 정하여 이후에 색을 비교하게 된다. 실제 이미지에서는 검은색이라고 정의하였어도 경계선에 있는 값은 정확히 검은색이 아니라 검은색에 가까운 회색인 경우가 많다. 따라서 이미지에서 문자색이어도 색의 비교는 유격을 주어야(색 값의 범위를 조금 더 넓혀야) 한다. RANGE_OF_COLOR_TO_CHECK는 문자 색으로 지정된 특정 색의 값에 더하거나 빼주어 문자 색의 범위를 넓게 만들어 주기 위한 값이다.

③ Point 구조체는 점의 값을 저장하는 구조체로, 평면에서 점의 위치는 x와 y 값으로 표현된다.

④ 이미지 파싱에서는 사각형으로 문자 이미지를 구분해야 하는데, Rect 구조체는 사각형을 표현할 수 있는 왼쪽 위의 시작점과 오른쪽 아래의 끝점 값을 가지고 정의한다.

⑤ Letter는 이미지 인식에서 가장 중요한 구조체로, 문자 이미지를 이미지 데이터로 변환

하여 저장한다. Letter의 value 변수는 문자의 값을 저장하기 위한 변수고, image 배열에 32×48 크기의 이미지 데이터를 저장한다.

⑥ Standard 구조체는 표준 데이터를 저장한다. 이번 장에서는 표준 데이터를 만드는 것을 구현하지만, 실제 OCR 프로그램은 프로그램이 실행되면 이미 만들어진 표준 데이터를 Standard 구조체에 넣는 작업을 가장 먼저 한다. 멤버 변수 count는 Standard 구조체에 저장된 기본 이미지 데이터의 총 개수를 저장한다.

⑦ Data 구조체는 Letter 구조체 변수와 Rect 구조체 변수를 가진다. Letter 구조체 변수는 인식하기 위한 이미지 데이터를 저장하고, Rect 구조체 변수는 해당 문자의 사각형 정보를 저장한다. 이미지 인식에서 사각형의 위치는 중요하다. 예를 들어, 영문 'L'의 소문자인 'l'과 마침표 '.'은 이미지 데이터에서 사각형 안이 문자 색으로 꽉 차게 된다. 이런 경우에는 사각형의 위치 정보로 문자를 구분할 수 있으며, 사각형의 가로와 세로의 비율을 사용하면 된다. 이번 장에서는 이 부분에 대하여 자세히 설명하지는 않지만 나중에 문서 사진에서 문자를 판독할 때 구현하겠다.

⑧ 인식하려는 모든 문자의 데이터는 AllData 구조체에 저장된다. AllData 구조체의 count 변수는 모든 문자의 개수 값을 가지며, data 변수는 모든 문자의 데이터를 저장한다.

⑨ COCR 클래스는 이미지 인식의 모든 연산을 함수로 구현한다. 프로그램의 [실행] 버튼이 선택되면 COCR 클래스의 함수를 실행하여 결과를 얻는다.

⑩ 인식하려는 사진의 이미지는 CImage 클래스에 담기며, COCR은 이 클래스를 가리키는 image 포인터Pointer를 사용하여 해당 이미지로의 접근이 가능해진다. 이처럼 포인터를 사용하여 이미지로 접근하는 이유는 같은 이미지를 여러 번 복사하여 메모리의 사용을 늘리는 일이 없게 하고 복사에 따른 실행 시간의 낭비도 막을 수 있기 때문이다.

⑪ standard 변수는 기본 이미지 데이터를 저장하는 변수로, 이번 장에서는 기본 이미지를 만든 후 standard 변수에 저장한다.

⑫ allData 변수에는 COCR 클래스에서 인식하려는 모든 데이터를 저장하며, 이미지 인식은 allData 변수의 모든 데이터를 차례대로 인식하면 된다.

⑬ allData에 있는 모든 이미지 중에서 현재 인식하려는 이미지로 접근이 가능하게 포인터 변수인 data를 사용하였다. 인식하려는 이미지 데이터를 하나씩 변경해 주고 이후에 실행되는 함수에서는 data 변수가 가리키는 이미지만을 인식하면 되기 때문에 구현이 조금 더 간결해질 수 있다.

⑭ 인식하려는 문자의 색 값을 저장한다. 프로그램에 따라 문자 색이 바뀌어도 이 값만 변경하면 되며 다른 부분에서의 코드 수정은 필요하지 않을 수 있다.

⑮ 기본 이미지 데이터를 만들기 위해 실행되는 함수로, [Run Standard] 버튼이 선택되었을 때 COCR의 CreateStandard( ) 함수만 실행하면 된다.

⑯ 이미지 파싱의 1단계를 수행하는 함수로, 전체 이미지에 대하여 라인을 구분한 후 다음 단계인 2단계의 함수를 순차적으로 실행한다.

⑰ 이미지 파싱의 2단계를 수행하는 함수로, 매개변수 yTop과 yBottom의 값을 받는다. 2단계는 하나의 라인에 대해 개별 문자를 나누는 작업이기 때문에 라인의 Y축에 대한 범위를 매개변수로 받는다. 2단계에서는 개별 문자에 대한 사각형을 알게 되는데, 사각형의 위치가 정확하지 않으므로 3단계를 통해 정확한 위치를 계산한다.

⑱ 이미지 파싱의 3단계를 수행하는 함수다. 매개변수로 한 문자의 사각형 값을 받으며 입력받은 사각형을 기준으로 세로축에 대한 정확한 값을 계산한다. 이때 만들어지는 사각형은 실제 완벽한 문자 이미지를 위한 사각형은 아니다. 이러한 이유로 이미지 파싱이 이미지 인식에서 가장 어려운 부분이라고 생각한다.

⑲ 이미지 파싱이 끝난 후에 모든 이미지를 사진 파일로 저장하기 위한 함수다. 매개변수의 fileNo는 파일 이름의 구분을 위한 것이며, rect는 전체 이미지에서 하나의 문자 이미지의 사각형 범위를 나타낸다.

이미지 인식을 위한 구조체가 정의되었으면 COCR 클래스의 함수를 만들어 필요한 기능을 구현하면 된다. [코드 2-8]은 프로그램 화면에서 [Run Standard] 버튼이 선택되었을 때 호출되는 COCR의 CreateStandard( ) 함수고, [코드 2-6]에서 기본 이미지 데이터를 만들기 위해 마지막에 이 함수 하나만 호출한다. 이미지 파싱은 여러 단계로 이루어져 있는데, 각 단계는 모두 순차적으로 진행되어 CreateStandard( ) 함수에서는 '1단계'를 수행하는 ParsingStepFirst( ) 함수를 실행하고, ParsingStepFirst( ) 함수에서 '2단계'를 수행하는 ParsingStepSecond( ) 함수를 실행한다. 마찬가지로 ParsingStepSecond( ) 함수에서는 '3단계'인 ParsingStepThird( ) 함수를 실행한다.

[코드 2-8] 기본 이미지 데이터를 위해 호출되는 함수 - COCR::CreateStandard 함수(OCR.cpp)

```
void COCR::CreateStandard(CImage *newImage)          //---①
{
    image = newImage;                                //---②
```

```
    colorToCheck = 50;                              //---③
    ParsingStepFirst();                             //---④
    for (int i=0; i<allData.count; i++)             //---⑤
        PrintImageToFile(i, &allData.data[i].rect);
}
```

① 기본 이미지 데이터를 만들기 위해 실행되는 함수다. 인식하려는 사진 파일을 저장하고 있는 CImage 클래스의 주소를 매개변수로 받는다.

② 입력받은 CImage 클래스의 주소를 COCR 클래스의 변수인 image에 복사한다. CImage 클래스를 가리키는 포인터 변수 image를 통하여 인식하려는 이미지에 접근할 수 있다.

③ 인식하려는 문자의 색을 50으로 정하였다. 이는 색을 표현하는 0~255 사이의 값들 중 0에 가까운 50으로 정의한 것이다. 실제 사진을 모니터에서 보면 이미지가 다소 부드럽게 보이곤 하는데, 이것은 색이 바뀌는 경계선의 값이 진한 검은색의 값인 0보다 다소 크기 때문이다. 이번 장에서 사용하는 사진은 밝은 회색 바탕에 검은색 글씨이기 때문에 초기 값을 충분히 높은 값인 50으로 설정하였다.

④ 이미지 파싱의 1단계를 실행하고 2단계와 3단계가 순차적으로 진행되므로 ParsingStepFirst( ) 함수만을 실행하는 것으로 이미지 파싱은 진행되며 결과는 allData에 저장된다. 즉, 이미지 파싱 후 allData 변수에는 인식하려는 문자의 총 개수가 저장되고, 모든 이미지 데이터는 자기만의 사각형 영역을 간직하게 된다. 이번 장에서는 이미지 파싱만 진행되며, 아직은 문자 데이터가 만들어지지 않았다.

⑤ 이미지 파싱 후 얻은 모든 문자 이미지를 각각 사진으로 출력한다. 기본 이미지 데이터를 만들기 위해 전체 이미지에 포함된 62개의 문자가 모두 작은 사진 파일로 만들어진다.

이미지 파싱에서 구현하는 기술은 픽셀 라인Pixel Line에 문자 색이 포함되었는지를 확인하고, 이전Previous 픽셀 라인이 문자가 포함된 라인인지 아닌지를 판단하기 위해서 사용한다.

[그림 2-13]은 픽셀 라인에 따라 영역을 구분하는 것을 보여 준다. 이때 사용하는 변수는 flagPrevLine과 isLetterLine 두 개다. flagPrevLine은 현재 픽셀 라인의 이전 픽셀 라인을 나타낸다. flagPrevLine이 false면 문자 색이 포함되지 않은 바탕 선이고, true면 문자 색이 포함된 문자 라인을 의미한다.

isLetterLine은 현재의 픽셀 라인을 의미한다. isLetterLine이 false면 현재의 라인이 바탕 라인이고, true면 문자를 포함한 라인이다. 즉, 두 개의 라인으로 경계선을 알아낼 수 있다. 바탕 라인(flagPrevLine = false)에서 문자 라인(isLetterLine = true)으로 변했다면 현재 라인이 문자 영역의 시작 경계선임을 나타내며, 문자 라인(flagPrevLine = true)에서 바탕 라인(isLetterLine = false)으로 변했다면 이전 라인이 문자 영역의 종료 경계선임을 알 수 있다. 즉, 위에서부터 순차적으로 라인을 확인하면서 두 라인의 변화로 문자 영역을 확인하면 된다. 두 개의 라인으로 경계선을 알아내는 방법은 모든 이미지 파싱 단계에서 사용하는 기본 개념이다.

그림 2-13 이미지 파싱의 기본 개념

[코드 2-9]는 이미지 파싱의 1단계를 구현한 코드다. 문서 이미지에서 모든 문자 행의 위 경계선과 아래 경계선을 알아낸 후, 아래 경계선이 확인되면 이미지 파싱의 2단계를 수행한다. 앞에서 설명한 이미지 파싱의 기본 개념만을 이해하면 이제부터 구현하는 코드는 쉽게 이해할 수 있다.

[코드 2-9] 1단계 - COCR::ParsingStepFirst() 함수(OCR.cpp)

```
void COCR::ParsingStepFirst()
{
    allData.count = 0;                          //---①
    data = &allData.data[0];                    //---②

    int xMax = image->GetWidth();               //---③
    int yMax = image->GetHeight();
```

```
    int x, y;
    COLORREF rgb;                                   //----④
    int yTop, yBottom;                              //----⑤
    bool isLetterLine;                              //----⑥
    bool flagPrevLine;                              //----⑦

    int xStart = (int)(xMax * 0.4);                 //----⑧
    int xEnd = (int)(xMax * 0.6);                   //----⑨

    flagPrevLine = false;                           //----⑩

    for (y=0; y<yMax; y++) {                        //----⑪
        isLetterLine = false;                       //----⑫

        for (x=xStart; x<xEnd; x++) {               //----⑬

            rgb = image->GetPixel(x,y);             //----⑭

            if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                isLetterLine = true;                //----⑮
                break;
            }
        }

        if (isLetterLine) {                         //----⑯
            if (!flagPrevLine) {
                yTop = y;
            }
        }
        else {                                      //----⑰
            if (flagPrevLine) {
                yBottom = y-1;

                ParsingStepSecond(yTop, yBottom);   //----⑱
            }
        }

        flagPrevLine = isLetterLine;                //----⑲
    }
}
```

① allData는 인식하려는 모든 문자의 정보를 저장하는 변수이며, allData의 멤버 변수인 count는 allData에 포함된 문자의 총 개수를 나타낸다. 여기서는 문자 데이터를 저장하기 전에 count를 0으로 초기화한다.

② data 변수는 현재 사용하려는 문자를 가리키는 포인터 변수로, 처음에는 allData의 첫 번째 문자를 가리킨다. 이미지 파싱으로 처음 얻게 되는 문자 데이터를 data 변수에 넣으면 된다. 이후부터 data 변수는 allData의 배열 변수를 순차적으로 가리키며 동일한 프로세스를 진행하면 된다.

③ 이미지 인식을 위한 사진의 가로와 세로의 크기를 xMax와 yMax에 저장한다. 이 크기는 픽셀의 개수다.

④ rgb는 한 픽셀의 RGB 데이터를 확인하기 위해 사용하는 변수다. 픽셀의 RGB 데이터를 rgb 변수에 저장한 후 rgb 변수에서 색의 값을 얻는다.

⑤ yTop과 yBottom은 이미지 파싱의 1단계에서 얻는 문자 행의 위 경계선과 아래 경계선의 Y축 값을 저장하는 변수로, 2단계를 수행할 때 이 두 개의 값을 전달한다.

⑥ isLetterLine은 현재 확인하는 라인의 특징을 알기 위해 사용하며, isLetterLine이 true면 현재 연산하는 라인이 문자 라인임을 의미하며, false면 바탕 라인임을 나타낸다.

⑦ flagPrevLine은 현재 확인하는 라인의 이전 라인의 특징을 기억하기 위해 사용된다. flagPrevLine이 true이면 이전 라인이 문자 라인임을 의미하며, false면 바탕 라인임을 나타낸다.

⑧ 가로 픽셀 라인이 문자 색을 포함하는지를 확인하기 위해서 이미지의 왼쪽 끝부터 오른쪽 끝까지 연산하지는 않는다. 여기서는 전체 이미지의 중간 부분만을 계산하는데, 시작 위치는 가로 라인에 0.4를 곱한 위치에서 시작한다. 즉, 사진의 가로 크기가 1,000 픽셀이라면 400 픽셀 위치부터 시작한다.

⑨ 가로 픽셀 라인의 중간 부분에 대한 연산에서 오른쪽 끝부분은 가로 라인에 0.6을 곱한 위치다. 사진의 가로 크기가 1,000 픽셀이라면 600 픽셀 위치까지 확인한다. 즉, 가로의 크기가 1,000 픽셀이지만, 연산하는 픽셀의 개수는 400부터 600까지인 200개만을 확인한다.

⑩ 이전 라인을 의미하는 flagPrevLine은 바탕 라인을 의미하는 false로 초기화한다.

⑪ 이미지의 Y축은 0부터 마지막 값인 yMax까지 순차적으로 증가하면서 가로 픽셀 라인을 검사한다. for 문 안에서 모든 문자 행의 위와 아래의 경계선이 계산되어 파싱 작업이 진행된다.

⑫ 가로 픽셀 라인을 검사하기 전에 항상 현재 라인(isLetterLine)의 값은 바탕 라인(false)으로 초기화한다.

⑬ 가로 픽셀 라인을 검사하는 for 문으로, 시작점인 xStart부터 끝점인 xEnd까지 순차적으로 증가하면서 모든 픽셀을 검사한다.

⑭ 이미지의 (x, y)에 위치한 픽셀의 RGB 데이터를 rgb 변수에 저장한다.

⑮ RGB의 Red 값을 얻는 GetRValue( ) 함수를 사용하여 Red 값을 가져와서 검은색에 가까운지를 확인한다. 문자 색의 정해진 범위에 위치하면 문자 라인을 의미하는 isLetterLine을 true로 설정하고 for 문 밖으로 나오게 된다. if 문에서 colorToCheck는 문자 색을 의미하는데, 여기서는 50으로 선언하였다. RANGE_OF_COLOR_TO_CHECK는 문자 색의 범위를 확장하는데, 여기서는 100으로 선언하였다. 즉, RGB의 Red 값이 150(=50+100)보다 작으면 문자 색으로 인식하게 된다. 여기서는 RGB에서 Red 값만을 가져왔지만 Green이나 Blue 값을 가져와서 연산하는 것도 가능하다. 문자 색이 특정 색이라면 Red, Green, Blue의 모든 값을 가지고 범위를 정해야 할 수도 있다. 이 코드에서 문자 색을 100 이하로 정한 것은 우리가 보는 검은색이 실제 사진에서는 정확히 검은색의 값을 갖지는 않기 때문이며, 바탕색과의 차이가 크다면 범위를 크게 정하는 것이 좋다.

⑯ 현재 라인이 문자 라인이고(isLetterLine == true) 이전 라인이 바탕 라인이면(flagPrevLine == false) 현재 라인이 문자 행의 위 경계선을 의미하므로 yTop에 현재 라인의 Y축 값인 y를 넣어서 위 경계선을 정의한다.

⑰ 현재 선이 바탕 라인이고(isLetterLine == false) 이전 라인이 문자 라인이면(flagPrevLine == true) 이전 라인이 문자 행의 아래 경계선을 의미하므로 yBottom에 이전 라인의 Y축 값인 y를 넣어서 아래 경계선을 정의한다.

⑱ 아래 경계선이 정해지면 문자 행의 위와 아래가 정해진 것이므로 이미지 파싱의 2단계를 진행하는 ParsingStepSecond( ) 함수를 실행하는데, 이때 위와 아래 경계선을 의미하는 yTop과 yBottom의 값을 매개변수로 전달한다.

⑲ 현재 픽셀 라인의 검사가 끝나면 현재 픽셀 라인은 이전 픽셀 라인으로 바뀌기 때문에 flagPrevLine의 값을 isLetterLine 값으로 변경한다.

현재 픽셀 라인과 이전 픽셀 라인을 비교하여 경계선을 알아내는 방법은 가로 또는 세로를 기준으로 하여 X축과 Y축의 경계선을 알아내는 방법이다. 앞에서 이미지 파싱의 1단계로 문자 행의 Y축 값인 위와 아래의 경계선을 알아내었으며, 2단계에서는 X축의 경계선을 알아내어 문자를 하나씩 구분하였다.

[코드 2-10]은 이미지 파싱의 2단계를 구현한 코드로, 앞에서와 마찬가지로 flagPrevLine과 isLetterLine 변수를 가지고 두 선의 성격을 구분한다. 이미

지 파싱의 1단계에서는 문자 행의 위와 아래 경계선이 정해졌다면, 2단계에서는 하나의 문자에 대한 사각형의 영역을 알아낸다. 물론 이 사각형은 정확한 사각형(문자가 꽉 찬 사각형)이 아니기 때문에 이후 3단계를 추가적으로 수행해야 한다.

2단계에서 계산되어 얻어진 사각형은 Data 구조체의 멤버 변수인 Rect 구조체에 사각형의 값이 저장되는데, 이 사각형의 구조체는 Call-By-Reference로 3단계로 전달되어 사각형의 값이 수정된다. [코드 2-10]에서 구현한 코드의 설명 중 [코드 2-9]에서 설명한 내용은 생략하며 필요한 내용만 추가로 설명하였다.

[코드 2-10] 2단계 - COCR::ParsingStepSecond() 함수(OCR.cpp)

```
void COCR::ParsingStepSecond(int yTop, int yBottom)
{
    int xMax = image->GetWidth();                    //---①

    int x, y;
    COLORREF rgb;
    bool isLetterLine;
    bool flagPrevLine;

    flagPrevLine = false;

    for (x=0 ; x<xMax; x++) {                        //---②

        isLetterLine = false;

        for (y=yTop; y<=yBottom; y++) {              //---③
            rgb = image->GetPixel(x,y);

            if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {

                isLetterLine = true;
                break;
            }
        }

        if (isLetterLine) {                          //---④
            if (!flagPrevLine) {
                data->rect.start.x = x;
                data->rect.start.y = yTop;
            }
        }
        else {                                           //---⑤
```

```
            if (flagPrevLine) {

                data->rect.end.x = x-1;
                data->rect.end.y = yBottom;

                ParsingStepThird(&data->rect);          //---⑥

                allData.count += 1;                     //---⑦
                data = &allData.data[allData.count];    //---⑧
            }
        }
        flagPrevLine = isLetterLine;
    }
}
```

① 2단계의 함수는 매개변수로 yTop과 yBottom의 값을 받는데, 문자 행의 위와 아래 경계선의 Y축 값을 의미하며, X축의 전체 길이를 순차적으로 증가하면서 경계선의 X축 값을 계산한다. X축의 계산을 위해 xMax는 인식을 위한 이미지의 가로축 픽셀의 전체 개수를 저장한다.

② X축은 왼쪽 끝의 값인 0부터 오른쪽 끝 xMax까지 순차적으로 증가하면서 문자 행의 모든 문자의 왼쪽과 오른쪽 경계선을 계산하며, 오른쪽 경계선이 계산된 후에는 이미지 파싱의 3단계를 진행한다.

③ 세로 픽셀 라인을 검사하여 문자 라인인지 바탕 라인인지를 알게 된다. 세로 라인의 범위는 매개변수로 입력받은 yTop과 yBottom이다. 즉, yTop부터 yBottom까지 순차적으로 증가하며 세로 라인 모든 픽셀의 색을 검사하게 된다. 세로 라인의 픽셀 중에서 하나라도 문자 색을 가지고 있으면 현재 라인이 문자 라인임을 정의(isLetterLine = true)하고 for 문을 종료한다.

④ 현재 라인이 문자 라인이고(isLetterLine == true) 이전 라인이 바탕 라인이면(flagPrevLine == false) 현재 라인이 문자의 왼쪽 경계선을 의미하므로 사각형의 시작점(x, y) 값을 저장한다.

⑤ 현재 라인이 바탕 라인이고(isLetterLine == false) 이전 라인이 문자 선이면(flagPrevLine == true) 이전 라인이 문자 행의 아래 경계선을 의미하며 사각형의 끝점(x, y) 값을 저장한다.

⑥ 문자의 사각형 영역이 정해졌으므로 이미지 파싱의 3단계를 진행한다.

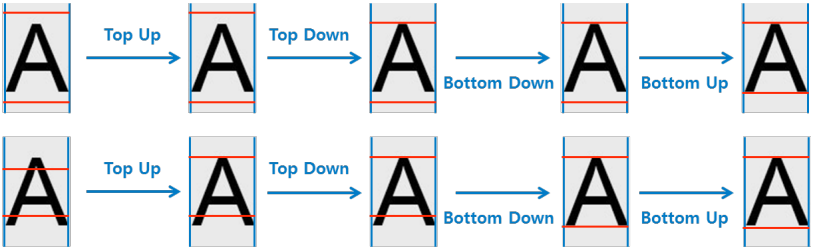
⑦ 현재 문자에 대한 이미지 파싱이 수행되었기 때문에 인식을 위한 문자의 총 개수를 1 증가시킨다.

⑧ 현재의 문자 데이터를 저장하는 구조체는 연산이 끝났기 때문에 포인터 변수인 data가 다음 빈Empty 구조체를 가리킨다. 배열Array의 인덱스Index는 0부터 시작하므로 [allData.count - 1]이 지금까지 연산이 수행된 구조체고 [allData.count]가 다음 빈 구조체를 의미한다.

이미지 파싱의 2단계까지 진행되면 하나의 문자에 대한 사각형 영역이 만들어진다. 그런데 이 사각형 영역이 이미지 데이터를 만들기 위한 정확한 영역은 아니기 때문에 3단계에서 사각형의 Y축에 대한 위와 아래의 경계선을 정확히 만들어 주는 작업을 진행해야 한다. 3단계에서 위와 아래 경계선을 보정해 주는 방법은 개별적으로 진행되는데, 먼저 위 경계선을 한 픽셀씩 위로 올리면서 문자 선인지 검사한 후, 다시 한 픽셀씩 아래로 내리면서 문자 선인지를 검사한다. 마찬가지로 아래 경계선을 아래로 한 픽셀씩 아래로 내리면서 문자 선인지 검사한 후, 다시 한 픽셀씩 위로 올리면서 문자 선인지를 검사한다.

[그림 2-14]는 위와 아래 경계선에 대한 보정 방법을 보여 주는데, 4단계로 진행된다. 먼저 위 경계선의 한 픽셀 위의 선을 검사한 후, 문자 선이면 위 경계선이 한 픽셀 위로 올라간다. 이와 같은 작업을 반복하게 되는데, 한 픽셀 위의 선이 바탕색일 때까지 진행한다([그림 2-14] Top Up). 이후에 위 경계선은 한 픽셀 아래의 라인을 검사하게 되는데, 한 픽셀 아래의 라인이 바탕 라인이면 위 경계선을 순차적으로 아래로 내리면서 문자 라인이 나올 때까지 진행한다([그림 2-14] Top Down). 위 경계선의 보정 작업과 마찬가지의 작업이 아래 경계선에서 진행되는데, 먼저 아래 경계선의 아래쪽을 검사하면서 바탕 . 이 나올 때까지 진행하고([그림 2-14] Bottom Down) 이후에 아래 경계선의 위쪽을 검사하면서 문자 라인이 나올 때까지 진행한다([그림 2-14] Bottom Up).

그림 2-14 이미지 파싱의 경계선 조정 방법



이미지 파싱의 1단계와 2단계는 한 번씩만 진행되지만, 3단계의 보정 작업은 여러 번 진행될 수도 있다. 이 책에서는 쉽게 이해하기 위해 초반에 3단계까지만 설명하였는데, OCR에서 가장 어려운 부분이 이미지 파싱이다. 이미지마다 특징이 다르기 때문에 다양한 방법으로 이미지 파싱이 진행된다. 이 책에서는 가장 기본이 되는 1, 2, 3단계를 순차적으로 진행하여 이미지 파싱의 결과를 확인할 예정이며, 추가적인 보정 작업이 왜 필요한지를 이후에 확인하고자 한다.

[코드 2-11]은 이미지 파싱의 3단계를 코드로 구현한 것으로, 4가지 방법(Top Up, Top Down, Bottom Down, Bottom Up)을 차례대로 진행한다.

[코드 2-11] 3단계(세로 보정 작업) - COCR::ParsingStepThird() 함수(OCR.cpp)

```
void COCR::ParsingStepThird(Rect *rect)
{
    int x, y;
    COLORREF rgb;
    bool isLetterLine;

    //------------------------ Letter Top ---------------------------
    for (y=rect->start.y; ; y--) {                    //---①

        isLetterLine = false;

        for (x=rect->start.x; x<=rect->end.x; x++) {

            rgb = image->GetPixel(x,y);

            if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                rect->start.y = y;
                isLetterLine = true;
```

```
            break;
        }
    }
    if (!isLetterLine)                          //---②
        break;
}
for (y=rect->start.y; ; y++)  {                 //---③

    isLetterLine = false;

    for (x=rect->start.x; x<=rect->end.x; x++) {

        rgb = image->GetPixel(x,y);

        if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
            rect->start.y = y;
            isLetterLine = true;
            break;
        }
    }
    if (isLetterLine)                           //---④
        break;
}
//------------------------ Letter Bottom --------------------------
for (y=rect->end.y; ; y++) {                    //---⑤

    isLetterLine = false;

    for (x=rect->start.x; x<=rect->end.x; x++) {
        rgb = image->GetPixel(x,y);

        if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
            rect->end.y = y;
            isLetterLine = true;
            break;
        }
    }
    if (!isLetterLine)                  //---⑥
        break;
}
for (y=rect->end.y;; y--) {                 //---⑦

    isLetterLine = false;

    for (x=rect->start.x; x<=rect->end.x; x++) {
        rgb = image->GetPixel(x,y);
```

```
            if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                rect->end.y = y;
                isLetterLine = true;
                break;
            }
        }
        if (isLetterLine)                    //---⑧
            break;
    }
}
```

① 문자의 위 경계선에 대한 'Top Up' 방법을 진행한다. 위Top 경계선의 y 값은 사각형의 시작점부터 시작하며, 위Up로 진행하기 때문에 y 값이 1씩 감소하게 된다. 여기서 for 문의 가운데 값인 조건문은 생략하였는데, for 문 안에서 조건 연산자 if를 사용하여 밖으로 나온다. for 문 안에서는 가로 라인을 검사하여 문자 라인과 바탕 라인을 구분하는데, isLetterLine에 값을 저장하며 문자 라인일 경우에 시작점의 y 값은 현재 라인으로 바뀌게 된다.

② 문자의 위 경계선을 위로 올리면서 진행하는 'Top Up' 방법은 검사하는 가로 라인이 바탕 라인이면 진행이 종료된다. 즉, 경계선은 바탕 라인을 포함하지 않는다.

③ 문자의 위 경계선에 대한 'Top Down' 방법을 진행한다. 위 경계선의 y 값은 사각형의 시작점부터 시작하며 아래로 진행하기 때문에 y 값이 1씩 증가한다. for 문 안에서 가로 라인을 검사하여 바탕 라인일 경우 y 값이 증가하면서 문자 라인이 나올 때까지 반복하여 진행한다.

④ 문자의 위 경계선을 아래로 내리면서 진행하는 'Top Down' 방법은 검사하는 가로 라인이 문자 라인이면 진행이 종료된다.

⑤ 문자의 아래 경계선에 대한 'Bottom Down' 방법을 진행한다. 아래 경계선의 y 값은 사각형의 끝점에서 시작하며 아래로 진행하기 때문에 y 값이 1씩 증가한다. for 문 안에서는 가로 라인을 검사하여 가로 라인이 바탕 라인일 때까지 계속해서 진행한다.

⑥ 'Bottom Down' 방법은 문자의 아래 경계선을 아래로 내리면서 진행하는 방법으로, 검사하는 가로 라인이 바탕 라인이면 진행이 종료된다.

⑦ 문자의 아래 경계선에 대한 'Bottom Up' 방법을 진행한다. 아래 경계선의 y 값은 사각형의 끝점에서 시작하며 위로 진행하기 때문에 y 값이 1씩 감소한다. for 문 안에서는 가로 라인을 검사하여 바탕 라인일 경우 y 값이 감소하면서 문자 라인이 나올 때까지 반복하여 진행한다.

⑧ 문자의 아래 경계선을 위로 올리면서 진행하는 'Bottom Up' 방법은 검사하는 가로 라인이 문자 라인이면 진행이 종료된다.

지금까지 이미지 파싱의 1, 2, 3단계를 구현하였다. 이미지 파싱이 정확히 이루어졌는지 확인하는 가장 좋은 방법은 파싱된 이미지를 눈으로 보는 것이다. 즉, 전체 이미지에서 문자마다 가지고 있는 각각의 영역에 대한 이미지를 개별 이미지 파일로 만드는 것이 좋다. 지금까지 구현한 프로그램의 단위 테스트를 진행하는 것이라고 생각하면 된다. 또한, 이미지 파싱을 정확히 수행하였는지 확인하여 오류가 없는 코드를 기반으로 추가 코드를 구현하는 것이 가장 좋은 방법이다. 프로그램을 구현하면 완성된 프로그램에는 포함되지 않지만 테스트를 위해 만들어야 하는 함수가 많아지곤 한다.

[코드 2-12]는 이미지 파싱이 이루어진 후 사각형 안의 모든 문자 이미지를 사진 파일로 출력하기 위해 구현한 `PrintImageToFile( )` 함수다. 이 함수에서 수행되는 과정은 간단하다. 출력하기 위한 사각형 이미지와 같은 크기의 새로운 이미지 클래스를 만든 다음, 사각형 이미지의 모든 픽셀을 새로운 이미지에 저장한 후 사진 파일로 저장한다.

[코드 2-12] 이미지 파싱 이후의 문자 이미지 사진 출력(OCR.cpp)

```
void COCR::PrintImageToFile(int fileNo, Rect *rect)          //---①
{
    CString strName;
    strName.Format(TEXT("./image%d.jpg"), fileNo);           //---②

    CImage newImage;                                         //---③
    int width, height;

    width = rect->end.x - rect->start.x + 1;                 //---④
    height = rect->end.y - rect->start.y + 1;

    if (width <=0 || height <= 0)
    {
        AfxMessageBox(_T("Error: 사진의크기가0보다작다."));
        return;
    }
```

```
    newImage.Create(width, height, 32);                          //---⑤

    for (int i=0; i<width; i++)                                  //---⑥
        for(int j=0; j<height; j++)
            newImage.SetPixel(i,j, image->GetPixel(rect->start.x+i, rect->start.
            y+j));

    newImage.Save(strName, Gdiplus::ImageFormatJPEG);            //---⑦
}
```

① 매개변수 fileNo와 rect 변수를 입력받아서 fileNo를 포함한 파일명의 사진 파일을 만든다. 이때, rect 변수는 문자 이미지의 사각형 영역 정보를 가지고 있으며 사진 이미지에서 해당 사각형 영역의 이미지를 사진 파일로 출력한다.

② 새롭게 만들어지는 사진 파일의 파일명을 정의한다. fileNo 숫자를 포함한 파일명으로 만든다. 예를 들어, fileNo가 3이라면 'image3.jpg'으로 파일명이 만들어지고, fileNo가 14라면 'image14.jpg'의 파일명이 만들어진다.

③ 새로운 이미지는 newImage 이름을 가진 CImage 클래스에 저장된다.

④ 새로운 이미지의 크기는 가로와 세로의 크기를 계산해서 width와 height에 저장한다. 이미지는 시작점과 끝점을 모두 포함하기 때문에 크기는 끝점에서 시작점을 뺀 값에 1을 더한 값이다. 예를 들어, 시작점의 x값이 20이고 끝점의 x값이 40이라면, 20부터 50까지의 점의 개수는 31 (=50-20+1)이 된다.

⑤ 새로운 이미지를 생성한다. 가로와 세로의 크기는 width와 height로 정의된다. 32는 픽셀당 Bit 수를 의미하는데, 32-bits는 4-Bytes며 항상 32를 사용하면 된다.

⑥ 새로운 이미지의 각 픽셀에 전체 사진 이미지에서 사각형에 해당하는 픽셀 값을 넣어 준다. 새로운 이미지는 크기가 정해졌으므로 모든 픽셀의 정보만 가지면 사진에서 필요한 모든 정보는 가졌다고 볼 수 있다.

⑦ 새로운 이미지를 사진 파일로 저장하는데, strName이라는 파일명으로 저장된다.

[그림 2-15]는 이미지 파싱의 1, 2, 3단계를 진행한 후 문자 이미지를 사진 파일로 출력한 결과다. 지금까지는 이미지 파싱의 가장 기본적인 부분을 구현하였다. 이는 사진에서 어떻게 문자 이미지를 가져오는지를 이해하기 위한 과정이었다.

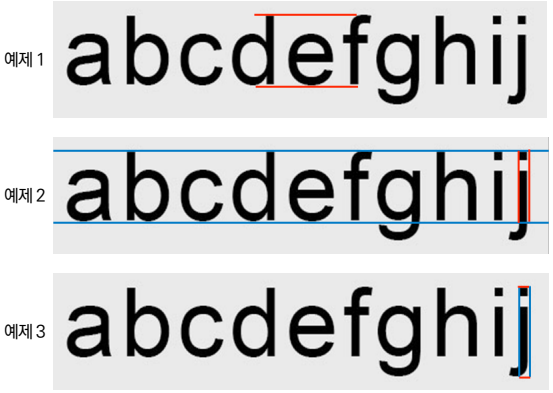
그림 2-15 이미지 파싱 1, 2, 3단계 진행 후 출력된 파일

[그림 2-15]에서 모든 문자 이미지 사진은 정상적으로 출력된 것처럼 보인다. 하지만 문자 이미지를 자세히 보면 'image9.jpg' 파일의 'j' 문자는 정상적으로 파싱되지 않았다. 이미지 파싱의 1, 2, 3단계를 진행한 것만으로는 모든 문자가 제대로 파싱되지 않음을 알 수 있다. 이미지 파싱은 하나의 방법을 모든 이미지에 적용할 수 없으며, 문자를 읽어 오기 위해 이미지마다 다른 방법을 적용하는 경우도 있다.

그렇다면 이제부터 'j' 문자를 정확히 얻어 오는 방법을 알아보자. 이미지 파싱에서 1, 2, 3단계가 기본이 된다고 하였다. 지금까지의 내용을 이해하였다면 이후의 작업은 보정 작업을 진행하는 3단계의 반복 작업이므로 쉽게 이해할 수 있다. .

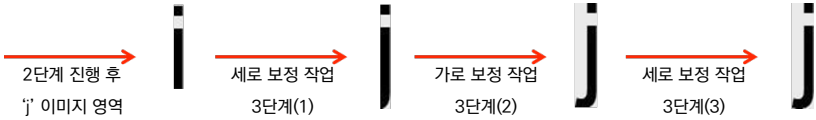
이미지 파싱의 1, 2, 3단계만으로 'j'를 제외한 모든 문자에 대한 이미지 영역을 정확히 얻을 수 있었다. [그림 2-16]은 'j'에 대한 이미지 파싱이 어떻게 진행되었는지를 보여 준다. 2단계 진행 후 3단계에서 Y축(세로 영역)에 대한 보정 작업을 진행하였기 때문에 'j' 문자의 왼쪽 아래 부분의 영역을 검사하지 못하였다. 지금까지 3단계는 Y축에 대한 보정 작업만을 진행하였는데, 'j' 문자의 경우에는 3단계의 보정 작업이 X축에 대해서도 필요함을 알 수 있다.

그림 2-16 소문자 'j'의 기본 이미지 파싱 진행 결과



[그림 2-17]은 3단계의 보정 작업을 가로와 세로를 반복하여 진행하여 정확한 문자의 영역을 알아내는 것을 보여 준다. 2단계 진행 후 'j' 문자는 'i' 문자와 같은 이미지를 얻으며, 세로 보정 작업 후 사각형 영역의 Y축의 값이 변하게 된다. 마찬가지로 X축에 대한 가로 보정 작업을 진행하게 되면 왼쪽의 숨겨진 영역을 포함한 새로운 영역을 얻는다. 세로와 가로 보정 작업이 한 번씩 진행되었지만, 아직도 정확한 보정 작업이 진행된 것은 아니다. X축의 값이 변하면서 왼쪽 영역은 포함되었지만, 그 밑의 미세한 영역은 다시 계산되어야 한다. 결과적으로 기본 이미지 영역을 정확히 얻기 위해서는 이미지 파싱 3단계의 보정 작업이 세로, 가로, 세로로 순차적으로 진행되어야 한다.

그림 2-17 추가 보정 작업 (가로&세로)



[코드 2-13]은 이전에 구현한 이미지 파싱 2단계의 코드에서 2단계가 진행된 후 3단계를 3번 진행하는 것이다. ParsingStepThird( ) 함수는 세로 보정 작업을 진행하고, ParsingStepThird2( ) 함수는 가로 보정 작업을 진행한다. 즉, ParsingStepThird( ) 함수는 이전에 구현하였고, 3단계의 가로 보정 작업을 위한 ParsingStepThird2( ) 함수만을 추가로 구현하였다. [코드 2-13]의 내용은 [코드 2-10]에서 설명한 것과 동일하며 단지 3단계를 진행하는 부분만 변경되었다.

[코드 2-13] 2단계 진행 후 3번의 보정 작업 수행 – 세로, 가로, 세로(OCR.cpp)

```
void COCR::ParsingStepSecond(int yTop, int yBottom)
{
    int xMax = image->GetWidth();

    int x, y;
    COLORREF rgb;
    bool isLetterLine;
    bool flagPrevLine;

    flagPrevLine = false;

    for (x=0 ; x<xMax; x++) {

        isLetterLine = false;

        for (y=yTop; y<=yBottom; y++) {
            rgb = image->GetPixel(x,y);

            if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {

                isLetterLine = true;
                break;
            }
        }

        if (isLetterLine) {
```

```
            if (!flagPrevLine) {
                data->rect.start.x = x;
                data->rect.start.y = yTop;
            }
        }
        else {
            if (flagPrevLine) {

                data->rect.end.x = x-1;
                data->rect.end.y = yBottom;

                ParsingStepThird(&data->rect);          //---①
                ParsingStepThird2(&data->rect);         //---②
                ParsingStepThird(&data->rect);          //---③

                allData.count += 1;
                data = &allData.data[allData.count];
            }
        }

        flagPrevLine = isLetterLine;
    }
}
```

① 3단계의 세로 보정 작업을 진행한다. 사각형의 영역을 나타내는 'data->rect'는 Call-By-Reference로 전달되며, 보정 작업 완료 후 'data->rect'의 값은 보정된 값으로 변경된다.

② 3단계의 가로 보정 작업을 진행한다. [코드 2-14]에서 확인할 수 있다.

③ 3단계의 세로 보정 작업을 한 번 더 진행한다. 가로와 세로의 보정 작업은 반복적으로 이루어질수록 더 정확한 값을 얻을 수 있는데, 지금까지의 내용에서 알 수 있듯이 이미지 파싱에서는 세로, 가로, 세로의 순서로 보정 작업을 3번만 진행하면 충분하다.

[코드 2-14]는 3단계의 가로 보정 작업인 ParsingStepThird2( ) 함수의 코드다. 세로 보정 작업에서 x와 y 변수를 서로 변경한 것이다. 즉, [코드 2-11]과 내용은 같으며 단지 y를 x로 변경하고 x를 y로 변경하였다. [코드 2-11]의 코드와 보정 방법을 이해하였다면 [코드 2-14]도 동일한 방법으로 이해하면 된다.

[코드 2-14] 3단계(가로 보정 작업) - COCR::ParsingStepThird2( ) 함수(OCR.cpp)

```
void COCR::ParsingStepThird2(Rect *rect)
{
    int x, y;
    COLORREF rgb;
    bool isLetterLine;

    //------------------------- Letter Left --------------------------
    for (x=rect->start.x; ; x--) {                          //---①

        isLetterLine = false;

        for (y=rect->start.y; y<=rect->end.y; y++) {        //---②

            rgb = image->GetPixel(x,y);

            if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                rect->start.x = x;
                isLetterLine = true;
                break;
            }
        }
        if (!isLetterLine)                                  //---③
            break;
    }
    for (x=rect->start.x; ; x++) {                          //---④

        isLetterLine = false;

        for (y=rect->start.y; y<=rect->end.y; y++) {

            rgb = image->GetPixel(x,y);

            if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                rect->start.x = x;
                isLetterLine = true;
                break;
            }
        }
        if (isLetterLine)                                   //---⑤
            break;
    }
    //------------------------ Letter Right ------------------------
    for (x=rect->end.x; ; x++) {                            //---⑥

        isLetterLine = false;
```

```
        for (y=rect->start.y; y<=rect->end.y; y++) {
            rgb = image->GetPixel(x,y);

            if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                rect->end.x = x;
                isLetterLine = true;
                break;
            }
        }
        if (!isLetterLine)                                  //---⑦
            break;
    }
    for (x=rect->end.x;; x--) {                             //---⑧

        isLetterLine = false;

        for (y=rect->start.y; y<=rect->end.y; y++) {
            rgb = image->GetPixel(x,y);

            if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                rect->end.x = x;
                isLetterLine = true;
                break;
            }
        }
        if (isLetterLine)                                   //---⑨
            break;
    }
}
```

① 문자 이미지의 왼쪽 경계선은 왼쪽으로 한 픽셀씩 이동하면서 문자 라인이 나오면 계속 진행하고 바탕 라인이면 종료한다. 즉, 사각형의 시작점의 x값은 1씩 줄어들면서 검사한다.

② 문자 이미지의 왼쪽 경계선의 모든 픽셀을 검사하는데, 한 픽셀이라도 문자 색이라면 문자 라인으로 인식한다.

③ 가로 보정 작업은 왼쪽 경계선이 왼쪽으로 이동하면서 검사할 때 바탕 라인이면 종료한다.

④ 왼쪽 경계선은 오른쪽으로 한 픽셀씩 이동하면서 바탕 라인이 나오면 계속 진행하고 문자 라인이면 종료한다.

⑤ 왼쪽 경계선이 오른쪽으로 이동하면서 검사할 때 문자 라인이 나오면 for 문을 종료한다.

⑥ 문자 이미지의 오른쪽 경계선은 오른쪽으로 한 픽셀씩 이동하면서 문자 라인이면 계속 진행하고 바탕 라인이면 종료한다.

⑦ 문자 이미지의 오른쪽 경계선이 오른쪽으로 이동하면서 검사할 때 바탕 라인이면 종료한다.

⑧ 오른쪽 경계선이 왼쪽으로 한 픽셀씩 이동하면서 바탕 라인이면 계속 진행하고, 문자 라인이면 종료한다.

⑨ 오른쪽 경계선이 왼쪽으로 이동하면서 검사할 때 바탕 라인이면 종료한다.

지금까지의 방법으로 이미지 파싱을 수행하면 정확한 문자 이미지를 얻을 수 있다. OCR에서 가장 어려운 부분이 이미지 파싱인데, 사진마다 문자 색이 다른 경우나 문자가 정확히 수평을 이루지 않은 경우도 있으므로 이미지마다 다양한 이미지 파싱 방법을 적용해야 하기 때문이다.

지금까지의 방법은 이미지 파싱의 가장 기본이 되는 기법이며 이것을 응용하여 사진 이미지에 따른 다양한 방법을 구현하면 생각보다 그리 어렵지는 않을 것이다. 다음 절에서는 문자 이미지로부터 연산이 가능한 데이터를 얻는 방법을 구현한다.

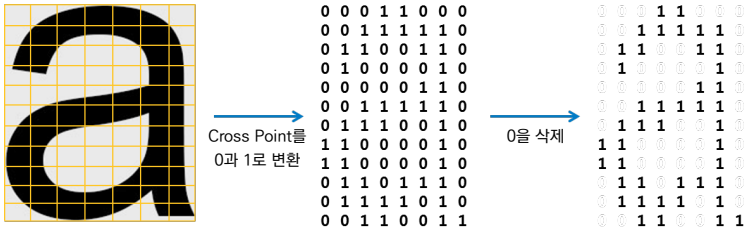
2.4 이미지 데이터 만들기

이미지 파싱으로 가로와 세로의 크기가 정해진 문자 이미지를 연산이 가능한 동일 크기의 이미지 데이터에 넣기 위해서는 가로와 세로를 일정한 크기로 분할하여, 분할된 위치의 픽셀에 대한 값을 0 또는 1로 정하여 해당 위치의 이미지 데이터에 넣으면 된다.

[그림 2-18]은 'a' 문자를 8×12 크기의 이미지 데이터에 넣는 방법이다. 'a' 이미지를 8개의 세로선과 12개의 가로선을 이용하여 분할하였다. 그리고 가로선과 세로선이 교차하는 위치Cross Point의 색이 문자 색Black Color이면 '1'을 넣고, 바탕 색Gray Color이면 '0'을 대입하였다. 가장 오른쪽의 1로만 표현된 'a'는 이미지 데이터로 변환한 후에도 'a'의 이미지를 표현할 수 있음을 확인하기 위해 만들었다. 각

문자 이미지는 크기가 모두 다르지만, 분할하는 선의 개수가 같기 때문에 선이 교차하는 위치Cross Point가 다를 뿐 이미지 데이터로 만들어진 후에는 모든 이미지 데이터의 크기는 동일해진다.

그림 2-18 문자 'a'를 8×12 크기의 이미지 데이터로 변환



문자 이미지를 분할하는 방법은 이미지 분할에 따른 교차하는 점들Cross Points의 차이Distance를 계산으로 알아낸 후 처음 시작하는 점부터 0 또는 1로 변환하여 이미지 데이터에 넣으면 된다. [그림 2-18]은 이해를 위해 8×12 크기의 이미지 데이터로의 변환을 보여 준 것이지만, 이 책에서는 32×48 크기의 이미지 데이터를 만든다.

[코드 2-15]는 기본 이미지를 만드는 함수의 코드다. 이미지 파싱 후에 이미지 데이터를 만드는 `MakeImageData( )` 함수를 추가하고, 정상적으로 프로그램이 동작하는지 확인하기 위해 모든 이미지 데이터를 순차적으로 파일에 기록하는 `PrintImageDataToFile( )` 함수를 사용하였다.

[코드 2-15] 이미지 데이터 만드는 함수 추가(OCR.cpp)

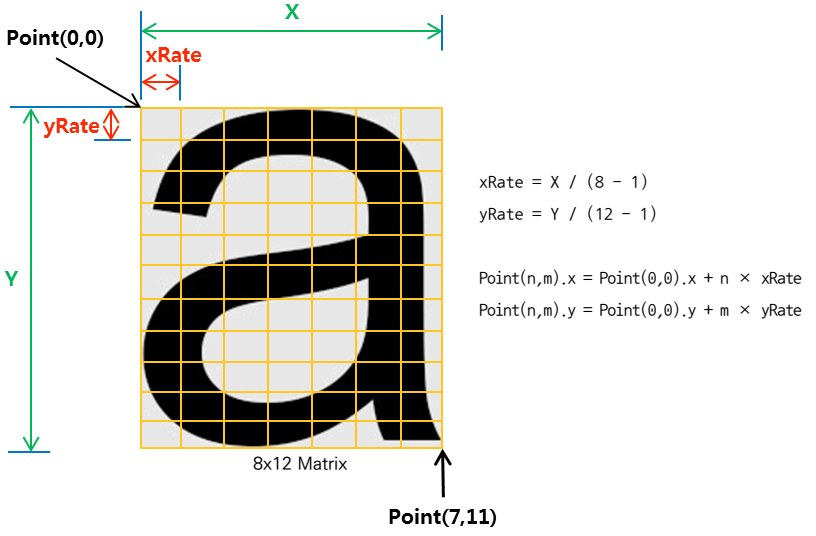
```
void COCR::CreateStandard(CImage *newImage)
{
    image = newImage;
    colorToCheck = 50;

    ParsingStepFirst();
    MakeImageData();                                          //---①

    for (int i=0; i<allData.count; i++)
        PrintImageDataToFile(i, &allData.data[i].letter);     //---②
}
```

① 이미지 파싱 후 모든 문자의 영역이 정해졌다. `MakeImageData()` 함수를 실행하여 각각의 문자에 대한 이미지 데이터를 만든다. 이미지 데이터를 만드는 함수는 이후에 모든 데이터 이미지를 만드는 데 사용한다.

② 하나의 이미지를 하나의 파일에 기록한다. 테스트를 위해 이미지가 정상적으로 만들어졌는지를 확인하는 함수로, 실제 프로그램이 완성되고 난 이후에는 사용되지 않는다. `PrintImageDataToFile()` 함수의 매개변수 `i`는 만들어지는 파일의 구분을 위한 정수값이다.

[그림 2-19]는 이미지 데이터로 변환하는 방법을 이해하기 위한 그림이다. 문자 이미지의 영역은 시작점과 끝점이 정해졌기 때문에 가로의 길이 X와 세로의 길이 Y를 알 수 있다. 또한, 8×12 크기로 나누기 위해서는 일정한 간격을 구해야 하는데, 가로에 대한 일정한 간격 xRate와 세로에 대한 간격 yRate도 계산으로 얻을 수 있다. 즉, [그림 2-19]의 수식 'xRate = X / (8 - 1)'과 같이 8개로 만들기 위한 길이 xRate는 가로의 길이에 X에서 가로의 개수 8보다 1 작은 값으로 나누면 된다. xRate와 yRate가 정해졌다면 문자 이미지에서 선들이 교차하는 지점의 위치는 시작점(Point(0,0))에서 xRate와 yRate를 가지고 계산할 수 있다.

그림 2-19 문자 'a'를 8×12 크기의 이미지 데이터로 변환하는 방법



[코드 2-16]은 영역을 가진 모든 문자 이미지를 이미지 데이터로 변환하는 함수이며, [그림 2-19]의 내용을 코드로 구현한 것이다. 물론 이미지 데이터의 크기 32×48로 계산되어 48개의 Integer(32-bits) 배열에 넣는 작업으로 이해하면 된다. 여기서 사용하는 연산자 중에 중요한 것은 Shift 연산자(>>)로[09], 비트 연산을 위해서 사용한다. 또한, '0x'를 사용하여 8진수로 32-bits의 정수를 초기화했다. 즉, 32-bits 크기 데이터에서 가장 왼쪽 bit만을 1로 만들기 위해서 '0x80000000'와 같이 초기화하였는데, 이것은 다음과 같이 이해하면 된다.

```
0x80000000 = 1000 0000 0000 0000 0000 0000 0000 0000 // 2진수
```

0x80000000로 초기화한 변수를 Shift 연산자를 사용하여 1을 가진 bit의 위치를 정하게 된다. 다음은 32-bits의 데이터를 왼쪽부터 13번째 bit 값만 1로 만드는 방법이다.

```
unsigned int buffer = 0x80000000; //10000000 00000000 00000000 00000000
buffer >>= 12;     //00000000 00001000 00000000 00000000
```

[코드 2-16]은 픽셀로 이루어진 문자 이미지를 32개의 가로선과 48개의 세로선을 만들어서 교차하는 위치의 픽셀의 rgb 값을 가지고, 문자 색이면 이미지 데이터의 해당 위치에 1을 넣는다. 이미지 데이터는 초기에 모든 bit 값을 0으로 초기화했기 때문에 문자 색인 경우에만 1을 삽입하면 된다. 이미지 데이터 연산 후 인식을 위한 문자의 정보를 가진 data의 Letter 구조체 변수인 letter는 이미지 데이터의 결과값을 갖게 된다.

[코드 2-16] 이미지 데이터 만들기(OCR.cpp)

```
void COCR::MakeImageData() {
```

09 비트 연산자인 Shift 연산자에 대한 자세한 설명은 『소수와 RSA 알고리즘으로 배우는 Big Number 연산』(한빛미디어, 2013)의 160~163페이지를 참조하면 된다.

```
    int i, j;

    for (i=0; i<allData.count; i++) {                          //---①

        Data *data = &allData.data[i];                         //---②
        Letter *letter = &data->letter;                        //---③
        Rect *rect = &data->rect;                              //---④

        float xRate, yRate;
        int x, y;
        unsigned int buffer;
        COLORREF rgb;

        for (j=0; j<48; j++)                                   //---⑤
            letter->image[j] = 0x00000000;

        xRate = (float)(rect->end.x - rect->start.x) / (32 - 1); //---⑥
        yRate = (float)(rect->end.y - rect->start.y) / (48 - 1); //---⑦

        for (y=0; y<48; y++) {                                 //---⑧
            for (x=0; x<32; x++) {                             //---⑨

                rgb = image->GetPixel((int)(rect->start.x + (x * xRate)), (int)
                (rect->start.y + (y * yRate)));                //---⑩
                if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                    buffer = 0x80000000;                       //---⑪
                    buffer >>= x;                              //---⑫
                    letter->image[y] |= buffer;                //---⑬
                }
            }
        }
    }
}
```

① allData에는 이미지 데이터를 만들기 위한 모든 이미지가 포함되어 있으며, for 문으로 모든 문자 이미지를 하나씩 이미지 데이터로 만든다.

② Data 구조체의 포인터 변수 data는 allData에 있는 allData.data 배열을 하나씩 가리킨다.

③ Letter 구조체의 포인터 변수 letter는 이미지 데이터를 만들기 위해 선택된 data의 data->letter 변수를 가리킨다.

④ 이후 코드의 단순화를 위해 Rect 구조체의 포인터 변수 rect는 현재 문자 이미지의 사각형 영역을 저장하는 data->rect 변수를 가리킨다.

⑤ 이미지 데이터가 저장되는 배열인 letter->image 변수를 0으로 초기화한다. image 변수는 unsigned int이고 32-bits 크기의 48개 배열이라서 32×48 bits로 이루어져 하나의 이미지 데이터를 저장한다. 즉, 48번의 for 문으로 모든 배열의 데이터를 0으로 초기화한다.

⑥ 이미지의 X축 간격을 의미하는 xRate를 계산한다. 이는 사각형의 가로 시작점과 끝점 간격을 32개의 선으로 나누었을 때 선과 선 사이의 간격을 의미한다.

⑦ 이미지의 Y축 간격을 의미하는 yRate를 계산한다. 이는 사각형의 세로 시작점과 끝점 간격을 48개의 선으로 나누었을 때 선과 선 사이의 간격을 의미한다.

⑧ 한 문자에 대한 문자 데이터를 얻기 위해 세로로 처음부터 48번의 for 문으로 마지막까지 반복한다.

⑨ 한 문자의 가로선에 대한 문자 데이터를 얻기 위해 가로로 처음부터 32번의 for 문으로 오른쪽 끝까지 반복한다.

⑩ 가로선과 세로선이 교차하는 위치에 해당하는 이미지의 rgb 값을 얻는다.

⑪ 해당 위치의 rgb가 문자 색이라면 해당 문자 데이터에 1을 넣는다. 여기서는 1을 넣기 위해 32개 bit의 제일 왼쪽 첫 bit를 1로 초기화한다.

⑫ 가장 왼쪽의 1의 값을 갖는 bit를 Shift 연산자를 사용하여 해당 위치의 bit만이 1이 되도록 설정한다.

⑬ '|' (or) 연산자로 문자 이미지 데이터인 image[y]의 해당 위치에 1을 만든다.

지금까지 특정 크기의 문자 이미지 데이터를 만드는 법을 구현하였다. 문자 이미지 데이터가 정확히 만들어졌는지 확인하는 가장 좋은 방법은 출력해서 직접 확인하는 것이다. [코드 2-17]과 같이 이미지를 파일에 출력하는 코드를 구현하여 [그림 2-20]과 같은 결과를 얻으면 된다. 즉, '32×48'개의 bit로 이루어진 문자 이미지 데이터를 '32×48'개의 0과 1로 이루어진 텍스트 파일을 만들어서 확인하면 된다.

[코드 2-17] 이미지 데이터 출력 - 0 & 1 Text File (OCR.cpp)

```
void COCR::PrintImageDataToFile(int fileNo, Letter *letter) {     //---①
    CString strName;
    strName.Format(TEXT("./ImageData%d.txt"), fileNo);             //---②
    FILE *fp;
```

```
    fp = fopen(LPSTR(LPCTSTR(strName)), "wt");                    //---③

    int x, y;
    unsigned int buffer;

    for (y=0; y<48; y++) {                                       //---④
        buffer = letter->image[y];                               //---⑤

        for (x=0; x<32; x++) {                                   //---⑥
            if (buffer & 0x80000000)                             //---⑦
                fputc('1', fp);
            else
                fputc('0', fp);
            buffer <<= 1;                                        //---⑧
        }
        fputc('\n', fp);                                         //---⑨
    }

    fclose(fp);
}
```

① `PrintImageDataToFile( )` 함수는 `fileNo`를 포함하는 파일 이름의 텍스트 파일에 특정 `letter`의 이미지 데이터를 출력한다.

② 파일 이름은 `fileNo`가 1이면 'ImageData1.txt'가 되고, 24면 'ImageData24.txt'로 만든다.

③ 생성되는 파일을 메모리에서 오픈Open하는 속성은 쓰기(w)와 텍스트(t) 모드로 생성된다.

④ 문자 이미지 데이터는 48개의 '`unsigned int`'(32-bits)며, `for` 문을 48번 반복하여 하나씩 처리한다.

⑤ 문자 이미지의 '`unsigned int`'를 하나씩 `buffer`에 담아서 처리하게 된다. 이와 같이 처리하는 이유는 ⑧에서 Shift 연산자를 사용하는데 원본 데이터에 변화를 주지 않기 위해서다.

⑥ 가로의 32-bits의 데이터를 하나씩 확인하기 위해 `for` 문으로 32번 반복한다.

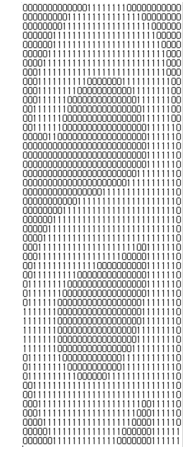
⑦ `buffer`의 가장 왼쪽 bit가 1이면 문자 '1'을 파일에 기록하며, 그렇지 않으면 '0'을 기록한다.

⑧ Shift 연산자를 사용하여 `buffer`의 모든 bit를 왼쪽으로 한 칸씩 이동시킨다.

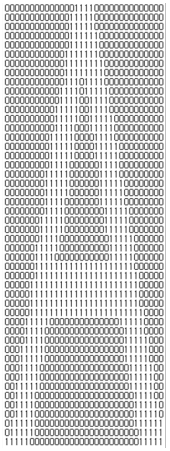
⑨ 생성되는 파일에 대한 모든 작업이 끝났기 때문에 `File Close`로 메모리에 있는 내용을 디스크에 저장한다.

지금까지 구현한 코드를 실행하면 'ImageData0.txt'부터 'ImageData61.txt'까지 62개의 텍스트 파일이 생성된다. [그림 2-20]은 그중에서 세 개 파일의 내용이다.

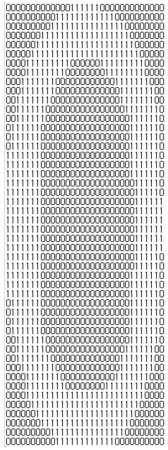
그림 2-20 이미지 데이터 출력 결과(소문자 'a', 대문자 'A', 숫자 '0')



ImageData0.txt : 'a'



ImageData26.txt : 'A'



ImageData52.txt : '0'

OCR의 기본은 인식하기 위한 기본 이미지 데이터를 먼저 만들고 난 후에 인식을 위한 사진을 읽고 데이터를 비교하여 출력하면 된다. 이번 장에서는 이미지 데이터를 얻는 방법을 구현하였는데, 이후에 진행하는 과정에 필요한 기본적인 개념이며, 이후에도 여기서 사용하는 코드를 반복 사용할 것이다. 또한, 기본 이미지 데이터를 만들고 난 이후에는 파일에서 관리하고 프로그램이 실행될 때 자동으로 파일에서 기본 이미지 데이터를 읽으면 된다.

2.5 기본 이미지 데이터 불러오기

지금까지 사진에서 문자를 추출Image Parsing하고 문자 이미지 데이터를 만드는 방법을 알아보았다. 이제는 문자 인식 프로그램을 실행할 때 저장해 둔 기본 이미지 데

이터를 파일에서 읽은 후 사용한다. 그렇다면 지금까지 구현한 기본 이미지 데이터를 파일에 저장하고 불러오는 방법을 알아보자.

[코드 2-18]은 기본 문자를 가지고 있는 이미지에서 기본 문자 데이터를 만들고, 순서에 따라 해당 문자로 인식한 후에 'standard.txt' 파일에 저장하도록 구현한 것이다. 또한, 이후에 추가되는 함수는 'standard.txt' 파일에서 기본 이미지 데이터를 읽어서 기본 데이터 구조체에 저장한 후, 확인을 위해 'standardout.txt' 파일에 다시 기록하는 것을 구현하였다. 이것은 함수들이 올바르게 구현되었는지를 위한 테스트라고 보면 된다. Text 파일로 기본 이미지 데이터 파일을 가지고 있는 것보다는 Binary 파일로 기본 이미지 데이터를 저장하는 것이 크기와 속도 측면에서 조금 더 나은 방법이라 할 수 있는데, Text 파일을 만드는 법을 이해한 이후에 Binary 파일에 저장하는 것도 구현하였다.

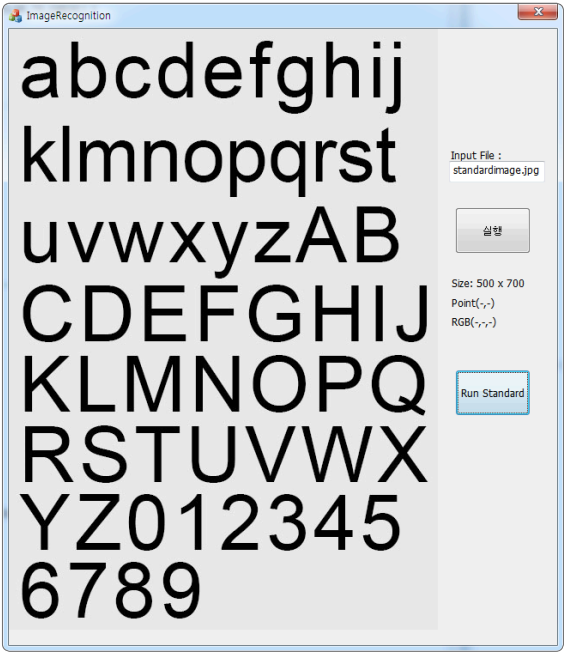
OCR 프로그램의 원리를 요약하면 다음과 같다.

1. 기본 이미지 데이터를 불러온다.
2. 인식을 위한 이미지로부터 문자 이미지를 파싱한다.
3. 파싱한 데이터를 이미지 데이터로 만든다.
4. 기본 이미지 데이터와 인식하기 위한 이미지 데이터를 비교한다.
5. 확률이 가장 높은 기본 이미지 데이터의 문자가 결과값이 된다.

이번 절에서는 기본 이미지 데이터를 만들고 불러오는 과정을 구현하며, 3장부터는 문서 사진에서 문자를 판독하는 방법을 구현한다.

[그림 2-21]은 기본 이미지 데이터를 만들기 위한 이미지로, a~z, A~Z, 0~9의 62개 문자와 숫자가 차례대로 정렬되어 있다.

그림 2-21 기본 이미지 데이터를 만들기 위한 이미지(standardimage.jpg)



[코드 2-18]에서 ①과 ②는 지금까지의 기본 이미지 데이터를 만들기 위한 이미지 파싱과 이미지 데이터를 만드는 함수를 실행한 것이다. 새롭게 봐야 할 부분은 이미지 데이터의 문자를 결정하는 ③부터다. 기본 이미지 데이터의 Letter 구조체는 하나의 결정된 문자를 value 값에 저장한다.

[코드 2-18] 기본 이미지 데이터를 만들어 텍스트 파일에 저장하고 불러오는 함수(OCR.cpp)

```
void COCR::CreateStandard(CImage *newImage) {

    image = newImage;
    colorToCheck = 50;

    ParsingStepFirst();                                    //---①
    MakeImageData();                                       //---②

    int i = 0;
    for (i=0; i<26; i++)
        allData.data[i].letter.value = 'a' + i;            //---③
    for (i=0; i<26; i++)
```

```
        allData.data[i+26].letter.value = 'A' + i;          //---④
    for (i=0; i<10; i++)
        allData.data[i+52].letter.value = '0' + i;          //---⑤

    PrintEveryImageDataInTextFile("standard.txt");          //---⑥
    GetStandardImageDataFromTextFile("standard.txt");       //---⑦

    PrintAllStandardImageToTextFile("standardout.txt");     //---⑧
}
```

① 기본 이미지 데이터를 만들기 위해 'standardimage.jpg' 파일에서 문자 이미지를 파싱하는데, 이 함수는 이전에 구현되었다.

② 이미지 파싱된 문자 이미지를 Letter 구조체의 이미지 데이터에 저장한다.

③ a~z까지 처음 26개의 문자를 기본 이미지 데이터인 Letter 구조체의 value에 넣는다. 기본 이미지 데이터의 소문자 알파벳인 26개 문자의 값을 결정한다.

④ A~Z까지 중간 26개의 문자를 기본 이미지 데이터인 Letter 구조체의 value에 넣는다. 기본 이미지 데이터의 대문자 알파벳인 26개 문자의 값을 결정한다.

⑤ 0~9까지 마지막 10개의 숫자 값을 기본 이미지 데이터인 Letter 구조체의 value에 넣는다.

⑥ 테스트를 위해 'standard.txt' 파일에 기본 이미지 데이터의 value 값과 이미지 데이터를 출력하여 시각적으로 확인한다.

⑦ 'standard.txt'에 저장된 기본 이미지 데이터를 불러와서 기본 이미지 데이터의 구조체인 Standard에 넣는다.

⑧ 테스트를 위해 Standard 구조체에 있는 기본 이미지 데이터를 다시 'standardout.txt' 파일에 저장한다. 'standard.txt' 파일과 'standardout.txt' 파일의 내용이 같으면 모든 함수들이 정상적으로 동작한 것이다.

이제부터 [코드 2-18]에서 구현된 세 개의 함수를 차례대로 구현하는데, 순서는 다음과 같다.

- [코드 2-19] PrintEveryImageDataInTextFile( ) : 기본 이미지 데이터를 텍스트 파일에 저장
- [코드 2-20] GetStandardImageDataFromTextFile( ) : 파일로부터 Standard 구조체 설정

• [코드 2-21] PrintAllStandardImageToTextFile( ) : Standard 구조체의 내용을 텍스트 파일에 저장

[코드 2-19] 기본 이미지 데이터를 텍스트 파일에 저장(OCR.cpp)

```
void COCR::PrintEveryImageDataInTextFile(char * fileName) {

    FILE *fp;
    fp = fopen(fileName, "wt");                    //---①

    int i, x, y;
    unsigned int buffer;                           //---②
    Letter *letter;                                //---③

    fprintf(fp, "%d\n", allData.count);            //---④

    for (i=0; i<allData.count; i++) {              //---⑤
        letter = &allData.data[i].letter;          //---⑥

        fprintf(fp, "\n%c\n", letter->value);      //---⑦

        for (y=0; y<48; y++) {                     //---⑧

            buffer = letter->image[y];             //---⑨

            for (x=0; x<32; x++) {                 //---⑩

                if (buffer & 0x80000000)           //---⑪
                    fputc('1', fp);
                else
                    fputc('0', fp);

                buffer <<= 1;                      //---⑫
            }
            fputc('\n', fp);                       //---⑬
        }
    }

    fclose(fp);                                    //---⑭
}
```

① 기본 이미지 데이터를 저장하기 위한 파일을 쓰기Write 모드(w)와 텍스트 모드(t)로 오픈한다.[10]

10 파일 입출력에 대한 내용은 『MFC 프로그래밍 : 주식 분석 프로그램 만들기』(한빛미디어, 2014) '2.2 File Data 읽기'를 참고하기 바란다.

② '32×48'의 이미지 데이터 크기에서 32-bits 단위로 데이터를 처리하기 위해 buffer 변수를 사용한다. 'unsigned int'로 buffer의 데이터 타입을 정한 것은 shift 연산을 수행할 때 가장 왼쪽의 부호 bit를 사용하지 않기 위해서다.

③ Letter 구조체를 가리키는 포인터 변수 letter를 사용하여 Letter 구조체를 차례대로 접근한다. 이와 같은 방법을 사용하는 것은 ⑥과 같이 Letter 구조체를 처음에 가리키고 난 이후에 letter 변수만을 사용하면 코드가 간결해지기 때문이다.

④ 모든 이미지 데이터의 개수를 파일의 가장 위에 기록한다. 이후에 파일로부터 데이터를 읽을 때는 데이터의 개수를 알기 때문에 모든 데이터를 불러오는 것이 용이하다.

⑤ 모든 이미지 데이터를 for 문으로 반복적으로 접근하면서 처리한다.

⑥ 포인터 변수 letter는 차례대로 모든 이미지 데이터의 Letter 구조체를 가리키며 처리한다.

⑦ 특정 이미지 데이터의 값Value을 파일에 기록한다.

⑧ 32×48 크기의 이미지 데이터를 48번 반복하여 32-bits씩 데이터를 처리한다. 'unsigned int'의 크기가 32-bits임을 이해하면 된다.

⑨ bit 연산을 정수형 변수인 buffer를 사용하여 차례대로 32-bits씩 buffer에 담아서 각각의 bit를 확인한다.

⑩ 정수에 담긴 32-bits를 for 문으로 32번 반복하여 개별 bit로 접근하여 처리한다.

⑪ buffer의 가장 왼쪽 bit를 bit 연산자 '&'를 사용하여 해당 bit가 1이면 '1'을 파일에 출력하고 1이 아니면 '0'을 파일에 출력한다.

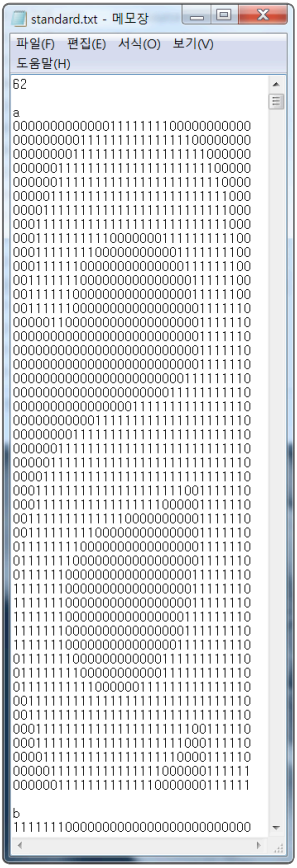
⑫ Shift 연산자를 사용하여 buffer의 모든 bit를 왼쪽으로 하나씩 움직이게 한다.

⑬ 32-bits의 확인이 끝났으면 줄을 바꾼다.

⑭ 파일에 모든 출력이 끝났으므로 파일을 Close하고 저장한다(『MFC 프로그래밍 : 주식 분석 프로그램 만들기』 '2.2 File Data 읽기' 참고).

[코드 2-19]를 실행하면 'standard.txt' 파일이 만들어진다. [그림 2-22]와 같이 가장 위에는 기본 이미지 데이터의 총 개수가 나타나고, 이후에 62개의 Letter 구조체의 값과 '1'과 '0'으로 이루어진 이미지 데이터 모양이 저장된다.

그림 2-22 텍스트 모드로 저장된 기본 이미지 데이터 파일(standard.txt)



[코드 2-20]은 OCR 프로그램이 실행될 때 가장 먼저 수행되는 기능으로, 텍스트 파일에 저장된 기본 이미지 데이터를 Standard 구조체에 저장하는 것을 구현하였다.

[코드 2-20] 텍스트 파일에 저장된 기본 이미지 데이터를 Standard 구조체에 넣기(OCR.cpp)

```
void COCR::GetStandardImageDataFromTextFile(char * fileName) {

    FILE *fp;
    fp = fopen(fileName, "rt");                    //---①

    int i, x, y, temp;
```

```
    char ch;
    unsigned int buffer;                          //---②
    Letter *letter;                               //---③

    fscanf(fp, "%d\n", &standard.count);          //---④

    for (i=0; i<standard.count; i++) {            //---⑤

        letter = &standard.letter[i];             //---⑥

        fscanf(fp, "\n%c\n", &letter->value);     //---⑦

        for (y=0; y<48; y++) {                    //---⑧

            letter->image[y] = 0x00000000;        //---⑨

            for (x=0; x<32; x++) {                //---⑩
                if ((ch = fgetc(fp)) == '1') {    //---⑪
                    buffer = 0x80000000;
                    buffer >>= x;
                    letter->image[y] |= buffer;
                }
            }
            ch = fgetc(fp);                       //---⑫
        }
    }

    fclose(fp);
}
```

① 기본 이미지 데이터를 가져오는 파일을 읽기Read 모드(r)와 텍스트 모드(t)로 오픈한다.

② 32-bits의 정수형 크기의 buffer 변수를 사용하여 이후에 '1'을 파일에서 읽으면 해당 위치의 bit를 1로 저장하는 연산을 수행한다.

③ 포인터 변수인 letter를 사용하여 Standard 구조체의 모든 Letter 구조체를 차례대로 가리킨다.

④ 기본 이미지 데이터의 총 개수를 저장하는 변수인 standard.count에 파일의 첫 데이터인 개수를 저장한다.

⑤ 모든 기본 데이터 이미지를 차례대로 접근하기 위해 for 문을 사용한다.

⑥ 포인터 변수인 letter가 모든 기본 이미지 데이터를 차례대로 접근하며 이후에 letter 변수를 사용하여 코드의 간결성을 유지한다.

⑦ 파일에서 문자를 읽어서 Letter 구조체의 value 변수에 저장한다.

⑧ 32×48 크기의 이미지 데이터를 for 문을 48번 반복하여 차례대로 데이터에 접근한다.

⑨ 하나의 기본 이미지 데이터의 32-bits 값을 0으로 초기화한다. 이후에 파일에서 '1'로 설정된 위치의 bit를 1로 설정하기만 하면 기본 이미지 데이터가 만들어진다.

⑩ 파일에 저장된 기본 이미지 데이터의 한 줄은 32개의 '0' 또는 '1'의 숫자로 이루어졌으며, for 문을 32번 반복하여 차례대로 접근한다.

⑪ 파일에 저장된 값이 '1'이면 해당 위치의 bit를 1로 설정한다. 임시 저장 변수인 buffer를 가장 왼쪽 bit만 1로 설정한 후에 Shift 연산자(>>)로 1인 bit의 위치를 변경한다. 그 후에 OR bit 연산자(|)로 해당 이미지 데이터의 bit를 1로 설정한다.

⑫ 줄바꿈 문자인 '\n'를 읽어서 다음 줄을 읽기 위한 데이터로 파일 포인터를 변경한다.

'standard.txt' 파일에서 데이터를 읽은 후 Standard 구조체를 만들었다면 OCR 프로그램의 시작은 준비되었다고 볼 수 있다. 프로그램을 만들 때는 항상 테스트를 통하여 프로그램이 정상적으로 만들어졌는지를 확인하는 것이 좋다. 테스트를 하지 않은 채 프로그램이 잘 만들어졌다고 생각하는 것은 큰 위험을 안고 가는 것이다. 이전에 구현한 것을 테스트를 통하여 다시 한 번 확인하는 것은 추후에 발생하는 에러로 인한 시간적 손실을 줄일 수 있다.

[코드 2-21]은 지금까지의 코드를 테스트하기 위해 구현한 함수다. Standard 구조체가 정확히 데이터를 갖고 있는지를 확인하기 위한 테스트다. [코드 2-21]은 [코드 2-19]와 거의 모든 부분이 같으며, 단지 AllData 구조체를 Standard 구조체로 접근하는 것만 다르기 때문에 중복된 설명은 생략한다.

[코드 2-21] Standard 구조체의 데이터를 텍스트 파일에 저장(OCR.cpp)

```
void COCR::PrintAllStandardImageToTextFile(char * fileName) {
    FILE *fp;
    fp = fopen(fileName, "wt");

    int i, x, y;
    unsigned int buffer;
    Letter *letter;

    fprintf(fp, "%d\n", standard.count);                    //---①

    for (i=0; i<standard.count; i++) {                      //---②
```

```
        letter = &standard.letter[i];                    //---③

        fprintf(fp, "\n%c\n", letter->value);

        for (y=0; y<48; y++) {

            buffer = letter->image[y];

            for (int x=0; x<32; x++) {

                if (buffer & 0x80000000)
                    fputc('1', fp);
                else
                    fputc('0', fp);

                buffer <<= 1;
            }
            fputc('\n', fp);
        }
    }

    fclose(fp);
}
```

① 파일의 가장 처음에 기본 이미지 데이터의 총 개수(standard.count)를 기록한다.

② for 문을 사용하여 모든 기본 이미지 데이터로 접근한다.

③ 포인터 변수 letter로 기본 이미지 데이터 구조체인 Letter 구조체에 접근한다.

이를 실행하여 'standardout.txt' 파일이 만들어지고 'standard.txt' 파일과 동일한 내용을 가지고 있다면 Standard 구조체는 정상적으로 만들어진 것이다. 텍스트 모드로 'standard.txt' 파일을 만든 목적은 시각적으로 확인하기 위해서다. 'standard.txt' 파일의 크기를 줄일 수 있다면 속도 측면에서 조금 더 빠르게 프로그램이 동작할 것이다. 물론 현재의 컴퓨터는 빠르게 연산을 수행하기 때문에 프로그램의 동작 시간에 대한 체감은 크지 않을 것이다. 하지만 프로그램의 크기를 조금 더 줄이고, 속도가 조금 더 빠르게 프로그램을 만드는 것은 프로그래머의 관점에서 추구해야 할 부분이다. 앞에서 언급한 적이 있듯이 Text 모드를 조금 더 빠르게 수행하는 방법은 Binary 모드로 파일을 바꾸는 방법이다.

[코드 2-22]부터 구현하는 방법은 Binary로 기본 이미지 데이터를 파일에 기록하고 읽는다. 파일의 크기가 약 1/4로 작아지고, 코드의 양도 줄고 상당히 효율적인 방법이다. 또한, 상용화 프로그램을 만든다면 Binary로 변경하는 것이 좋다. [코드 2-22]는 [코드 2-18]과 거의 모든 부분이 동일하며, 단지 Binary 파일에 저장하고 읽는 두 개의 함수만 다르므로 중복된 부분은 설명을 생략한다.

[코드 2-22] 기본 이미지 데이터를 만들고 Binary 파일에 저장하고 불러오는 함수(OCR.cpp)

```
void COCR::CreateStandard(CImage *newImage) {
    image = newImage;
    colorToCheck = 50;

    ParsingStepFirst();
    MakeImageData();

    int i = 0;
    for (i=0; i<26; i++)
        allData.data[i].letter.value = 'a' + i;
    for (i=0; i<26; i++)
        allData.data[i+26].letter.value = 'A' + i;
    for (i=0; i<10; i++)
        allData.data[i+52].letter.value = '0' + i;

    PrintEveryImageDataInBinaryFile("standard.bin");          //---①
    GetStandardImageDataFromBinaryFile("standard.bin");       //---②

    PrintAllStandardImageToTextFile("standardout.txt");       //---③
}
```

① 기본 이미지 데이터를 Binary 파일에 저장하는 함수다.

② Binary 파일에 저장된 기본 이미지 데이터를 읽은 후 Standard 구조체에 저장하는 함수다.

③ Binary 파일에 저장된 기본 이미지 데이터를 정확히 읽어서 Standard 구조체에 저장했는지를 확인하는 테스트 함수로, [코드 2-21]에서 구현한 함수와 동일하다.

[코드 2-22]에 나온 두 개의 함수를 차례대로 구현하는데, 순서는 다음과 같다.

- [코드 2-23] PrintEveryImageDataInBinaryFile ( ) : 기본 이미지 데이터를 Binary 파일에 저장

- [코드 2-24] GetStandardImageDataFromBinaryFile ( ) : Binary 파일에서 Standard 구조체 설정

[코드 2-23]은 [코드 2-19]와 거의 동일하므로 변경된 부분만을 설명한다.

[코드 2-23] 기본 이미지 데이터를 Binary 파일에 저장(OCR.cpp)

```
void COCR::PrintEveryImageDataInBinaryFile(char * fileName) {
    FILE *fp;
    fp = fopen(fileName, "wb");                          //---①

    int i, x, y;
    Letter *letter;

    fprintf(fp, "%d\n", allData.count);

    for (i=0; i<allData.count; i++) {

        letter = &allData.data[i].letter;

        fprintf(fp, "%c\n", letter->value);

        for (y=0; y<48; y++)
            fprintf(fp, "%d\n", letter->image[y]);       //---②
    }

    fclose(fp);
}
```

① 기본 이미지 데이터를 저장하는 파일을 쓰기 모드(w)와 바이너리Binary 모드(b)로 오픈한다.

② 32-bits의 데이터는 정수값을 의미하므로 개별 bit로 저장하는 것이 아니라 정수형 값으로 한 번에 저장하면 된다. 즉, 하나의 데이터 이미지는 48개의 정수형 값으로 Binary에 저장되며, 줄바꿈 문자인 '\n'도 Binary에 저장된다. 코드의 양이 상당히 줄어들고 속도도 크게 향상한다.

[그림 2-23]은 'standard.bin' 파일에 저장된 기본 이미지 데이터다. 시각적으로 어떤 문자의 이미지 데이터인지 알 수는 없다. 하지만 [그림 2-24]에서 볼 수 있듯이 Binary로 저장된 파일(standard.bin)의 크기가 텍스트로 저장된 파일(standard.txt) 크기의 26%로 줄어들었다.

그림 2-23 Binary 파일에 저장된 기본 이미지 데이터(standard.bin)

그림 2-24 기본 이미지 데이터 파일의 크기 비교(Binary와 텍스트)

[코드 2-24]는 Binary 파일에서 데이터를 읽어서 Standard 구조체를 만드는 방법을 구현한 것으로, [코드 2-20]과 거의 동일하기 때문에 변경된 부분만을 설명한다.

[코드 2-24] Binary 파일에 저장된 기본 이미지 데이터를 Standard 구조체에 넣기(OCR.cpp)

```
void COCR::GetStandardImageDataFromBinaryFile(char * fileName) {
    FILE *fp;
    fp = fopen(fileName, "rb");                    //---①

    int i, x, y;
    Letter *letter;

    fscanf(fp, "%d\n", &standard.count);

    for (i=0; i<standard.count; i++) {

        letter = &standard.letter[i];

        fscanf(fp, "%c\n", &letter->value);

        for (y=0; y<48; y++)
```

```
            fscanf(fp, "%d\n", &letter->image[y]);          //---②
    }

    fclose(fp);
}
```

① 기본 이미지 데이터를 가져오는 파일을 읽기 모드(r)와 바이너리 모드(b)로 오픈한다.

② 32-bits의 데이터는 정수값을 의미하기 때문에 개별 bit로 읽는 것이 정수형 값으로 한 번에 읽어 오는 것이다. 즉, 하나의 데이터 이미지는 48개의 정수형 값으로 Binary로 기록되었기 때문에 for 문을 48번 반복하여 48개의 정수를 읽어 온다. 프로그램 코드의 양이 상당히 줄어들고 속도도 크게 향상한다.

이번 장에서는 기본 이미지 데이터를 만드는 것을 구현하였다. 기본 이미지 데이터는 62개의 기본적인 문자만으로 구현하였기 때문에 실제 OCR 프로그램에서는 추가적인 특수 문자가 많이 필요하다.

3장에서는 문서 이미지에서 문자를 판독하는 방법을 구현하는데, 단계적으로 추가적인 부분을 구현하면서 문서 이미지의 OCR을 알아볼 것이다. 프로그램을 구현하면서 한 번에 좋은 프로그램을 완성한다는 것은 실제로 불가능하다. 기본적인 것을 구현한 후에 문제점Bug을 하나씩 풀어 나가면서 조금 더 좋은 프로그램을 만들어 가는 작업이 프로그래밍이라고 이해하는 것이 좋다. 이 세상에 존재하는 프로그램의 대부분은 발견되지 않은 문제점을 가지고 있고, 그래서 상용화된 프로그램도 끊임없는 Patch 작업을 한다. Patch 작업을 통하여 단계적으로 더 좋은 프로그램이 되어 간다고 이해하자.

chapter 3

문서 이미지 OCR

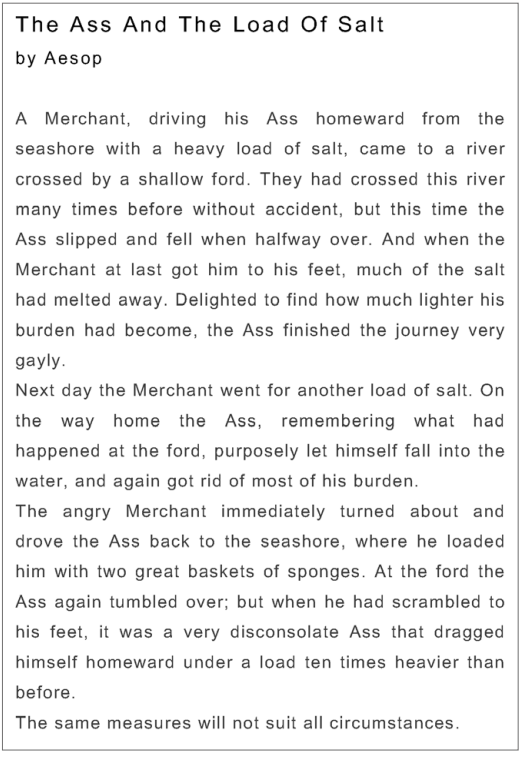
이번 장에서는 문서 이미지에서 문자를 판독해 내는 OCR 프로그램을 단계적으로 구현한다. 이전 장에서 기본적인 부분은 구현되었으나, OCR 프로그램의 다양한 부분에 대한 깊이 있는 분석과 추가적으로 구현해야 할 부분이 상당히 많다. 기본적인 내용으로 시작하지만, 내용이 진행됨에 따라 OCR에서 생각해야 할 부분을 조금 더 구체적으로 알 수 있으며, 이번 장이 끝나더라도 독자가 접하는 문서에 따라 추가적인 부분을 직접 구현할 수 있는 힘을 기르기를 바란다.

이 장에서 사용하는 문서 이미지는 '이솝 이야기'의 내용으로 [그림 3-1]과 [그림 3-2]의 두 개의 파일이다. 먼저 프로그램에서는 [그림 3-1]의 첫 번째 내용만을 가지고 OCR 프로그램이 구현되며, 모든 구현이 끝난 이후에는 [그림 3-2]의 내용도 OCR을 진행하여 프로그램의 완성도를 검증할 예정이다.

실생활에서도 문서 이미지 OCR을 많이 사용하고 있는데, 대표적으로 고급 복사기에서 많이 사용한다. 즉, OCR 기능을 가진 고급 복사기에서는 문서를 스캔하면 Word 파일로 결과가 만들어진다. 복사기는 문서를 스캔하여 문서 이미지를 얻으며, 복사기에 설치된 자체 프로그램이 문서 이미지를 판독하여 Word 문서 파일로 다시 만들어 준다. 이 기술이 발전하여 번역 프로그램과 결합할 경우를 생각해 보자. 책을 복사기로 복사하면 각 나라의 언어로 번역된 책이 자동적으로 만들어질 수도 있을 것이다. 아직까지는 번역 프로그램의 완성도가 아주 낮기 때문에 자동 번역에 대한 기능을 단시간에 만들기는 힘들다. 그러나 OCR가 자동 번역기와 함께하는 순간 우리 생활에 엄청난 변화를 가져다줄 것이다.

OCR 기술과 관련하여 생각해 볼 수 있는 또 한 가지는 문서를 읽어 주는 프로그램과의 결합이다. 복사기에서 책을 복사하면 성우가 책을 읽어 주는 소리 파일이 만들어진다. 이 기술은 OCR로 인식이 끝난 문서 파일에 인간의 음성으로 읽어 주는 기술을 접목하면 된다. 인간의 음성으로 읽어 주는 기술은 이미 시장에서 찾아볼 수 있으나 기술이 완벽하지는 않다. 이는 두 말이 이어져 발음되는 연음 처리에 대한 기술이 아직 완벽하지 않기 때문이다. 어쨌든, OCR 기술이 번역, 낭독 등의 기술과 함께할 때 아주 멋진 생활의 편리함을 가져올 것임은 분명하다.

그림 3-1 이솝 우화 문서 이미지(aesop_1.png: The Ass And The Load Of Salt)



The Ass And The Load Of Salt

by Aesop

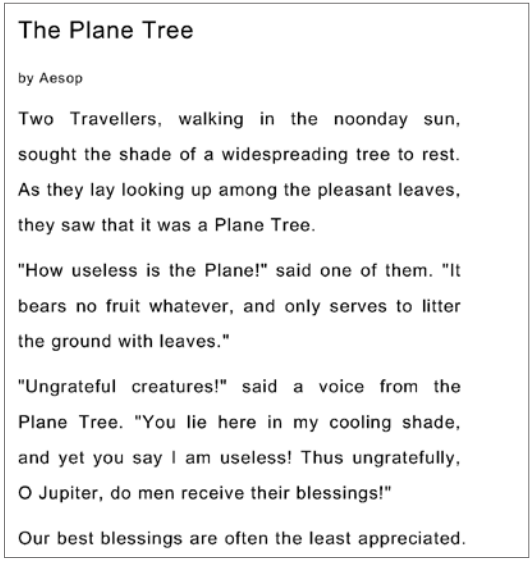
A Merchant, driving his Ass homeward from the seashore with a heavy load of salt, came to a river crossed by a shallow ford. They had crossed this river many times before without accident, but this time the Ass slipped and fell when halfway over. And when the Merchant at last got him to his feet, much of the salt had melted away. Delighted to find how much lighter his burden had become, the Ass finished the journey very gayly.

Next day the Merchant went for another load of salt. On the way home the Ass, remembering what had happened at the ford, purposely let himself fall into the water, and again got rid of most of his burden.

The angry Merchant immediately turned about and drove the Ass back to the seashore, where he loaded him with two great baskets of sponges. At the ford the Ass again tumbled over; but when he had scrambled to his feet, it was a very disconsolate Ass that dragged himself homeward under a load ten times heavier than before.

The same measures will not suit all circumstances.

그림 3-2 이솝 우화 문서 이미지 (aesop_2.png: The Plane Tree)



The Plane Tree

by Aesop

Two Travellers, walking in the noonday sun, sought the shade of a widespreading tree to rest. As they lay looking up among the pleasant leaves, they saw that it was a Plane Tree.

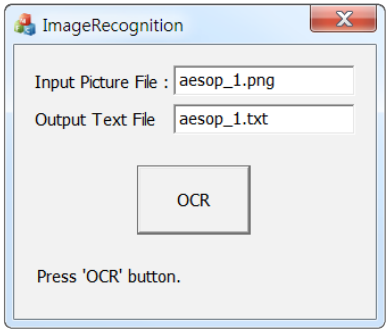
"How useless is the Plane!" said one of them. "It bears no fruit whatever, and only serves to litter the ground with leaves."

"Ungrateful creatures!" said a voice from the Plane Tree. "You lie here in my cooling shade, and yet you say I am useless! Thus ungratefully, O Jupiter, do men receive their blessings!"

Our best blessings are often the least appreciated.

[그림 3-3]은 이번 장에서 구현되는 프로그램의 실행 화면이다. 파일의 이름을 입력하고 'OCR' 버튼만 누르면 실행되는 프로그램으로, 상당히 간단하다. 'Input Picture File'에는 인식을 위한 문서 이미지 파일 이름을 넣고, 'Output Text File'에는 인식이 끝난 후의 결과 문자들을 입력하기 위한 파일 이름을 넣는다. 'OCR' 버튼을 누르게 되면 이후에 진행되는 모든 OCR 관련 프로그램 함수들이 실행된다.

그림 3-3 문서 이미지 OCR 프로그램 실행 화면



[코드 3-1]은 [그림 3-3]의 [OCR] 버튼을 눌렀을 때 실행되는 함수로, 'ImageRecognitionDlg.cpp' 파일에 있는 OnBnClickedBtnRunocr( ) 함수다.[01] ImageRecognitionDlg.cpp 파일에서는 버튼을 눌렀을 때 실행되는 [코드 3-1]의 내용이 가장 중요하며 OCR에서 중심이 되는 함수들은 'OCR.h'와 'OCR.cpp'에서 구현한다.

[코드 3-1] OCR 버튼을 눌렀을 때 실행되는 함수(ImageRecognitionDlg.cpp)

```
void CImageRecognitionDlg::OnBnClickedBtnRunocr()
{
    UpdateData(TRUE);

    m_message.Format(_T("OCR is Running..."));                    //---①
    UpdateData(FALSE);

    if (image != NULL)
        image.Destroy();

    HRESULT hResult = image.Load(m_inputfile);                    //---②

    if (FAILED(hResult)) {
        m_message.Format(_T("Error : Can't Open Input File."));
        UpdateData(FALSE);
        return;
    }

    ocr->RunOCR(&image, m_outputfile, 50);                        //---③

    m_message.Format(_T("OCR is Completed..."));                  //---④
    UpdateData(FALSE);
}
```

① 실행 화면 하단의 'Press 'OCR' button.' 메시지가 'OCR is Running…'으로 바뀐다. UpdateData(TRUE)는 화면의 내용을 프로그램의 각 멤버 변수 내용으로 업데이트하고, UpdateData(FALSE)는 멤버 변수의 내용이 화면의 내용을 바꾸기 때문에 m_message의 내용이 화면 하단의 메시지를 바꾸게 된다.

② 문서 이미지 파일의 이름을 저장하고 있는 m_inputfile 변수를 사용하여 이미지를 로

01 MFC 프로그램에서 버튼 등을 포함한 도구와 해당 도구와 연동되는 멤버 변수의 추가에 대해서는 『MFC 프로그래밍 : 주식 분석 프로그램 만들기』(한빛미디어, 2014)에서 구체적으로 설명하였으므로 여기서는 설명을 생략한다.

드한다. 즉, 문서 이미지가 CImage 클래스 변수인 image에 저장된다.

③ COCR 클래스의 멤버 함수인 RunOCR을 실행해 OCR을 진행한다. 이때 전달되는 파라미터는 이미지를 담고 있는 image, 출력 파일 이름인 m_outputfile, RGB에서 검은색(문자 색)을 나타내는 50이다.

④ OCR이 실행되면 화면 하단에 'OCR is Completed…' 메시지로 종료를 알려 준다.

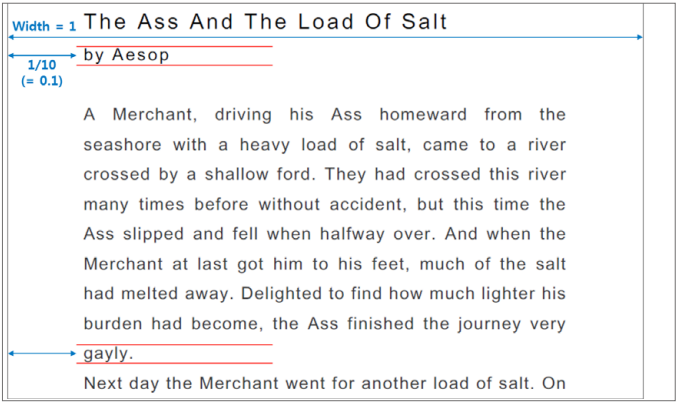
Dialog 파일에서의 구현은 이처럼 간단하다. 이후에는 확률을 이용한 OCR의 기본 개념부터 OCR에 추가로 구현해야 하는 내용을 단계적으로 알아본다.

3.1 확률을 이용한 OCR

이전까지 기본 이미지 데이터를 만드는 방법을 알아보았다. 이것은 사전이 만들어진 것으로 이해하면 된다. 지금부터 진행되는 OCR의 실행 단계는 하나의 문자를 이미지 파싱을 거쳐서 이미지 데이터를 만든 후, 사전과 같은 기본 이미지 데이터와 비교하여 특정 문자로 결정하는 단계다. 3장에서는 처음부터 단계적으로 OCR을 진행하게 되는데, 먼저 확률을 이용한 OCR이 어떻게 이루어지는지 알아볼 것이다. 이것이 OCR의 가장 기본이 되는 기술이다. 그러나 확률을 이용해서 문자를 결정하는 방법만으로는 인식의 한계를 알게 될 것이다. 기본 62개의 문자보다 더 많은 특수 문자가 있으며, 줄바꿈Enter과 공백Space 등의 부분도 구현해야 하기 때문이다.

[그림 3-4]는 이번 장에서 인식할 문서 이미지인데, 이미지 파싱을 어디서부터 해야 하는지를 보기 위한 그림이다. 문서 이미지에서는 이미지 파싱이 중간이 아닌 앞쪽에서 이루어져야 한다. 심한 경우에는 한 줄에서 앞쪽에 짧은 한 단어만이 존재할 수 있기 때문이다. 프로그램에서는 이 부분을 문서의 가로Width 크기에서 비율Rate로 비교하여 앞쪽에서 0.1부터 시작하여 0.3까지 픽셀을 검사하여 한 줄을 인식하는 것으로 구현하였다.

그림 3-4 이미지 파싱의 기준



[코드 3-2]는 COCR 클래스의 헤더header 파일로, 이미지 파싱을 위해 시작되고 끝나는 가로의 비율을 처음에 0.1과 0.3으로 정의하였다. 이후에 구현되는 함수를 추가로 클래스의 멤버 함수로 정의하였다. 이는 2장에서 사용한 내용과 거의 유사하며 확률을 이용하여 특정 문자로 인식하는 함수를 추가로 구현하였다.

[코드 3-2] COCR 클래스 멤버 함수(OCR.h)

```
#define RATE_START_FOR_PARSING    0.1                                    //---①
#define RATE_END_FOR_PARSING      0.3                                    //---②

##### 생략 #####

class COCR {

private:
    CImage *image;
    Standard standard;                                                   //---③
    AllData allData;
    Data *data;

    int colorToCheck;

public:
    COCR(void);
    ~COCR(void);

    void RunOCR(CImage *image, CString outFileName, int colorLetter);//---④
    void GetStandardImageDataFromBinaryFile(char * fileName);          //---⑤
```

```
    void ParsingStepFirst();
    void ParsingStepSecond(int yTop, int yBottom);
    void ParsingStepThird(Rect *rect);
    void ParsingStepThird2(Rect *rect);
    void MakeImageData();
    void FindLetterValue();                                    //---⑥
    void StoreLetterToTextFile(CString outFileName);           //---⑦
};
```

① RATE_START_FOR_PARSING을 0.1로 정의한다. 상수 변수의 이름에서 알 수 있듯이 이미지 파싱을 위한 시작 비율을 나타낸다([그림 3-3] 참조).

② RATE_END_FOR_PARSING은 0.3으로 정의하며 이미지 파싱을 위한 마지막 부분의 비율을 나타낸다.

③ 기본 이미지 데이터를 저장하는 standard 변수를 정의한다. 나머지 변수는 2장과 동일한 목적으로 사용되었다.

④ OCR을 실행하는 함수로, 매개변수로 이미지 클래스 변수인 image, 출력 파일의 이름, 문자로 인식하기 위한 색의 값을 가진다.

⑤ 프로그램이 실행되면 자동으로 COCR 클래스가 생성되고, 생성 클래스에서 자동으로 GetStandardImageDataFromBinaryFile( ) 함수를 실행한다. 즉, 프로그램이 실행될 때 바이너리 파일로부터 기본 이미지 데이터를 읽어서 Standard 구조체에 넣는다.

⑥ 확률을 이용하여 Standard 구조체에 저장된 값과 비교하여 인식하려는 문자의 결과값을 얻게 된다.

⑦ 인식된 이미지의 결과들을 outFileName의 파일 이름에 텍스트 형태로 저장한다.

[코드 3-3]은 COCR의 생성자와 프로그램 실행 화면에서 [OCR] 버튼을 선택할 때 실행되는 함수를 구현한 것이다. 인식하려는 문서에서 이미지 파싱을 진행한 후 각각의 문자를 이미지 데이터로 만든다. 그다음 기본 이미지 데이터와 비교하여 특정 문자로 인식하고 결과 파일에 저장한다.

[코드 3-3] COCR 생성자 및 OCR 실행 함수 RunOCR(OCR.cpp)

```
COCR::COCR(void)
{
    GetStandardImageDataFromBinaryFile("standard.bin");    //---①
}
```

```
COCR::~COCR(void)
{

}

void COCR::RunOCR(CImage *newImage, CString outFileName, int colorLetter) {

    image = newImage;                                    //---②
    colorToCheck = colorLetter;                          //---③

    for (int i=0; i<MAX_COUNT_DATA; i++)
        allData.data[i].isFixed = false;                 //---④

    ParsingStepFirst();                                  //---⑤
    MakeImageData();                                     //---⑥

    FindLetterValue();                                   //---⑦

    StoreLetterToTextFile(outFileName);                  //---⑧
}
```

① 프로그램이 실행될 때 COCR 클래스의 생성자가 실행된다. 바이너리 파일에서 기본 이미지 데이터를 읽어서 기본 데이터 구조체를 만든다.

② 포인터 변수 image로 인식하려는 이미지를 가리킨다. 이후에는 image 변수로 해당 이미지에 접근하게 된다.

③ 문자 색을 colorToCheck의 변수에 저장한다.

④ allData는 인식하기 위해 파싱된 모든 문자를 저장한다. 아직 문자가 정의되지 않았기 때문에 isFixed 변수를 false로 초기화한다. 이후에 문자가 확률을 통하여 특정 문자로 결정되면 isFixed 변수는 true가 된다.

⑤ 이미지 파싱을 진행하는 함수로, 2장에서 구현한 것과 동일하다. 단지, 2장에서는 문자선 가운데를 기준으로 나누었다면, 이번에는 앞쪽을 기준으로 나누었다는 차이가 있다. ParingStepFirst( ) 함수의 설명은 2장에서 했으므로 여기서는 생략한다.

⑥ 이미지 파싱된 문자 이미지를 이미지 데이터에 넣는 함수로, 이 함수도 2장에서 구현한 것과 동일하므로 여기서는 설명을 생략한다.

⑦ 기본 이미지 데이터와 비교하여 확률이 가장 높은 문자를 인식하려는 문자의 결과값으로 정의한다.

⑧ OCR의 인식이 끝난 상태이므로 결과값을 텍스트 파일에 저장한다.

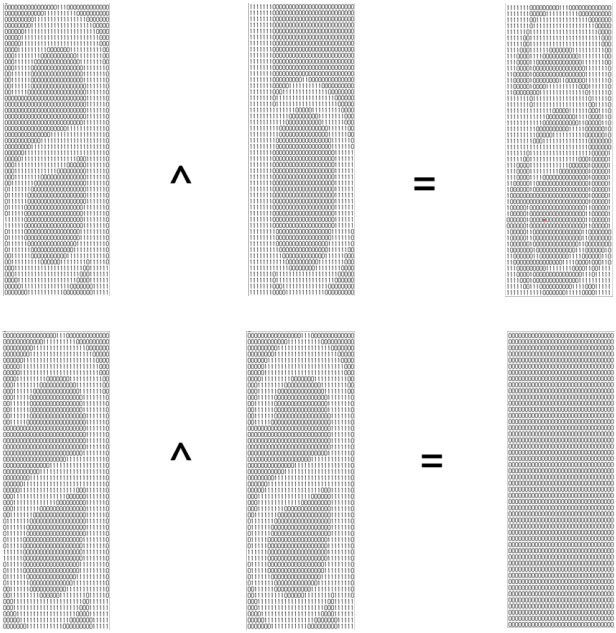
프로그램이 실행되면 자동으로 기본 이미지 데이터를 파일에서 읽어서 Standard 구조체를 만들어야 한다. [코드 3-4]는 기본 이미지 데이터를 만들어 주는 함수다. 즉, 파일을 열어서 메모리에 파일을 넣은 후 차례대로 데이터를 얻어서 Standard 구조체에 넣어 주고 파일을 닫는다. [코드 3-4]의 내용은 [코드 2-24]와 동일하므로 설명은 생략한다.

[코드 3-4] standard.bin 파일로부터 기본 이미지 데이터 생성(OCR.cpp)

```
void COCR::GetStandardImageDataFromBinaryFile(char * fileName) {
    FILE *fp;
    fp = fopen(fileName, "rb");                          //---①

    int i, x, y;
    Letter *letter;

    fscanf(fp, "%d\n", &standard.count);                 //---②

    for (i=0; i<standard.count; i++) {

        letter = &standard.letter[i];                    //---③

        fscanf(fp, "%c\n", &letter->value);              //---④

        for (y=0; y<48; y++)
            fscanf(fp, "%d\n", &letter->image[y]);       //---⑤
    }

    fclose(fp);                                          //---⑥
}
```

확률을 이용한 인식에는 XOR(^) 연산자를 사용한다. 비교하는 두 개의 bit가 같은 값이면 0, 다른 값이면 1의 결과값을 가진다. [그림 3-5]는 1장에서 이미 설명한 두 개의 이미지 데이터를 XOR 연산으로 얻은 결과다. 이미지가 비슷할수록 더 많은 bit가 0의 값을 갖게 된다.

그림 3-5 XOR(^) 연산자를 사용한 확률 계산 - 0이 많을수록 이미지가 더욱 비슷하다



[코드 3-5]는 이미지 가로 한 줄의 32개 bit에서 0의 개수를 알아내는 코드다. 32-Bits 중에서 가장 왼쪽에 있는 한 bit만을 1로 설정한 후에 해당 bit를 순차적으로 오른쪽으로 이동하면서 AND(&) 비트 연산자를 이용하여 0의 개수를 확인한다.

[코드 3-5] 정수형 변수의 32-bits에서 0의 개수를 세는 방법

```
count = 0;
for (x=0; x<32; x++) {              //for 문을 이용하여 32번 반복

    bit = 0x80000000;               //32-bits의 가장 왼쪽 bit를 1로 초기화
    bit >>= x;                      //왼쪽의 1 bit의 위치를 오른쪽으로 이동
    bit = bit & buffer;             //& 연산자를 통하여 buffer의 해당 bit를 확인

    if (!bit)                       //해당 bit가 0이면
        count += 1;                 //count를 1 증가한다.
}
```

[코드 3-6]은 확률을 이용하여 문자를 찾는 방법이다. 이미지 데이터에서 기본 이미지 데이터와 비교하여 0의 값을 갖는 비트의 개수가 많은 것을 선택하여 해당 문자를 결정한다. 이미지 파싱한 모든 문자의 이미지 데이터는 allData에 저장되어 있으며, 각각의 이미지 데이터는 기본 이미지 데이터와 비교한 후, 0의 개수가 가장 많은 기본 이미지 데이터의 문자를 현재 이미지 데이터의 문자로 확정한다.

[코드 3-6] 확률을 이용한 문자 찾기(OCR.cpp)

```
void COCR::FindLetterValue() {
    int count, maxCount;                                          //---①
    unsigned int buffer, bit;                                     //---②
    int i, j, x, y;

    for (i=0; i<allData.count; i++) {
        Data *data = &allData.data[i];                            //---③
        Letter *letter = &data->letter;
        Rect *rect = &data->rect;

        maxCount = 0;                                             //---④

        for (j=0; j<standard.count; j++) {                        //---⑤
            count = 0;

            for (y=0; y<48; y++) {
                buffer = letter->image[y] ^ standard.letter[j].image[y]; //---⑥
                for (x=0; x<32; x++) {                            //---⑦

                    bit = 0x80000000;
                    bit >>= x;
                    bit = bit & buffer;

                    if (!bit)
                        count += 1;
                }
            }

            if (count > maxCount) {                               //---⑧
                letter->value = standard.letter[j].value;
                maxCount = count;
            }
        }
    }
}
```

① count 변수는 비교하는 이미지 데이터와 기본 이미지 데이터에서 XOR(^)를 수행한 후 0의 개수를 세기 위해 사용한다. maxCount는 62개의 기본 이미지 데이터와의 비교에서 0의 개수가 가장 큰 수를 저장하기 위해 사용한다.

② buffer는 이미지 데이터의 한 줄을 의미하는 정수형 값을 저장하는 데 사용하고, bit는 한 bit만을 1로 설정한 후에 buffer에서 특정 위치의 bit 값을 확인하는 데 사용한다.

③ 모든 이미지 데이터는 allData 변수에 저장되고, 포인터 변수인 data는 차례대로 이미지 데이터를 가리키며 이후의 연산이 수행된다.

④ 기본 이미지 데이터와 비교하기 전에 maxCount를 0으로 초기화한다.

⑤ 기본 이미지 데이터의 개수만큼 for 문을 사용하여 처음부터 끝까지 XOR 연산으로 0의 개수를 확인하여 해당 문자를 결정한다.

⑥ 비교하려는 이미지 데이터(letter)와 기본 이미지 데이터(Standard.letter)에 XOR 연산을 수행한 결과를 정수형 변수인 buffer에 저장한다.

⑦ 이미지 데이터의 가로 한 줄은 32-bits의 정수형이며, for 문을 이용하여 32번 반복하여 32개의 모든 bit를 확인한다. bit 값이 0이면 count를 1 증가시킨다.

⑧ 현재 비교한 이미지 데이터의 0의 개수가 최대값인 maxCount보다 크면 현재의 기본 이미지 데이터의 문자로 결정하고, 최대값 maxCount는 현재의 값인 count 값을 가진다.

[코드 3-6]을 실행하여 인식할 이미지의 모든 문자를 결정지었다면, [코드 3-7]을 실행하여 텍스트 파일에 결과를 저장한다.[02]

[코드 3-7] 결과 데이터를 출력 파일에 저장(OCR.cpp)

```
void COCR::StoreLetterToTextFile(CString outFileName) {
    FILE *fp;
    fp = fopen((char*)((LPCSTR)(outFileName)), "wt");          //---①

    for (int i=0; i<allData.count; i++)
        fputc(allData.data[i].letter.value, fp);               //---②

    fclose(fp);
}
```

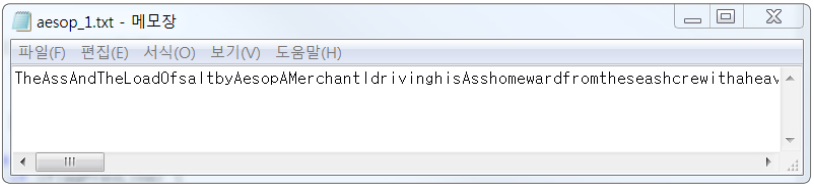
① outFileName 변수의 이름을 가진 파일을 쓰기와 텍스트 모드로 오픈한다.

02 파일 입출력에 대한 내용은 『MFC 프로그래밍 : 주식 분석 프로그램 만들기』(한빛미디어, 2014)의 '2.2 File Data 읽기'에서 자세히 설명하였으므로 참고하기 바란다.

② 모든 이미지 데이터의 문자 값을 파일에 순차적으로 저장한다.

[그림 3-6]은 지금까지 구현한 프로그램을 실행한 결과다. 인식하려는 이미지와 현저히 다르게 모든 문자가 일렬로 연결되었다. 이는 단지 문자만을 인식했을 뿐 줄바꿈과 공백 문자가 없기 때문이다.

그림 3-6 확률을 이용한 OCR



지금까지 기본 이미지 데이터와 비교하는 내용을 구현하였다. 이제 줄바꿈과 공백 등을 추가하여 읽기 쉬운 결과값을 내기 위해 단계적으로 OCR의 기능을 만들어 보자.

3.2 줄바꿈 문자 추가하기

OCR에서 줄바꿈 문자인 '\n'를 추가하는 것은 간단하다. 이미지 파싱을 진행할 때 두 번째 단계에서 한 줄에서의 문자를 나누고 난 이후 마지막에 '\n' 문자의 이미지 데이터를 추가하면 된다. 또한, 줄바꿈을 위해 추가된 이미지 데이터는 확정된 값으로 정하여 이후에 기본 이미지 데이터와 비교하지 않으면 된다. 그래서 Date 구조체에 문자가 확정되었다는 표시를 위해 isFixed 변수를 추가한다. 즉, isFixed가 true면 확률을 이용한 이미지 인식에서 제외한다.

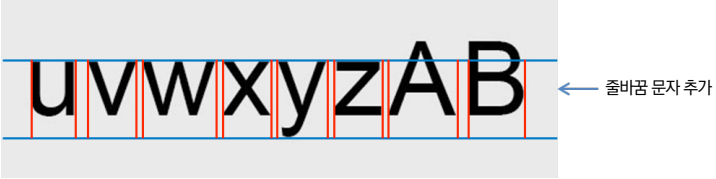
[코드 3-8] Data 구조체에 isFixed 변수 추가(OCR.h)

```
struct Data {
    bool isFixed;                    //---①
    Letter letter;
    Rect rect;
};
```

① 줄바꿈 문자인 '\n'이 구조체에 추가 저장되면 isFixed 값이 true이며, 이외의 확률을 이용한 이미지 데이터의 확인을 위한 구조체는 isFixed 값이 false를 가진다.

[그림 3-7]은 [그림 2-9]와 같으며, 이미지 파싱의 2단계를 진행한 후 마지막에 줄바꿈 문자인 '\n'의 이미지 데이터를 추가한 것이다. 이렇게 줄바꿈 문자가 삽입되면 OCR에서 한 줄마다 줄바꿈이 이루어져 결과 파일에서 인식하려는 이미지와 같은 줄의 개수로 결과 파일에 저장된다.

그림 3-7 이미지 파싱 2단계 - Letter 파싱 마지막에 줄바꿈 문자 삽입



[코드 3-9]는 [코드 2-10]의 이미지 파싱 2단계와 거의 동일한 코드로, 마지막에 줄바꿈 문자의 이미지 데이터를 추가하는 4줄의 코드가 추가되었다.

[코드 3-9] 줄바꿈 문자의 이미지 데이터 삽입(OCR.cpp)

```
void COCR::ParsingStepSecond(int yTop, int yBottom) {
    int xMax = image->GetWidth();

    int x, y;
    COLORREF rgb;
    bool isLetterLine;
    bool flagPrevLine;

    flagPrevLine = false;

    for (x=0 ; x<xMax; x++) {

        isLetterLine = false;

        for (y=yTop; y<=yBottom; y++) {
            rgb = image->GetPixel(x,y);

            if (GetRValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {

                isLetterLine = true;
                break;
            }
```

```
        }

        if (isLetterLine) {
            if (!flagPrevLine) {
                data->rect.start.x = x;
                data->rect.start.y = yTop;
            }
        }
        else {
            if (flagPrevLine) {
                data->rect.end.x = x-1;
                data->rect.end.y = yBottom;

                ParsingStepThird(&data->rect);
                ParsingStepThird2(&data->rect);
                ParsingStepThird(&data->rect);

                allData.count += 1;
                data = &allData.data[allData.count];
            }
        }

        flagPrevLine = isLetterLine;
    }

    data->isFixed = true;                    //---①
    data->letter.value = '\n';               //---②

    allData.count += 1;                      //---③
    data = &allData.data[allData.count];     //---④
}
```

① 현재의 이미지 데이터 구조체에서 isFixed를 true로 설정하여 확률을 이용한 기본 이미지 데이터와의 비교를 진행하지 않는다.

② 현재의 이미지 데이터 구조체에서 문자의 값은 줄바꿈 문자인 '\n'의 값을 가진다.

③ 이미지 데이터 구조체의 개수를 하나 증가시킨다.

④ 현재의 포인터 변수인 data는 다음의 비어 있는 구조체를 가리킨다. 이렇게 하여 줄바꿈 문자를 저장하는 이미지 데이터의 추가가 완료된다.

[코드 3-10]은 확률을 이용한 이미지 데이터를 비교하는 [코드 3-6]과 거의 동일한 코드로, 중간에 이미지 데이터가 확정되었는지를 확인하는 isFixed 값을 비교

하는 조건문 하나가 추가되었다. [코드 3-6]에서 코드에 대한 설명을 하였기에 자세한 설명은 생략한다.

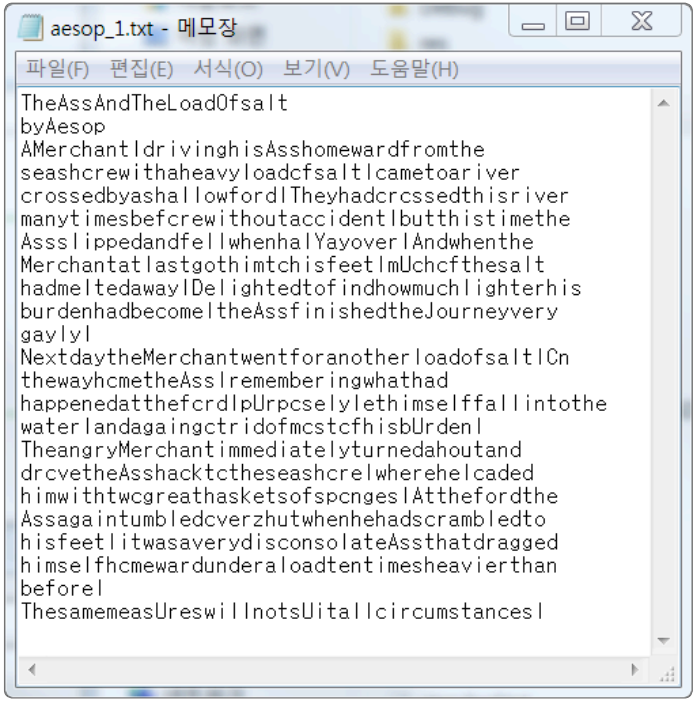
[코드 3-10] 확률을 이용한 이미지 데이터의 비교에서 줄바꿈 문자는 제외(OCR.cpp)

```
void COCR::FindLetterValue() {
    int count, maxCount;
    unsigned int buffer, bit;
    int i, j, x, y;

    for (i=0; i<allData.count; i++) {
        Data *data = &allData.data[i];
        Letter *letter = &data->letter;
        Rect *rect = &data->rect;

        if (!data->isFixed) {                    //—①
            maxCount = 0;

            for (j=0; j<standard.count; j++) {
                count = 0;

                for (y=0; y<48; y++) {
                   buffer = letter->image[y] ^ standard.letter[j].image[y];

                    for (x=0; x<32; x++) {
                        bit = 0x80000000;
                        bit >>= x;
                        bit = bit & buffer;

                        if (!bit)
                            count += 1;
                    }
                }
                if (count > maxCount) {
                    letter->value = standard.letter[j].value;
                    maxCount = count;
                }
            }
        }
    }
}
```

① 이미지 데이터의 isFixed가 true면 줄바꿈 문자의 이미지 데이터이므로 비교를 진행하지 않으며, isFixed가 false면 기본 이미지 데이터와의 비교를 진행한다.

[그림 3-8]은 지금까지 구현된 코드를 실행했을 때의 결과 파일이다. 줄바꿈 문자가 추가되어 원본 이미지 파일과 같은 줄의 문서 내용이 되었다. 하지만 아직 완벽하지는 않다. 단어들을 구분할 수 있게 공백이 추가된다면 조금 더 좋은 문서 내용이 될 것이다.

그림 3-8 줄바꿈(\n) 문자 추가 후 OCR 결과



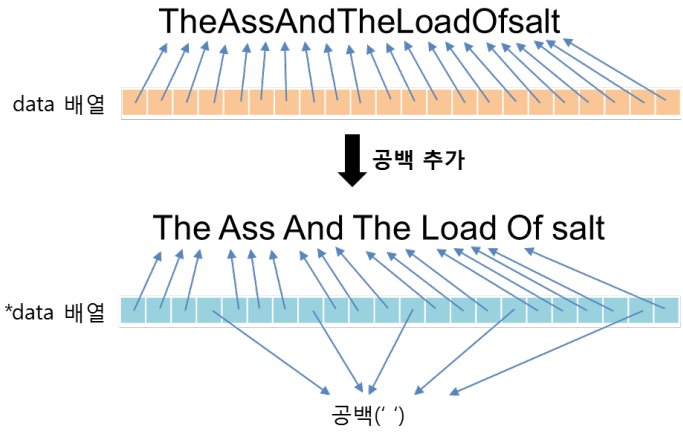
3.3 공백 문자 삽입하기

줄바꿈 문자를 삽입하여 결과 파일이 문서 이미지와 같은 줄의 개수로 만들어졌지만, 아직 단어 사이의 구분이 없어서 문서를 읽기는 힘들다. 즉, 단어와 단어 사이에 공백Space을 넣어 주어야 한다. 이번에는 공백을 단어 사이에 삽입하는 코드를 구현하겠다.

간단하게 생각하면 이미지 데이터에 공백 이미지 구조체를 넣는 방법을 생각할 수 있다. 그러나 이 방법을 사용하면 공백 문자를 중간에 삽입할 때마다 뒤에 존재하는 이미지 데이터 배열이 순차적으로 하나씩 뒤로 밀리게 되며, 이는 공백을 추가할 때마다 항상 진행되기 때문에 프로그램의 속도가 현저히 늦어질 수 있다. 그래서 이 책에서는 이미지 데이터 배열을 포인터로 만들어서 포인터 변수가 처음부터 순차적으로 이미지 데이터를 가리키며 진행되다가 문자 사이의 간격이 일정 범위를 넘으면 공백 이미지 구조체를 추가하여 가리키도록 구현하였다. 이미 만들어진 이미지 데이터 구조체 배열을 변경하지 않고, 이미지 데이터 구조체를 가리키는 포인터 배열을 추가하여 속도를 향상시키면서 프로그램의 구현을 단순화했다.

[그림 3-9]는 공백 이미지 구조체를 하나 만들어서 단어 사이에 공백이 필요한 부분에서 포인터 변수가 공백을 가리키게 된다. 이러한 방식이 진행되기 때문에 결과를 텍스트로 출력할 때는 포인터 배열을 사용하여 결과 파일을 얻는다.

그림 3-9 포인터 배열인 *data를 만든 후 하나의 공백 구조체를 추가하여 단어 사이에 추가



[그림 3-10]은 OCR.h 헤더 파일에 추가된 코드다. 이미지 데이터의 포인터 변수를 위해 AllDataPointer 구조체를 추가하였고, AllDataPointer 구조체는 data라는 이미지 데이터 포인터 배열을 포함한다. 『소수와 RSA 알고리즘으로 배

우는 Big Number 연산 - 구조체와 자료구조의 이해』(한빛미디어, 2014)의 포인터 부분을 이해한 독자라면 AllData의 data 배열의 개수보다 AllDataPointer의 포인터 data 배열의 개수가 2배지만, 메모리에서는 훨씬 더 작은 공간을 차지하는 것을 이해할 수 있을 것이다. 이는 포인터 변수가 메모리의 주소값을 가져서 모든 포인터 변수는 4-Bytes의 크기를 갖기 때문이다.

그림 3-10 이미지 데이터 배열인 data와 포인터 배열로서 추가된 *data

```
struct AllData {
            int count;
            Data data[MAX_COUNT_DATA];                    //data 배열
};

struct AllDataPrinter {
            int count;
            Data *data[MAX_COUNT_DATA + MAX_COUNT_DATA];    //*data 배열
};
```

[코드 3-11]은 OCR.h 헤더 파일의 일부분으로, 이미지 데이터에 공백을 추가하기 위해 추가된 AllDataPointer 구조체와 공백을 위해 구현된 함수를 보여 준다. 이번 장에서 OCR의 단계를 순차적으로 구현하는 이유는 인식하려는 문자 이미지 파일이 다양하여 문서에 따라서 독자가 직접 구현해야 하는 부분도 있기 때문이다. [코드 3-11]에서 가장 중요한 부분은 AllDataPointer 구조체로, 이후 함수에서 어떻게 포인터 배열을 사용하는지 이해하는 것이 중요하다.

[코드 3-11] 포인터 이미지 데이터 구조체 추가 및 COCR 클래스(OCR.h)

```
struct AllData {
    int count;
    Data data[MAX_COUNT_DATA];
};

struct AllDataPointer {                                  //---①
    int count;
    Data *data[MAX_COUNT_DATA + MAX_COUNT_DATA];         //---②
};

class COCR {
private:
```

```
    CImage *image;
    Standard standard;
    AllData allData;
    AllDataPointer allDataPointer;                          //---③
    Data *data;

    Data letterSpace;                                       //---④

    int colorToCheck;

public:
    COCR(void);
    ~COCR(void);

    void RunOCR(CImage *image, CString outFileName, int colorLetter);
    void GetStandardImageDataFromBinaryFile(char * fileName);
    void ParsingStepFirst();
    void ParsingStepSecond(int yTop, int yBottom);
    void ParsingStepThird(Rect *rect);
    void ParsingStepThird2(Rect *rect);
    void MakeImageData();
    void MakeLetterData(Rect *rect);
    void FindLetterValue();
    void AddSpaceValue();                                   //---⑤
    void AddSpaceValueInLine(int *index, int start, int end); //---⑥
    int CalculateGapSpace(int start, int end);              //---⑦
    void StoreLetterToTextFile2(CString outFileName);       //---⑧
}
```

① OCR의 이미지 데이터를 저장하는 AllData에 공백을 추가하기 위해 포인터 배열을 포함하는 AllDataPointer 구조체를 추가한다.

② 이미지 데이터를 가리키는 포인터 배열로 data 배열을 사용한다.

③ allData 변수와 마찬가지로 포인터 배열을 포함하는 구조체를 위해 allDataPointer를 선언한다. 이미지 데이터로 접근하는 포인터 배열은 allDataPointer 변수를 사용하여 접근할 수 있다.

④ 공백 이미지 데이터를 letterSpace를 사용하며 만들었다. 이후에 공백이 추가될 때마다 이미지 데이터를 가리키는 포인터 변수는 하나의 letterSpace 이미지 데이터 변수를 가리키게 된다.

⑤ 공백을 추가하는 함수로, OCR이 실행될 때 마지막에 이 함수를 실행하면 공백이 추가된다.

⑥ 공백을 추가하는 AddSpaceValue( ) 함수는 한 줄마다 공백을 추가하는 AddSpaceValueInLine( ) 함수를 반복해서 수행하여 공백을 추가한다.

⑦ 공백은 단어 사이에 추가되어야 한다. 단어 사이의 간격과 단어 안에 존재하는 문자 사이의 간격은 다르므로 문자 사이의 간격이 일정 간격 이상으로 넓다면 공백을 추가하여 단어의 구분으로 인식하게 해야 한다. CalculateGapSpace( ) 함수는 구분이 되는 간격을 계산한다.

⑧ 공백을 추가한 이후에 결과값을 텍스트 파일에 저장하는 함수다. 이전에는 allData 변수를 사용하여 결과를 저장했지만, 여기서는 이미지 데이터를 가리키는 포인터 배열을 포함하는 allDataPointer 변수를 사용하여 결과 파일을 만든다.

[코드 3-12]는 공백 이미지 데이터를 COCR 클래스의 생성자에서 정의한다. 즉, 프로그램이 실행될 때 COCR 클래스가 생성되는데, 이때 공백 이미지 데이터를 생성한다. 또한, 공백 이미지 데이터는 단 하나만 만들고 포인터 배열이 공백 값을 가질 때 여기서 정의된 공백 이미지 데이터만을 가리키면 된다.

[코드 3-12] COCR 생성자에 공백 이미지 데이터 구조체 설정(OCR.cpp)

```
COCR::COCR(void)
{
    GetStandardImageDataFromBinaryFile("standard.bin");

    letterSpace.letter.value = ' ';                    //---①
    letterSpace.isFixed = true;                        //---②
}
```

① 공백 이미지 데이터의 문자 값을 빈칸(‘ ’)으로 정의한다.

② 이전에 줄바꿈을 위해 추가한 isFixed 변수값도 true로 설정하여 확정된 이미지 데이터로 설정한다.

[코드 3-13]은 프로그램 실행 화면에서 [OCR] 버튼을 눌렀을 때 실행되는 함수로, 공백을 추가하는 함수인 AddSpaceValue( )를 이미지 인식의 마지막에 추가하였다. 이미지 인식이 진행되고 난 다음 줄바꿈이 추가된 이후에 단어 사이에 공백이 추가된다.

[코드 3-13] OCR 실행 함수에서 공백 추가 함수 실행(OCR.cpp)

```
void COCR::RunOCR(CImage *newImage, CString outFileName, int colorLetter) {
    image = newImage;
    colorToCheck = colorLetter;

    for (int i=0; i<MAX_COUNT_DATA; i++)
        allData.data[i].isFixed = false;

    ParsingStepFirst();
    MakeImageData();

    FindLetterValue();

    AddSpaceValue();                        //---①

    StoreLetterToTextFile2(outFileName);    //---②
}
```

① OCR의 마지막 과정에서 단어 사이에 공백이 추가되는 함수가 실행된다.

② OCR의 결과를 가지고 있는 포인터 배열을 사용하여 텍스트 파일에 결과를 출력한다.

[코드 3-14]는 단어 사이에 공백을 추가하는 함수의 코드다. 줄바꿈 문자를 확인하여 한 줄씩 구분을 할 수 있기 때문에 줄바꿈 문자가 있을 때마다 한 줄씩 공백을 삽입하는 AddSpaceValueInLine( ) 함수를 실행한다. 코드를 이해하기 위해서는 세 가지 변수(index, start, end)를 정확히 이해하는 것이 좋다. index는 포인터 배열인 allDataPointer의 data를 구분하기 위한 위치값으로 사용되고, start는 allData에 있는 배열 한 줄의 시작 위치를 나타내며, end는 한 줄의 끝 위치를 위해 사용한다.

[코드 3-14] 전체 문서에 공백을 추가하는 함수(OCR.cpp)

```
void COCR::AddSpaceValue() {
    int index, start, end;                          //---①

    index = 0;                                      //---②
    start = 0;

    for (int i=0; i<allData.count; i++) {           //---③

        if (allData.data[i].letter.value == '\n') { //---④

            end = i;                                //---⑤
```

```
            AddSpaceValueInLine(&index, start, end);     //---⑥
            start = i + 1;                               //---⑦
        }
    }
    allDataPointer.count = index;                        //---⑧
}
```

① index는 allDataPointer의 포인터 배열인 data의 위치를 지정하는 데 사용하는 변수고, start와 end는 문서의 한 줄을 구분하는 데 사용하는 변수로 allData의 이미지 데이터 배열의 위치값을 가진다.

② allDataPointer의 포인터 배열은 0부터 시작되어 순차적으로 값이 증가하므로 index의 초기값을 0으로 설정하며, 이후에 1씩 증가하게 된다. 문서의 이미지 데이터 배열도 0부터 시작하기 때문에 start를 0으로 초기화하며, 공백이 포함되지 않은 이미지 데이터 배열의 한 줄의 시작값이기 때문에 프로그램이 진행되면서 큰 값으로 변한다.

③ 변수 i는 allData의 이미지 데이터 배열 data의 위치[index]다. 0부터 순차적으로 1씩 증가하면서 for 문을 사용하여 공백이 없는 배열을 차례대로 반복하여 처리한다. for 문 안에서는 한 줄의 시작과 끝을 찾아서 한 줄씩 공백을 넣는 작업을 반복한다.

④ 공백이 포함되지 않은 이미지 데이터 배열의 문자를 처음부터 순차적으로 확인하면서 줄바꿈 문자가 나오면 한 줄의 끝임을 알게 되므로 한 줄에 공백을 넣는 함수를 실행한다.

⑤ 한 줄의 끝은 줄바꿈 문자기 때문에 끝의 위치를 나타내는 end 값은 '\n'의 이미지 데이터의 위치값인 i로 설정한다.

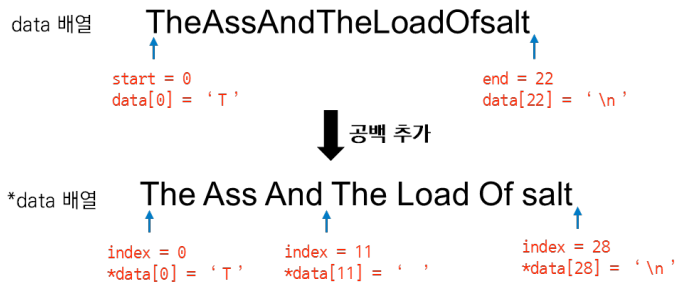
⑥ 한 줄에 공백을 삽입하는 함수를 실행한다(자세한 설명은 [코드 3-15]에서 하겠다). 이 함수가 실행되어 allDataPointer의 포인터 배열이 가리키는 것은 공백 이미지 데이터를 포함하고, index는 Call-By-Reference 방법을 사용하므로 AddSpaceValueInLine ( ) 함수 안에서도 계속해서 증가한다.

⑦ 한 줄의 공백을 추가하면 다음 한 줄의 시작 위치인 start는 현재 위치(i)의 문자인 '\n'의 다음 문자이므로 'start = i + 1;'이 되어야 한다.

⑧ 문서 이미지 데이터에 공백 추가가 완료되고, allDataPointer의 포인터 배열의 개수는 index 값을 가진다.

[그림 3-11]은 공백이 포함되지 않은 한 줄의 이미지 데이터 배열에서 start와 end의 위치를 보여 주는데, 공백이 추가된 이후에 포인터 배열의 위치를 나타내는 index의 값이 어떻게 변하는지 알 수 있다. 그래서 문서의 한 줄에 공백을 추가하는 함수인 AddSpaceValueInLine( )의 매개변수는 index, start, end가 된다. start와 end 값은 함수가 끝난 이후에 값이 바뀔 필요가 없으므로 Call-By-Value로 값이 전달되며, index는 함수의 실행 이후에 위치가 변경되어야 하므로 Call-By-Reference로서 값이 전달된다. 즉, index는 AddSpaceValueInLine( ) 함수가 실행될 때마다 증가된 값으로 계속 사용하게 된다.

그림 3-11 이미지 데이터 배열에서 한 줄 의 시작과 끝, 공백이 추가된 포인터 배열에서의 위치를 나타내는 index



[코드 3-15]는 한 줄에 공백을 추가하는 함수다. 여기서 중요한 부분은 현재의 문자와 바로 앞 문자 사이의 공간의 길이가 특정 길이보다 길면 공백 문자를 삽입하게 된다는 것이다. 또한, 매개변수로서 start, end, index를 받는다. 한 줄의 시작과 끝을 알게 되고, 새롭게 만들어지는 allDataPointer 변수 안의 포인터 배열이 차례대로 이미지 구조체를 가리키게 된다. 포인터 배열의 위치값을 가지고 있는 index를 증가하면서 이미지 데이터를 순차적으로 가리키게 된다. 매개변수 index는 Call-By-Reference 방법을 사용하여 메모리에 저장된 값이 변하므로 함수의 실행이 완료되어도 증가된 index의 값을 다른 곳에서 사용할 수 있다.

[코드 3-15] 문서 한 줄에 공백 추가하는 함수(OCR.cpp)

```
void COCR::AddSpaceValueInLine(int *index, int start, int end) {      //---①
    int i;
    int gapSpace = CalculateGapSpace(start, end);                     //---②

    allDataPointer.data[*index] = &allData.data[start];               //---③
    *index += 1;

    for (i=start+1; i<end; i++) {                                     //---④
        Data *prevData = &allData.data[i-1];                          //---⑤
        Data *currData = &allData.data[i];                            //---⑥

        if ((currData->rect.start.x - prevData->rect.end.x) > gapSpace) { //---⑦
            allDataPointer.data[*index] = &letterSpace;               //---⑧
            *index += 1;
        }
        allDataPointer.data[*index] = &allData.data[i];               //---⑨
        *index += 1;
    }
    allDataPointer.data[*index] = &allData.data[i];                   //---⑩
    *index += 1;
}
```

① 한 줄에 공백을 삽입하는 함수로, 매개변수 start, end, index를 받는다. start와 end는 Call-By-Value 방식이며, index는 Call-By-Reference 방식이다. 즉, start와 end는 값을 받아서 사용하고 함수가 종료된 이후 메모리에서 지워지며, index는 포인터를 사용하여 특정 메모리에 저장되어 이 함수를 호출한 곳에서 변경된 값을 계속 사용할 수 있다.

② gapSpace는 두 문자 사이의 공백의 길이를 판단하는 기준이다. gapSpace보다 길면 공백을 추가하고, gapSpace보다 짧으면 공백을 추가하지 않는다. CalculateGapSpace( ) 함수는 한 줄에서 공백의 기준이 되는 길이를 계산하는 함수로, 한 줄의 시작과 끝의 위치를 매개변수로 가진다(이 함수에 대해서는 [코드 3-16]에서 자세히 설명한다).

③ 한 줄의 첫 문자 앞에는 공백을 추가할 필요가 없다. allDataPointer에서 첫 문자의 이미지 데이터 주소를 가리킨다. 포인터 배열의 위치값인 index는 1이 증가하며, 포인터 배열의 다음 변수를 사용한다.

④ 한 줄에서 현재의 문자와 이전 문자의 거리를 계산하여 공백을 넣기 때문에 allData의 이미지 데이터 배열의 위치값인 i는 한 줄의 두 번째 문자부터 시작하여 마지막 문자인 줄바꿈 문자 전까지 진행되어야 한다.

⑤ 포인터 변수인 prevData는 이전 문자의 이미지 데이터를 가리킨다.

⑥ 포인터 변수인 currData는 현재 문자의 이미지 데이터를 가리킨다.

⑦ 현재 문자가 시작되는 X축 값에서 이전 문자가 종료되는 X축 값을 빼면 두 문자 사이의 공백의 거리를 알 수 있고, 이 공백의 거리가 gapSpace 값보다 크면 공백을 삽입한다.

⑧ allDataPointer의 포인터 배열 data의 현재 위치값에 공백 이미지 데이터 구조체의 주소를 저장하여 공백 문자를 가리키게 한다. 그리고 index를 1 증가시켜서 다음 위치의 배열이 문자 이미지 데이터를 가리키도록 준비한다.

⑨ 현재 문자의 이미지 데이터를 allDataPointer의 포인터 배열이 가리키게 한다. 또한, index를 1 증가시켜서 다음 위치의 배열이 문자 이미지 데이터를 가리키도록 준비한다.

⑩ 한 줄에서 공백이 모두 추가되면 마지막에 줄바꿈 문자를 포인터 배열에 추가한다. 이로써 한 줄에 공백을 넣는 작업이 완료된다.

[코드 3-16]은 문자 사이에 공백을 넣는 기준이 되는 거리를 계산하는 함수다. 일반적으로 한 줄에 공백이 10개 정도 있으면 문자는 40개가 조금 넘게 있는데, 이를 바탕으로 단어 사이의 공백의 길이를 계산한다. 이 함수에서 중요한 것은 totalGap과 count 변수다. totalGap은 한 줄의 모든 문자 사이의 길이를 더하고, count는 문자의 개수를 나타낸다. 마지막에 totalGap / (count * 0.7)을 계산하여 공백의 기준이 되는 값을 얻는다. 두 문자 사이의 길이를 계산하는 방법은 [코드 3-15]와 같은 방법으로 구현한다. 한 줄의 시작과 끝을 알기 때문에 현재의 문자 이미지 데이터(currData)와 이전의 문자 이미지 데이터(prevData)를 얻어서 두 문자 사이의 길이를 얻는다.

[코드 3-16] 문자 사이 간격 기준을 반환하는 함수 – 이 값보다 크면 공백을 추가(OCR.cpp)

```
int COCR::CalculateGapSpace(int start, int end) {
    int totalGap, count;                                    //---①

    totalGap = 0;                                           //---②
    count = 0;

    for (int i=start+1; i<end; i++) {

        Data *prevData = &allData.data[i-1];                //---③
        Data *currData = &allData.data[i];                  //---④
```

```
        totalGap += (currData->rect.start.x - prevData->rect.end.x);    //---⑤
        count += 1;                                                     //---⑥
    }

    return ((int)(totalGap / (count * 0.7)));                           //---⑦
}
```

① totalGap은 모든 문자 사이의 공백의 길이를 합한 값을 저장하고, count는 모든 문자의 개수를 나타낸다.

② totalGap과 count는 0으로 초기화되며, for 문 안에서 계산될 때마다 값은 증가하게 된다.

③ 포인터 변수인 prevData는 이전 문자의 이미지 데이터를 가리킨다.

④ 포인터 변수인 currData는 현재 문자의 이미지 데이터를 가리킨다.

⑤ 이전 문자와 현재 문자의 X축의 간격을 계산하여 totalGap에 더한다.

⑥ 문자의 개수를 나타내는 count는 for 문이 진행될 때마다 1씩 증가한다.

⑦ 앞에서 설명한 바와 같이 한 줄에 10개의 공백이 있으면 일반적으로 40개가 조금 넘는 문자가 있다. 그래서 전체 문자의 개수에서 0.7을 곱하여 문자의 개수를 감소시킨 후에 totalGap에서 나누면 공백의 기준이 되는 값을 얻을 수 있다(이때 0.7은 실제 프로그램을 하면서 알게 된 count에 곱하는 값 중에서 가장 좋은 결과값을 얻을 수 있는 값이다).

지금까지 구현한 내용으로 allDataPointer 변수는 공백이 추가된 포인터 배열을 가지게 된다. 이전에는 allData에서 이미지 데이터 배열로 결과 파일을 만들었지만, 이제부터는 allDataPointer를 사용하여 결과 파일을 만들어야 한다. [코드 3-17]은 allDataPointer를 사용하여 텍스트 결과 파일을 만드는 방법이다. [코드 3-6]과 거의 동일하며, 단지 allData 대신에 allDataPointer로 결과 파일을 만드는 것이 다를 뿐이다.

[코드 3-17] 이미지 데이터 포인터 배열의 값을 결과 파일에 출력

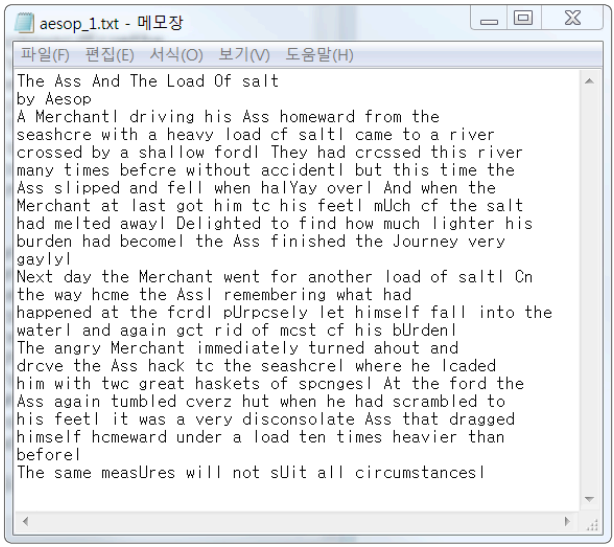
```
void COCR::StoreLetterToTextFile2(CString outFileName) {
    FILE *fp;
    fp = fopen((char*)((LPCSTR)(outFileName)), "wt");

    for (int i=0; i<allDataPointer.count; i++)              //---①
        fputc(allDataPointer.data[i]->letter.value, fp);
```

```
    fclose(fp);
}
```

① allDataPointer에 존재하는 문자의 개수만큼 for 문을 반복하면서 처음부터 순차적으로 파일에 문자를 출력한다.

[그림 3-12]는 출력된 결과 문서다. 문자 사이에 공백이 정확히 추가되었음을 알 수 있다. 줄바꿈 문자와 공백이 추가되어서 읽기 쉬운 문서가 만들어졌지만, 특정 단어에서는 아직 인식률이 높지 않다. 마침표, 쉼표, 따옴표 등이 인식되지 않았다. 이는 이전에 기본 이미지 데이터 파일을 만들 때 62개의 기본 문자와 숫자만을 만들었기 때문이다.

그림 3-12 단어 사이에 공백을 추가한 후 OCR 결과



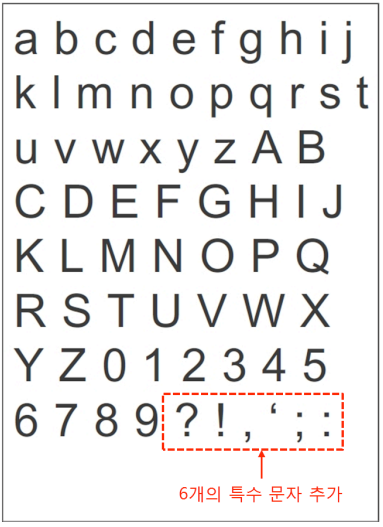
3.4 특수 문자 추가하기

이전에 만든 기본 이미지 데이터는 52개의 알파벳과 10개의 숫자였다. 그러나 인식하려는 문서 이미지에는 특수 문자가 몇 개 포함되어 있다. 따라서 인식률을 높

이려면 특수 문자도 기본 이미지 데이터에 포함되어야 한다. 그래서 [그림 3-13]과 같이 문서 이미지에서 새롭게 나온 특수 문자를 위해 기본 이미지 데이터에 6개의 특수 문자를 추가하였다. 이번 절에서는 기본 이미지 데이터에 특수 문자를 포함하여 만든 'standard.bin' 파일을 OCR 프로그램에 추가하여 특수 문자의 이미지 인식률을 높여 보겠다. 문서에 6개의 특수 문자 외에 다른 특수 문자가 포함되어 있다면 기본 이미지 데이터에 해당 특수 문자를 추가로 만들면 된다.

[그림 3-13]처럼 포함하려는 문자를 기본 데이터를 위한 이미지에 추가해 보자.

그림 3-13 특수 문자가 포함된 기본 문자 이미지(standardImage.jpg)



특수 문자를 추가할 때 특수 문자를 문자로 프로그램 코드에 넣으면 프로그램 오류가 발생할 수 있다. 예를 들어, 작은따옴표Apostrophe 문자는 C 언어에서 C 언어에서 문자Character 값을 넣는 데 사용하는 예약어다. 그래서 작은따옴표(') 문자를 이미지 데이터의 결과값으로 넣으려면 ASCII 코드표를 사용하여 16진수 값으로 넣어야 한다. 다음 두 줄을 보면 첫 번째 예에서 에러가 발생한다.

```
allData.data[65].letter.value = '''; // 작은따옴표(')는 예약어이므로 에러 발생
```

```
allData.data[65].letter.value = 0x27; //작은따옴표(')의 16진수 값인 0x27 사용
```

[그림 3-14]의 ASCII 코드표에서 작은따옴표는 16진수 값이 '0x27'이므로 이 값을 결과값으로 문자 변수에 넣으면 된다.

0x27(16진수) = 0010 0111(2진수) = 39(10진수)

이를 이해하면 [그림 3-14]의 ASCII 코드표를 보는 법은 쉽게 알 수 있다. 프로그램에서 특수 문자는 ASCII 코드표에서 16진수 값으로 문자의 결과를 넣는다.

그림 3-14 ASCII 코드표(특수 문자의 16진수 값)

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

[코드 3-18]은 2장에서 구현한 기본 이미지 데이터를 만드는 코드의 실행 함수인 `CreateStandard( )`의 내용이다. [그림 3-13]의 특수 문자를 포함한 이미지를 인식하여 기본 데이터를 만들고, 함수의 중간에 문자 6개의 내용을 삽입하는 것이 추가되었다. 모든 특수 문자는 16진수 값으로 결과값을 가지며 텍스트 모드에서 결과값을 출력하면 정상적인 문자 데이터가 파일에 쓰인다.

[코드 3-18] 특수 문자의 결과가 포함된 기본 이미지 데이터를 만드는 함수

```
void COCR::CreateStandard(CImage *newImage) {
    image = newImage;
    colorToCheck = 50;

    ParsingStepFirst();
    MakeImageData();

    int i = 0;
    for (i=0; i<26; i++)
        allData.data[i].letter.value = 'a' + i;
    for (i=0; i<26; i++)
        allData.data[i+26].letter.value = 'A' + i;
    for (i=0; i<10; i++)
        allData.data[i+52].letter.value = '0' + i;

    allData.data[62].letter.value = 0x3F;//'?';              //---①
    allData.data[63].letter.value = 0x21;//'!';              //---②
    allData.data[64].letter.value = 0x2C;//',';              //---③
    allData.data[65].letter.value = 0x27;//'''';             //---④
    allData.data[66].letter.value = 0x3B;//';';              //---⑤
    allData.data[67].letter.value = 0x3A;//':';              //---⑥

    PrintEveryImageDataInTextFile( " standard.txt " );
    //GetStandardImageDataFromTextFile("standard.txt");

    PrintEveryImageDataInBinaryFile("standard.bin");
    GetStandardImageDataFromBinaryFile("standard.bin");

    PrintAllStandardImageToTextFile("standardout.txt");
}
```

① 물음표(?)의 16진수 값인 '0x3F'를 문자 값으로 대입한다.

② 느낌표(!)의 16진수 값인 '0x21'을 문자 값으로 대입한다.

③ 쉼표Comma(,)의 16진수 값인 '0x2C'를 문자 값으로 대입한다.

④ 작은따옴표(')의 16진수 값인 '0x27'을 문자 값으로 대입한다.

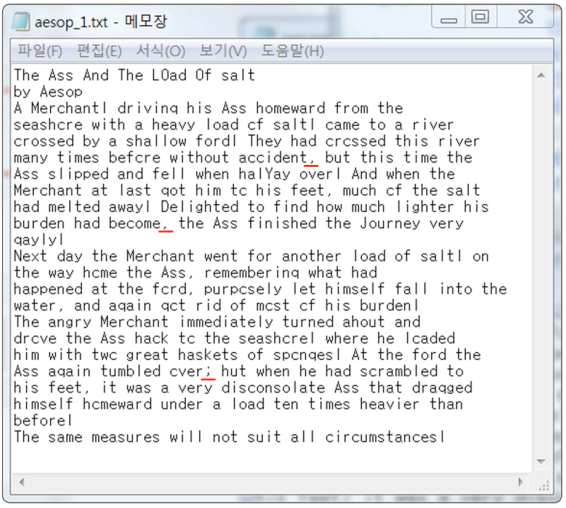
⑤ 쌍반점Semicolon(;)의 16진수 값인 '0x3B'를 문자 값으로 대입한다.

⑥ 쌍점Colon(:)의 16진수 값인 '0x3A'를 문자 값으로 대입한다.

특수 문자가 포함된 기본 이미지 데이터를 가지고 OCR 프로그램을 실행하면 결과는 [그림 3-15]와 같다. 쉼표(,)와 쌍반점(;)이 정상적으로 인식되었으나 아직 마

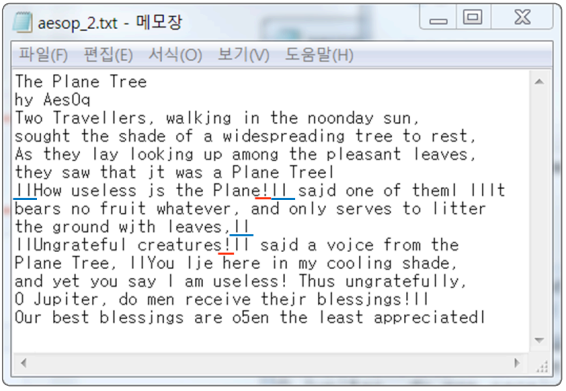
침표는(.)의 정상적으로 인식되지 않았다. 다음 절에서 마침표를 포함한 다른 문자는 어떻게 인식하는지 알아보자.

그림 3-15 쉼표(,)와 쌍반점(;) 인식(aesop_1.txt)



[그림 3-16]은 'aesop_2.txt' 결과 파일이다. 느낌표(!)는 정상적으로 인식되었으나, 큰따옴표(")는 마침표와 마찬가지로 '1'로 인식되었다. 지금까지 상태로 OCR은 80% 이상의 인식률을 보이나 다음 절에서 추가로 코드를 구현하여 인식률을 더 높이는 작업을 진행해 보겠다.

그림 3-16 느낌표(!)는 인식하였으나 큰따옴표(")는 인식하지 못함(aesop_2.txt)

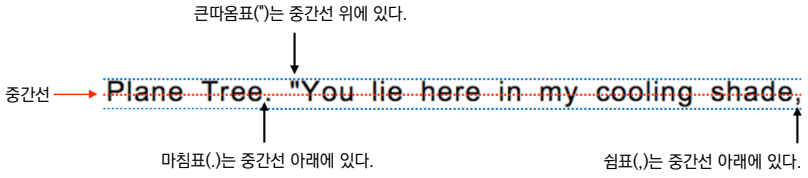


3.5 마침표, 쉼표, 따옴표 인식

줄바꿈, 공백에 이어 특수 문자까지 추가하여 조금 더 읽기 쉬운 결과를 내는 인식이 이루어졌다. 하지만 마침표(.)는 '1'로 인식되었고 큰따옴표(")는 '11'로 인식되어 문장을 읽는 것이 매끄럽지는 않다. 이번에는 마침표(.), 쉼표(,), 작은따옴표('), 큰따옴표(")를 인식하는 코드를 구현하고자 한다. 이러한 특수 문자의 특징은 일반 문자보다 크기가 작아 가로와 세로가 일정한 이미지 데이터에 넣으면 이미지가 확장되어 일반 문자로 인식된다는 점이다. 이렇게 크기가 작은 특수 문자는 위치를 기반으로 인식 프로그램에 적용하는 방법이 있다.

[그림 3-17]을 보면 한 줄에 마침표, 큰따옴표, 쉼표가 있다. Y축을 기준으로 한 줄의 가운데에 중간선MidLine을 그리면 세 특수 문자의 위치를 구분하기가 쉽다. 즉, 일반적인 알파벳은 중간선에 모두 걸쳐 있지만, 세 개의 특수 문자는 중간선의 위쪽이나 아래쪽에 위치하게 된다. 따라서 한 줄에서 중간선의 위치를 정하고, 마침표와 쉼표 이미지의 위쪽 경계선은 중간선보다 아래에 있고, 큰따옴표 이미지의 아래쪽 경계선은 중간선보다 위에 있는 특징을 바탕으로 프로그램을 구현하면 된다. 이번 절에서는 문자 이미지의 사각형 위치를 기반으로 문자를 정의한다. 이처럼 OCR은 문서 이미지에서 문자의 특징을 가지고 구체적으로 프로그램을 진행할 때 인식률이 높은 프로그램으로 발전한다는 것을 알아두자.

그림 3-17 한 줄 Y축의 중간선을 기준으로 위치에 따라 문자 정의



[코드 3-19]는 Data 구조체에 isDeleted라는 변수를 추가하였다. 이전에 추가한 isFixed 변수는 줄바꿈 문자를 추가하기 위해 존재하는 변수고, isDeleted

변수는 이미 생성된 이미지 데이터를 사용하지 않기 위해 추가한 변수다. 지금까지의 프로그램에서 큰따옴표(")는 두 개의 문자 '1' 이미지 데이터로 인식되었다. 즉, 하나의 큰따옴표는 하나의 이미지 데이터를 만들어야 하는데, 두 개로 생성된 것이다. 따라서 하나의 이미지 데이터는 큰따옴표 문자로 결과를 변경하고, 나머지 하나의 이미지 데이터는 지워야 한다. 이때 배열로 이미 생성된 이미지 데이터를 지우려면 코드상에서 다소 번거로운 작업이 이루어져야 한다. isDeleted 변수를 사용하여 isDeleted가 true면 해당 이미지 데이터는 사용하지 않고, isDeleted가 false면 정상적인 이미지 데이터로 처리를 한다. 이렇게 변수 하나를 추가하여 코드를 조금 더 단순화시키는 방법은 프로그램을 만들면서 많이 사용한다.

[코드 3-19] 이미지 데이터 구조체에 Flag 변수 추가(OCR.cpp)

```
struct Data {
    bool isFixed;                //---①
    bool isDeleted;              //---②
    Letter letter;
    Rect rect;
};

struct AllData {
    int count;
    Data data[MAX_COUNT_DATA];
};

struct AllDataPointer {
    int count;
    Data *data[MAX_COUNT_DATA + MAX_COUNT_DATA];
};
```

① isFixed 변수가 true면 해당 Data 구조체의 문자는 결과가 확정된 것이고, false면 기본 이미지 데이터와 비교하여 인식을 진행해야 한다. isFixed 변수는 줄바꿈 문자를 위해 추가한 변수다.

② isDeleted 변수가 true면 해당 Data 구조체는 결과 파일에 출력을 하지 않는(사용하지 않는) 구조체고 false면 사용하는 이미지 데이터인데, 인식을 진행하면서 지워야 하는 이미지 데이터가 필요할 때 사용한다.

[코드 3-20]은 COCR 클래스의 생성자와 [OCR] 버튼을 선택하였을 때 실행되는 RunOCR( ) 함수다. MakeCommaPeriod( ) 함수와 MakeQuotationMark( ) 함수가 마지막에 추가되었다. 여기서 중요한 것은 공백 문자 이미지 데이터는 COCR 클래스의 생성자에서 초기화되는데, 일반적인 이미지 데이터인 allData는 RunOCR( ) 함수 안에서 초기화된다는 것이다. 공백 문자 이미지 데이터는 프로그램이 실행될 때 한 번만 초기화되는 것이 좋다. 일반적인 이미지 데이터는 [OCR] 버튼이 선택될 때마다 새롭게 초기화되어 인식을 진행해야 하기 때문이다.

[코드 3-20] Flag 변수 isFixed와 isDeleted 초기화 및 OCR을 위한 추가 함수(OCR.cpp)

```
COCR::COCR(void)
{
    GetStandardImageDataFromBinaryFile("standard.bin");

    letterSpace.letter.value = ' ';
    letterSpace.isFixed = true;                    //---①
    letterSpace.isDeleted = false;
}

void COCR::RunOCR(CImage *newImage, CString outFileName, int colorLetter) {

    image = newImage;
    colorToCheck = colorLetter;

    for (int i=0; i<MAX_COUNT_DATA; i++) {         //---②
        allData.data[i].isFixed = false;
        allData.data[i].isDeleted = false;
    }

    ParsingStepFirst();
    MakeImageData();

    FindLetterValue();

    AddSpaceValue();
    MakePeriodApostrophe();                        //---③
    MakeQuotationMark();                           //---④

    StoreLetterToTextFile2(outFileName);
}
```

① 프로그램이 실행되면 COCR 클래스가 생성되며, 이때 공백 문자의 이미지 데이터는 초기화된다. isFixed는 true로 문자의 결과는 정해져서 인식을 진행할 필요가 없다. isDeleted는 false로 선언하여 지울 필요가 없이 결과 파일에 문자를 출력한다.

② [OCR] 버튼이 선택될 때마다 allData의 모든 이미지 데이터는 초기화된다. isFixed가 false이므로 기본 이미지 데이터와의 비교를 통한 인식을 진행한다. isDeleted를 false로 선언하여 이후에 지워야 하는 이미지 데이터가 존재할 때 isDeleted를 true로 변경한다. 큰따옴표(")를 만들기 위한 함수에서도 isDeleted 변수를 사용한다.

③ 한 줄의 중간선을 기준으로 마침표(.)와 작은따옴표(')를 인식하는 함수다.

④ 두 개의 연속되는 작은따옴표(')가 있으면 하나의 큰따옴표(")로 변경해 주는 함수다.

[코드 3-21]은 한 줄에서 중간선의 Y축 위치를 구하는 함수로, 한 줄에 존재하는 모든 문자의 Y축의 중간값을 더한 후 문자의 개수로 나누어서 평균값을 반환한다. 한 문자의 Y축의 중간값은 문자를 이미지 파싱한 후에 얻은 사각형의 시작점과 끝점을 더한 후 2로 나누어 문서 이미지에서의 Y축 값을 얻으면 된다. 함수는 매개변수로 한 줄의 첫 번째 문자의 위치를 받으며, 한 줄의 마지막은 줄바꿈 문자이므로 쉽게 한 줄의 문자를 모두 확인할 수 있다.

[코드 3-21] 한 줄에서 세로 Y축의 중간값 계산하기(OCR.cpp)

```
int COCR::CalculateMidlineValue(int index) {          //---①

    int count = 0;                                    //---②
    float sumHeight = 0;                              //---③

    for (int i=index; i<allData.count; i++) {         //---④

        if (allData.data[i].letter.value == '\n')     //---⑤
        break;

        sumHeight += ((allData.data[i].rect.end.y + allData.data[i].rect.start.
        y) / 2);
        count += 1;                                   //---⑥
    }

    return (int)(sumHeight / count);                  //---⑦
}
```

① 한 줄의 Y축의 중간선 값을 반환하는 함수로, 매개변수로서 한 줄의 첫 번째 문자의 위치 값을 받는다.

② count 변수는 문자의 개수를 저장하기 위해 사용되며, 0으로 초기화된 후에 문자가 하나씩 중간선 값이 계산될 때마다 1씩 증가한다.

③ sumHeight는 문자의 Y축 중간값들을 더하기 위해 사용하고, 함수의 마지막에 문자의 개수로 나누어서 한 줄의 평균 중간값을 계산하기 위해 사용한다.

④ 함수의 매개변수로 받은 index는 한 줄의 첫 번째 문자의 위치이며, 이미지 데이터의 마지막까지 for 문을 진행한다. 그러나 ⑤에서와 같이 줄바꿈 문자의 이미지 데이터가 나오면 for 문을 종료한다.

⑤ 현재 검사하는 문자가 줄바꿈 문자면 한 줄의 마지막까지 진행된 것이므로 분기 명령어(break;)로 for 문을 빠져 나온다.

⑥ 한 문자의 Y축 중간값은 이미지 영역(rect)에서 시작점과 끝점의 Y축 값들을 더한 후 2로 나누면 된다. sumHeight 변수는 모든 문자의 중간값을 모두 더하고, count 변수는 문자의 개수를 증가시킨다.

⑦ 모든 문자의 중간선 값을 더한 sumHeight를 문자의 개수를 저장하는 count로 나누면 중간선의 평균값이 되며 함수는 이 값을 반환한다.

중간선의 값이 계산되면 이 값을 기준으로 위 또는 아래에 있는 문자를 마침표 또는 작은따옴표로 변경하면 된다. 쉼표는 이전에 특수 문자로서 인식되어서 쉼표로 인식된 문자가 중간선 위에 있다면 작은따옴표(')로 인식하면 된다.

[코드 3-22]는 마침표와 작은따옴표 문자를 인식하는 함수로, 한 줄마다 중간선의 값을 계산하면서 문자를 하나씩 비교하게 된다. 여기서는 for 문을 선언할 때 이미지 데이터의 위치로 사용되는 변수 i를 증가하는 것이 아니라 함수 안에서 i가 1씩 증가하면서 프로그램이 진행된다. 참고로, while(true)와 for(;;)가 무한 루프를 만든다는 것을 이해하면 분기를 위한 변수가 어디서 사용되어도 괜찮다는 것을 쉽게 이해할 수 있을 것이다.

[코드 3-22] 마침표(.)와 쉼표(,) 만들기(OCR.cpp)

```
void COCR:: MakePeriodApostrophe() {
```

```
    int avgMidline;

    for (int i=0; i<allData.count;) {                                    //---①

        avgMidline = CalculateMidlineValue(i);                           //---②

        while (allData.data[i].letter.value != '\n') {                   //---③

            if (allData.data[i].rect.start.y > avgMidline) {             //---④
                if ((allData.data[i].letter.value == 'I') ||
                    (allData.data[i].letter.value == 'l')) {

                    allData.data[i].letter.value = '.';
                }
            } else if (allData.data[i].rect.end.y < avgMidline) {        //---⑤
                if ((allData.data[i].letter.value == 'I') ||
                    (allData.data[i].letter.value == 'l') ||
                    (allData.data[i].letter.value == ',')) {

                    allData.data[i].letter.value = 0x27;//'''';
                }
            }

            i += 1;                                                      //---⑥
        }
        i += 1;                                                          //---⑦
    }
}
```

① for 문은 문자 이미지 데이터의 처음부터 끝까지 진행된다. 앞에서 언급한 바와 같이, i는 for 문을 선언할 때가 아닌 for 문 안에서 1씩 증가한다. i가 for 문을 선언할 때 증가하지 않고 함수 안에서 증가하는 방법은 프로그래밍에서 종종 사용한다.

② 한 줄의 중간선을 계산하여 문서 이미지의 Y축 값을 가져온다. 함수의 매개변수로 한 줄의 시작 문자의 위치를 나타내는 i를 함수에 전달한다.

③ 줄바꿈 문자는 한 줄의 마지막 문자이므로 줄바꿈 문자가 나오기 전까지 앞에서 얻은 중간선을 사용하여 한 줄에 있는 문자를 검사하여 마침표와 작은따옴표 문자를 찾는다.

④ 이미지 파싱에서 얻은 사각형 위의 선이 중간선 값보다 크면 중간선 아래에 있는 것이고, 이 문자가 대문자 'I'나 소문자 'l'로 인식되었다면 마침표 문자로 결정한다.

⑤ 이미지 파싱에서 얻은 사각형 아래의 선이 중간선 값보다 작다면 중간선 위에 있고, 이 문자가 대문자 'I', 소문자 'l', 쉼표로 인식되었다면 작은따옴표 문자로 결정한다.

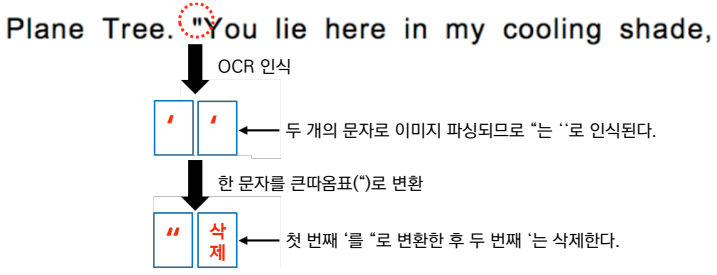
⑥ while 문은 한 줄 안에서 진행되는데, 한 문자의 검사가 종료될 때마다 문자의 위치값인 i를 1 증가시켜서 다음 문자의 검사를 진행한다.

⑦ 한 줄의 문자 검사가 완료되면 i를 1 증가시켜서 줄바꿈 문자의 다음 문자를 다음 줄의 첫 문자로 정하고 다음 줄의 검사를 진행한다.

마침표 문자와 작은따옴표 문자의 인식으로 조금 더 나은 문서를 만들었다. 그러나 문서 이미지에서의 큰따옴표는 두 개의 작은따옴표 문자로 인식되었다. 그렇다면 두 개의 작은따옴표와 큰따옴표를 구분하는 방법은 무엇일까?

[그림 3-18]은 두 개의 작은따옴표(')를 하나의 큰따옴표(")로 변환하는 방법으로, 작은따옴표 문자가 두 개 연속하여 발생하면 하나의 큰따옴표 문자를 만들어 준다. 두 개의 작은따옴표로 인식된 것은 두 개의 문자 이미지 데이터가 만들어진 것이며, 앞에 있는 작은따옴표 문자를 큰따옴표 문자로 치환하고 뒤에 있는 작은따옴표 문자의 이미지 데이터를 삭제하면 된다. 실제 프로그램에서는 이미지 데이터는 배열로 생성되므로 하나의 이미지 데이터를 메모리에서 삭제하는 것은 불가능하다. 그래서 이전에 만들어 놓은 Data 구조체에 있는 isDeleted 변수를 true로 설정하여 이미지 데이터를 사용하지 않게 설정한다.

그림 3-18 큰따옴표(")는 두 개의 작은따옴표(')로 인식, 큰따옴표(")의 변환 작업 필요



[코드 3-23]은 [그림 3-18]에서의 큰따옴표 만드는 방법을 코드로 구현한 것으로, 프로그램의 예약어로 큰따옴표와 작은따옴표가 정의되었기 때문에 ASCII 코드 값으로 대체했다. 즉, 0x27은 작은따옴표 문자를 나타내고 0x22는 큰따옴표 문자를 나타내는 ASCII 코드 값이다.

[코드 3-23] 큰따옴표(“) 만들기(OCR.cpp)

```
void COCR::MakeQuotationMark() {

    for (int i=0; i<allData.count; i++) {
        if (!(allData.data[i].isDeleted) && allData.data[i].letter.value ==
        0x27) {
            if (allData.data[i+1].letter.value == 0x27) {       //---①
                allData.data[i].letter.value = 0x22;//'"'       //---②
                allData.data[i+1].isDeleted = true;             //---③
            }
        }
    }
}
```

① 이미지 데이터의 처음부터 마지막까지 검사가 진행되며, 현재(i)의 이미지 데이터가 지워지지 않았고, 문자가 0x27로서 작은따옴표(‘) 값을 가지고 있다면 다음 조건에서 바로 다음 문자(i+1)가 작은따옴표 값인지를 검사하여 큰따옴표(“)로 만들어 준다.

② 두 개의 연속한 문자가 작은따옴표 문자이므로 현재(i) 위치의 문자를 큰따옴표의 ASCII 값인 0x22로 변경한다.

③ 큰따옴표로 치환한 바로 다음(i+1) 문자의 이미지 데이터 구조를 사용하지 않기 위해서 isDeleted를 true로 설정한다. 이후에 결과를 출력할 때 isDeleted가 true인 이미지 데이터의 문자를 결과값으로 출력하지 않게 된다.

[그림 3-19]와 [그림 3-20]은 지금까지 구현한 프로그램을 실행한 텍스트 파일의 결과다. [그림 3-19]에서는 마침표가 정확히 인식되었으며, [그림 3-20]에서는 큰따옴표가 정확히 인식되었다.

OCR은 100%의 인식률을 갖는 것이 상당히 어렵지만 최대한 인식률을 높이는 방법으로 프로그램을 구현해야 한다. 어떤 부분에서 인식이 안 되는지를 알고, 해당 문자의 인식률을 높이는 방법을 계속하여 연구하면서 더 나은 인식률을 갖는 프로그램이 되도록 해야 한다.

[그림 3-19]를 다시 확인해 보자. 첫 줄을 보면 ‘Load’라는 단어가 ‘LOad’로 인식되었다. 다음 절에서는 문장의 첫 문자는 대문자로, 문장의 중간에는 소문자로

인식하는 프로그램을 구현하여 조금 더 정확한 결과를 만드는 OCR 프로그램을 만들어 보겠다.

그림 3-19 마침표(.) 인식 결과(aesop_1.txt)

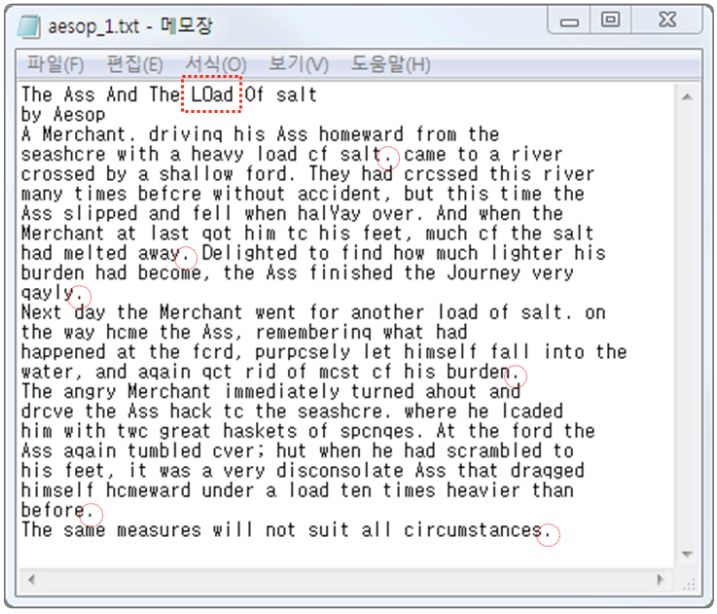
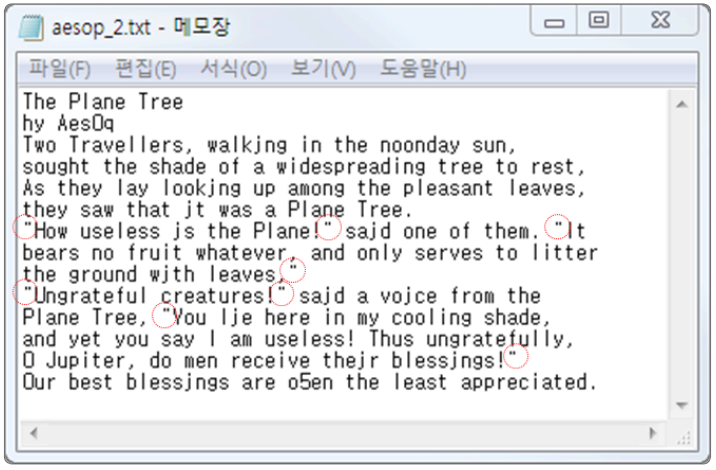


그림 3-20 큰따옴표(“) 인식 결과(aesop_2.txt)



3.6 대문자와 소문자 보정

알파벳에는 대문자와 소문자가 동일하게 인식되는 문자가 존재한다. 소문자 a와 대문자 A는 모양이 확연하게 다르기 때문에 인식 프로그램에서 대문자와 소문자를 확인하는 작업이 필요하지 않다. 그러나 [그림 3-21]과 같이 이미지 데이터에 인식할 문자를 넣으면 이미지상으로 비슷한 단어가 여럿 있다. 예를 들어, 소문자 c와 대문자 C는 가로와 세로의 크기가 일정한 이미지 데이터에 넣으면 같은 이미지를 갖게 된다. 소문자 l과 대문자 I는 Arial 문자형에서는 동일한 이미지 데이터를 가진다.

이번에 구현하는 항목은 문서 이미지에서 문장이 시작될 때는 대문자를 쓰고, 문장 중간에는 소문자를 쓰는 코드다. [그림 3-21]과 같이 9쌍의 대문자와 소문자를 문장에 따라 변형하는 작업이다.

그림 3-21 대문자와 소문자가 같은 모양일 경우 상호 변환(OCR.cpp)

이는 인식의 마지막에 진행하는 중요한 작업 중 하나다. 이전까지는 한 줄에 따른 인식 처리를 진행하였지만, 이제부터는 한 문장을 기준으로 대문자와 소문자를 변경한다. 문장의 기준이 되는 것은 마침표(.), 물음표(?), 느낌표(!), 큰따옴표(") 등이 될 수 있다. 이 책에서는 문장이 끝나는 기준을 마침표와 큰따옴표만으로 진행한다. 실제로 상용화 프로그램을 만들기 위해서는 더욱 세세한 부분까지 신경 쓰면서 프로그램을 구현해야 한다.

[코드 3-24]는 프로그램 화면에서 [OCR] 버튼이 선택했을 때 실행되는 함수다. 함수의 마지막에 대문자와 소문자를 변경하는 함수인 ChangeBigSmallLetter( ) 함수를 추가하였다. 추가한 함수의 구현은 [코드 3-26]에서 확인할 수 있다. 이 프로그램에서 더 나은 인식을 위해 추가로 구현해야 하는 것들이 있으면 추가 함수를 만들면 되고, 대부분은 [코드 3-24]의 RunOCR( ) 함수 안에서 추가로 구현된 함수를 실행하면 된다.

[코드 3-24] OCR 버튼 선택 시 실행되는 함수에 대문자/소문자 보정 함수 추가(OCR.cpp)

```
void COCR::RunOCR(CImage *newImage, CString outFileName, int colorLetter) {

    image = newImage;
    colorToCheck = colorLetter;

    for (int i=0; i<MAX_COUNT_DATA; i++) {
        allData.data[i].isFixed = false;
        allData.data[i].isDeleted = false;
    }

    ParsingStepFirst();
    MakeImageData();

    FindLetterValue();

    AddSpaceValue();
    MakeCommaPeriod();
    MakeQuotationMark();

    ChangeBigSmallLetter();                    //---①

    StoreLetterToTextFile2(outFileName);
}
```

① 이전까지 진행된 OCR 구현 함수의 마지막에 ChangeBigSmallLetter( ) 함수를 추가하여 전체 문서 결과에서 문장에 따른 대문자와 소문자의 변환을 진행한다.

[코드 3-25]는 대문자와 소문자를 변경할 때 이를 쉽게 구현하기 위해 추가한 3개의 함수를 보여 준다. 하나의 문자인 value 매개변수가 알파벳, 대문자 또는 소문자인지를 검사하여 참(true) 또는 거짓(false)을 반환한다. 프로그램에서 자주 호출하는 코드는 이렇게 필요한 보조 함수를 만들어 사용하는 것이 좋다.

[코드 3-25] 대소문자 변경을 위한 세 가지 함수 - (OCR.cpp)

```
bool COCR::isAlphabet(char value) {

    if (value >= 'A' && value <= 'Z')                //---①
        return true;
    else if (value >= 'a' && value <= 'z')           //---②
        return true;

    return false;
}

//-------------------------------------------------------
bool COCR::isBigLetter(char value) {

    if (value >= 'A' && value <= 'Z')                //---③
        return true;

    return false;
}

//-------------------------------------------------------
bool COCR::isSmallLetter(char value) {

    if (value >= 'a' && value <= 'z')                //---④
        return true;

    return false;
}
```

① 함수의 이름을 isAlphabet( )과 같이 질문하는 문구로 만드는 것이 좋다. 함수의 이름만으로 무엇을 처리하는 함수인지를 쉽게 알 수 있다. 'A'와 'Z' 사이에 있는 문자는 대문자 알파벳이므로 true를 반환한다.

② 'a'와 'z' 사이에 있는 문자는 소문자 알파벳이므로 ①과 같이 true를 반환한다.

③ isBigLetter( )는 대문자인지를 판별하는 함수로, 'A'와 'Z' 사이에 존재하면 true를 반환하고 아니면 false를 반환한다.

④ isSmallLetter( )는 소문자인지를 판별하는 함수로, 'a'와 'z' 사이에 존재하면 true를 반환하고 아니면 false를 반환한다.

대문자와 소문자의 변경 작업은 특수 문자에서는 필요하지 않고 알파벳에서만 진행되므로 [코드 3-25]와 같은 보조 함수를 구현하였다. [코드 3-26]에서는 만들어진 보조 함수를 사용하여 대문자와 소문자의 변경 작업을 진행한다. [코

드 3-26]에서 isFirstChar 변수가 있는데, 프로그램에서 가장 중요한 변수다. isFirstChar가 true면 이후에 검사하는 문자는 대문자여야 하며, 첫 대문자를 변경한 이후에는 isFirstChar를 false로 변경하여 소문자로 변경해야 한다. 또한, 문장이 끝나는 마침표나 큰따옴표가 나오면 isFirstChar를 true로 다시 변경하여 이후에 나오는 첫 알파벳 문자를 대문자로 만들어 준다. [코드 3-26]은 지금까지 인식된 모든 문자를 for 문을 사용하여 처음부터 끝까지 검사하고, isFirstChar가 true일 때와 false일 때를 기준으로 대문자와 소문자의 변경 작업을 진행한다.

[코드 3-26] 대문자와 소문자를 변경하는 보정 함수 – OCR.cpp)

```
void COCR::ChangeBigSmallLetter() {

    bool isFirstChar = true;                          //---①
    char *ch, *chNext;                                //---②

    for (int i=0; i<(allData.count-1); i++) {         //---③

        ch = &(allData.data[i].letter.value);         //---④
        chNext = &(allData.data[i+1].letter.value);   //---⑤

        if (isFirstChar) {                            //---⑥

            if (isSmallLetter(*ch)) {                 //---⑦

                ChangeSmallToBigLetter(ch);
                isFirstChar = false;

            } else if (isBigLetter(*ch)) {            //---⑧

                isFirstChar = false;

            } else if (*ch == '0') { //숫자0          //---⑨
                if(isAlphabet(*chNext))
                    *ch = 'O'; //알파벳O

                isFirstChar = false;
            }
        } else {                                      //---⑩

            if (isBigLetter(*ch)) {                   //---⑪

                ChangeBigToSmallLetter(ch);
```

```
            } else if (*ch == '0') { //숫자0          //----⑫
                if(isAlphabet(*chNext))
                    *ch = 'o'; //알파벳o
            }

            if ((*ch == '.') || (*ch == 0x22)) {      //----⑬
                isFirstChar = true;
            }
        }
    }
}
```

① 문서 이미지에서 가장 처음에 시작되는 알파벳 문자는 대문자이므로 isFirstChar를 true로 설정한다. 문장의 첫 문자 이후에는 isFirstChar를 false로 설정하여 소문자로 변경 작업이 진행된다.

② 현재의 문자를 가리키는 포인터 변수 ch와 현재 문자의 바로 다음 문자를 가리키는 포인터 변수 chNext를 선언한다. 알파벳 'O'를 숫자 '0'으로 인식할 경우를 비교하기 위해서다. 현재의 문자가 숫자 '0'으로 인식되었는데, 바로 다음 문자가 알파벳이라면 대문자 'O'나 소문자 'o'로 변경해 주어야 한다. 즉, 일반적으로 문서에서 숫자 '0'은 다른 숫자와 이웃하여 있으므로 알파벳의 중간에 숫자 '0'이 있다면 알파벳이 숫자로 잘못 인식된 것이다. 또한, 현재의 문자를 포인터 변수인 ch로 만든 이유는 이후에 대문자와 소문자를 변경할 때 Call-By-Reference 방법을 사용하기 위해서다.

③ allData에 있는 모든 이미지 데이터를 처음부터 끝까지 대문자와 소문자의 변환 작업을 진행한다. 문서의 가장 마지막은 줄바꿈 문자고, 현재 문자의 다음 문자를 확인하기 때문에 for 문은 이미지 데이터의 마지막 바로 전까지만 진행한다.

④ 문자를 가리키는 포인터 변수인 ch는 현재 이미지 데이터의 문자 결과를 가리킨다. 이처럼 포인터 변수를 사용하는 것은 Call-By-Reference 방법을 사용하여 이후에 다른 함수의 매개변수로 전달하여 함수 실행 후의 결과값을 메모리상에서 변경하기 위해서다.

⑤ 다음 문자를 가리키는 포인터 변수인 chNext는 다음 문자의 이미지 데이터의 문자 결과를 가리킨다.

⑥ isFirstChar가 true면 현재 문자는 대문자로 만드는 과정을 진행한다.

⑦ 현재 문자가 소문자면 isSmallLetter(*ch)는 true고 현재 문자인 ch는 ChangeSmallToBigLetter(ch) 함수를 사용하여 대문자로 변경해야 한다. isSmallLetter(*ch)는 ch를 Call-By-Value로 전달하고, ChangeSmallToBig

Letter(ch)는 Call-By-Reference로 전달한다. 포인터를 정확히 이해한다면 Call-By-Value는 값을 복사하여 전달하는 것이고 Call-By-Reference는 메모리의 주소를 전달하여 ChangeSmallToBigLetter(ch) 함수에서 ch가 가리키는 메모리의 값을 변경하는 것을 이해할 수 있을 것이다.[03] 소문자를 대문자로 변경한 후에는 문장의 중간에 있는 문자이므로 isFirstChar를 false로 설정하여 소문자로 변경하는 작업을 진행한다.

⑧ 문장의 첫 문자가 대문자면 isFirstChar를 false로 설정하여 다음 문자부터는 소문자로 변환을 진행한다.

⑨ 첫 문자가 숫자 '0'이면 다음 문자가 알파벳인지를 검사한다. 다음 문자가 알파벳이면 숫자 '0'을 대문자 'O'로 변환하고, 다음 문자가 숫자면 숫자 '0'을 유지한다. 또한, 첫 문자를 검사하였기에 이후의 문자를 소문자로 변환하기 위해 isFirstChar을 false로 설정한다.

⑩ isFirstChar가 false인 경우에 진행되는 항목으로, 마침표나 큰따옴표 문자가 나오기 전까지 모든 알파벳을 소문자로 만든다.

⑪ 현재의 문자가 대문자인지를 검사하여 대문자면 소문자로 변환한다. ⑦에서와 마찬가지로 isBigLetter(*ch) 함수는 Call-By-Value를 사용하였으며, ChangeBigToSmallLetter(ch) 함수에서는 Call-By-Reference를 사용하여 ChangeBigToSmallLetter(ch) 함수가 종료된 후에는 포인터 변수인 ch가 가리키는 메모리의 값이 변경된다.

⑫ 첫 문자가 숫자 '0'이면 다음 문자가 알파벳 문자인지를 검사한다. 다음 문자가 알파벳이면 숫자 '0'을 소문자 'o'로 변환하고, 다음 문자가 숫자면 숫자 '0'을 유지한다.

⑬ 현재 문자가 마침표 또는 큰따옴표면 이후의 첫 알파벳 문자를 대문자로 만들기 위해 isFirstChar 변수를 true로 설정한다. 큰따옴표 문자의 ASCII 코드 16진수 값은 '0x22'다.

[코드 3-27]은 대문자를 소문자로 변환하거나 소문자를 대문자로 변환하는 함수를 구현한 것으로, [그림 3-20]의 내용을 프로그램으로 만든 것이다. [코드 3-27]은 대문자와 소문자의 변환 작업이 필요한 9개 알파벳의 변환을 구현하였다. 함수의 이름에서 알 수 있듯이 ChangeBigToSmallLetter( ) 함수는 대문자를 소

03 이 내용은 『소수와 RSA 알고리즘으로 배우는 Big Number 연산 – 구조체와 자료구조의 이해』(한빛미디어, 2014)에 자세히 설명되어 있다.

문자로 변환해 주는 함수고, ChangeSmallToBigLetter( ) 함수는 소문자를 대문자로 변환해 주는 함수다. 이 두 함수에서 중요한 것은 Call-By-Reference를 사용하여 매개변수인 value가 가리키는 메모리의 값을 직접 치환한다는 것인데, C/C++ 프로그램에서 중요한 내용인 포인터의 사용을 충분히 이해하고 있어야 한다.

[코드 3-27] New Data 구조체

```
void COCR::ChangeBigToSmallLetter(char *value) {              //---①

    if (*value == 'I')    //대문자'I'
        *value = 'l';     //소문자'l'
    else if (*value == 'C')
        *value = 'c';
    else if (*value == 'O')
        *value = 'o';
    else if (*value == 'S')
        *value = 's';
    else if (*value == 'U')
        *value = 'u';
    else if (*value == 'V')
        *value = 'v';
    else if (*value == 'W')
        *value = 'w';
    else if (*value == 'X')
        *value = 'x';
    else if (*value == 'Z')
        *value = 'z';
}

//-----------------------------------------------
void COCR::ChangeSmallToBigLetter(char *value) {              //---②

    if (*value == 'l')    //소문자'l'
        *value = 'I';     //대문자'I'
    else if (*value == 'c')
        *value = 'C';
    else if (*value == 'o')
        *value = 'O';
    else if (*value == 's')
        *value = 'S';
```

```
    else if (*value == 'u')
        *value = 'U';
    else if (*value == 'v')
        *value = 'V';
    else if (*value == 'w')
        *value = 'W';
    else if (*value == 'x')
        *value = 'X';
    else if (*value == 'z')
        *value = 'Z';
}
```

① 대문자를 소문자로 치환하는 함수다. 문서 이미지의 문자형이 Arial이므로 대문자 'I'와 소문자 'l'은 같은 이미지 데이터를 가진다. 문자가 대문자 'I'이라면 소문자 'l'로 변경해야 한다. 나머지 8개의 문자는 대문자와 소문자가 같은 이미지 데이터로 저장되기 때문에 각 문자를 검사하여 대문자면 소문자로 변환해 준다.

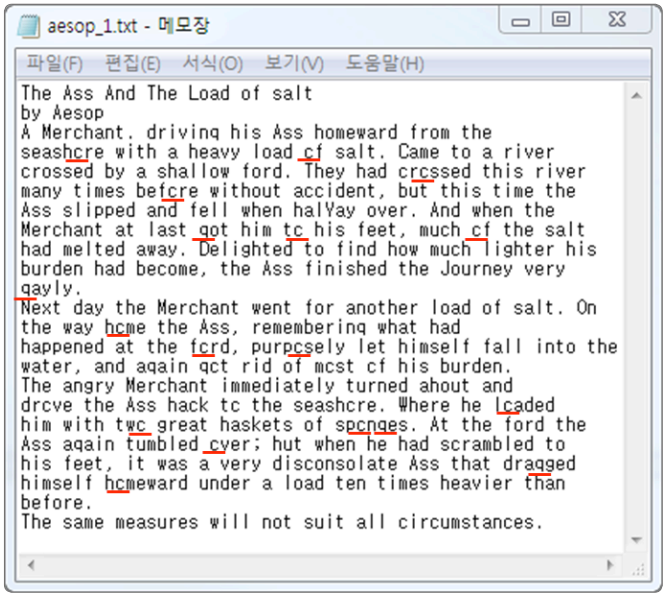
② 소문자를 대문자로 치환하는 함수다. 소문자 'l'은 이미지 데이터가 같은 대문자 'I'로 치환하고, 나머지 8개의 알파벳은 자신의 소문자를 대문자로 변환한다.

지금까지 구현한 프로그램은 인식의 거의 모든 부분을 보여 준다. OCR 프로그램을 구현하려면 지금까지의 내용을 응용하여 인식률을 높일 수 있게 추가로 구현하면 된다. 지금까지의 내용을 이해했다면 충분히 추가로 구현할 수 있을 것이다.

[그림 3-22]는 첫 번째 문서 이미지의 인식 결과로, 특정 문자에서 인식률이 높지 않다. 'o'를 'c'로 인식하고, 'g'를 'q'로 인식한다. 지금까지 구현한 프로그램에서 이 부분을 해결하기 위해 추가로 구현해야 할 것이 있을까?

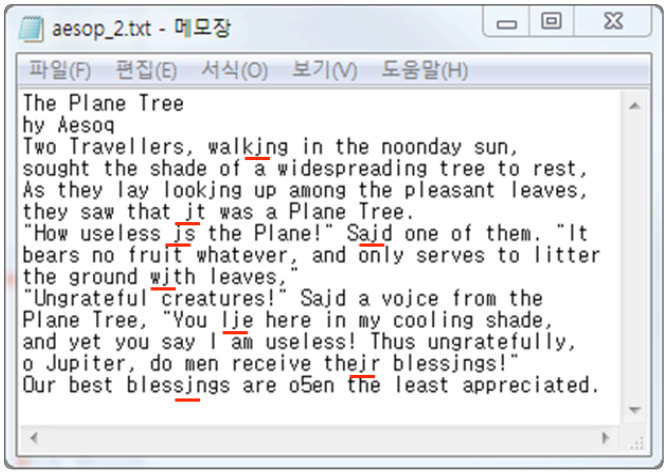
이 문제를 해결하기 위해 다음 장에서 설명하는 문서 이미지의 픽셀에 대하여 이해할 필요가 있다. 문서 이미지에서 문자는 검은색이고 바탕은 흰색이지만, 문자와 바탕의 경계선은 검은색과 흰색으로 정확히 구분되지 않는다. 확대해 보면 경계선에 회색 픽셀이 분포하고 있음을 알 수 있다. 즉, 문자와 바탕의 경계선이 정확히 구분되지 않아서 이러한 결과를 가져온 것이다. 다음 절에서는 이 부분을 어떻게 해결하는지를 자세히 설명하겠다.

그림 3-22 대문자와 소문자 변환 결과_1 (aesop_1.txt)



[그림 3-23]에서는 'i'를 'j'로 인식하였다.

그림 3-23 대문자와 소문자 변환 결과_2 (aesop_2.txt)

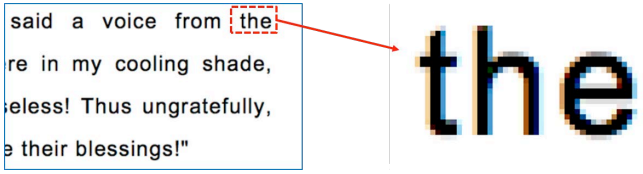


3.7 문자 색의 범위 조정으로 인식률 높이기

앞에서 기본적인 방법을 모두 구현하였지만, 특정 문자의 인식이 제대로 이루어지지 않았다. [그림 3-24]를 살펴보자. 왼쪽을 보면 문자 부분만 검은색으로 나오는 것처럼 보인다. 그러나 'the' 문자를 확대하면 오른쪽과 같이 경계 부분에서 검은색도 흰색도 아닌 픽셀들이 존재함을 알 수 있다. 즉, 지금까지 만든 프로그램이 잘못된 것이 아니라 이미지의 특성상 이미지 파싱이 명확히 이루어지지 않아서 인식률이 떨어졌던 것이다. 물론 문서 이미지의 문자가 크다면 경계 부분의 픽셀에 대한 문제점이 인식률에 크게 영향을 미치지 않는다. 하지만 대부분의 문서 이미지 문자는 작은 편이어서 경계선의 색이 인식률에 크게 영향을 미친다.

그렇다면 이 문제를 어떻게 푸는 것이 좋을까? 이는 문자 색인 검은색의 범위를 확대하는 것으로 해결하면 된다. 즉, 경계선의 회색도 문자 색으로 판단하는 프로그램을 만들면 된다.

그림 3-24 문자와 바탕의 경계 픽셀은 회색



[코드 3-28]은 'OCR.h' 헤더 파일의 윗부분에 정의된 상수다. RANGE_OF_COLOR_TO_CHECK 상수의 값을 130으로 변경하였는데, 이 상수는 문자 색의 범위를 넓이는 상수다. 이 값을 100에서 130으로 증가하여 문자 색으로 인식하는 범위를 더 넓게 하여 경계선의 회색도 일정한 값을 넘으면 문자 색으로 인식하게 변경한 것이다. 문자 색의 범위를 넓히는 방법만으로도 인식률은 상당히 높아진다.

[코드 3-28] 문자 색의 범위를 130으로 증가(OCR.h)

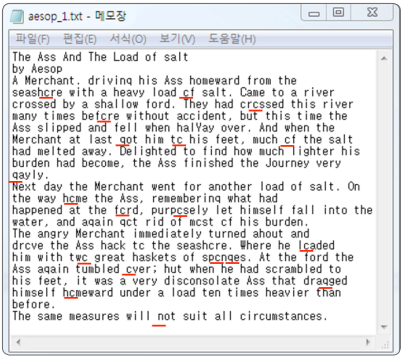
```
#define MAX_COUNT_STANDARD          80
#define MAX_COUNT_DATA              2000
```

```
#define RANGE_OF_COLOR_TO_CHECK      130          //---①

#define RATE_START_FOR_PARSING       0.1
#define RATE_END_FOR_PARSING         0.3
```

① 문자 색으로 판단하는 범위를 100에서 130으로 증가한다. 추가된 함수가 없음에도 인식률이 향상된다.

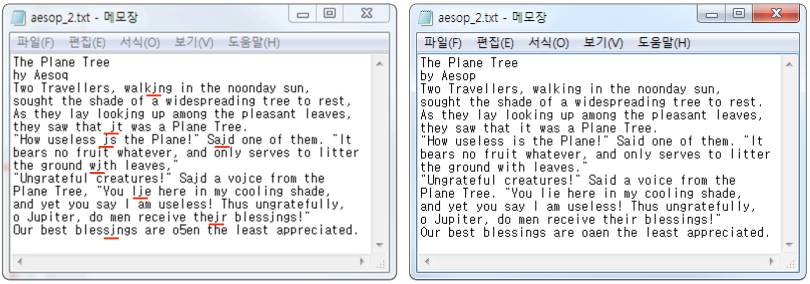
[그림 3-25]와 [그림 3-26]은 문자 색의 범위를 넓힌 결과다. 문자 색의 범위의 변경만으로도 인식률이 상당히 향상되어 이전에 특정 문자들이 인식 안 되는 문제가 해결하였었다. 대부분 문서 이미지는 읽기에 불편함이 없더라도 문자를 확장해 보면 경계선에서 문자와 바탕의 중간색으로 된 경우가 많아 문자 색의 범위를 넓게 해주는 것이 중요하다.

그림 3-25 색의 범위를 확대하여 인식률을 높인 결과(aesop_1.txt)



o를 c로 인식, g를 q로 인식　　색의 범위를 확대하여 인식률을 향상한 결과

그림 3-26 색의 범위를 확대하여 인식률을 높인 결과(aesop_2.txt)



인식을 위한 기본적인 부분은 이번 장에서 모두 구현되었다. 지금까지 프로그램을 이해한 독자라면 OCR 프로그램을 어떻게 만들어야 하는지를 알게 되었을 것이다. 이제 문서 이미지의 특성에 맞추어 인식에 필요한 추가 부분만 구현하면 된다.

현재 OCR 프로그램은 주차장에서 사진을 찍어서 차량 번호를 확인하는 것에 가장 많이 활용되고 있다. 그 외에도 다양한 방법으로 응용이 가능하다. 예를 들어, 신분증을 복사하는 일이 잦은 관공서나 은행에서 신분증만을 스캔하는 기계를 만들어 그 내용을 판독하는 데 활용하는 것, 경찰이 신분증이나 차량 사진을 찍어 신원 조회를 하는 프로그램을 만드는 것 등도 좋을 것이다.

소프트웨어는 우리의 삶에 편리함과 재미를 주는 쪽으로 발전해 왔으며, 앞으로도 재미있는 프로그램이 우리의 삶을 즐겁게 해 줄 것이라 확신한다. 또한, 많은 직업이 생겨나고 없어지는 상황에서 프로그래머의 수요는 항상 늘어날 것이다. 실력을 갖추어 자신의 몸값을 높인다면 전문직으로서 오랫동안 일할 수 있는 시장이 형성될 것이라 확신한다. 앞으로 5년 안에 우리나라에서 프로그래머에 대한 대우는 상당히 좋아질 것이다. 프로그래밍이 단시간에 익힐 수 있는 것이 아니며, 고급 프로그래머는 많은 시간 동안 고통을 통해 만들어질 수밖에 없기 때문에 사회에서 더 많은 고급 프로그래머를 요구한다면 소프트웨어 산업도 서서히 변화될 것이다.

chapter 4

자동차 번호판 인식

요즘 쇼핑몰이나 대형 건물의 주차장에 설치된 번호판 인식 시스템을 쉽게 볼 수 있다. 차량의 정면에서 사진을 찍어서 번호판의 숫자와 문자를 인식하는 것이다. 이 책에서는 이해를 위해 핵심적인 부분만을 설명하였으나, 실제 번호판 시스템도 필자의 방법과 비슷할 것이라 생각한다. 물론 실제 시스템에서는 인식률을 높이기 위해 훨씬 복잡한 프로그래밍이 진행되었을 것이다.

[그림 4-1]은 한 주차장의 주차 위치 확인 시스템의 화면으로, 이 주차장은 곳곳에 주차 위치를 확인하는 모니터가 있고, 자신의 차량이 어떻게 사진에 찍혔는지를 확인할 수도 있다. 오래전에 필자가 이 주차장을 방문했을 때는 자동차 앞면 사진에서 글자는 흰색이고 나머지는 검은색으로 나타나는 사진을 봤었다. 그 사진은 특수 카메라로 찍은 사진이었다고 판단되며, 실제 사진을 활용한 것보다 이미지 인식 프로그래밍이 조금 더 쉬웠을 것이다.

그림 4-1 주차 위치 확인 시스템

지금까지 이미지 인식 프로그램을 이해한 독자라면 이미지 인식에서 가장 힘든 부분이 이미지 파싱이라는 것을 알 것이다. 자동차 번호판 인식에서도 이미지 파싱 부분만 이해하면 이미지 인식의 원리는 동일하므로 나머지 부분은 이전 장에서와 같은 인식 방법을 사용하면 된다.

모든 번호판을 인식하려면 프로그램이 상당히 복잡해지기 때문에 이 책에서는 최대한 단순화하여 한 가지 형태의 번호판만을 인식할 예정이다. 따라서 최근 번호판 형태 즉, 글자는 검은색이고 바탕은 흰색인 번호판만 활용하겠다.

현재는 고가의 번호판 인식 시스템이 대중화되었지만 자동차 번호판 인식을 응용하여 스마트폰 애플리케이션을 만든다면 다양하게 응용될 수 있다. 예를 들어, 스마트폰이 대중화되었으므로 교통경찰관이 자신의 스마트폰을 사용하여 차량 사진을 찍고, 경찰청 시스템에서 차량의 정보를 받아 단속에 쉽게 사용할 수 있을 것이다. 비용적인 측면에서도 모든 교통경찰관이 활용할 수 있으려면 기기 개발보다 시스템 개발이 쉬울 것이다. 하드웨어를 대체할 수 있는 소프트웨어는 비용 절감은 물론 친환경적이라고 볼 수 있다. 생산에 있어서 공간을 차지하지 않고 폐기물이 없기 때문이다.

4.1 사진 크기 변경 및 흑백 사진 만들기

스마트폰으로 찍은 사진은 생각보다 크기가 크다. 프로그램에서 사용하는 자동차 번호판 사진을 2011년에 아이폰4 카메라로 찍었는데 해상도가 Full-HDTV보다 더 높았다. 번호판 사진에서 번호판의 크기가 작다면 모든 픽셀을 가지고 이미지 인식을 진행해도 되지만, 크기가 큰 사진은 픽셀 수가 많기 때문에 사진의 크기를 줄이고 이미지 인식을 진행하는 것이 좋다. 또한, 이미지 인식에서 이미지 데이터는 0과 1로만 이루어진 구조체를 만드는 것이기 때문에 흑백 사진으로 바꾸어서 검은색과 흰색으로 문자와 바탕을 구분하는 것도 좋다.

자동차 번호판 인식에서 사용하는 사진들은 사이즈가 큰 컬러 사진이므로 이번 절에서는 사진의 크기를 줄이고 흑백 사진을 만들어서 사진을 단순화시키는 작업을 진행하겠다. 사진의 크기를 줄여도 인식 결과에 영향을 주지 않는다면 사진의 크기를 줄이는 것이 처리 속도를 빠르게 할 수 있는 가장 좋은 방법일 것이다.

[그림 4-2]는 스마트폰으로 찍은 번호판 사진의 크기를 줄이고 흑백으로 만든 결과다. 약 1/6 정도로 사진의 크기를 줄였음에도 사진의 이미지는 깨지지 않았다. 작은 이미지를 크게 만들 때 이미지가 깨지는 것은 어쩔 수 없지만, 큰 이미지를 아주 작게 만드는 것이 아니라면 이미지를 눈으로 보는 것에는 큰 영향을 주지 않는다. [그림 4-2]에서 왼쪽의 원본 이미지는 흰색 차여서 컬러 사진임에도 흑백 사진과 구분하기가 힘들지만, 앞으로 구현되는 프로그램 코드를 통하여 흑백 사진으로 어떻게 만들게 되는지를 알게 될 것이다.

그림 4-2 사진 크기를 줄이고 흑백 사진으로 만든다

[코드 4-1]은 번호판 인식의 설정값, 구조체와 클래스의 구조를 보여 준다. 이전 장에서 설명한 문서 OCR의 내용과 비슷하다. 문서 OCR에서는 영문과 숫자를 함께 기본 이미지 데이터에서 사용하였지만, 자동차 번호판 OCR에서는 숫자와 문자를 나누어서 두 개의 기본 이미지 데이터를 만들었다는 데에 차이가 있다. 자동차 번호판은 숫자와 문자의 위치가 정해져 있으므로 숫자 이미지는 숫자 기본 이미지 데이터와만 비교하는 것이 실행 속도에서 가장 좋기 때문이다. 이렇게 특정 위치에 특정 문자가 있는 경우에는 비교하는 데이터를 정해 놓는 것이 좋다. 번

호판의 문자는 한글이기 때문에 번호판에 쓰인 한글을 모두 기본 데이터 이미지에서 담게 되면 비교해야 할 개수가 많아진다. 다행히도 최근의 번호판은 한글이 한 글자만 존재하기 때문에 여러 글자일 때보다 이미지 인식 속도가 빨라진다.

[코드 4-1] 번호판 인식의 설정 값 및 정의된 구조체와 COCR 클래스(OCR.h)

```
#define MAX_COUNT_STANDARD          80          //—①
#define MAX_COUNT_DATA              7           //—②
#define RANGE_OF_COLOR_TO_CHECK     80          //—③
#define RATE_START_FOR_PARSING      0.3         //—④
#define RATE_END_FOR_PARSING        0.7         //—⑤

struct Point {
    int x;
    int y;
};

struct Rect {
    Point start;
    Point end;
};

struct Letter {
    CString value;
    unsigned int image[48];
};

struct StandardNumber {                         //—⑥
    int count;
    Letter letter[10];
};

struct StandardLetter {                         //—⑦
    int count;
    Letter letter[MAX_COUNT_STANDARD];
};

struct Data {
    Letter letter;
    Rect rect;
};

struct AllData {
    int count;
```

```
    Data data[MAX_COUNT_DATA];
};

//------------------------------------------------------------------
class COCR {

private:
    CImage *image;                                          //---⑧
    StandardNumber standardNumber;                          //---⑨
    StandardLetter standardLetter;                          //---⑩
    AllData allData;
    Data *data;

    int colorToCheck;

public:
    COCR(void);
    ~COCR(void);

    void RunOCR(CImage *newImage);                          //---⑪
    CImage * ResizeInBlackWhiteImage(CImage *newImage);     //---⑫
};
```

① 기본 이미지 데이터의 최대 개수를 정의한 것이다. 프로그램에서 한글 문자 1개의 총 가짓수를 80개까지 만들 수 있다. 총 개수가 많아지면 이 값을 증가시키면 된다.

② 자동차 번호판에서 인식하기 위한 문자 이미지의 총 개수는 7개다.

③ 인식을 위해 글자로 인식할 범위를 정한 것이다. 검은색은 0의 값을 갖기 때문에 실제 글자로 인식하는 범위는 255개의 값 중에서 0~80까지의 값이다.

④ 사진의 이미지 파싱을 위한 시작 범위는 왼쪽에서 30%의 위치부터 시작한다.

⑤ 사진의 이미지 파싱을 위한 종료 범위는 왼쪽에서 70%까지 픽셀 값을 확인한다.

⑥ 숫자를 위한 기본 이미지 데이터는 10개의 `Letter` 구조체를 가지고 있다.

⑦ 문자를 위한 기본 이미지 데이터 구조체를 정의하였으며, 한글 문자를 이미지 데이터로 가지고 있다.

⑧ 인식을 위한 번호판 사진의 클래스를 가리키는데, 원본 이미지가 아니라 축소 작업을 진행한 흑백 사진의 번호판 이미지의 클래스를 가리킨다.

⑨ 숫자 기본 이미지 데이터를 가지고 있는 변수이며, 프로그램에서 숫자 기본 이미지 데이터는 이 변수를 사용하여 접근한다.

⑩ 문자 기본 이미지 데이터를 가지고 있는 변수이며, 프로그램에서 문자 기본 이미지 데이

터는 이 변수를 사용하여 접근한다.

⑪ [OCR] 버튼을 클릭하면 COCR 클래스의 RunOCR( ) 함수가 실행된다.

⑫ 원본 이미지를 가지고 이미지를 축소하고 흑백 사진으로 변환하는 함수로, 변환된 이미지를 반환한다.

[코드 4-2]에서는 사진의 크기를 변경하고 흑백 사진으로 만드는 것을 구현하였는데, 이 두 작업은 ResizeBlackWhiteImage( ) 함수 안에서 동시에 진행하였다. 작은 사이즈의 이미지에 저장하기 전에 흑백 사진으로 만드는 것이 효율적이기 때문에 한 함수에서 구현하였다.

이 책의 부록에 사진 크기를 변경하는 프로그램(A.1 사진 크기 변경하기)과 흑백 사진으로 만드는 프로그램(A.3 흑백 사진 만들기)을 구분하여 설명하였다. 사진의 크기를 축소하는 방법은 2장의 [그림 2-18]에서 설명한 것과 비슷한 방법을 사용한다. 가로와 세로를 기준으로 일정한 간격으로 픽셀을 건너뛰면서 해당 픽셀을 읽고 작은 크기의 사진에 차례대로 넣으면 된다. 흑백 사진을 만드는 방법은 규칙이 있으며, RGB의 개별적인 세 값으로 하나의 값을 만들어서 세 개(Red, Green, Blue)의 위치에 넣으면 된다. RGB의 세 가지 색을 가지고 한 가지 값을 만드는 것은 다음과 같다. 세 가지 색이 흑백 사진에서 영향을 주는 비율이라고 이해하면 된다.

그림 4-3 흑백 사진에 영향을 미치는 RGB 비율

- Red : 30%
- Green : 59%
- Blue : 11%

➔ ((Red * 30) + (Green * 59) + (Blue * 11)) / 100

흑백 사진에 영향을 미치는 RGB 비율에서 Green이 59%이기 때문에 Green의 값이 변하면 흑백 사진의 색에 다른 값보다 많은 변화를 주게 된다. 반대로 Blue의 값은 11%이므로 거의 영향을 안 준다고 이해하면 된다.

[코드 4-2] 인식 속도를 위해 사진의 크기를 줄이고 흑백 사진으로 만든다.(OCR.cpp)

```
void COCR::RunOCR(CImage *newImage) {                    //---①
```

```
    image = ResizeInBlackWhiteImage(newImage);                          //---②

    image->Save("resizeBlackWhite.jpg", Gdiplus::ImageFormatJPEG);      //---③
}

CImage * COCR::ResizeInBlackWhiteImage(CImage *newImage) {              //---④

    static CImage image;                                                //---⑤

    COLORREF rgb;
    BYTE byte;

    int width = 400;                                                    //---⑥
    int height = 300;

    float xRate = (float)(newImage->GetWidth()) / width;                //---⑦
    float yRate = (float)(newImage->GetHeight()) / height;

    if (image != NULL)                                                  //---⑧
        image.Destroy();

    image.Create(width, height, 32);                                    //---⑨

    for (int i=0; i<width; i++) {
        for(int j=0; j<height; j++) {
            rgb = newImage->GetPixel((int)(xRate * i), (int)(yRate * j));//---⑩

            byte = (GetRValue(rgb) * 30 +                               //---⑪
                    GetGValue(rgb) * 59 +
                    GetBValue(rgb) * 11) / 100;

            rgb = RGB(byte, byte, byte);                                //---⑫

            image.SetPixel(i,j, rgb);                                   //---⑬
        }
    }

    return &image;                                                      //---⑭
}
```

① [실행] 버튼을 클릭하면 실행되는 함수로, 사진의 크기를 축소하고 흑백 사진으로 만든다. 이번 절에서는 축소된 사진을 확인하기 위해 'resizeBlackWhite.jpg' 파일에 저장하지만, 이후부터 추가되는 함수는 이 함수에서 실행하게 된다.

② 원본 사진을 축소하고 흑백 사진을 만드는 함수인 `ResizeBlackWhiteImage( )` 함수를 실행한 후 결과 이미지 클래스를 결과값으로 받아서 포인터 변수인 `image`가 가리키게 된다.

③ 결과 이미지를 확인하기 위해 'resizeBlackWhite.jpg' 파일에 저장한다.

④ 원본 사진을 축소하고 동시에 흑백 사진을 만드는 함수다.

⑤ 결과 이미지를 저장하기 위한 CImage 클래스로, static으로 정의하였기 때문에 프로그램이 종료될 때까지 image 변수는 메모리에서 지워지지 않는다.

⑥ 크기가 큰 원본 사진은 가로와 세로가 '400×300' 크기의 이미지로 축소된다.

⑦ 사진의 크기를 줄이기 위해서는 원본 사진에서 일정한 간격으로 건너 뛰면서 픽셀을 가져와야 한다. xRate는 가로의 간격 값을 갖고, yRate는 세로의 간격 값을 가진다.

⑧ image가 이전에 정의되었는지를 확인한 후 이전에 정의되었다면 NULL이 아니므로 image 클래스를 메모리에서 지워야 한다. 프로그램이 종료되기 전에 [OCR] 버튼이 여러 번 클릭되어 재실행되어도 문제없이 프로그램이 동작하게 하기 위해서다.

⑨ 축소한 이미지를 저장할 작은 사이즈의 CImage 클래스를 생성한다. 클래스가 생성된다는 것은 '400×300' 크기의 이미지 클래스가 메모리에 올라가는 것을 의미한다.

⑩ 일정한 간격으로 픽셀을 가져온다. 픽셀을 가져온다는 것은 해당 픽셀의 RGB 값을 얻는 것을 의미한다.

⑪ RGB의 세 가지 색인 Red, Green, Blue의 값을 rgb 변수에서 얻어 와서 흑백 사진의 rgb 값에 넣을 하나의 값을 계산한다.

⑫ 흑백 사진을 만들기 위한 RGB는 세 가지 값인 Red, Green, Blue의 값이 모두 동일한 것을 의미한다. 따라서 동일한 값을 넣어서 rgb를 새로운 값으로 만들어 준다.

⑬ 축소를 위해 생성된 image의 해당 위치의 픽셀에 흑백 사진의 rgb 값을 넣는다.

⑭ image는 축소된 흑백 사진의 이미지를 가지고 있으며, 메모리에 저장된 이 결과값의 메모리 주소를 반환하게 된다. image 변수는 static으로 저장되었으므로 함수가 종료되어도 메모리에서 지워지지 않는다.

이미지 인식에서 문자가 크다면 프로그램의 실행 속도를 빠르게 하기 위해 이미지를 축소하는 것이 필요하다. 물론 사이즈를 줄이지 않고 프로그램을 만들어서 처리하는 방법도 있지만, 코드의 구현이 복잡해질 수 있다. 이번 절을 이해한 독자라면 사진을 줄이고 흑백 사진을 만드는 것이 얼마나 간단한지를 알게 되었을 것이다.

4.2 문자 이미지 추출

자동차 번호판에서 문자 이미지를 얻기 위한 이미지 추출Image Parsing은 사진의 특성을 이용한다. 즉, 사진에서 문자 이미지의 공통되는 특징을 파악하여 문자 부분만을 발췌해야 한다. [그림 4-4]는 번호판 사진에서 문자 이미지를 얻은 결과로, 7개의 개별 문자를 가져온다. 현재 발급되는 번호판은 6개의 숫자와 1개의 한글로 이루어져 있다(일부 번호판은 일부러 보이지 않게 함).

그림 4-4 번호판 이미지 파싱 결과

이 책에서 다루는 프로그램은 모든 번호판을 완벽히 이미지 파싱을 진행하지는 못한다. 교육의 목적으로 한글 문자를 파싱하는 방법을 최대한 간단히 구현하려고 했기 때문이다. 실제 상용 프로그램을 만들기 위해서는 이 책에서 설명하는 것보다 추가적인 작업을 많이 진행해야 한다.

[그림 4-5]를 보면 번호판의 문자 이외에 차량의 다른 위치에도 검은색과 흰색이 나타나는 것을 알 수 있다. [그림 4-5]에서 왼쪽 사진은 테두리가 검은색이어서 문자 색으로 인식하게 되고, 오른쪽 사진은 번호판에 벌레와 같은 이물질이 붙

어서 작게 검은색으로 인식하게 된다. 이러한 경우가 종종 발생하는데, 번호판 인식의 오류를 만드는 요인이다. 이를 해결하기 위해 번호판 문자의 특징을 살펴보자. 사진에서의 공통된 형식은 글자 크기가 일정하다는 것, 글자는 차례대로 배열된다는 것 등이 있다. 문자의 검은색과 문자가 아닌 위치에 있는 검은색을 구분하려면 문자 이미지의 크기를 비교하면 된다. 따라서 2장의 문서 인식과는 달리, 글자의 크기에 따라서 글자인지 아닌지를 판단하는 알고리즘을 추가해야 한다.

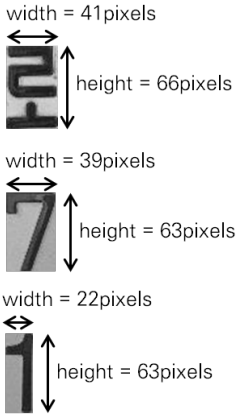
그림 4-5 이미지 파싱에서 문자로 인식할 수 있는 오류

번호판 테두리가 문자로 인식되는 오류

벌레가 번호로 인식되는 오류

[그림 4-6]은 [그림 4-5]에서 발생하는 오류를 해결하는 방법이다. 번호판 문자의 특징을 살펴보면 첫 번째는 숫자를 포함한 모든 문자는 세로의 길이가 가로의 길이보다 길다는 점이다. 두 번째는 세로가 가로의 길이에 4를 곱한 값보다 작다는 점이다. 이 점은 [그림 4-5]의 왼쪽 사진의 테두리를 글자로 인식하지 않으면서 1과 같이 가로보다 세로가 훨씬 긴 숫자를 인식하는 데 꼭 필요한 조건이다. 마지막으로 번호판은 사진 안에서 꽉 차게 찍히기 때문에 인식을 위한 사진의 축소를 진행한 후에도 세로의 길이는 항상 50픽셀보다 크다는 점이다. 이와 같이, 문자의 크기와 비율을 가지고 문자인지 판단하는 알고리즘을 이미지 파싱을 진행하면서 꼭 추가해야 한다.

그림 4-6 번호판 문자로 인식하기 위한 크기의 조건



번호판 문자의 가로와 세로 법칙

1. 세로가 가로보다 크다.(height 〉 width)
   예: 모든 문자가 동일하다.
2. 세로가 가로의 4배보다 작다.(height 〈 4 × width)
   예: 가장 큰 비율은 숫자 '1'
3. 견본 사진의 특성상 세로는 50픽셀보다 크다.(height 〉 50)

[코드 4-3]은 OCR 프로그램을 실행할 때 생성되는 생성자와 프로그램 화면에서 [OCR] 버튼을 클릭하였을 때 실행되는 RunOCR( ) 함수를 구현한 코드다. RunOCR( ) 함수에 추가된 항목은 이미지 파싱을 진행을 시작하는 ParsingStepFirst( ) 함수와 개별 문자 이미지의 결과를 각각의 사진 파일로 저장하는 PrintImageToFile( ) 함수다. 이미지 파싱은 사진의 특성을 잘 파악해야 하며, 기본 원리는 2장에서 진행된 이미지 파싱과 크게 다르지 않다.

[코드 4-3] COCR 생성자와 OCR 버튼을 선택했을 때 실행되는 RunOCR( ) 함수(OCR.cpp)

```
COCR::COCR(void) {
    colorToCheck = 0;                                    //----①
}

void COCR::RunOCR(CImage *newImage) {
    image = ResizeInBlackWhiteImage(newImage);           //----②

    ParsingStepFirst();                                  //----③

    for (int i=0; i<MAX_COUNT_DATA; i++)
        PrintImageToFile(i, &allData.data[i].rect);      //----④
}
```

① 프로그램을 실행할 때 COCR 클래스가 생성되며, 이때 문자 영역의 색인 검은색의 값으로

colorToCheck이 설정된다. 색의 범위는 1-byte의 영역인 0~255 사이인데, 0은 검은색을 의미한다.

② 이전 절에서 진행한 사진의 크기를 줄이고 흑백 사진으로 만드는 함수다.

③ 이번 절에서 추가된 이미지 파싱을 진행하는 함수다. 1단계를 진행하는 함수를 실행하여 이후에 2단계와 3단계를 차례대로 수행하여 결과를 가져온다.

④ 테스트를 위해 7개의 개별 문자 이미지를 사진 파일에 차례로 저장한다.

이 책에서 사용하는 자동차 번호판은 번호판만을 사진으로 크게 담아 번호판의 문자를 가운데 정렬하였다. 이미지 파싱의 첫 번째 단계로 문자가 배열된 위와 아래의 경계선을 찾아야 하는데, 사진의 중간을 기준으로 위와 아래의 경계선을 찾아가면 된다. [그림 4-7]은 문자가 있는 곳에서 세로선(Y축)의 가운데를 기준으로 시작하여 위와 아래의 경계선을 찾는 방법이다. 먼저 사진의 중앙선에서 시작하여 위로 한 줄의 픽셀을 차례대로 검사하여 문자 색(검은색)이 없는 바탕의 줄이 나올 때까지 진행한다. 또한, 마찬가지 방법으로 아래쪽 경계선을 찾으면 된다.

그림 4-7 이미지 파싱 1단계는 사진의 가운데에서 시작한다.

[코드 4-4]는 이미지 파싱의 1단계다. 기본 알고리즘은 2장에서 설명한 이미지 파싱과 동일하다. 다른 점이 있다면, 문서 이미지에서는 기울어지지 않은 문서를 기준으로 했기 때문에 한 픽셀이라도 문자 색(검은색)이 나오면 문자 선으로 인식했지만, 번호판 사진은 조금이라도 기울어져 있기 때문에 한 픽셀 줄에서 3개 이상 문자 픽셀이 나와야만 경계선으로 인식하게 구현하였다는 점이다. 또한, 자동차는 도로에서 달리기 때문에 이물질이나 벌레 등으로 인하여 번호판 사진이 깨끗

하지 않아 1 픽셀이 검은색이라고 하여 문자 선으로 인식하는 것은 좋지 않다. 이미지 파싱의 진행은 OCR의 시작이기 때문에 함수가 시작될 때 이미지 데이터 결과값을 저장하는 value 변수를 '\0'(NULL) 값으로 초기화한다. [코드 4-4]는 [그림 4-7] 이미지 파싱 1단계의 진행 방법을 차례대로 구현한 것이다.

[코드 4-4] 번호판 이미지 파싱 1단계(OCR.cpp)

```
void COCR::ParsingStepFirst() {

    int i, x, y, count;
    COLORREF rgb;
    int yTop, yBottom;
    bool isLetterLine;

    allData.count = 0;
    data = &allData.data[0];
    for(i=0; i<MAX_COUNT_DATA; i++) {
        allData.data[i]letter.value = "\0";              //---①
    }

    int xMax = image->GetWidth();                        //---②
    int yMax = image->GetHeight();
    int yMiddle = (int)(yMax / 2);

    int xStart = (int)(xMax * RATE_START_FOR_PARSING);     //---③
    int xEnd = (int)(xMax * RATE_END_FOR_PARSING);

    //---------- 문자가 있는 곳의 위 경계선을 찾는다. ----------
    for (y=yMiddle; y>0; y--) {                          //---④

        isLetterLine = false;
        count = 0;

        for (x=xStart; x<xEnd; x++)   {
            rgb = image->GetPixel(x,y);

            if (GetGValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                count += 1;                              //---⑤
            }
            if (count > 3) {                             //---⑥
                isLetterLine = true;
                break;
            }
        }
```

```
        if (!isLetterLine) {                          //---⑦
            yTop = y+1;
            break;
        }
    }

    //---------- 문자가 있는 곳의 아래 경계선을 찾는다. ----------
    for (y=yMiddle; y<yMax; y++) {                    //---⑧

        isLetterLine = false;
        count = 0;

        for (x=xStart; x<xEnd; x++)   {

            rgb = image->GetPixel(x,y);

            if (GetGValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                count += 1;                           //---⑨
            }
            if (count > 3) {                          //---⑩
                isLetterLine = true;
                break;
            }
        }

        if (!isLetterLine) {                          //---⑪
            yBottom = y-1;
            break;
        }
    }

    ParsingStepSecond(yTop, yBottom);                 //---⑫
}
```

① 번호판의 문자 이미지는 7개며, 이미지의 결과값을 저장하는 alldata 안의 문자 값을 '\0'(NULL) 값으로 초기화한다.

② xMax는 사진의 가로 길이를 저장하고, yMax는 세로 길이를 저장하며, yMiddle은 사진의 세로축의 중간값을 가진다.

③ xStart는 시작 위치를 나타내는데, RATE_START_FOR_PARSING이 0.3이기 때문에 가로축의 왼쪽에서 30%의 위치에서 픽셀을 읽기 시작한다. xEnd는 종료 위치를 나타내며, 왼쪽에서 70%의 위치까지 픽셀을 읽기 위해 사용한다.

④ 이미지 파싱 1단계에서 가장 먼저 진행하는 것은 위 경계선을 알기 위해서다. Y축을 기

준으로 중간에서 시작하여 y 값을 1씩 감소하여 경계선을 위로 증가시키면서 글자 선을 확인한다.

⑤ 하나의 픽셀이 검은색이면 문자를 포함한 픽셀이기 때문에 count를 하나 증가시킨다. 하나의 픽셀이 검은색이라고 곧바로 문자 선으로 인식하는 것은 좋지 않다.

⑥ 검사를 하는 현재의 픽셀 선에서 검은색 픽셀이 3개 이상이면 문자 라인으로 인식하고 isLetterLine을 true로 설정하여 문자 선임을 결정하고 for 문을 빠져 나온다.

⑦ 위 경계선을 찾는 것은 문자 라인에서 시작하는 것이므로 문자 라인이 아닌 바탕 라인이라면 경계를 찾는다. 그래서 현재의 Y축 값이 바탕 라인이라면 바로 아래 라인인 y+1이 위 경계선이 된다.

⑧ 아래 경계선을 찾는 코드다. Y축의 중간에서 시작하여 한 픽셀씩 아래로 내려가며 바탕 라인이 나오면 아래 경계선을 찾은 것이다. 아래 경계선을 찾는 과정은 ④부터 ⑦까지의 설명과 같다.

⑨ 아래 경계선을 찾는 과정에서 바탕 라인에 도달하면 바로 위의 Y축 라인이 아래 경계선이기 때문에 y-1의 값이 아래 경계선이 된다.

⑩ 이미지 파싱 1단계에서 위와 아래의 경계선이 정해졌기 때문에 2단계를 진행한다.

[코드 4-5]는 2장에서 구현한 이미지 파싱 2단계인 [코드 2-10]과 거의 동일하다. [코드 2-10]과 다른 점은 문자 선을 인식할 때 3개 이상의 픽셀들이 검은색일 때 문자 라인으로 판단한다는 점이다. 코드 내용은 [코드 2-10]에서 자세히 설명하였으므로 생략한다.

[코드 4-5] 번호판 이미지 파싱 2단계(OCR.cpp)

```
void COCR::ParsingStepSecond(int yTop, int yBottom) {

    int xMax = image->GetWidth();

    int x, y, count;
    COLORREF rgb;
    bool isLetterLine;
    bool flagPrevLine;

    flagPrevLine = false;

    for (x=0 ; x<xMax; x++) {
        isLetterLine = false;
        count = 0;
```

```
        for (y=yTop; y<=yBottom; y++) {
            rgb = image->GetPixel(x,y);

            if (GetGValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                count += 1;
            }
            if (count > 3) {                                    //---①
                isLetterLine = true;
                break;
            }
        }

        if (isLetterLine) {
            if (!flagPrevLine) {
                data->rect.start.x = x;
                data->rect.start.y = yTop;
            }
        } else {
            if (flagPrevLine) {
                data->rect.end.x = x-1;
                data->rect.end.y = yBottom;

                ParsingStepThird(&data->rect);                  //---②

                if (IsLetterRect(&data->rect)) {                //---③
                    allData.count += 1;
                    data = &allData.data[allData.count];
                }

                if (allData.count == 7)                         //---④
                    break;
            }
        }
        flagPrevLine = isLetterLine;
    }

    if (allData.count != 7)                                     //---⑤
        ::AfxMessageBox(_T("Image Parsing Error : Take Photo Again."));
}
```

① 개별 문자로 이미지를 나누는 파싱 2단계에서도 1단계와 마찬가지로 한 줄에 3개 이상의 검은색이 있으면 문자로 인식하고 아니면 바탕으로 인식한다.

② 하나의 문자에서 왼쪽 경계선과 오른쪽 경계선을 찾은 후 이미지 파싱의 보정 작업인 3

단계를 진행한다.

③ 개별 문자에 대한 이미지 파싱의 모든 단계를 진행한 이후에는 파싱된 문자가 숫자나 한글의 문자인지를 확인해야 한다. [그림 4-6]에서 설명한, 문자로 인식하기 위한 알고리즘을 진행하는 것이다. 문자라면 allData에 추가하고 아니면 추가하지 않고 다음 문자를 계속 찾는다.

④ 7개의 문자를 모두 찾았다면 이미지 파싱을 종료하기 위해 for 문을 빠져 나온다.

⑤ 7개의 문자를 다 찾지 않고 이미지 파싱이 종료되었다면 에러 메시지를 출력한다.

[코드 4-6]은 2장에서 구현한 이미지 파싱 3단계인 [코드 2-11]과 거의 동일하다. 다른 점은 문자 라인을 인식할 때 3개 이상의 픽셀들이 검은색일 때 문자 라인으로 판단한다는 점이다.

[코드 4-6] 번호판 이미지 파싱 3단계(OCR.cpp)

```
void COCR::ParsingStepThird(Rect *rect) {

    int x, y, count;
    COLORREF rgb;
    bool isLetterLine;

    int height = image->GetHeight();

    //------------------------- Letter Top ---------------------------
    for (y=rect->start.y; y>=0; y--) {              //---①

        isLetterLine = false;
        count = 0;

        for (x=rect->start.x; x<=rect->end.x; x++) {

            rgb = image->GetPixel(x,y);

            if (GetGValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                count += 1;
            }
            if (count > 3) {                        //---②
                rect->start.y = y;
                isLetterLine = true;
                break;
            }
        }
        if (!isLetterLine)
```

```
        break;
    }
    for (y=rect->start.y; y<height; y++)  {
        isLetterLine = false;
        count = 0;

        for (x=rect->start.x; x<=rect->end.x; x++) {
            rgb = image->GetPixel(x,y);

            if (GetGValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                count += 1;
            }
            if (count > 3) {
                rect->start.y = y;
                isLetterLine = true;
                break;
            }
        }
        if (isLetterLine)
            break;
    }

    //------------------------- Letter Bottom ---------------------------
    for (y=rect->end.y; y<height; y++) {                    //---③

        isLetterLine = false;
        count = 0;

        for (x=rect->start.x; x<=rect->end.x; x++) {

            rgb = image->GetPixel(x,y);

            if (GetGValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                count += 1;
            }
            if (count > 3) {
                rect->end.y = y;
                isLetterLine = true;
                break;
            }
        }
        if (!isLetterLine)
            break;
    }
    for (y=rect->end.y; y>=0; y--) {
```

```
        isLetterLine = false;
        count = 0;

        for (x=rect->start.x; x<=rect->end.x; x++) {
            rgb = image->GetPixel(x,y);

            if (GetGValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                count += 1;
            }
            if (count > 3) {
                rect->end.y = y;
                isLetterLine = true;
                break;
            }
        }
        if (isLetterLine)
            break;
    }
}
```

① 하나의 문자 이미지에서 위 경계선의 보정 작업을 진행한다.

② 검사하는 픽셀 선에 3개 이상의 검은색이 있으면 문자 선으로 인식한다.

③ 문자 이미지의 아래 경계선의 보정 작업을 진행한다.

[코드 4-7]은 이미지 파싱이 완료 후에 이미지가 문자 이미지인지를 판단하는 `IsLetterRect( )` 함수다. [그림 4-6]에서 설명한 문자의 크기와 비율을 구현한 함수라고 이해하면 된다.

[코드 4-7] 파싱된 이미지가 문자 이미지인지 아닌지를 판별하는 함수(OCR.cpp)

```
bool COCR::IsLetterRect(Rect *rect) {
    int width = rect->end.x - rect->start.x;
    int height = rect->end.y - rect->start.y;

    if (height < 50)                                    //---①
        return false;

    if ((height > width) && (height < (4 * width)))     //---②
        return true;

    return false;
}
```

① 세로의 길이가 50 픽셀 이하이면 문자가 아니므로 false를 반환한다. 즉, 문자 이미지의 세로의 길이는 항상 50 이상이어야 한다.

② 세로의 길이가 가로의 길이보다 크고 가로 길이의 4배보다 작으면 문자 이미지이므로 true를 반환한다. 즉, 항상 세로의 길이는 가로의 길이보다 크며 가로 길이의 4배를 넘지 않는다. 숫자 1의 이미지 같은 경우에도 세로의 길이가 가로의 길이보다 훨씬 크지만 가로 길이의 4배를 넘지는 않는다.

[코드 4-8]은 이미지 파싱된 하나의 문자 이미지를 사진 파일에 저장하는 함수로, 테스트를 위해 만들었다. 번호판 사진의 전체 이미지에서 이미지 파싱 후에 만들어진 이미지 영역을 같은 크기의 새로운 CImage 클래스를 만들어서 사진 파일에 저장한다. [코드 4-8]은 2장의 [코드 2-12]와 같기 때문에 설명은 생략한다.

[코드 4-8] 파싱된 문자 이미지를 파일에 저장하는 함수(OCR.cpp)

```
void COCR::PrintImageToFile(int fileNo, Rect *rect) {
    CString strName;
    strName.Format(TEXT("./image%d.jpg"), fileNo);

    CImage newImage;
    int width, height;

    width = rect->end.x - rect->start.x + 1;
    height = rect->end.y - rect->start.y + 1;

    if (width <=0 || height <= 0) {
        AfxMessageBox(_T("Error: 사진의크기가0보다작다."));
        return;
    }

    newImage.Create(width, height, 32);

    for (int i=0; i<width; i++)
        for(int j=0; j<height; j++)
        newImage.SetPixel(i,j, image->GetPixel(rect->start.x+i, rect->start.
        y+j));

    newImage.Save(strName, Gdiplus::ImageFormatJPEG);
}
```

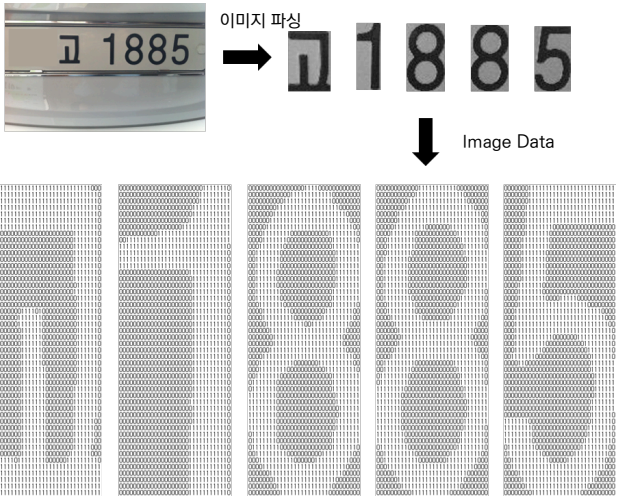
지금까지 진행된 이미지 파싱은 2장에서의 코드와 거의 동일하며 1단계에서의 접근 방법만이 다를 뿐이다. OCR에서 가장 어려운 부분이 이미지 파싱이며, 전

체 사진에서 문자 영역만을 정확히 가져올 수 있다면 OCR 프로그램은 쉽게 만들 수 있다.

4.3 문자 이미지 데이터 변환

이미지 파싱이 진행된 후 만들어진 문자 이미지는 문자 이미지 데이터로 변환되어야 한다. 이는 2장에서 진행된 문자 이미지 데이터 만드는 방법과 동일하다. [그림 4-8]은 문자 이미지 데이터를 만든 결과다. 최대한 수평으로 번호판 사진을 찍는다 하더라도 대부분의 사진은 약간의 기울기를 가지고 있기 때문에 그림과 같이 약간 기울어진 이미지 데이터의 그림이 나타난다. 이미지 데이터를 만드는 방법까지 진행된 이후에는 번호판 사진의 데이터를 가지고 기본 이미지 데이터를 만들어야 하며, 약간 기울어진 문자 이미지 데이터를 일정한 문자의 모양으로 직접 수작업을 진행해야 한다. 이는 번호판 사진에서 정확한 글자의 모양을 얻는 것은 거의 불가능하기 때문이다. 필자는 프로그램 구현 후에 기본 이미지 데이터를 만드는 데 0과 1의 값을 변경하면서 상당히 많은 시간을 소요했다.

그림 4-8 문자 이미지 데이터로 변환한 결과



[코드 4-9]는 화면의 [OCR] 버튼을 클릭하였을 때 실행되는 함수다. 이미지 데이터를 만드는 함수인 MakeImageData( ) 함수와 결과값을 확인하기 위한 테스트 함수가 추가되었다.

[코드 4-9] 이미지 파싱 후 이미지 데이터를 만들고 결과를 확인하는 함수 추가(OCR.cpp)

```
void COCR::RunOCR(CImage *newImage) {
    image = ResizeInBlackWhiteImage(newImage);

    ParsingStepFirst();

    MakeImageData();                                    //---①

    for (int i=0; i<MAX_COUNT_DATA; i++)
        PrintImageDataToFile(i, &allData.data[i].letter);    //---②

    PrintEveryImageDataInTextFile("EveryImageData.txt");    //---③
}
```

① MakeImageData는 이미지 파싱을 진행한 후 만들어진 문자 이미지를 이미지 데이터로 만드는 함수다.

② 테스트를 위해 구현한 것으로 하나의 이미지 데이터를 하나의 텍스트 파일에 저장한다.

③ 테스트를 위해 구현한 것으로 7개의 문자 이미지 데이터를 하나의 텍스트 파일에 저장한다.

[코드 4-10]은 문자 이미지를 문자 이미지 데이터로 변환하는 함수다. 2장의 [코드 2-16]과 동일하므로 설명은 생략한다.

[코드 4-10] 문자 이미지를 문자 이미지 데이터로 변환하는 함수(OCR.cpp)

```
void COCR::MakeImageData() {
    int i, j;

    for (i=0; i<allData.count; i++) {
        Data *data = &allData.data[i];
        Letter *letter = &data->letter;
        Rect *rect = &data->rect;

        float xRate, yRate;
        int x, y;
        unsigned int buffer;
        COLORREF rgb;
```

```
        for (j=0; j<48; j++)
            letter->image[j] = 0x00000000;

        xRate = (float)(rect->end.x - rect->start.x) / (32 - 1);
        yRate = (float)(rect->end.y - rect->start.y) / (48 - 1);

        for (y=0; y<48; y++) {
            for (x=0; x<32; x++) {
                rgb = image->GetPixel((int)(rect->start.x + (x * xRate)), (int)
                (rect->start.y + (y * yRate)));

                if (GetGValue(rgb) < colorToCheck + RANGE_OF_COLOR_TO_CHECK) {
                    buffer = 0x80000000;
                    buffer >>= x;
                    letter->image[y] |= buffer;
                }
            }
        }
    }
}
```

[코드 4-11]은 이미지 데이터를 텍스트 파일에 저장하는 함수다. 2장의 [코드 2-17]과 동일하므로 설명을 생략한다.

[코드 4-11] 하나의 이미지 데이터를 텍스트 파일에 저장하는 함수(OCR.cpp)

```
void COCR::PrintImageDataToFile(int fileNo, Letter *letter) {
    CString strName;
    strName.Format(TEXT("./ImageData%d.txt"), fileNo);

    FILE *fp;
    fp = fopen(LPSTR(LPCTSTR(strName)), "wt");

    int x, y;
    unsigned int buffer;

    for (y=0; y<48; y++) {
        buffer = letter->image[y];

        for (x=0; x<32; x++) {
            if (buffer & 0x80000000)
                fputc('1', fp);
            else
                fputc('0', fp);
```

```
            buffer <<= 1;
        }
        fputc('\n', fp);
    }
    fclose(fp);
}
```

[코드 4-12]는 모든 이미지 데이터를 하나의 텍스트 파일에 저장하는 함수로, 2장의 [코드 2-19]와 거의 동일하므로 설명을 생략한다. [코드 2-19]와 한 가지 다른 점은 2장에서는 영문만을 다루었기 때문에 1-Byte로 문자의 표현이 가능했지만, 자동차 번호판 인식에서는 한글을 다루기 때문에 2-Byte에 문자를 CString 변수를 사용한다는 점이다.

[코드 4-12] 구현된 모든 이미지 데이터들을 하나의 텍스트 파일에 저장하는 함수(OCR.cpp)

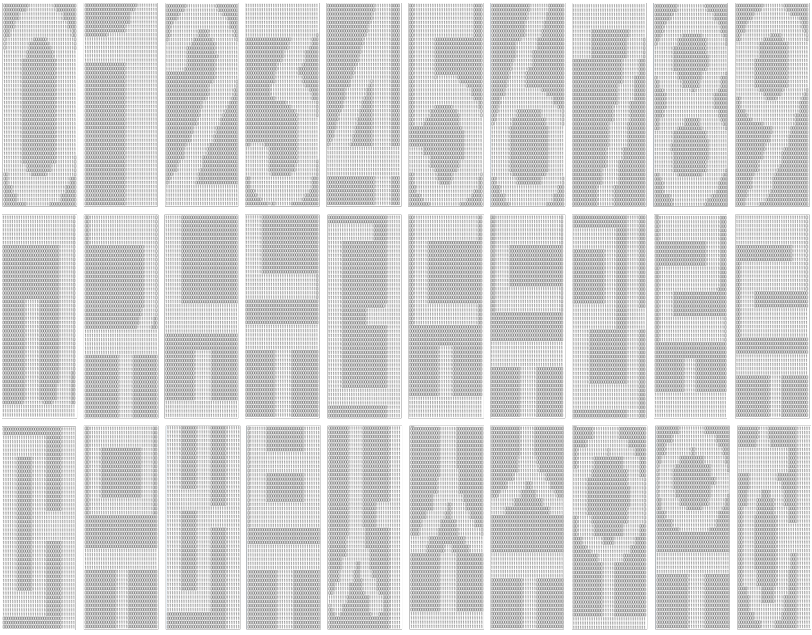
```
void COCR::PrintEveryImageDataInTextFile(char * fileName) {
    FILE *fp;
    fp = fopen(fileName, "wt");

    int i, x, y;
    unsigned int buffer;
    Letter *letter;

    fprintf(fp, "%d\n", allData.count);

    for (i=0; i<allData.count; i++) {
        letter = &allData.data[i].letter;

        fprintf(fp, "\n%s\n", letter->value.GetString());     //---①

        for (y=0; y<48; y++) {
            buffer = letter->image[y];

            for (x=0; x<32; x++) {
                if (buffer & 0x80000000)
                    fputc('1', fp);
                else
                    fputc('0', fp);

                buffer <<= 1;
            }
            fputc('\n', fp);
        }
```

```
    }
    fclose(fp);
}
```

① 자동차 번호판 인식은 한글을 사용하기 때문에 value 변수는 CString 타입이며, 값을 가져오기 위해 CString의 GetString ( ) 함수를 사용한다.

이미지 데이터를 만드는 것까지 진행되었기 때문에 기본 이미지 데이터를 만들 수 있는 방법이 생겼다. 기본 이미지 데이터를 만들려면 한글이 다른 모든 번호판의 사진을 찍어서 텍스트 파일에 저장된 한글 이미지를 가지고 standard.txt에 계속 추가하는 작업을 진행하면 된다. [그림 4-9]는 필자가 다수의 번호판을 인식한 후 다양한 한글을 가지고 기본 이미지 데이터를 만든 것이다. 모든 이미지 데이터는 '32×48' 크기의 0과 1의 조합으로 만들어지는 것이 기본이다. 다음 절에서는 만들어진 기본 이미지 데이터를 가지고 인식하는 방법을 진행한다.

그림 4-9 기본 이미지 데이터 - 텍스트 모드 사진



4.4 번호판 인식

자동차 번호판 인식은 기본 이미지 데이터를 만들고 난 이후에 확률을 이용한 인식을 진행하면 되는데, '3.1 줄바꿈 문자 추가하기'에서 설명한 내용과 비슷하다. [그림 4-10]은 이번 절에서 구현하는 프로그램을 실행한 결과다. 사진의 번호판 문자가 메시지 박스에 텍스트로 나타난다. 2장과 3장에서 설명한 내용이 이미지 인식의 가장 중요한 내용이었으며 4장의 번호판 인식은 대상을 자동차 번호판으로 변경한 것뿐이라고 이해하면 된다.

그림 4-10 번호판 인식 결과(이미지 파일: 17.jpg)

이 책에서는 자동차 번호판 인식에서 기본 이미지 데이터를 만드는 방법을 생략하였다. 실제로 프로그램을 만들 때에는 다양한 종류의 번호판을 인식한 후 모든 한글을 이미지 데이터로 변환한 다음 개별적으로 수정 작업을 진행하면 된다.

[코드 4-13]은 프로그램을 실행할 때 이미 만들어진 기본 이미지 데이터를 불러와서 구조체에 넣는 함수를 실행하는 것이다.

[코드 4-13] 프로그램 실행 시 기본 이미지 데이터를 파일에서 불러오기(OCR.cpp)

```
COCR::COCR(void) {
    GetStandardImageDataFromTextFile("standard.txt");       //--①
    PrintAllStandardImageToTextFile("standardout.txt");     //--②
    colorToCheck = 0;
}
```

① 텍스트 파일에 있는 기본 이미지 데이터를 불러와서 구조체에 넣는 함수를 실행한다.

② 테스트를 위해 구조체에 담긴 모든 기본 이미지 데이터를 파일에 출력한다.

[코드 4-14]는 [OCR] 버튼을 클릭하였을 때 실행되는 함수다. 이미지 파싱과 이미지 데이터를 만드는 작업을 수행한 후 마지막으로 인식을 진행하고 결과값을 메시지 박스에 출력한다.

[코드 4-14] 번호판 인식을 진행하는 함수와 결과를 출력하는 함수 추가(OCR.cpp)

```
void COCR::RunOCR(CImage *newImage) {
    image = ResizeInBlackWhiteImage(newImage);

    ParsingStepFirst();

    MakeImageData();

    FindValue();                          //---①

    ShowResultMessage();                  //---②
}
```

① 확률을 이용하여 번호판의 인식을 진행하는 함수를 실행한다.

② 인식된 결과값을 메시지 박스에 출력하는 함수를 실행한다.

[코드 4-15]는 인식을 진행하는 함수로, 함수 안에서 숫자를 인식하는 함수와 한글을 인식하는 함수를 실행한다. 즉, 처음 두 문자는 숫자로 인식을 진행하고 세 번째 글자만 한글로 인식을 진행한 후에 나머지 문자는 다시 숫자로 인식을 진행한다.

[코드 4-15] 번호판 인식을 진행하는 함수(OCR.cpp)

```
void COCR::FindValue() {
    int i;

    for (i=0; i<allData.count; i++) {          //---①
        Data *aData = &allData.data[i];        //---②

        if (i == 2)                            //---③
            FindLetterValue(aData);            //---④
        else
            FindNumberValue(aData);            //---⑤
```

```
    }
}
```

① 모든 데이터의 인식을 차례대로 진행한다. 여기서 count의 값은 항상 7(번호판 문자의 개수)이기 때문에 7번 반복 수행한다.

② 인식을 위한 데이터를 차례대로 접근한다.

③ 배열은 0부터 시작하므로 변수 'i' 값이 2이면 세 번째 문자를 나타내므로 한글을 인식하는 함수를 실행하고, 나머지는 숫자를 인식하는 함수를 실행한다.

④ FindLetterValue( )는 한글을 인식하는 함수다.

⑤ FindNumberValue( )는 숫자를 인식하는 함수다.

[코드 4-16]은 숫자를 인식하는 함수로, 3장의 [코드 3-6]과 거의 동일하므로 자세한 설명은 생략한다.

[코드 4-16] 번호판에서 숫자를 인식하는 함수(OCR.cpp)

```
void COCR::FindNumberValue(Data *aData) {
    int count, maxCount;
    unsigned int buffer, bit;
    int i, x, y;

    Letter *letter = &(aData->letter);

    maxCount = 0;

    for (i=0; i<standardNumber.count; i++) {            //---①
        count = 0;

        for (y=0; y<48; y++) {

            buffer = letter->image[y] ^ standardNumber.letter[i].image[y];

            for (x=0; x<32; x++) {

                bit = 0x80000000;
                bit >>= x;
                bit = bit & buffer;

                if (!bit)
                    count += 1;
            }
        }
```

```
        if (count > maxCount) {;
            letter->value = standardNumber.letter[i].value;
            maxCount = count;
        }
    }
}
```

① 숫자를 위한 기본 이미지 데이터는 standardNumber 구조체에 저장되었기 때문에 모든 숫자의 이미지 데이터는 이 구조체와 비교를 진행한다.

[코드 4-17]은 번호판의 한글을 인식하는 함수로, 3장의 [코드 3-6]과 거의 동일하므로 자세한 설명은 생략한다.

[코드 4-17] 번호판에서 한글 인식하는 함수(OCR.cpp)

```
void COCR::FindLetterValue(Data *aData) {
    int count, maxCount;
    unsigned int buffer, bit;
    int i, x, y;

    Letter *letter = &(aData->letter);

    maxCount = 0;

    for (i=0; i<standardLetter.count; i++) {            //---①
        count = 0;

        for (y=0; y<48; y++) {
            buffer = letter->image[y] ^ standardLetter.letter[i].image[y];

            for (x=0; x<32; x++) {

                bit = 0x80000000;
                bit >>= x;
                bit = bit & buffer;

                if (!bit)
                    count += 1;
            }
        }
        if (count > maxCount) {
            letter->value = standardLetter.letter[i].value;
            maxCount = count;
        }
```

```
    }
}
```

① 번호판의 한글을 위한 기본 이미지 데이터는 standardLetter 구조체에 저장되므로 모든 한글의 이미지 데이터는 이 구조체와 비교를 진행한다.

[코드 4-18]은 기본 이미지 데이터를 텍스트 파일에서 가져오는 함수를 구현한 것이다. 2장의 [코드 2-20]과 거의 동일하다. 다른 점이 있다면 2장과 3장에서는 영문 인식이기 때문에 문자를 저장하기 위해 1-Byte의 저장 공간을 사용하였지만, 여기서는 하나의 한글을 저장하기 위해서 2-Bytes의 메모리 공간을 차지한다는 점이다. 또한, 텍스트에 저장된 기본 이미지 데이터는 처음 10개가 숫자이고 나머지는 한글의 이미지 데이터이기 때문에, 숫자와 한글의 기본 이미지 데이터를 나누어서 구조체에 저장한다.

[코드 4-18] 기본 이미지 데이터를 텍스트 파일에서 가져오는 함수(OCR.cpp)

```
void COCR::GetStandardImageDataFromTextFile(char * fileName) {
    FILE *fp;
    fp = fopen(fileName, "rt");

    int i, x, y, temp;
    char ch, str[4];
    unsigned int buffer;
    Letter *letter;

    fscanf(fp, "%d\n", &standardLetter.count);        //---①

    standardNumber.count = 10;                        //---②
    standardLetter.count -= 10;                       //---③

    for (i=0; i<standardNumber.count; i++) {          //---④
        letter = &standardNumber.letter[i];

        fscanf(fp, "\n%s\n", str);                    //---⑤
        letter->value = str;

        for (y=0; y<48; y++) {
            letter->image[y] = 0x00000000;

            for (x=0; x<32; x++) {
                if ((ch = fgetc(fp)) == '1') {
```

```
                buffer = 0x80000000;
                buffer >>= x;
                letter->image[y] |= buffer;
            }
        }
        ch = fgetc(fp);
    }
}

for (i=0; i<standardLetter.count; i++) {        //---⑥

    letter = &standardLetter.letter[i];

    fscanf(fp, "\n%s\n", str);
    letter->value = str;

    for (y=0; y<48; y++) {

        letter->image[y] = 0x00000000;

        for (x=0; x<32; x++) {
            if ((ch = fgetc(fp)) == '1') {
                buffer = 0x80000000;
                buffer >>= x;
                letter->image[y] |= buffer;
            }
        }
        ch = fgetc(fp);
    }
}
fclose(fp);
}
```

① 기본 이미지 데이터가 저장된 텍스트 파일에서 기본 이미지 데이터의 전체 개수를 읽는다.

② 숫자 기본 이미지 데이터의 개수는 10개다.

③ 한글 기본 이미지 데이터의 개수는 전체 개수에서 숫자의 개수인 10을 빼면 된다.

④ 숫자 기본 이미지 데이터 10개를 구조체에 저장한다.

⑤ 번호판에서 기본 데이터의 값은 2-Bytes에 저장되므로 문자열로 파일에서 읽는다.

⑥ 한글 기본 이미지 데이터를 구조체에 저장한다.

[코드 4-19]는 2장의 [코드 2-19]와 거의 동일하므로 자세한 설명은 생략한다. 테스트를 위해 구현한 함수이며 기본 이미지 데이터에 저장된 모든 값을 텍스트 파일에 저장한다.

[코드 4-19] 테스트를 위해 기본 이미지 데이터를 텍스트 파일에 저장하는 함수(OCR.cpp)

```
void COCR::PrintAllStandardImageToTextFile(char *  e(fp);
}
```

[코드 4-20]은 인식된 결과를 메세지 박스에 출력하는 함수로, 7개의 인식된 결과값을 문자열로 만들어서 출력한다.

[코드 4-20] 번호판 인식 결과를 MessageBox에 출력하는 함수(OCR.cpp)

```
void COCR::ShowResultMessage() {

    CString result;

    for (int i=0; i<7; i++)
        result.AppendFormat(_T("%s"), allData.data[i].letter.value); //---①

    ::AfxMessageBox(result);                                          //---②
}
```

① 인식될 결과를 result 문자열에 차례대로 추가한다.

② 결과를 메시지 박스에 출력한다.

2장과 3장의 내용을 이해하였다면 번호판으로 인식 대상만 변했을 뿐 그 방법은 거의 동일함을 알 수 있다. 이와 같이, OCR은 사전이라고 할 수 있는 기본 이미지 데이터를 만든 후에 확률을 이용하여 결과값을 도출하면 된다. 이 책에서는 쉽게 설명하기 위해 프로그램 코드를 최대한 간단히 만들어 작업한 것이므로 지금까지의 내용만으로 모든 한글을 인식할 수는 없다. 따라서 상용화할 수 있는 프로그램이 만들어진 것은 아니다. 그러나 지금까지의 내용을 자신의 것으로 만든 독자라면 조금 더 어려운 이미지 인식도 패턴Pattern을 찾아서 쉽게 프로그램을 만들 수 있을 것이다. 더하여, 이 책에서는 구현하지 않았지만, 확률을 이용한 이미지 인식에

서 기본 이미지 데이터와의 일치 확률이 일정한 범위(예: 80%) 이상 될 경우에만 결과값을 도출하는 것도 정확도를 위한 하나의 방법이다.

지금까지의 내용이 이 책에서 설명하는 가장 기본적인 OCR의 구현이다. 문자 인식 프로그램을 볼 때마다 어떻게 구현했을까 궁금해하던 독자라면 이제는 OCR의 모든 것이 이해되었을 것이다.

4.5 이미지 데이터 변형

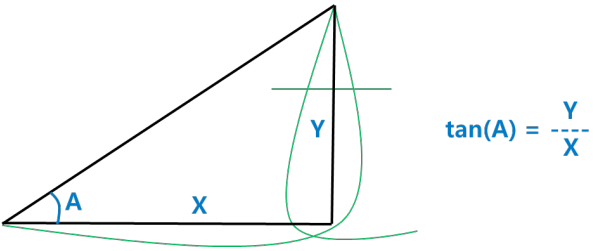
지금까지는 거의 정확한 이미지를 가지고 OCR을 진행했다. 그러나 문자 이미지는 사진을 찍는 각도에 따라서 기울어져 있는 경우가 많다. 이번 절에서 설명하려는 내용은 이처럼 정확하지 않은 이미지 데이터의 모양을 변형하면서 인식을 진행하는 방법의 기초가 되는 내용이다. 실제 OCR에서는 이러한 이미지 변형 작업을 진행하면서 인식률을 높이는 내용이 추가된다. 하지만 이 책은 배움을 목적으로 진행하기 때문에 이미지 데이터의 변형만을 확인하고 이해할 것이다. 실제로 작업할 때 OCR에 이미지 데이터 변형을 추가한다면 다양한 이미지 데이터 변형을 진행한 후에 확률을 이용한 인식을 진행하면 된다. 수학의 삼각함수를 적절히 이용해야 하기 때문에 이미지 데이터 변형이 이해하기 어려울 수가 있다(사실 이미지를 변형하는 포토샵 같은 프로그램은 수학을 많이 사용하는 소프트웨어 중의 하나다). 물론 이번 절을 이해하지 않아도 간단한 OCR 프로그램을 만드는 데에는 문제가 없을 것이다.

이 책에서 다루는 내용에는 중학교 수학 시간에 배우는 삼각함수가 포함된다. 구현된 코드에는 삼각함수의 탄젠트tan, Tangent를 사용하였다. [그림 4-11]은 삼각함수의 tan 공식을 보여 주는데, 가로 길이를 기준으로 각도에 따라서 세로 길이를 알 수 있다. C/C++ 프로그램을 구현할 때는 제공되는 수학 함수를 사용하기 위해서 'math.h' 헤더 파일을 코드의 가장 위에 추가하면 된다. 다음과 같이 `include`를 사용하여 추가한다. 이와 같이 'math.h' 파일을 추가하면 수학에서 사용하는

거의 모든 공식을 사용할 수 있으며 수학 함수의 tan도 tan ( ) 함수를 사용하여 쉽게 구현할 수 있다.

```
#include "math.h"
```

그림 4-11 삼각함수 tan 공식



이번 절에서는 이미지 데이터의 문자를 기울이고, 줄이고, 늘리는 다양한 방법을 구현한다. [코드 4-21]은 앞으로 구현하는 함수를 실행하고 결과를 하나의 텍스트 파일에 저장하여 문자 이미지를 확인하는 코드다.

[코드 4-21] OCR 버튼 클릭 시 이미지 데이터를 변형하는 함수 실행(OCR.cpp)

```
void COCR::RunOCR(CImage *newImage) {
    Letter *testLetter, *result;

    testLetter = &(standardNumber.letter[8]);          //---①
    PrintImageDataToFile(0, testLetter);

    result = TiltLetterRight(testLetter);              //---②
    PrintImageDataToFile(1, result);

    result = TiltLetterLeft(testLetter);               //---③
    PrintImageDataToFile(2, result);

    result = MakeThinLetter(testLetter);               //---④
    PrintImageDataToFile(3, result);

    result = StretchHeight(testLetter);                //---⑤
    PrintImageDataToFile(4, result);
}
```

① testLetter는 원본 문자를 저장하는 Letter 변수이며, 여기서는 숫자 '8'의 이미지 데이터를 가리킨다. PrintImageDataToFile( ) 함수를 사용하여 0번째 텍스트 파일(ImageData0.txt)에 원본 이미지 데이터를 저장한다.

② 이미지를 오른쪽으로 기울이는 함수를 실행한다. 결과는 1번째 텍스트 파일(ImageData1.txt)에 저장한다.

③ 이미지를 왼쪽으로 기울이는 함수를 실행한다. 결과는 2번째 텍스트 파일 (ImageData2.txt)에 저장한다.

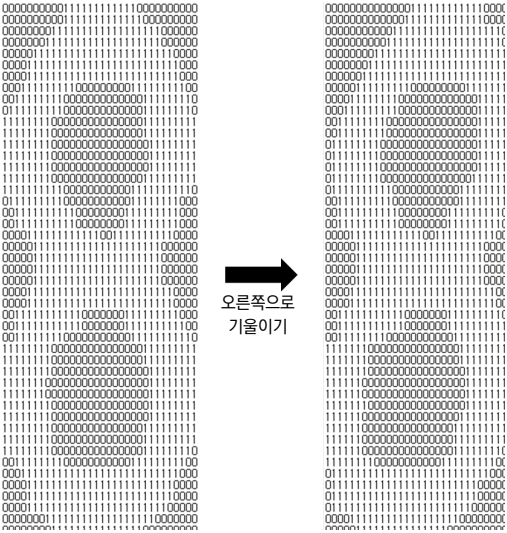
④ 이미지의 가로 폭을 줄이는 함수를 실행한다. 결과는 3번째 텍스트 파일(ImageData3.txt)에 저장한다.

⑤ 이미지를 세로로 길게 늘여 주는 함수를 실행한다. 결과는 4번째 텍스트 파일(ImageData4.txt)에 저장한다.

프로그램을 구현할 때는 함수 안의 내용을 작성하지 않고 함수 이름만을 정의한 이후에 [코드 4-21]과 같이 테스트를 위해 함수를 실행하는 것을 작성하고 컴파일하는 것이 좋다. 이렇게 구조적인 부분을 구현한 이후에 각각의 함수를 하나씩 구현해 나간다. 오랫동안 프로그램을 만들면서 필자가 배운 것은 이렇게 구조적으로 만들어 나가는 것이 속도도 빠르고 프로그래밍을 한결 편하게 진행할 수 있다는 것이다.

[그림 4-12]은 숫자 '8'의 이미지 데이터를 오른쪽으로 기울이는 함수를 구현한 결과다. 이탤릭Italic체가 되었다고 이해할 수 있다. 이미지의 기울기는 세로축의 중간값을 기준으로 위쪽은 오른쪽으로 기울이면 되고, 아래쪽은 왼쪽으로 기울이면 된다. 이때 수학 함수의 탄젠트가 사용되는데, 세로축의 중간에 가까운 것은 기울임 정도가 작고 중간에서 멀어질수록 조금 더 많이 데이터를 이동하여 기울임을 크게 한다. 자세한 사항은 [코드 4-22]의 코드를 보면서 이해하면 된다.

그림 4-12 문자 이미지 데이터를 오른쪽으로 기울이기(ImageData1.txt)



[코드 4-22]는 이미지 데이터를 오른쪽으로 기울이는 코드를 구현한 것으로, 여기서 사용하는 수학 함수는 tan( )이다. 이미지 데이터의 세로는 48개이므로 가운데를 중심으로 위쪽 24개는 오른쪽으로 기울이는 코드를 구현하고, 아래쪽 24개는 왼쪽으로 기울이는 코드를 구현하였다. 이미지를 오른쪽으로 기울이는 각도는 20도이며, 가장 위와 가장 아래 끝의 이동 범위를 정한 후에 그 비율에 맞추어서 모든 값을 이동하게 된다.

[코드 4-22] 문자 이미지 데이터를 오른쪽으로 기울이는 함수(OCR.cpp)

```
Letter* COCR::TiltLetterRight(Letter * letter) {
    const float PI = (float) 3.141592;                          //---①
    const int DEGREE_TO_TILT = 20;                              //---②

    static Letter aLetter;
    int y;

    int xToMove = (int)(tan( DEGREE_TO_TILT * PI / 360 ) * 24);  //---③

    for (y=0; y<24; y++) {                                      //---④
        aLetter.image[y] = letter->image[y];                    //---⑤
        aLetter.image[y] >>= (int)(xToMove * (23 - y) / 23);    //---⑥
```

```
    }
    for (y=24; y<48; y++) {                                          //---⑦
        aLetter.image[y] = letter->image[y];
        aLetter.image[y] <<= (int)(xToMove * (y - 24) / 23);         //---⑧
    }
    return &aLetter;                                                 //---⑨
}
```

① 삼각함수를 계산하기 위해 사용되는 파이(π) 값이다.

② 기울어지는 각도를 나타내며 여기서는 오른쪽으로 기울어지는 각도가 20도라는 의미를 가진다.

③ 세로축의 중간을 기준으로 위쪽 끝을 길이가 24라고 할 때, xToMove는 위쪽 끝의 가로축으로 이동해야 하는 거리 값을 삼각함수로 계산한 후에 저장한다.

④ 세로축의 가운데를 기준으로 위쪽의 이미지 데이터를 차례대로 오른쪽으로 이동한다.

⑤ 새로 만들어지는 이미지의 가로 한 줄은 원본 이미지의 가로 한 줄로 초기화한다.

⑥ 세로줄의 위치에 따라서 우측으로 이동시키며 이미지 데이터의 위치를 변경한다. 이때, 세로축의 가운데에 가까운 이미지 데이터는 이동이 미미하지만, 위쪽 끝에 가까운 이미지 데이터일수록 오른쪽으로 크게 이동한다.

⑦ 세로축의 가운데를 기준으로 아래쪽의 이미지 데이터를 차례대로 왼쪽으로 이동한다.

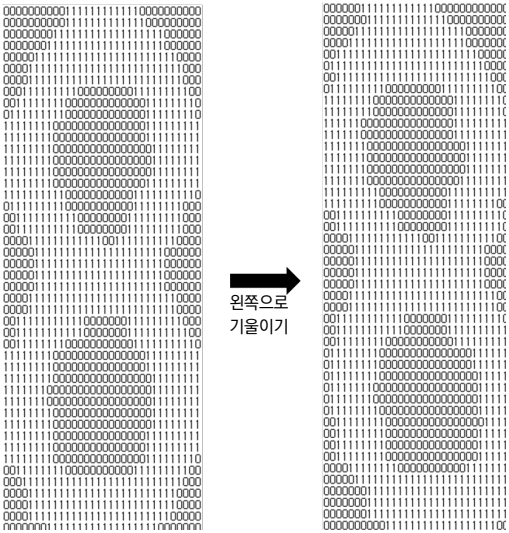
⑧ 세로줄의 가운데를 기준으로 아래 부분을 왼쪽으로 이동시키는 것이다. ⑥의 내용과 동일하며 이동 방향만 다르다.

⑨ 결과값은 메모리에 저장되어 있으며 메모리 주소만을 반환한다.

프로그래밍의 방법에는 정해진 답이 없다. 우측으로 기울어진 이미지 데이터를 만들기 위해서 필자는 [코드 4-22]와 같이 구현하였다. 이미지 변환은 사진에서도 가능한데 이와 같은 원리로 진행하면 된다. 하지만 사진을 가지고 이미지 변환 작업을 진행한 후에 OCR을 진행하면 프로그램 속도가 상당히 느려진다. 물론 예외가 있다. 사진에서 이탤릭 문자체가 있다면 이미지 파싱이 어려워서 사진 이미지를 가지고 직접 변환 작업을 진행한 후에 이미지 파싱을 진행하는 것이 좋을 수 있다. OCR은 경우에 따라서 다양한 방법으로 프로그램을 만들 수 있지만, 이미지의 규칙을 가진 사진을 찍는 것이 가장 좋은 방법이다.

[그림 4-13]은 문자 이미지 데이터를 왼쪽으로 기울이는 함수의 실행 결과다. [그림 4-12]와 기울어지는 방향만 다르며 기본적인 개념은 동일하기 때문에 설명은 생략한다.

그림 4-13 문자 이미지 데이터를 왼쪽으로 기울이기(ImageData2.txt)



[코드 4-23]은 [코드 4-22]와 거의 모든 내용이 같으며, 단지 Shift 연산자의 방향만 변경되었기에 자세한 설명은 생략한다.

[코드 4-23] 문자 이미지 데이터를 왼쪽으로 기울이는 함수(OCR.cpp)

```
Letter* COCR::TiltLetterLeft(Letter * letter) {
    const float PI = (float) 3.141592;
    const int DEGREE_TO_TILT = 20;

    static Letter aLetter;
    int y;

    int xToMove = (int)(tan( DEGREE_TO_TILT * PI / 360 ) * 24);

    for (y=0; y<24; y++) {
        aLetter.image[y] = letter->image[y];
        aLetter.image[y] <<= (int)(xToMove * (23 - y) / 23);     //---①
    }
```

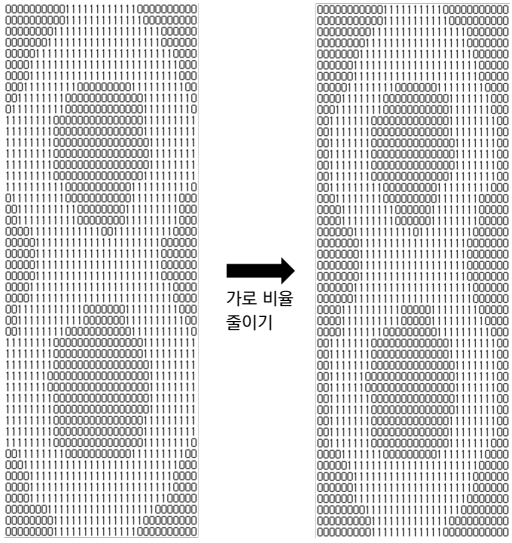
```
    for (y=24; y<48; y++) {
        aLetter.image[y] = letter->image[y];
        aLetter.image[y] >>= (int)(xToMove * (y - 24) / 23);     //---②
    }
    return &aLetter;
}
```

① 세로줄의 위치에 따라서 왼쪽으로 이동시키며 이미지 데이터의 위치를 변경한다. 이때, 세로축의 가운데에 가까운 이미지 데이터는 이동이 미미하지만, 위쪽 끝에 가까운 이미지일수록 왼쪽으로 크게 이동한다.

② 세로줄의 가운데를 기준으로 아래 부분을 오른쪽으로 이동시키는 것이다. 1의 내용과 동일하지만 이동 방향만 다르다.

[그림 4-14]는 문자 이미지 데이터의 가로 폭을 줄인 결과다. 가로의 가운데를 기준으로 왼쪽의 이미지 데이터는 오른쪽으로 옮겨 주고, 오른쪽의 이미지 데이터는 왼쪽으로 옮겨 준다. 이미지 데이터의 이동 시에 좌측과 우측의 끝 부분은 0으로 값이 채워짐을 알 수 있다. 프로그램의 구현에서는 하나씩 이동하면서 값을 변경해 주는 것은 효과적이지 않으며, 모든 값을 0으로 초기화한 후에 1의 값을 갖는 위치의 값만을 1로 변경해 주는 것이 속도와 구현 측면에서 더 효과적이다.

그림 4-14 문자 이미지 데이터의 가로 폭을 줄이기(ImageData3.txt)



[코드 4-24]는 이미지 데이터의 가로 폭을 줄이는 함수를 구현한 것이다. 문자 이미지 데이터를 90% 크기로 줄이게 된다. 함수의 시작 부분에서 상수 변수 rate의 값을 0.9로 정의하여 90%의 크기로 줄이도록 설정하였는데, 이와 같이 상수 변수를 사용하면 이후에 줄이는 비율을 쉽게 조정할 수 있다.

[코드 4-24] 문자 이미지 데이터의 가로 폭을 줄이는 함수(OCR.cpp)

```
Letter* COCR::MakeThinLetter(Letter *letter) {
    const float rate = (float) 0.9;                    //---①

    static Letter aLetter;
    unsigned int buffer;
    int x, y;

    for (y=0; y<48; y++)                               //---②
        aLetter.image[y] = 0x00000000;

    for (y=0; y<48; y++) {

        for (x=0; x<16; x++) {                         //---③

            buffer = 0x00010000;                       //---④
            buffer <<= x;                              //---⑤

            if (buffer & letter->image[y]) {           //---⑥
                buffer = 0x00010000;
                buffer <<= (int)(rate * x);            //---⑦
                aLetter.image[y] |= buffer;
            }
        }
        for (x=0; x<16; x++) {                         //---⑧

            buffer = 0x00008000;                       //---⑨
            buffer >>= x;

            if (buffer & letter->image[y]) {           //---⑩
                buffer = 0x00008000;
                buffer >>= (int)(rate * x);            //---⑪
                aLetter.image[y] |= buffer;
            }
        }
    }
    return &aLetter;
}
```

① 이미지 데이터의 가로 폭을 90%로 줄이기 위해 상수 변수 rate의 값을 0.9로 설정한다.

② 새로운 이미지 데이터의 모든 값을 0으로 초기화한다.

③ 가로축의 가운데 선을 기준으로 왼쪽의 이미지 데이터를 오른쪽으로 축소한다.

④ 가로축의 가운데 선을 기준으로 왼쪽의 가장 오른쪽에 있는 값만을 1로 초기화한다.

⑤ 원본 이미지 데이터의 왼쪽 특정 위치의 값이 1인지를 확인하기 위해 buffer의 값을 해당 위치의 값만 1로 만들려고 Shift 연산자를 사용하였다.

⑥ 원본 이미지 데이터의 왼쪽 특정 위치의 값이 1인지를 확인하여 1이면 if 문 안의 프로세스를 수행한다.

⑦ 원본 이미지 데이터에서 1의 값을 가진 데이터의 위치를 축소하여 buffer의 새로운 위치에 1로 설정한 후 새로운 이미지의 해당 위치에 1을 추가한다.

⑧ 가로축의 가운데 선을 기준으로 오른쪽의 이미지 데이터를 왼쪽으로 축소한다.

⑨ 가로축의 가운데 선을 기준으로 왼쪽의 가장 오른쪽에 있는 값만을 1로 초기화한다.

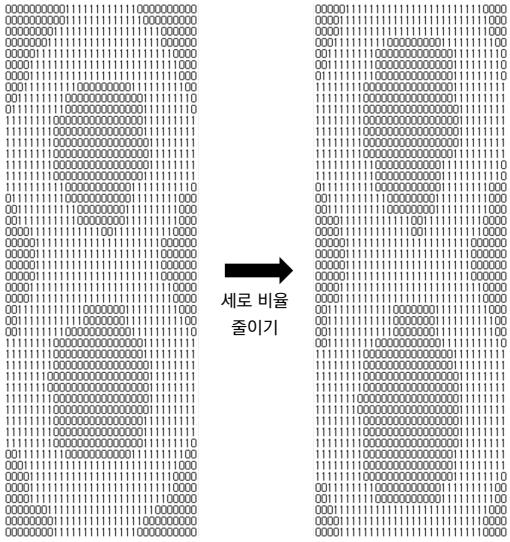
⑩ 원본 이미지 데이터의 오른쪽 특정 위치의 값이 1인지를 확인하여 1이면 if 문 안의 프로세스를 수행한다.

⑪ 이미지 데이터의 왼쪽에서 진행한 방법과 같이 오른쪽에서도 축소된 새로운 위치에 1을 설정한 후 새로운 이미지의 해당 위치에 1을 추가한다.

[그림 4-14]에서 가로축을 중심으로 축소하였기 때문에 [코드 4-24]를 응용하면 세로축에서의 축소도 쉽게 구현할 수 있다.

[그림 4-15]는 문자 이미지 데이터를 세로축을 기준으로 위아래로 확장한 함수의 실행 결과다. 세로축의 가운데를 중심으로 위쪽은 이미지 데이터를 일정한 비율로 위쪽으로 이동하는 것이고, 아래쪽은 이미지 데이터를 아래로 이동한 것이다. [그림 4-15]의 세로축 확장을 응용하면 쉽게 가로축에서의 확장도 구현할 수 있다.

그림 4-15 문자 이미지 데이터를 위/아래로 확장하기(ImageData4.txt)



[코드 4-25]는 문자 이미지 데이터를 위아래로 확장하는 함수를 구현한 것으로, 20%의 비율로 확장을 진행하였다. 상수 변수 rateToMove 값을 0.2로 설정한 것은 20% 비율로 확장하기 위해서고 rateToMove 값을 변경하여 쉽게 비율을 조정할 수 있다.

[코드 4-25] 문자 이미지 데이터를 위아래로 확장하는 함수(OCR.cpp)

```
Letter* COCR::StretchHeight(Letter *letter)
{
    const float rateToMove = (float) 0.2;                    //---①

    static Letter aLetter;
    int y;

    for (y=0; y<48; y++)                                     //---②
        aLetter.image[y] = 0x00000000;

    for (y=0; y<24; y++)                                     //---③
        aLetter.image[y] = letter->image[y + (int)((24 - y) * rateToMove)];

    for (y=47; y>=24; y--)                                   //---④
        aLetter.image[y] = letter->image[y - (int)((y - 24) * rateToMove)];
```

```
    return &aLetter;
}
```

① 세로축을 기준으로 위아래로 확장하는 비율은 20%이므로 변수 상수 `rateToMove` 값을 `0.2`로 설정한다.

② 새로운 이미지 데이터의 모든 값을 `0`으로 초기화한다.

③ 세로축의 가운데를 기준으로 위쪽은 위로 확장되며, 모든 세로축 데이터가 `rateToMove`의 비율에 의하여 일정하게 확장된다.

④ 세로축의 가운데를 기준으로 아래쪽은 아래로 확장되며, 가장 아래쪽에서부터 순차적으로 진행된다.

지금까지 이미지 데이터의 변형을 설명하였는데, 사진의 이미지 변형도 이와 같은 방법으로 이루어진다. 즉, 사진도 가로와 세로가 정해져 있고, 특정 위치에 픽셀이 존재하기 때문에 픽셀의 위치를 수학적으로 계산해서 변경해 주면 된다. 포토샵과 같은 사진 프로그램도 이와 같은 원리에 기반한다. 포토샵과 같은 프로그램을 프로그래밍적으로 분석한다면 포토샵의 레이어Layer 하나가 사진 객체 하나라고 볼 수 있고, 다수의 레이어가 있다는 것은 다수의 사진을 하나의 파일에서 관리한다는 의미로 해석할 수 있다.

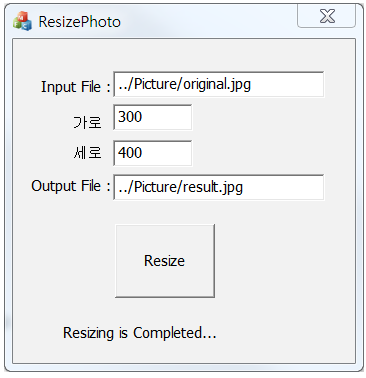
부록

A.1 사진 크기 변경하기

사진의 크기를 변경하는 것은 의외로 간단하다. 이미 이 책의 '4.1 사진 크기 변경 및 흑백 사진 만들기'에서 설명하였으나 여기서는 개별적으로 사진의 크기만을 변경하는 간단한 프로그램을 설명하겠다.

[그림 A-1]은 구현하려는 프로그램의 실행 화면이다. 가로와 세로의 크기를 특정 값으로 설정할 수 있고, 가로나 세로의 값을 0으로 설정하면 원본 사진의 가로와 세로의 비율을 유지하면서 크기를 변경할 수 있다. [Input File]에는 축소를 위한 원본 사진의 파일 이름을 입력하고, [Output File]에는 축소된 사진의 파일 이름을 입력한다.

그림 A-1 사진 크기 변경하는 프로그램 실행 화면



[그림 A-2]는 원본 사진과 축소를 진행한 결과 사진이다. 원본 사진은 DSLR 카메라로 찍어서 사진의 크기가 '2700×4050'이지만, 프로그램을 실행하여 사진의 크기를 '300×400'으로 축소하였다. 일반 사용자는 사진 편집 프로그램 중에서 크기를 줄여 주는 프로그램을 가장 많이 사용하는데, 이때 자신이 직접 만든 프로그램을 사용한다면 더 많은 재미를 느끼게 될 것이다. 자신이 원하는 프로그램을 직접 만들면서 컴퓨터 게임보다 더 알찬 시간을 보낼 수 있다는 것은 프로그래머로 살아가는 재미 중의 하나다. 컴퓨터 게임이 시간을 소비하는 일이라면 프로그래밍은 창조적으로 시간을 할애하는 작업이라고 할 수 있다.

그림 A-2 사진의 크기 줄이기 결과

original.jpg: 2700 × 4050

result.jpg: 300 × 400

[코드 A-1]은 사진의 크기를 변경하는 함수를 구현한 것이다. 가로나 세로 중 하나의 값이 0일 경우에는 원본 사진의 가로와 세로의 비율을 유지하면서 사진의 크기를 축소하도록 구현하였다. 실제 프로그램을 실행할 때 가로나 세로의 값 하나를 0으로 설정한 후 프로그램을 실행하여 결과를 확인하면 이해가 조금 더 용이할 것이다.

[코드 A-1] 사진의 크기를 변경하는 함수(PhotoProcess.cpp)

```
void CPhotoProcess::ResizePhoto(CImage *image, int width, int height, CString
outputFile) {
    CImage newImage;

    if (width == 0 && height == 0) {                          //---①
        AfxMessageBox(_T("Error: 사진의크기를다시설정하세요."));
        return;
    }

    int widthImage = image->GetWidth();                       //---②
    int heightImage = image->GetHeight();

    if (width == 0)                                           //---③
        width = (int)((widthImage * height) / heightImage);
    else if (height == 0)                                     //---④
        height = (int)((heightImage * width) / widthImage);

    float xRate, yRate;

    xRate = (float)(widthImage) / (width);                    //---⑤
    yRate = (float)(heightImage) / (height);

    newImage.Create(width, height, 32);                       //---⑥

    for (int i=0; i<width; i++)                               //---⑦
        for(int j=0; j<height; j++)
            newImage.SetPixel(i,j, image->GetPixel((xRate * i), (yRate * j)));

    newImage.Save(outputFile, Gdiplus::ImageFormatJPEG);      //---⑧
}
```

① 가로와 세로의 값 모두가 0이면 프로그램의 오류이므로 에러 메시지를 보여 준다.

② 원본 이미지의 가로와 세로의 값을 widthImage와 heightImage 변수에 저장한다.

③ 프로그램 화면에서 가로의 값이 0이면 세로의 값을 기준으로 원본 이미지의 가로와 세로의 비율을 유지하는 새로운 값으로 가로의 값을 설정한다.

④ 프로그램 화면에서 세로의 값이 0이면 가로의 값을 기준으로 원본 이미지의 가로와 세로의 비율을 유지하는 새로운 값으로 세로의 값을 설정한다.

⑤ 원본 이미지를 축소하기 위해 중간에 건너뛰어야 하는 가로와 세로의 값을 xRate와 yRate에 저장한다. 이 비율에 의해서 일정한 간격으로 원본 이미지의 픽셀을 얻어서 새로운 이미지에 넣으면 된다.

⑥ 가로와 세로의 크기를 가지고 새로운 이미지를 생성한다.

⑦ 일정한 비율로 원본 이미지의 픽셀을 얻은 후 새로운 이미지의 모든 픽셀에 넣어 준다.

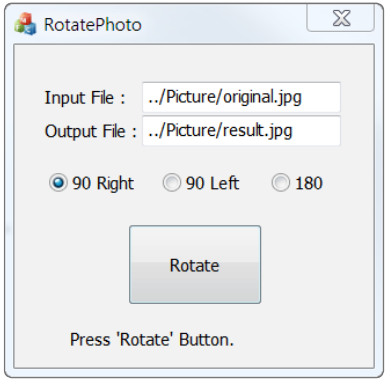
⑧ 축소된 새로운 이미지를 사진 파일에 저장한다.

사진을 축소하는 코드는 간단하며 이 코드를 응용하여 조금 더 유용한 프로그램을 만들 수 있을 것이다. 필자는 코드를 간단히 하기 위해 실행 화면에서 사진 파일 이름을 직접 타이핑하게 만들었지만, 마우스로 파일을 드래그하여 자동으로 넣어 주는 프로그램을 구현한다면 조금 더 편리한 프로그램이 만들어질 것이다.

A.2 사진 회전하기

[그림 A-3]은 사진을 회전하는 프로그램의 실행 화면이다. 사진을 회전하는 방법도 간단하여 시계 방향(90 Right) 회전, 시계 반대 방향(90 Left) 회전, 180도로 회전하도록 선택할 수 있다.

그림 A-3 사진을 회전하는 프로그램 실행 화면



[그림 A-4]는 시계 방향(90 Right)으로 사진을 회전한 결과다.

그림 A-4 시계 방향(90 Right) 회전 결과

original.jpg: 2700 × 4050

result.jpg: 4050 × 2700

[코드 A-2]는 사진을 회전하는 함수를 구현한 것으로, 함수의 매개변수 degree에 따라 사진을 회전한다. degree가 0이면 시계 방향(90 Right)으로, 1이면 시계 반대 방향(90 Left)으로, 2이면 180도로 회전을 수행한다.

[코드 A-2] 사진을 회전하는 함수(PhotoProcess.cpp)

```
void CPhotoProcess::RotatePhoto(CImage *image, int degree, CString outputFile)
{
    CImage newImage;

    int width = image->GetWidth();
    int height = image->GetHeight();

    if (degree == 0) {                                          //---①
        newImage.Create(height, width, 32);                     //---②

        for (int i=0; i<height; i++)                            //---③
        for(int j=0; j<width; j++)
            newImage.SetPixel(i,j, image->GetPixel(width - j - 1, height - i - 1));

    } else if (degree == 1) {                                   //---④
        newImage.Create(height, width, 32);

        for (int i=0; i<height; i++)
        for(int j=0; j<width; j++)
            newImage.SetPixel(i,j, image->GetPixel(j, i));
```

```
    } else if (degree == 2) {                                    //---⑤
        newImage.Create(width, height, 32);
        for (int i=0; i<width; i++)
        for(int j=0; j<height; j++)
            newImage.SetPixel(i,j, image->GetPixel(width - i -1, height - j - 1));
    }
    newImage.Save(outputFile, Gdiplus::ImageFormatJPEG);         //---⑥
}
```

① 원본 이미지를 시계 방향(90 Right)으로 회전하는 프로세스를 실행한다.

② height와 width의 크기를 가진 새로운 이미지를 생성한다.

③ 원본 이미지에서 모든 픽셀을 하나씩 가져와서 새로운 이미지에 차례대로 넣어 준다.

④ 원본 이미지를 시계 반대 방향(90 Left)으로 회전하는 프로세스를 실행한다.

⑤ 원본 이미지를 180도 회전하는 프로세스를 실행한다.

⑥ 회전이 완료된 새로운 이미지를 사진 파일에 저장한다.

사진을 회전하게 하는 프로그램을 많이 사용하지는 않지만, 이 코드는 픽셀의 위치를 변경하는 프로그래밍 연습으로 아주 좋은 예가 될 수 있다. 많은 사진을 찍은 후 확인하다 보면 간혹 사진의 방향이 올바르지 않을 때가 있는데, 윈도우에서는 이러한 경우에 쉽게 사진을 회전시키는 환경을 제공한다. 이 경우 이와 같은 원리로 변경 작업이 이루어진 것이라고 보면 된다.

A.3 흑백 사진 만들기

'4.1 사진 크기 변경 및 흑백 사진 만들기'에서 사진의 크기를 변경하고 컬러 사진을 흑백 사진으로 만드는 것에 대하여 설명하였다. 여기서는 흑백 사진을 만드는 방법만을 설명하겠다.

컬러 사진은 RGB[Red, Green, Blue]의 세 가지 색의 값이 개별적이다. 흑백 사진은 RGB 값을 가지고 있지만 세 가지 색이 모두 동일한 값을 가지고 있다. 따라서 컬

러 사진을 흑백 사진으로 만들기 위해서는 RGB의 세 가지 값을 하나의 값으로 만들어서 RGB의 세 가지 값에 동일하게 넣어 주면 된다.

[그림 A-4]는 컬러 사진의 RGB에서 세 가지 값을 가지고 하나의 값으로 만드는 수식이다. 세 가지 색이 흑백 사진에 영향을 미치는 비율로 수식이 이루어졌다. 즉, 100을 기준으로 할 때 Red는 흑백 사진에 30의 영향을 주고, Green은 59로 가장 큰 영향을 미치며, Blue는 11로 가장 미미한 영향을 준다.

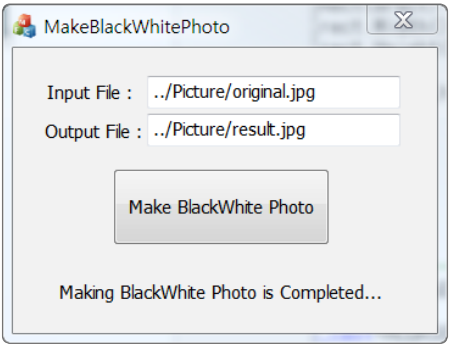
그림 A-4 흑백 사진에 영향을 미치는 RGB 비율

- Red : 30%
- Green : 59%
- Blue : 11%

➔ ((Red * 30) + (Green * 59) + (Blue * 11)) / 100

[그림 A-5]는 여기서 구현한 프로그램의 실행 화면이다. [Input File]에는 컬러 사진의 파일 이름을 넣고, [Output File]은 흑백 사진으로 만들어진 후에 저장되는 사진 파일 이름을 넣는다.

그림 A-5 흑백 사진을 만드는 프로그램 실행 화면



[그림 A-6]은 프로그램을 실행한 후의 결과다. 컬러 사진(왼쪽)이 크기가 같은 흑백 사진(오른쪽)으로 변경되었다. 컬러 사진이 흑백 사진으로 변경되었어도 사진의 명암은 그대로이며 분위기 있는 사진이 되었다.

그림 A-6 흑백 사진을 만든 결과

original.jpg: 2700 × 4050

result.jpg: 2700 × 4050

[코드 A-3]은 흑백 사진을 만드는 함수를 구현한 것이다. 사진의 모든 픽셀을 차례대로 가져와 RGB의 값을 하나의 값으로 계산한 후 다시 RGB의 세 가지 색에 동일한 값을 넣은 다음 픽셀을 변경하는 원리다.

[코드 A-3] 흑백 사진을 만드는 함수(PhotoProcess.cpp)

```
void CPhotoProcess::MakeBlackWhitePhoto(CImage *image, CString outputFile) {
    CImage newImage;
    COLORREF rgb;

    int width = image->GetWidth();
    int height = image->GetHeight();

    newImage.Create(width, height, 32);                          //---①

    for (int i=0; i<width; i++) {
        for(int j=0; j<height; j++) {

            rgb = image->GetPixel(i, j);                         //---②

            BYTE byGray = (GetRValue(rgb) * 30 +
                           GetGValue(rgb) * 59 +
                           GetBValue(rgb) * 11) / 100;           //---③
```

```
            rgb = RGB(byGray, byGray, byGray);                    //---④
            newImage.SetPixel(i, j, rgb);                         //---⑤
        }
    }
    newImage.Save(outputFile, Gdiplus::ImageFormatJPEG);          //---⑥
}
```

① 원본 이미지와 동일한 크기의 새로운 이미지를 생성한다.

② 원본 이미지의 픽셀을 차례대로 가져온다.

③ 원본 이미지의 RGB의 세 가지 색을 가지고 하나의 값으로 계산한다.

④ 하나의 값을 RGB의 세 가지 색에 동일하게 대입한다.

⑤ 새롭게 만들어진 픽셀을 새로운 이미지에 넣는다.

⑥ 새로운 이미지를 사진 파일에 저장한다.

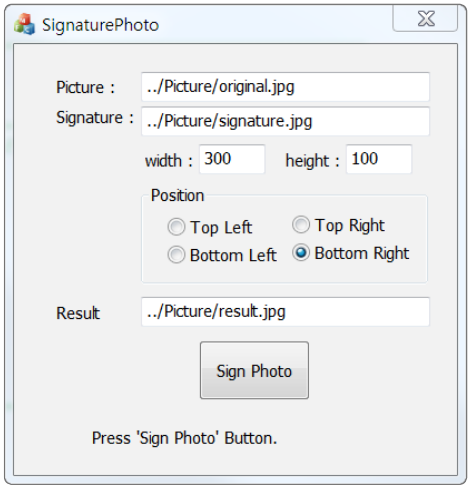
필자가 어린이였을 때는 흑백 사진과 흑백 TV의 시대였으며 컬러 TV가 출현하면서 새로운 시대가 되었음을 느꼈다. 컬러 시대인 요즈음은 사진을 찍을 때 일부러 흑백 사진을 찍곤 하는데 흑백 사진이 주는 과거로의 감정을 느끼게 된다. 세상에는 다양한 색이 존재하지만 간혹 흑백으로 연출되는 세상을 바라보는 것도 좋다.

A.4 서명 넣기

인터넷에 사진을 올릴 때 자신의 서명Signature을 넣는 것이 중요할 수 있다. 자신의 홈페이지를 전문적으로 관리하는 사람이라면 자신이 찍은 사진에 서명해서 올리는 것이 저작권 보호에도 좋으며 자신의 홈페이지 홍보에도 좋은 방법이다. 이번 절에서는 사진에 서명을 넣는 방법을 설명한다.

[그림 A-7]은 프로그램 실행 화면이다. 서명의 가로와 세로의 길이는 [width]와 [height]에 넣고 서명의 위치는 [Position] 항목에서 선택할 수 있다.

그림 A-7 사진에 서명을 삽입하는 프로그램 실행 화면



[그림 A-8]은 사진에 넣을 서명 이미지다. 예제에서 사용하는 사진에 노란색 서명이 눈에 잘 들어오기 때문에 서명의 색을 노란색으로 정의하였다. 서명을 사진에 넣는 방법은 글씨를 기준으로 할 수도 있고, 서명의 색이 다양하다면 바탕색을 하나의 색으로 통일해서 바탕색이 아닌 경우에 사진의 픽셀을 서명으로 변경할 수도 있다. 이 책에서 구현하는 코드는 바탕색을 흰색으로 정해서 픽셀이 흰색이 아닌 경우에 사진의 서명이 들어갈 부분의 픽셀을 서명의 픽셀로 변경하는 것으로 구현하였다.

그림 A-8 서명 이미지

[그림 A-9]는 사진에 서명을 넣은 결과다. 오른쪽 사진의 하단에 'kimsehoon'이라는 서명이 삽입되었다. 이 책에서는 서명을 간단하게 만들었지만, 자신만의 서명을 포토샵으로 멋있게 만들어 넣으면 사진 찍는 일이 더 재미있을 것이다.

그림 A-9 사진에 서명을 삽입한 결과

original.jpg: 864 x 1296

result.jpg: 864 x 1296

[코드 A-4]은 사진에 서명을 삽입하는 함수다. 서명을 넣을 위치의 시작점인 startX와 startY의 위치를 설정한 후에 서명의 바탕색인 흰색이 아닌 픽셀을 차례대로 사진에 넣는 프로세스를 구현한 것이다.

[코드 A-4] 사진에 서명(Signature)을 삽입하는 함수(PhotoProcess.cpp)

```
void CPhotoProcess::SignPhoto(CImage *image, CImage *signature, int position,
int width, int height, CString resultFile) {
    CImage newSignature;

    newSignature.Create(width, height, 32);                    //---①

    float xRate, yRate;
    xRate = (float)(signature->GetWidth()) / width;            //---②
    yRate = (float)(signature->GetHeight()) / height;

    for (int i=0; i<width; i++)                    //---③
        for(int j=0; j<height; j++)
        newSignature.SetPixel(i,j, signature->GetPixel((xRate * i), (yRate * j)));

    int widthImage = image->GetWidth();            //---④
    int heightImage = image->GetHeight();

    int startX, startY;                            //---⑤
```

```
    if (position == 0) {                                    //---⑥
        startX = 30;
        startY = 30;
    } else if (position == 1) {                             //---⑦
        startX = widthImage - width - 30;
        startY = 30;
    } else if (position == 2) {                             //---⑧
        startX = 30;
        startY = heightImage - height - 30;
    } else if (position == 3) {                             //---⑨
        startX = widthImage - width - 30;
        startY = heightImage - height - 30;
    }

    COLORREF rgb;

    int R = 255;
    int G = 255;
    int B = 255;
    int gap = 100;

    for (int x=0; x<width; x++) {                           //---⑩
        for(int y=0; y<height; y++) {
            rgb = newSignature.GetPixel(x,y);

            if (!((GetRValue(rgb) > (R-gap) && GetRValue(rgb) < (R+gap)) &&
                (GetGValue(rgb) > (G-gap) && GetGValue(rgb) < (G+gap)) &&
                (GetBValue(rgb) > (B-gap) && GetBValue(rgb) < (B+gap)))) {

                image->SetPixel(x+startX, y+startY, newSignature.GetPixel(x,y));
            }
        }
    }
    image->Save(resultFile, Gdiplus::ImageFormatJPEG);
}
```

① 서명의 이미지를 설정한 서명의 크기에 맞추기 위해서 width와 height의 길이를 가진 새로운 서명 이미지를 생성한다.

② 서명 이미지의 크기를 조정하기 위해 사진의 크기 줄이기와 같은 방법으로 새로운 서명 이미지를 만들어 준다. 이를 위해 서명에서 픽셀을 가져오도록 건너뛰는 비율을 xRate와 yRate에 계산하여 넣어 준다.

③ 서명의 원본 이미지를 새로운 크기의 서명 이미지에 넣는다. 사진의 크기를 줄이는 방법

과 동일한 프로세스로 진행된다.

④ 서명을 넣을 원본 이미지의 가로와 세로의 길이를 가져온다.

⑤ 원본 이미지의 서명을 넣을 위치의 시작점을 저장하기 위한 변수다.

⑥ 원본 이미지에서 서명을 넣을 위치는 왼쪽 위이며 시작점을 설정한다.

⑦ 원본 이미지에서 서명을 넣을 위치는 오른쪽 위이며 시작점을 설정한다.

⑧ 원본 이미지에서 서명을 넣을 위치는 왼쪽 아래이며 시작점을 설정한다.

⑨ 원본 이미지에서 서명을 넣을 위치는 오른쪽 아래이며 시작점을 설정한다.

⑩ 서명 이미지에서 바탕색(흰색)이 아닌 픽셀을 원본 이미지의 서명 위치에 넣어 준다. 프로세스 안의 `if` 문은 바탕색이 아닌 경우에 픽셀을 대치하는 코드다. 여기에서 RGB의 모든 값을 기준으로 조건문을 만든 것은 서명 이미지의 바탕색이 흰색이 아닌 다른 색일 경우에도 사용할 수 있는 코드를 만들기 위해서다.

부록에서 설명한 내용은 사진을 가지고 진행할 수 있는 가장 기본적인 것이다. 각 절에서 개별적으로 다룬 내용을 바탕으로 독자가 하나의 프로그램으로 구현하여 사용한다면 프로그램 만드는 작업도 재미있지만 사진을 찍는 일 또한 즐거울 것이다.

글을 마치며

오래전에 OCR 프로그램을 만들었으나 프로그램 코드를 저작권 등록만 하고 사용할 일이 없었다. 'How-to Series'의 세 번째 주제를 OCR로 정하고 난 이후에 이전 프로그램의 코드를 책으로 공개할 수 있게 다듬는 작업과 문서 OCR을 새롭게 구현하는 작업 등 책으로 정리하는 데 10개월이 소요되었다. OCR은 프로그래밍이 생각보다 간단할 수 있음에도 지금까지 만드는 방법을 공개한 사람이 없었다. 이 책을 통하여 더 많은 문자 인식 프로그램이 만들어질 수 있을 것을 기대한다. 스마트폰의 애플리케이션으로 만들진 않았지만, 이 책의 코드를 응용한다면 OCR 기술을 이용한 다양한 애플리케이션을 만들 수 있을 것이다.

최근에 수학식을 사진으로 찍어서 계산하는 애플리케이션을 보고 세계의 언론이 감탄한 적이 있다. 그러나 기술적으로 보면, 이 책에서 다룬 이미지 파싱과 이미지 변형을 적절히 응용한다면 구현할 수 있는 내용이다. 필자도 음성 인식에 대하여 생각을 해 보았지만, FFT[Fast Fourier Transform]을 이용하여 소리를 데이터로 바꾸어 사용해야 하는지, 아니면 다른 방법이 있는지는 알지를 못한다. 하지만 음성 인식도 이미지 인식과 접근하는 방법은 같다. 즉, 비교할 수 있는 기본 데이터 타입을 만든 후, 인식을 위한 데이터를 통계적으로 가장 크게 일치하는 것을 선택하면 된다. 물론 기존에 만들어 놓은 기본 데이터가 없으면 인식은 불가능하다.

필자가 독자에게 바라는 것은 이미지 인식의 기본적인 방법을 이해하여 더 멋진 프로그램을 만드는 것이다. 이 책을 통하여 기본기를 익혔다면 직접 자신만의 프로그램을 만들면서 본인의 프로그래밍 방법을 몸으로 체득하는 것이 좋다. 소프트

웨어는 공간에서 물건을 만드는 것이 아니라 컴퓨터 안에서 가상의 물건을 만드는 것이다. 공간을 차지하는 물건을 만드는 사람만이 장인이 아니라, 프로그래머도 장인과 같다. 단시간에 모든 것을 체득할 수 있는 것이 아니다. 프로그래머는 소프트웨어를 만들면서 고통의 시간을 얼마나 많이 경험했느냐가 실력으로 고스란히 드러난다.

현재 우리나라의 IT 시장은 환경이 열악하기 때문에 고급 인력이 해외로 많이 나가고 있다. 하지만 2020년 안에 우리나라의 IT 시장은 변할 수밖에 없다. IT 시장에는 항상 고급 인력이 필요하며 소수의 고급 인력을 채용하기 위해서 시장이 자연스럽게 변화할 것이다. 따라서 현재의 젊은 사람들이 지금부터 실력을 배양해 나간다면 5년 뒤에는 좋은 프로그래머로서 새롭게 맞이하는 시대에 발맞춰 나갈 수 있을 것이다.

프로그래밍을 한다는 것은 치매가 오기 전까지 일할 수 있다는 것이기도 하다. 컴퓨터 게임으로 시간을 버리는 것이 아니라, 프로그래밍을 통하여 게임보다 더 즐거운 세상을 경험하며, 더하여 즐기면서 항상 더 나은 무엇인가를 만들고 있다는 쾌감을 느끼는 것이 좋다. 앞으로 더 많은 사람들이 프로그래밍의 재미를 느끼기를 바란다.

필자는 개인의 경력을 만들어 나가기 위해 'How-to Series'를 집필하기 시작했지만, 이제는 내가 아는 것을 다른 사람에게 전달해야겠다고 생각하고 있다. 많은 사람들이 자신의 앎을 공유할 때 더 좋은 정보를 다른 사람들이 얻을 수 있다. 넓은 의미로 보면 우리나라의 IT 산업을 위해 작은 보탬이 될 것이다.

'How-to Series' 두 번째 책을 편집해 주신 한빛미디어의 정지연 과장님이 세 번째 책도 다시 맡아 주심에 감사드린다.

한빛 리얼타임

한빛 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾기도 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 First – 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 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 좀 더 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 어려운 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.