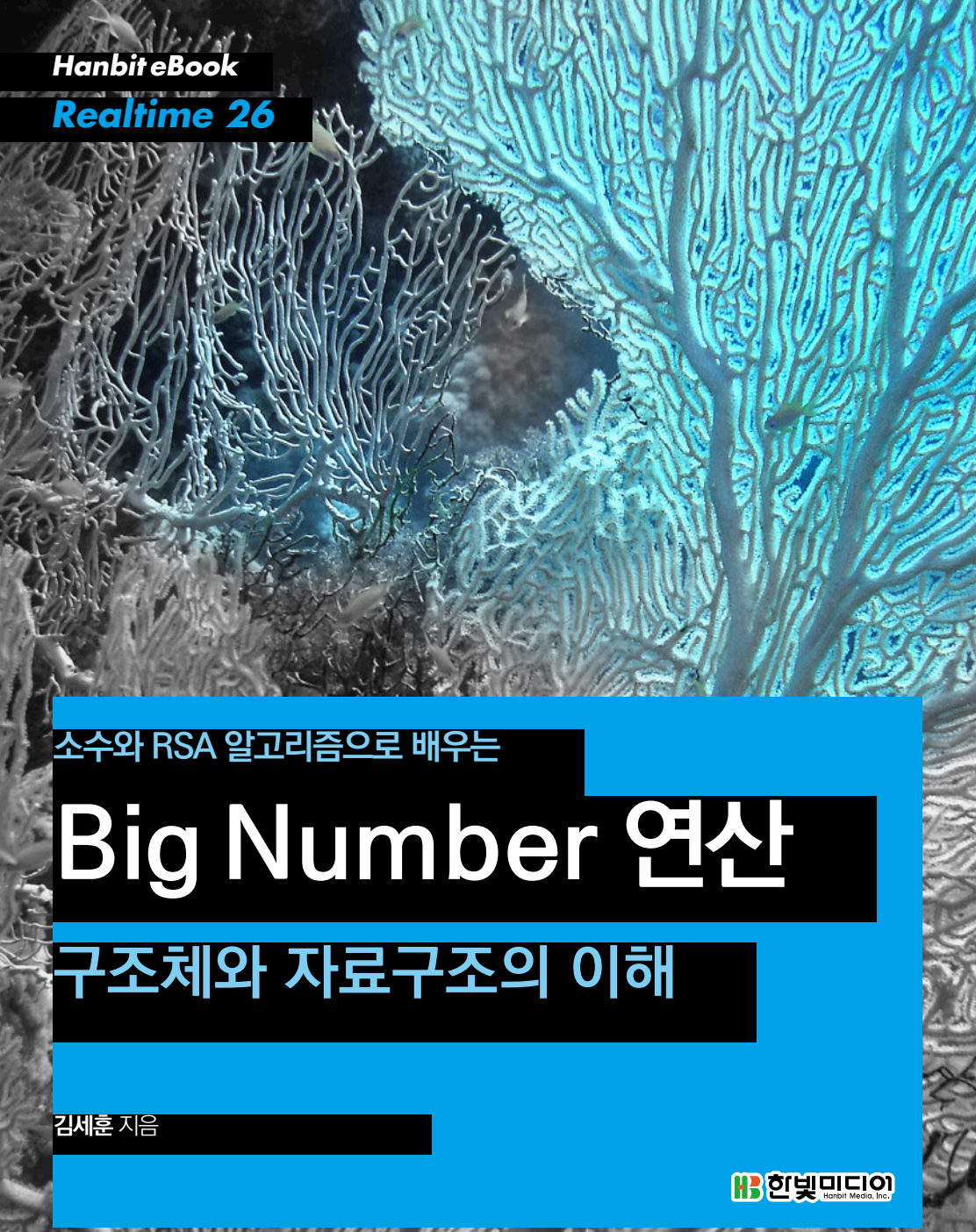


Hanbit eBook

Realtime 26

소수와 RSA 알고리즘으로 배우는

Big Number 연산

구조체와 자료구조의 이해

김세훈 지음

한빛미디어 Hanbit Media, Inc.

소수와 RSA 알고리즘으로 배우는

Big Number 연산

구조체와 자료구조의 이해

소수와 RSA 알고리즘으로 배우는 **Big Number 연산 – 구조체와 자료구조의 이해**

초판발행 2013년 5월 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 978-89-6848-607-4 15000 / **정가** 9,900원

책임편집 배용석 / **기획** 이중민 / **편집** 김연숙
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 박진희
마케팅 박상용, 박주훈, 정민하

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

Published by HANBIT Media, Inc. Printed in Korea

지은이_ **김세훈**

동국대학교에서 수학을 전공하고, University of Waterloo에서 컴퓨터 과학을 전공해 두 개의 학사 학위를 받았다. 현재 (주)에임투지 기술 연구소에서 선임연구원으로 재직 중이다. 70살이 되었을 때도 멋진 프로그래머라는 말을 듣는 것이 꿈이며 지금도 그 꿈을 만들어 가는 중이다.

저자 서문

이 책은 프로그래밍을 공부하기 위한 'How-to Series'의 첫 번째 주제며, 구조체와 자료구조를 이해하여 컴퓨터가 가진 한계를 소프트웨어로 어떻게 극복할 수 있는지를 알아본다. 예를 들어 계산기로 '1 / 3'의 연산을 실행하면 0.3333333333이라는 결과가 도출된다. 그러나 '0.3333333333 × 3'의 연산을 실행하면 1이 되지 않고 0.9999999999라는 결과가 나온다. 즉, 계산기가 스스로 처리할 수 있는 수의 한계가 존재한다는 뜻이다. 이 책에서 설명하는 Big Number는 이러한 수의 한계를 극복하기 위한 노력의 일부라 할 수 있다.

우주로 로켓을 쏘려면 슈퍼컴퓨터로는 연산해야 한다고 들은 적이 있다. 그렇다면 슈퍼컴퓨터는 우리가 사용하는 일반적인 컴퓨터와 무엇이 다를까? 어쩌면 하드웨어 성능이 월등히 뛰어나서 연산을 빠르게 할 뿐, 소프트웨어에는 큰 변화가 없을지도 모른다. 그렇다면 하드웨어의 한계를 소프트웨어로 극복해야만 한다. 따라서 Big Number는 하드웨어의 한계를 소프트웨어로 극복하려는 방법의 하나일 수 있으며, 여러분이 더 나은 소프트웨어를 만들 수 있게 도와주는 시작일 수 있다.

Big Number 연산은 사칙 연산과 고급 연산을 다루는데, 정수만으로 연산을 하는 만큼 알고리즘의 대부분은 누구나 쉽게 이해할 수 있다. 즉, 이 책에서는 쉬운 연산 과정을 코드로 어떻게 구현하는지를 다룬 후, 사칙 연산으로는 큰 소수를 구하는 연산을 실행하고 고급 연산은 현재 가장 많이 사용하는 공개키 암호화 알고리즘인 RSA의 연산을 실행한다.

이 책의 프로그래밍 언어는 C에 기반을 두고 있다. 그리고 가장 중요한 내용은 구조체를 이용해 새로운 자료형Data Type을 만드는 것이다. 즉, 구조체structure로 C의 자료형에는 존재하지 않는 새로운 자료형을 만드는 것이다.

사실 자료형을 만드는 구조체가 발전해 C++의 클래스class가 되었다. 그만큼 프로그래밍에서 구조체가 얼마나 중요한지를 알 수 있다. 필자의 기준에서 C를 잘 이해한다는 것은 프로그램에서 사용하는 자료형을 가장 먼저 이해할 수 있어야 한다는 것을 말한다. 따라서 이 책에서 다루는 새로운 자료형을 이해하고 나면, 어떠한 자료형도 자유자재로 만들 수 있을 것이다.

Big Number 연산의 응용은 크게 두 가지 주제로 나눌 수 있다. 첫째는 '소수Prime Number'고, 둘째는 암호 알고리즘 'RSA'의 구현이다. 소수의 개념은 중학교 수학 수준이며, 'RSA'는 고등학교 수학 수준이다. 단지 여기에 C 기반의 프로그래밍 기법이 가미된 것으로 모르는 부분이 있을 때는 C 입문서를 보면서 이해하면 된다.

프로그래밍은 프로그래밍 언어를 완전히 익힌 후 하는 것이 아니라 프로그래밍 언어의 도움을 받는 것이다. 외우는 분야가 아니고 이해하는 일이 중요하다. 필요한 정보는 책이나 인터넷을 통해 얻을 수 있으므로 이 책을 보기 위해 C 입문서를 정독할 필요는 없다. 혹시 C 입문서를 다 읽고도 프로그래밍을 구현하지 못하겠다면 주제를 정하고 무작정 프로젝트를 진행해보기 바란다. 내가 아는 프로그래밍이란 모든 것을 알고 하는 것이 아니라 그때마다 필요한 것을 맞추어 조립하는 장난감 같은 것이다. 이제 이 책을 통해 내 자식 같은 프로그램을 출가시키려 한다. 내 코드가 어떤 누군가의 인생에 조금이나마 도움이 될 수 있기를 바란다.

마지막으로 필자의 첫 책인 Big Number 연산을 어머니 문언연 여사에게 바친다.

집필을 마치며

김세훈

대상 독자 및 도서 구성

초급 **초중급** 중급 중고급 고급

이 책의 독자는 C 기초를 이해한 초 · 중급자를 대상으로 하며 알고리즘 학습에 도움을 주는 기본적인 수학 함수들을 코드로 어떻게 구현하는지를 설명한다. Big Number라는 주제는 펜으로 계산하기 어려운 큰 수에 대한 것이지만, 여기서 다루는 알고리즘은 초등학교의 사칙 연산부터 시작하므로 쉽게 이해를 할 수 있을 것이다. 큰 소수를 구하는 일은 사칙 연산만으로 계산이 이루어지므로 간단할 수 있으며, 고급 연산은 현재 가장 많이 사용하는 공개키 암호화 알고리즘인 RSA의 이해를 돕기 위해 다룬다. 하지만 RSA를 위한 고급 연산도 결국은 고등학교 수학의 범위를 벗어나지는 않는다. 실제 RSA 연산은 이 책에서 다루는 방법으로 코드를 구현하지 않지만 RSA를 이해할 때 Big Number의 고급 연산으로 많은 도움을 받을 수 있을 것이다.

이 책에 수록된 소스 코드는 특정 플랫폼에 종속되지 않으므로 윈도우 또는 리눅스에서 실행할 수 있으며, 'How-to Series'는 프로그래밍 방법론에 초점을 맞추어 쓴 책이다. 일방적인 지식의 전달이라기보다는 '이러한 방법도 있구나'라고 이해할 수 있기를 바란다.

그리고 이 책에서는 설명하지 않지만, 이 책의 소스 코드를 C++로 구현한 예제 파일을 함께 제공한다. 이 책의 내용을 끝까지 공부한 다음 C++로 구현한 소스 코드를 스스로 분석해본다면 독자 여러분의 프로그래밍 실력도 한층 더 성장할 것으로 기대한다.

예제 테스트 환경 및 예제 파일 다운로드

사용 프로그램	설명
Visual Studio 2008 이상	코드 예제는 윈도우 환경에서 테스트했다
모든 운영체제	운영체제에 종속되지 않는 프로그램이다

● 예제 파일 다운로드

: http://www.hanb.co.kr/exam/2607

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook 입니다.

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

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

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

   500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내시는 선배, 전문가, 고수분에게는 보다 쉽게 집필하실 기회가 되리라 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 일정한 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 보완되고 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여, DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT기기에서 자유롭게 활용하실 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해, 독자 여러분이 언제 어디서 어떤 기기를 사용하시더라도 편리하게 전자책을 보실 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

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

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

1 | 왜 Big Number인가?

프로그래밍 언어는 우리가 일상적으로 사용하는 언어와 비슷하다. 각 나라의 언어가 다양하듯 프로그래밍 언어도 여러 가지 종류가 있으며, 그중 하나의 언어를 선택해 책을 쓰듯 우리는 프로그래밍 언어 하나를 선택하여 프로그램을 만들고 있다.

그런데 일상 언어에서 중요한 것이 문법이 아니라 내용의 전달이듯 프로그래밍 언어에서 정의한 연산자들을 사용하는 것은 기초일 뿐이다. 중요한 것은 함수 사이의 데이터 전달이다. 이러한 의미로 보면 프로그램을 이해하는 데 있어 가장 중요한 것은 전달하는 데이터가 무엇이냐를 아는 것인데, 만약 회사에서 C로 만든 프로그램이 있을 때 전달하는 데이터가 무엇인지 잘 알려면 구조체structure가 어떻게 이루어져 있는지 파악해두는 것이 좋다.

프로그래밍 언어의 기초를 다루는 책에서는 대부분 구조체를 간단히 다루는데, 필자는 실무에서 가장 중요한 것이 구조체라는 점을 강조하고 싶다. 구조체가 발전해 클래스class가 되었으며 C에서 구조체를 잘 다루게 되면 프로그램의 구조를 잘 설계할 수 있다. 물론 프로그램을 만드는 일은 기본적으로 함수를 생성해 나가는 작업이지만 함수를 만들기 전에 구조체를 만들어야 내용을 담아 전달하기 위한 도구로 함수를 사용한다. 즉, 필자가 이 책에서 Big Number를 다루는 이유는 구조체 개념을 이해하는 일이 앞으로의 프로그래밍 인생에 많은 도움이 되기 때문이다.

빅데이터 시대에 접어들면서 이러한 일은 더 빈번하게 발생할 것이다. 하지만 지금부터 설명하는 Big Number는 메모리memory가 허락하는 한 어떠한 수도 표현할 수 있다. 즉, Big Number는 대량의 데이터를 처리할 때 수반되는 하드웨어의 한계를 소프트웨어로 극복할 수 있는 기초 개념이다.

2 | 새로운 자료형 정의

Big Number의 핵심은 구조체로 새로운 자료형을 정의하는 일이다. 이 새로운 자료형을 정의하다 보면 중급 프로그래머로 도약하는 데 필요한 알고리즘을 이해할 수 있게 될 뿐만 아니라 메모리 중심의 자료구조를 이해하면서 큰 데이터를 자유롭게 다룰 수 있는 원리를 이해할 수도 있게 된다.

지금부터는 Big Number를 저장하기 위해 구조체로 새로운 자료형을 만들어서 필요한 연산을 실행할 것이다. 이 책에서는 10진수로 저장하는 구조체와 2진수로 저장하는 구조체 두 가지를 다룬다. 지금부터 살펴보도록 하자.

2.1 포인터

구조체를 잘 사용하려면 먼저 포인터를 제대로 이해할 수 있어야 한다.

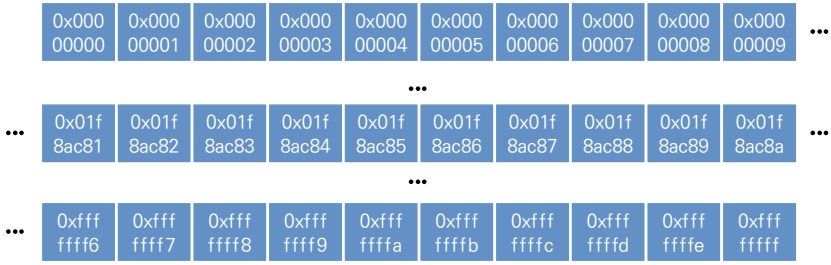
포인터를 잘 사용한다는 것은 하드웨어의 한계를 파악하고 소프트웨어로 하드웨어를 효율적으로 사용할 수 있다는 의미이기도 하다. 자바Java 같은 객체 지향 프로그래밍 언어는 포인터 연산을 없애 프로그래머를 편하게 해주었다고 한다. 하지만 이러한 객체 지향 프로그래밍 언어는 포인터 연산만 없앴을 뿐 객체Object를 포인터 개념으로 사용한다. C/C++가 어려운 이유 중 하나는 포인터 때문이며, C/C++로 프로그램을 만들어본 사람은 다른 프로그래밍 언어를 쉽게 배울 수 있지만 포인터를 사용해보지 않은 프로그래머는 C/C++를 어렵게 생각한다. 사실 포인터를 자유자재로 사용할 수 있다면 다른 프로그래밍 언어를 다루는 것은 따분한 일이라고 생각할 정도다. 포인터를 잘 다룬다는 것은 메모리를 잘 이해한다는 것이므로 먼저 메모리에 관해 알아야 한다.

1차원 메모리RAM 구조를 생각해보자. 물론 하드웨어 구조상으로는 2차원이지만, 여기서 다루는 메모리 개념은 순차적인 것으로서 자신의 주소를 가지므로 1차원 구조라고 생각해도 무방하다. 그림 2-1은 주소를 가진 1차원 행렬로 각 셀cell은 1바이트를 가지게 된다.

그림 2-1 순차적인 1차원 메모리 구조

위와 같은 메모리 구조는 2차원 배열이어도 순차적인 번호를 갖게 되며, 순차적인 번호는 address라고 불리는 메모리의 주소다. 실제 메모리 관리는 운영체제가 담당하므로 프로그래머는 메모리 구조를 자세히 알 필요는 없다. 단지 운영체제가 할당하는 메모리 주소만 알면 메모리를 읽고 쓸 수 있다. 그림 2-2는 32비트 운영체제에서 어떻게 주소를 지정하는지를 보여준다.

그림 2-2 주소로 순차 표현한 메모리 구조

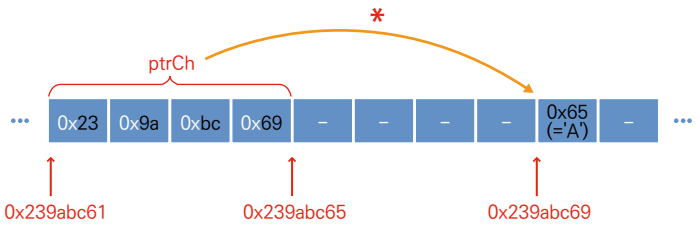


그림에서 볼 수 있듯이 32비트 운영체제에서는 2^{32}바이트(약 4GB)의 메모리만 관리할 수 있다. 그래서 일반 컴퓨터에서 4GB 이상의 메모리를 사용하려면 64비트 운영체제를 사용해야 한다. 우리가 만드는 프로그램은 메모리의 첫 부분부터 사용하는 것이 아니라 운영체제가 메모리의 중간 부분을 사용하도록 허가하는 것이다.

즉, 프로그래머가 만든 프로그램은 운영체제가 허가한 메모리 안에서만 동작하고, 포인터는 말 그대로 '가리키는 것'으로 메모리를 가리키며, 이 메모리의 주소를 가지고 동작한다고 보면 된다.

어떤 메모리 공간에 데이터가 존재한다면 포인터는 데이터가 존재하는 메모리의 주소를 가리킨다. 즉, 어떤 변수가 포인터 변수라면 해당 변수는 메모리의 주소값만을 가지는 것이며, 32비트(4바이트)의 공간만을 가진다. 즉, 모든 포인터 변수의 크기는 4바이트다. 예를 들어 char *ptrCh = 'A';라고 입력한 코드는 그림 2-3과 같은 메모리 구조를 가지게 된다.

그림 2-3 포인터 변수의 메모리 구조



포인터는 변수 앞에 '*'(애스터리스크)를 입력해 선언하며 '가리킨다'는 의미로 해석하면 이해하기 쉽다. 위 그림에서 볼 수 있듯이 ptrCh 변수는 4바이트의 메모리 공간에 주소값 0x239abc69를 가진다. 그런데 지시자인 '*'가 ptrCh 변수 앞에 붙으면 ptrCh 변수 안에 저장한 주소를 읽은 후 메모리 안의 내용을 가져온다.

따라서 포인터를 이해한다는 것은 위 *ptrCh라는 포인터 변수와 포인터가 가리키는 실제 변수 ptrCh를 구분할 줄 안다는 것이다. 즉, char *ptrCh는 ptrCh 변수 크기가 char 자료형의 크기인 1바이트가 아니라 주소값을 저장하는 4바이트고, *ptrCh(즉, 포인터가 가리키는 대상)의 자료형이 char라는 의미다.

포인터 기호 '*'와 항상 같이 사용하는 것은 메모리 주소를 나타내는 '&'(앰퍼샌드)다. 이 기호를 변수 앞에 붙이면 주소값을 가져오라는 의미다. 다음 코드를 살펴보면 알 수 있다.

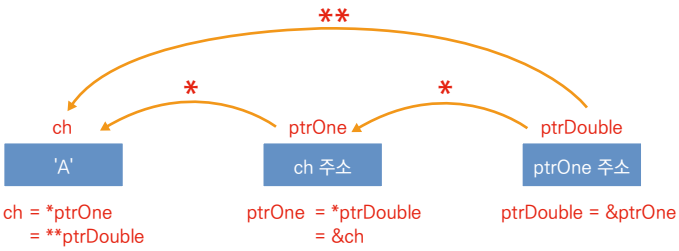
코드 2-1 '&'의 사용 예

```
char ch, *ptrCh;  // 문자를 저장하는 변수 'ch'와 포인터 변수 'ptrCh' 선언
ch = 'A';         // 'ch' 변수에 'A'라는 문자를 삽입
ptrCh = &ch;      // 'ptrCh' 변수에 'ch'의 주소값을 삽입
```

위와 같이 '&'를 이용해 변수의 주소값을 포인터 변수에 대입하면 포인터 변수는 ch를 가리킨다. 즉, ptrCh = &ch; 구문으로 포인터 변수(ptrCh)에 생명을 주었다고 이해하자(ptrCh = &ch; 구문을 선언하기 전의 ptrCh 변수는 임의의 값을 저장하는데, 이는 프로그램에서 아무 쓸모가 없는 데이터다).

이번에는 '*'를 두 번 쓰는 이중 포인터를 알아보기로 하자. 이중 포인터는 말 그대로 두 번 가리켜야 원하는 데이터를 얻는다고 이해하면 된다. 즉, 한 번 가리킨 곳은 주소값을 가지며, 다시 한 번 더 가리킨 곳에 원하는 데이터가 있는 것이다. 이중 포인터는 잘 사용하지는 않지만 포인터를 더 잘 이해하는 데는 많은 도움이 된다. 그림 2-4는 이중 포인터를 어떻게 사용하는지 보여준다.

그림 2-4 이중 포인터



코드 2-2를 보면 이중 포인터를 이해할 수 있다. 코드만 보고 메모리가 어떻게 할당되는지를 생각할 수 있으면 포인터를 충분히 이해했다고 생각해도 좋다.

코드 2-2 이중 포인터의 이해

```
#include <stdio.h>

int main()
{
    char **ptrDouble, *ptrOne, ch = 'A';

    ptrOne = &ch;
    ptrDouble = &ptrOne;

    printf("ptrDouble address : 0x%p \n\n", &ptrDouble);

    printf("ptrDouble value : 0x%p \n", ptrDouble);
    printf("ptrOne address : 0x%p \n\n", &ptrOne);

    printf("ptrOne value : 0x%p \n", ptrOne);
    printf("ch address : 0x%p \n\n", &ch);

    printf("**ptrDouble value: %c \n", **ptrDouble);
    printf(" *ptrOne value: %c \n", *ptrOne);
    printf(" ch value: %c \n\n", ch);
}
```

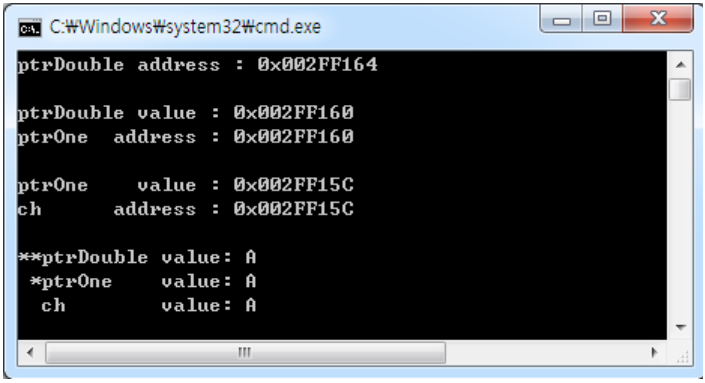
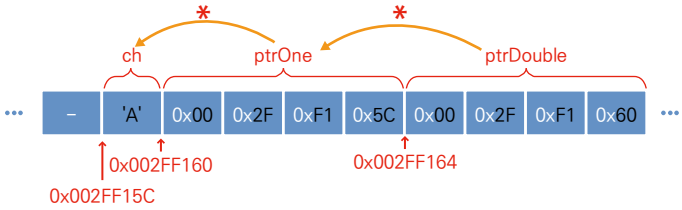


그림 2-5는 실행 결과가 메모리에서 어떻게 구조화되는지를 보여준다. 단일 포인터(ptrOne 변수)와 이중 포인터(ptrDouble 변수)는 주소값을 가진 단순 변수임을 알 수 있다. 그리고 포인터 지시자인 '*'에 의해 다른 곳을 찾아간다.

그림 2-5 코드 2-2 실행 결과에 따른 메모리 구조



정리하면 포인터가 메모리 주소를 저장하는 변수라고 이해하는 순간, 메모리 구조를 이해할 수 있으며 포인터를 자유자재로 사용할 수 있다. 우리가 흔히 접하는 '참조에 의한 호출Call-By-Reference'은 대상을 메모리에 그대로 두고 주소값을 전달하여 대상을 제어한다고 이해하면 된다. 사실 포인터를 이해하고 나면 다른 프로그래밍 언어가 재미없어질 것이다.

포인터는 Big Number 연산을 이해하는 데 가장 중요한 내용이다. 즉, 큰 수를 만들게 되면 메모리에 수를 저장하고 Big Number는 포인터로 메모리의 주소만을 기억하게 된다. 그리고 이 메모리 주소를 부르는 방식으로 연산을 실행해 실제 처리해야 하는 데이터양을 줄여 연산 속도를 빠르게 만들기도 한다.

2.2 구조체

일반 C 입문서에서는 구조체structure를 비중 있게 설명하지 않는다. 생각보다 간단하기 때문이다. 그러나 실무에서 프로그래밍을 하다 보면 구조체는 상당히 중요하며 아주 많이 사용된다는 것을 알게 된다. 또한 Big Number의 모든 수는 구조체로 표현하는데, 이때 큰 수 하나가 하나의 구조체라고 생각하면 이해하기 쉽다.

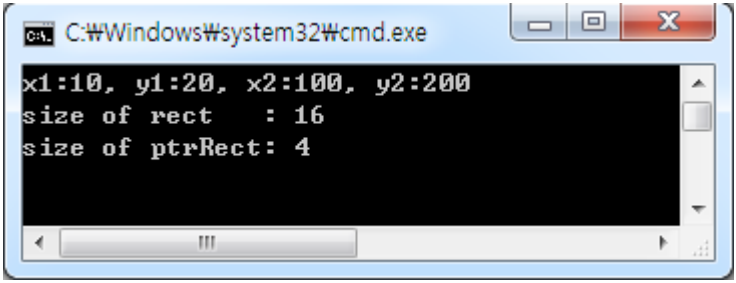
어떤 프로그램을 이해한다는 것은 소스 코드에 포함된 구조체를 완벽히 이해하는 것이다. 구조체는 프로그래머가 만들 수 있는 새로운 자료형이며, 구조체 개념이 발전해 클래스class가 되었다. 즉, 객체 지향 프로그래밍Object-Oriented Programming, OOP이라는 개념은 구조체를 기반으로 만들어졌다고 말할 수 있다.

> **NOTE** 여러분이 어떤 회사에 입사한다면 책에서 보던 기본 자료형(int, char, float, bool 등)이 없는 소스 코드를 볼 수도 있다. 그렇다고 난감해하지 말자. 구조체가 그 회사에 특화되도록 소스 코드를 바꾸어 놓은 것이다. 즉, 회사에서 구조체로 새로운 자료형을 만들어서 사용하는 것이며, 어떤 구조체가 만들어졌는지를 파악하고 나면, 소스 코드를 분석하는 데 큰 문제가 없을 것이다.

이 책에서 설명하는 구조체는 기존의 자료형에 기반을 두고 새로운 자료형을 만드는 것이라고 이해하자. 코드 2-3은 정수형(int)이 모여 점(Point)이 되고, 점(Point)가 모여 사각형(Rectangle)이 되는 구조체의 예제다.

코드 2-3 구조체 예제 - Point & Rect

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

struct Rect {
    Point ptLeftTop;
    Point ptRightBottom;
} rect, *ptrRect; // --- ❶

int main()
{
    // --- ❷
    rect.ptLeftTop.x = 10;
    rect.ptLeftTop.y = 20;
    rect.ptRightBottom.x = 100;
    rect.ptRightBottom.y = 200;

    ptrRect = &rect;

    printf("x1:%d, y1:%d, x2:%d, y2:%d \n",
        // --- ❸
        ptrRect->ptLeftTop.x, ptrRect->ptLeftTop.y,
        ptrRect->ptRightBottom.x, ptrRect->ptRightBottom.y);

    // --- ❹
    printf("size of rect   : %d \n", sizeof(rect));
    printf("size of ptrRect: %d \n", sizeof(ptrRect));
}
```



소스 코드를 살펴보면 다음과 같다.

❶에서는 구조체 변수를 선언했다. rect는 변수로서 메모리에 구조체 공간의 크기(16바이트)를 할당하며, ptrRect는 구조체를 가리키는 포인터 변수로서 메모리의 주소값을 저장하는 4바이트의 공간만 할당한다. 이처럼 구조체를 정의하고 변수를 바로 선언하는 코드는 생각보다 많이 사용한다.

❷에서는 구조체 안에 있는 변수 x와 y에 접근하려고 '.' 기호를 사용한다.

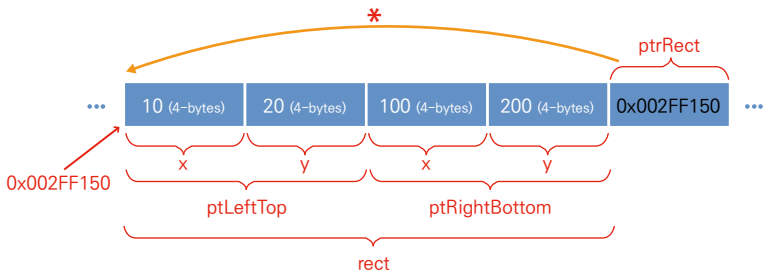
❸에서는 구조체를 가리키는 포인터 변수인 prtRect에서 구조체가 존재하는 메모리로 접근하려고 화살표 모양인 '->' 기호를 사용한다.

❹에서는 rect 변수와 ptrRect 포인터 변수의 크기를 출력한다.

코드 2-3에서 중요한 점은 구조체가 메모리에 어떻게 할당되는지 아는 것이다. 위 소스 코드에 있는 rect와 ptrRect 변수는 그림 2-6처럼 메모리에 할당한다.

또한 rect와 ptrRect 변수는 전역 변수이므로 코드 2-3을 실행하면 메모리의 스택 Stack에 할당해 프로그램 실행이 끝날 때까지 존재하게 된다.

그림 2-6 구조체 변수 rect와 구조체 포인터 변수 ptrRect의 메모리 할당



구조체를 메모리에 할당할 때는 구조체 안에 존재하는 변수의 순서가 중요하다. 구조체로 자료형을 변환해야 하는 경우가 발생하기 때문이다.

2.3 malloc( ) 함수와 free( ) 함수

컴파일러는 우리가 만든 소스 코드를 기계어로 바꾼 후 실행 파일로 만든다. 실행 파일을 실행하면 운영체제는 알아서 메모리를 할당하고 그 안에서 프로그램이 동작한다. 이때 운영체제에서 사용을 허가한 메모리는 프로그램 하나를 위해 여러 개로 나눈다.

나누는 영역은 크게 보면 코드가 존재하는 영역과 프로그램이 동작하는 동안 데이터가 기록되고 지워지는 영역이다. 코드가 존재하는 영역은 프로그램을 실행하는 동안은 변화가 없으나 변수 등의 데이터 영역을 위해 사용하는 영역은 일반적으로 스택Stack과 힙Heap이라는 두 부분으로 나누어지며 프로그램이 실행되는 동안 크기가 수시로 변한다. 예를 들어 스택은 프로그램에서 함수를 실행하면 함수 안에 존재하는 변수값을 저장하고, 힙은 프로그래머가 인위적으로 메모리를 할당하여 사용한다(사실 스택과 힙 외에도 전역 변수(또는 정적 변수static variable)를 저장하는 '데이터'라는 영역도 있으나 이해를 쉽게 하기 위해 생략했다).

그림 2-7은 프로그램을 실행할 때 메모리 영역이 어떻게 나누어지는지를 보여준다.

그림 2-7 프로그램이 사용하는 메모리 영역의 구조

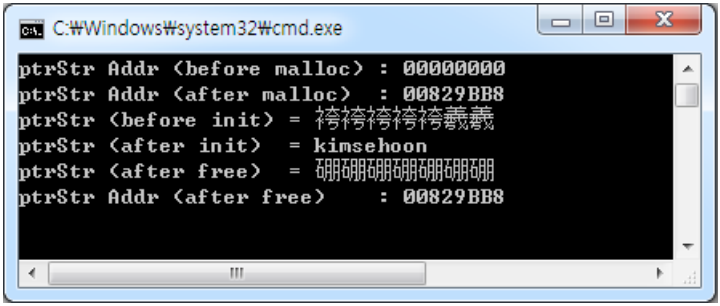
앞으로 설명하게 될 malloc() 함수는 힙 영역을 사용하는 것이며, 메모리 용량이 허락하는 한 원하는 만큼 프로그래머가 사용할 수 있다.

'메모리 할당'을 영어로는 'Memory Allocation'이라고 표현한다. 그래서 메모리를 할당한다는 뜻으로 함수 이름을 malloc이라고 지었다. 그리고 할당한 메모리를 없애주는 함수는 '자유롭게 놔준다'는 의미로 free라고 이름 지었다.

코드 2-4 malloc() 함수와 free() 함수

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i;

    char str[10] = "kimsehoon";
    char *ptrStr;

    printf("ptrStr Addr (before malloc) : %p\n", ptrStr); // --- ❶
```

```
    ptrStr = (char *)malloc(10); // --- ❷

    printf("ptrStr Addr (after malloc)  : %p\n", ptrStr);

    printf("ptrStr (before init) = %s \n", ptrStr); // --- ❸

    // --- ❹
    for(i = 0; i < 10; i++)
        ptrStr[i] = str[i];

    printf("ptrStr (after init) = %s \n", ptrStr);

    free(ptrStr); // --- ❺

    printf("ptrStr (after free) = %s \n", ptrStr);
    printf("ptrStr Addr (after free) : %p\n", ptrStr); // --- ❻
}
```



소스 코드의 중요 부분을 살펴보자.

❶에서의 ptrStr 포인터 변수는 malloc() 함수 사용 전에는 가리키는 곳이 없으므로 임의의 값으로 초기화되었다.

❷에서는 힙 영역에 10바이트를 할당한다. malloc() 함수의 실행 결과는 항상 포인터 변수에 저장하며, (char *)등과 같이 자료형 변환을 해주어야 한다. 따라서 ptrStr 포인터 변수는 할당한 메모리의 제일 앞 번지수를 저장하게 된다.

❸에서는 malloc() 함수로 메모리 영역을 할당받았지만 어떠한 값도 입력하지 않았기에 쓰레기값Garbage Value으로 초기화되었다.

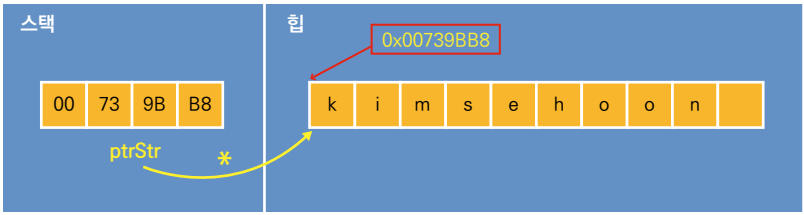
❹에서는 malloc() 함수로 할당받은 힙 영역에 데이터를 삽입한다.

❺에서는 malloc() 함수로 할당받았던 메모리 영역을 free() 함수로 해제한다. 그래서 ptrStr 변수가 가리키는 곳은 다시 쓰레기값을 가진다.

❻에서는 free() 함수로 힙 영역을 삭제해도 ptrStr 변수값에는 변화가 없으므로 힙 영역을 계속 가리킨다.

그림 2-8은 코드 2-4의 실행 결과에 따른 메모리 구조를 표현한 것이다. 실제 프로그램에서 스택과 힙 두 영역을 함께 사용함을 알 수 있다.

그림 2-8 malloc() 함수에 의한 ptrStr 지역 변수의 메모리 구조



지역 변수인 ptrStr은 스택 영역에 저장했으며, malloc() 함수로 할당한 메모리 공간은 힙 영역에 자리한다. ptrStr 변수는 힙 영역에 할당한 메모리 공간을 가리키는 주소값만을 저장한다.

프로그래밍하다 보면 malloc() 함수를 많이 사용하게 되는데, 힙 영역에 할당한 메모리 공간을 free() 함수로 해제하지 않으면 프로그램이 사용하는 메모리가 계속 커지게 된다. 코드가 길면 프로그래머의 실수로 free() 함수를 생략하는 경우도 발생하는데, 처음에는 프로그램에 문제가 생기지 않기 때문에 무심코 지나쳤다가 나중에 어려움을 겪을 수도 있다. 이러한 이유로 메모리 할당은 종종 어렵게 느껴지곤 한다.

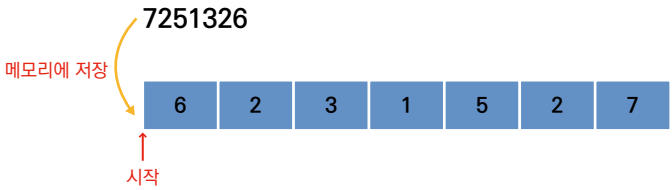
Big Number에서 다루는 수는 크기가 정해진 것이 아니므로 수를 저장하는 메모리 크기가 일정하지 않다. 따라서 수를 생성할 때 malloc() 함수로 수를 저장하는 메모리를 확보해야 하며, 연산 중에 필요 없는 수는 free() 함수로 메모리의 공간을 삭제해야 한다.

2.4 구조체를 이용한 새로운 자료형

이제부터 본격적으로 Big Number를 살펴보겠다. 먼저 구조체로 새로운 자료형을 만드는 원리를 살펴보자.

정수의 크기를 판단하는 기준은 오른쪽이다. 즉, 7251326이라는 숫자는 '$7\times10^6 + 2\times10^5 + 5\times10^4 + 1\times10^3 + 3\times10^2 + 2\times10^1 + 6\times10^0$'으로 표현하며, 오른쪽으로부터 얼마만큼 떨어져 있느냐가 해당 숫자의 크기를 결정한다. 그렇다면 메모리 관점에서는 어떻게 표현할까? 메모리는 값을 저장할 위치가 결정되면 오른쪽으로 증가한다. 즉, 숫자를 메모리에서 표현하려면 숫자 오른쪽 부분을 메모리의 왼쪽 부분에 위치시킨다는 뜻이다. 그리고 수학에서는 소수점이 중요한데, 고정된 소수점의 위치를 프로그램상에서 메모리의 시작 위치로 생각하면 된다. 실제 프로그래밍 언어에서 이를 고려할 필요는 없지만, 우리가 만드는 새로운 자료형은 그림 2-9와 같이 숫자를 표현한다고 기억하자.

그림 2-9 숫자를 메모리상에 표현



위 그림과 같이 10진수의 한 자리 숫자를 메모리상 1바이트만큼의 공간에 입력한다면 10진수의 표현에 따라서 메모리의 크기도 정해질 것이다. 1바이트에서 표현할 수 있는 숫자는 0~255일 테지만, 여기서는 1바이트에서 표현하는 숫자를 0~10까지로 한정한다고 가정하자. 새로운 자료형을 만들 때의 장점은 프로그래머 마음대로 모든 것을 정의할 수 있다는 점이다. 따라서 그림 2-9는 코드 2-5와 같이 선언할 수 있다.

코드 2-5 숫자 7251326을 새로운 자료형으로 표현

```
unsigned char decimal[7] = { 0x06, 0x02, 0x03, 0x01, 0x05, 0x02, 0x07 };
```

그렇다면 Big Number 연산을 위한 새로운 자료형은 어떻게 정의해야 할까? 정수의 특징에 따라야 한다. 먼저 음수인지 양수인지를 표시하는 sign 변수가 필요하고, 값을 저장하고 있는 메모리 공간도 필요하다. 그런데 값을 저장하는 공간은 숫자의 크기에 따라 크기가 유동적일 것이다.

그렇다면 메모리상에서 숫자를 담는 공간을 어떻게 표시해야 할까? 해결 방법은 메모리 공간을 할당할 때 시작하는 곳을 알고, 크기 정보를 가지면 된다. 그래서 새로운 자료형인 BIG_NUMBER라는 구조체는 코드 2-6과 같이 선언한다.

코드 2-6 정수 연산을 위한 BIG_NUMBER 구조체

```
struct BIG_NUMBER {
    unsigned char *ptrSpace;  // ptrSpace는 저장 공간의 시작 번지를 저장
    int size;                 // 저장 공간의 크기(바이트 크기)를 저장
    bool sign;                // 부호 변수(false:양수, true:음수)
};
```

코드 2-6에서는 정수만을 다룬다. 그렇다면 실수를 위한 새로운 자료형은 어떻게 만들까? 소수점의 위치를 저장하는 변수를 추가하면 된다. 프로그래밍에서의 실수 연산은 정수 연산을 실행한 후 소수점 위치를 바꿔주면 된다. 실수 연산을 위한 구조체는 코드 2-7이며 소수점 위치를 저장하는 변수만 추가했다.

코드 2-7 실수 연산을 위한 BIG_FLOAT 구조체

```
struct BIG_FLOAT {
    unsigned char *ptrSpace;  // ptrSpace는 저장 공간의 시작 번지를 저장
    int size;                 // 저장 공간의 크기(바이트 크기)를 저장
    int decimalPoint;         // 소수점 위치를 저장
    bool sign;                // 부호 변수(false:양수, true:음수)
};
```

Big Number 연산에서 실수는 다루지 않을 것이다. 하지만 Big Number 연산을 이해한다면 실수 연산을 구현하는 일도 어렵지 않을 것이다.

2.5 BIG_DECIMAL 구조체

이번에는 앞서 배운 개념을 살려 실제로 이 책에서 사용할 구조체를 만들어보자. 여기서는 Big Number를 10진수로 표현하려고 BIG_DECIMAL이라는 구조체를 정의한다.

이 구조체는 Big Number에 해당하는 모든 수를 표현하게 되는데, 실제 수는 메모리에 저장하며 구조체 안의 포인터가 실제 수를 가리킨다. 구조체의 구현은 코드 2-8과 같다.

코드 2-8 BIG_DECIMAL 구조체

```
struct BIG_DECIMAL { // --- ❶
    unsigned char *digit; // --- ❷
    int size; // --- ❸
    bool sign; // --- ❹
};
```

❶에서는 BIG_DECIMAL이라는 구조체를 정의한다.

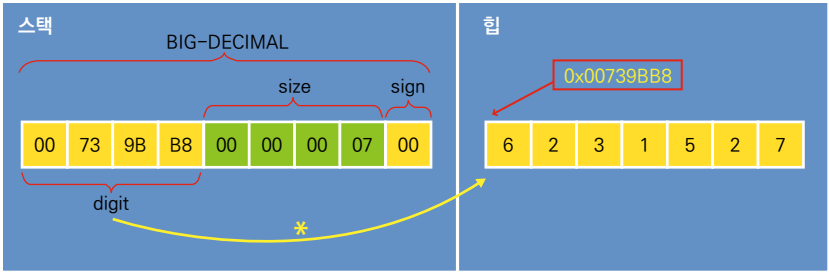
❷에서 digit은 malloc() 함수로 힙 영역에 할당한 주소의 시작 번지를 가지는 포인터 변수며, malloc() 함수에 할당한 공간에 실제 10진수 값을 저장한다. 10진수의 각 자릿수에 해당하는 digit이라는 포인터 변수 하나는 1바이트 크기를 저장하므로 char로 선언했다.

❸에서 size 변수는 digit 포인터 변수가 가리키는 힙 영역에 할당한 메모리 공간의 바이트 크기를 저장한다. 즉, 10진수 숫자의 총 개수라고 보면 된다.

❹에서 sign 변수는 정수 부호를 나타내며 음수는 true, 양수는 false로 표현한다.

BIG_DECIMAL 구조체는 그림 2-10과 같이 스택과 힙 영역 모두에 값을 가지므로, BIG_DECIMAL 구조체의 크기가 항상 9바이트더라도 힙 영역에 할당한 공간을 고려한다면 실제 크기는 유동적이라고 보는 것이 좋다. 그림 2-10은 7251326이라는 10진수가 BIG_DECIMAL 구조체에서 어떻게 표현되는지를 보여준다.

그림 2-10 메모리상에서의 BIG_DECIMAL 구조체



위 그림에서 BIG_DECIMAL 구조체는 스택 영역에 있는 9바이트 크기의 메모리에 저장한다.

처음 4바이트는 포인터 변수인 digit에 해당하며, 실제 값을 저장한 힙 영역의 주소를 저장한다. 구조체의 두 번째 변수인 size는 힙 영역에 할당한 메모리 공간의 크기를 저장한다. 즉, 위 그림에서 size 변수값이 7이면 힙 영역에서 7바이트의 공간을 사용한다고 보면 된다. 마지막으로 sign 변수값이 0이면 BIG_DECIMAL 구조체로 표현하는 숫자가 양수임을 나타낸다.

코드 2-9는 BIG_DECIMAL 구조체로 실제 값을 생성하는 CreateDecimal() 함수로, 문자열과 크기를 매개변수로 받아서 BIG_DECIMAL 구조체로 표현하는 양의 정수를 만든다.

코드 2-9 BIG_DECIMAL 구조체를 이용해 실제 값을 생성하는 CreateDecimal() 함수

```
BIG_DECIMAL CreateDecimal(unsigned char *str, int size) // --- ❶
{
    BIG_DECIMAL decimal;

    decimal.digit = (unsigned char *)malloc(size); // --- ❷
```

```
    // --- ❸
    for(int i = 0; i < size; i++)
        decimal.digit[i] = str[size-i-1] - 48;

    // --- ❹
    decimal.size = size;
    decimal.sign = false;

    return decimal; // --- ❺
}
```

❶에서는 BIG_DECIMAL 구조체를 이용하는 함수를 선언하며, 매개변수로 문자열과 10진수의 크기를 입력받는다. 함수의 사용법은 CreateDeciama((unsigned char*)"7251326", 7)처럼 매개변수로 10진수 문자열(7251326)을 저장하고, 두 번째로 문자열의 길이(7)를 저장한다.

❷에서는 BIG_DECIMAL 구조체를 이용해 실제 값을 저장하려는 공간을 malloc() 함수로 할당한다.

❸에서는 ❷에서 할당받은 공간에 매개변수로 입력받은 값을 삽입한다. for문을 통해 매개변수 문자열의 오른쪽 값은 할당받은 공간의 왼쪽부터 채워지게 된다. 48을 빼는 이유는 '0'이라는 문자의 아스키코드 값이 48이기 때문이다(문자로 받는 값을 실제 값으로 바꿔줄 때는 아스키코드 값을 가지고 처리한다). 즉, 0~9라는 문자열의 실제 값이 48~57이므로 48을 빼주어 실제 값을 0~9로 만들어주는 것이다(이해가 안 되는 분은 'APPENDIX A. 아스키코드' 부분을 참고하기 바란다).

❹에서는 BIG_DECIMAL 구조체의 size 변수와 sign 변수의 값을 입력한다.

❺에서는 반환하는 값이 구조체므로 메모리 관점에서 본다면 BIG_DECIMAL 구조체 크기인 9바이트를 복사한다고 보면 된다. 실제 값을 저장한(malloc() 함수로 할당한) 힙 영역은 지워지지 않고 값을 저장하며, 반환하는 BIG_DECIMAL 구조체 안 힙 영역에 있는 메모리 주소를 저장해 전달하는 것이다.

코드 2-10은 BIG_DECIMAL 구조체로 생성한 decimal이라는 구조체 변수값을 콘솔에 출력하는 함수로, decimal 변수값을 10진수 문자열로 출력한다.

코드 2-10 BIG_DECIMAL 구조체 출력(콘솔)

```
void printDecimal(BIG_DECIMAL decimal) // --- ❶
{
    int i;

    // --- ❷
    if(decimal.sign)
        printf("-");

    // --- ❸
    for(i = decimal.size-1; i >= 0; i--)
        printf("%c", decimal.digit[i]+48);
    printf("\n");
}
```

❶의 printDecimal() 함수는 BIG_DECIMAL 구조체 변수 decimal의 값을 콘솔에 출력한다. 이러한 출력 함수는 초기에 꼭 만들어야 한다. 왜냐하면 테스트 목적으로 가장 많이 사용하기 때문이다. 보통 테스트 코드 작성에 다소 인색한 편인데, 실제 프로그래밍에서는 테스트 코드를 많이 작성할수록 좋다.

❷에서는 decimal 변수의 부호를 출력한다. sign 변수값이 1이면 음수고, 0이면 양수다.

❸에서는 10진수의 왼쪽부터 차례대로 출력한다. 48을 더해주는 이유는 실제 값을 우리가 알아볼 수 있는 문자열로 출력하기 위해서다. 즉, 실제 값인 0~9에 48을 더해 48~57의 값을 만들면 숫자를 나타내는 아스키 문자로 출력된다.

decimal 변수값의 크기가 클 때는 파일에 출력할 필요가 있다. 코드 2-11은 콘솔 대신 파일에 출력하려는 값을 기록하는 함수로, 콘솔 출력에 사용된 printf() 함수를 파일로 출력하는 fprintf() 함수로 대체한 것 외에는 소스 코드 내용이 같다.

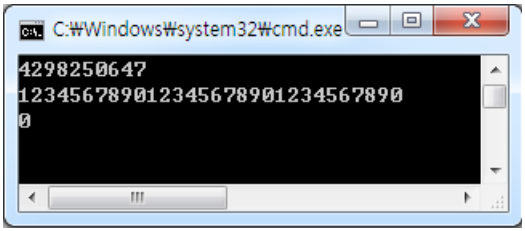
코드 2-11 BIG_DECIMAL 구조체 출력(파일)

```
void fprintDecimal(FILE *fp, BIG_DECIMAL decimal)
{
    int i;

    if(decimal.sign)
        fprintf(fp, "-");
    for(i = decimal.size-1; i >= 0; i--)
        fprintf(fp, "%c", decimal.digit[i]+48);
    fprintf(fp, "\n");
}
```

코드 2-12는 decimal 구조체 변수값을 출력하는 테스트 코드다. 실제 프로그램을 구현할 때는 위에서 설명한 소스 코드보다 테스트 코드를 먼저 작성하는 것이 좋다. 왜냐하면 프로그램을 다 만든 후에 수정하는 것보다는 문제가 있는지를 테스트하면서 프로그램을 만들어 가는 방법이 좋기 때문이다.

코드 2-12 BIG_DECIMAL 구조체 테스트

```
int main()
{
    BIG_DECIMAL decimal;

    decimal = CreateDecimal((unsigned char *)"4298250647",10); // --- ❶
    printDecimal(decimal);

    decimal = CreateDecimal(
        (unsigned char *)"12345678901234567890123456789O",30
    ); // --- ❶
    printDecimal(decimal);

    decimal = CreateDecimal((unsigned char *)"0",1); // --- ❶
    printDecimal(decimal);

    // --- ❷
    FILE *fp;
    if((fp=fopen("result.txt", "wt")) == NULL)
        printf("file open error. \n");

    fprintDecimal(fp, decimal);
        fclose(fp);
}
```



❶에서는 BIG_DECIMAL 구조체를 이용하는 CreateDecimal() 함수를 사용한다. 10진수의 문자열을 넣고 자료형 변환을 해주며 숫자 길이를 입력한다.

❷에서는 출력하려는 값을 저장할 파일을 생성한다.

NOTE BIG_DECIMAL 구조체를 이해했다면 여러분도 Big Number 연산을 직접 구현할 수 있다. 필자는 이해를 돕고 구현을 쉽게 하려고 10진수의 각 자릿수에 해당하는 digit이라는 포인터 변수 각각을 1바이트에서 표현했으나 다른 구조체를 만들어 Big Number를 표현할 수도 있으며 어디까지나 프로그래머가 마음대로 정할 문제다. 즉, 이 책에 소개한 것이 정답은 아니라는 뜻이다.

프로그래밍은 소스 코드를 작성하는 일만 말하는 것이 아니다. 소스 코드 작성은 극히 일부분에 해당하는 작업일 뿐이며, 어떻게 구조화할지 생각하는데 소스 코드 작성 시간의 몇 배 혹은 몇십 배를 소요할 수도 있다. 따라서 어떻게 연산을 실행할지를 스스로 생각해보는 것도 좋은 훈련이다.

여러분이 어떤 회사에 입사한다면 책에서 보던 기본 자료형(int, char, float, bool 등)이 없는 소스 코드를 볼 수도 있다. 그렇다고 난감해하지 말자. 구조체가 그 회사에 특화되도록 소스 코드를 바꾸어 놓은 것이다. 즉, 회사에서 구조체로 새로운 자료형을 만들어서 사용하는 것이며, 어떤 구조체가 만들어졌는지를 파악하고 나면, 소스 코드를 분석하는 데 큰 문제가 없을 것이다.