Hanbit eBook
Realtime 11

Thinking About
C++ STL 프로그래밍

최흥배 지음

한빛미디어
Hanbit Media, Inc.

Thinking About:

C++ STL 프로그래밍

지은이_ **최흥배**

2003년부터 지금까지 보드 게임부터 시작해서 MMORPG 게임까지 다양한 온라인 게임의 서버 프로그램을 만들어 왔다. 게임 개발자이다 보니 프로그래밍 언어 중 C++을 주력 무기로 사용하고 C#을 보조 언어로 사용하고 있다. 다른 프로그래머들과 기술이나 경험을 공유하는 것을 좋아해서 게임 개발자 커뮤니티나 세미나 강연을 통해서 교류하고 있다. 요즘은 C++11 프로그래밍과 Linux 플랫폼 프로그래밍, Mono, Node.js에 대해서 스터디하고 있다. 웹이 대중화되기 전부터 프로그래밍 공부를 해서 그런지 아직도 기술을 배울 때는 책을 더 선호하며 지금도 매달 새로운 프로그래밍 책을 보고 있다. 현재 티쓰리엔터테인먼트 삼국지천 팀에서 근무하고 있다.

- 블로그 : http://jacking.tistory.com/
- 트위터: @jacking75

Thinking About: C++ STL 프로그래밍

초판발행 2012년 12월 21일

지은이 최흥배 / **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 978-89-7914-993-7 15560 / **비매품**

책임편집 배용석 / **기획** 스마트미디어팀 / **편집** 김제룡, 박민근, 이정재, 조진희, 한상곤
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 스마트미디어팀
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

저자 서문

C++는 대학생 때부터 공부를 시작해서 시간 상으로는 공부한지 15년이 넘었다. 하지만 여전히 공부할 것이 있어서 난감함을 주지만 그만큼 깊게 팔 것이 많고 자유도가 높아서 내가 가장 좋아하는 프로그래밍 언어다. 프로그래밍 공부를 막 시작했을 때는 C++가 프로그래밍 언어 중 가장 인기 있는 언어였는데, 웹 시대가 시작되면서 다른 언어에 자리를 내어주고 지금은 올드한 느낌을 주는 언어가 되었다.

하지만 C++은 아직 죽은 언어가 아니다. 여전히 하드웨어 제어나 고성능 프로그램을 만들 때 주력 언어로 사용하는 등 나름 자신만의 영역을 튼튼히 가지고 있다. 특히 게임 개발에서는 C++이 많은 사랑을 받고 있다. 얼마 전까지 비디오 게임 및 온라인 게임 개발에 주로 사용되다가 요즘은 큰 인기를 얻고 있는 스마트 폰 게임 개발에서 높은 성능과 멀티 플랫폼 대응을 위해서 사용되어 오히려 이전보다 사용 영역이 더 늘어 난 것 같다.

C++이 최고 인기 프로그래밍 언어의 자리를 내주게 된 이유는 여러 가지 있지만 가장 큰 이유 중 하나가 생산성이다. 그래서 C++ 세계에서는 이 생산성이라는 부분을 어떻게 하면 향상시킬 수 있을까 고민을 하는데, 이 고민을 덜어주는 것이 바로 STL 즉 표준 템플릿 라이브러리(Standard Template Library)다.

2000년 초반만 하더라도 STL의 효용성에 대해서 찬반이 있었는데, 지금은 STL을 사용하지 않고 C++ 프로그래밍을 한다는 것은 상상하기 힘들다. 아주 특별한 경우가 아닌 이상, C++ 프로그래밍에서는 STL을 필수적으로 사용하고 있다.

STL 덕분에 C++ 프로그래밍의 난이도는 내려가고 이전에 비해서 생산성이 증가 되었다. 그래서 작년 말에 나온 C++ 새로운 표준인 C++11에서도 STL을 중요하게 여겨 다양한 기능이 추가 되었다. 앞으로 나올 새로운 표준에도 STL에 다양한 기능이 들어갈 예정이다.

글을 집필할 때, 최대한 STL의 실용적인 부분에 중점을 두었다. 기존에 나온 STL 책들은 설명이 중심이고 샘플 코드가 실제 사용하는 것과 거리감이 있어서 쉽게 와닿지 않았다. 그래서 게임 개발에서 사용했던 경험을 떠올리면서 최대한 실용적인 경우를 토대로 설명하였다. 이 책은 STL 전체를 설명하는 대신 자주 사용하고 꼭 알고 있어야 하는 것들만 추려서 설명한다. 이 책에 나온 것만 알고 있어도 STL을 사용하는 데 거의 어려움을 느끼지 않으리라 생각한다.

C++ 프로그래머라면 STL 공부는 필수입니다. 이 책을 통해서 STL을 빠르고 쉽게 공부할 수 있기 바랍니다.

끝으로 언제나 다양한 프로그래머 모임에 참여해주시는 등 나에게 큰 힘이 되어 주신 임영기님, 온라인 서버 프로그래머 커뮤니티를 만들어 주신 이지현님, 커뮤니티 운영하느라 고생 많으신 이욱진님, 게임 프로그래머가 되기 전부터 스터디를 통해서 알게 되어 지금은 같은 게임 프로그래머로서 일하면서 많은 도움을 주고 있는 조진현군에게 고마움을 전하고 싶다. 앞으로도 지금처럼 게임 업계에서 재미있는 게임을 만들기 바랍니다.

집필을 마치며,

최홍배

편집자 소개

김제룡

훈민정음이라는 워드프로세서를 개발한 넥스소프트에서 사회생활을 시작하였다. VK MOBILE이라는 핸드폰 제작 회사에서 임베디드 시스템 개발도 해보았지만, 어려서부터 동경하던 게임개발의 꿈을 버리지 못해 게임개발자로 전향한 프로그래머다. 이스트소프트에서 카발온라인1 개발에 참여하였고, 현재는 카발온라인2 개발 및 서비스에 주력하고 있다. 외부활동보다는 조용하게 개발자의 내공을 쌓는 것과 새로운 것을 경험해보는 것을 좋아한다.

박민근

10년간 NC, NTL-inc, 네오위즈 게임즈 등에서 “드래곤볼 온라인”, “야구의 신” 등의 온라인 게임을 개발한 게임 프로그래머다. 현재는 새로운 모바일 회사에서 그토록 염원하던 미소녀TCG를 개발 중인 솔로 오덕 프로그래머다. “게임 개발자 랩소디”라는 팟캐스트를 운영하였으며, 다양한 게임개발 관련 세미나에서 강연을 하고 있다.

이정재

윈도우 분야의 여러 분야를 공부하고 있는 윈도우 개발자다. 좀 더 깊고, 좀 더 많은 경험을 하지 못한 것을 아쉬워하면서 좀 더 나은 개발자가 되고 싶은 꿈을 향해 노력하고 있다. 최근에는 안드로이드 앱 개발을 하고 있다.

조진희

나에게 꿈을 심겨준 컴퓨터와 함께 15년의 세월을 보내고 20대 마무리에 있는 연세대학교 알고리즘연구실 대학원생이다. Positive mind로 항상 살아가기에 힘쓰며, 곧 사회에 나갈 준비를 하는 취준생이기도 하다. 30대에는 보다 다이나믹한 삶을 계획하고 있다.

한상곤

대학원에서 보안시스템 설계를 공부하고 있는 대학원생이다. 평소에 오픈소스 관련 프로젝트를 친구들과 함께 신나게 진행하고 있으며 현재는 파이썬에 관심을 가지고 공부하고 있다. 오프소스도 관심이 많아서 우분투 관련 모임에 꾸준히 참여하고 있다.

편집을 마치며

요즘 전 세계에는 스마트폰의 광풍이 불고 있습니다. 안드로이드와 아이폰의 전쟁이라고 할 수 있는 흐름이 있어서, 안드로이드를 개발하기 위한 자바와 아이폰을 개발하기 위한 오브젝티브 C 언어는 마치 IT개발의 세계를 양분하는 프로그래밍 언어의 전부인듯한 느낌을 받기도 합니다.

STL은 C++을 사용하는 많은 분들이 사용하는 라이브러리입니다. STL 도서의 필요성을 의심하지는 않지만 스마트폰이 유행하는 이 시점에서 STL에 대한 책을 낸다고 했을 때, 시기가 맞지 않는 것은 아닐까라는 생각이 들기도 했습니다.

그러나 예전에 이 책의 근간이 된 인터넷 문서를 통해 STL에 대한 이해를 넓히고, 좀 더 친근하게 다가갈 수 있었던 경험이 생각났고, 제가 받았던 도움과 경험을 다른 사람들에게 나누어 줄 수 있는 기회라고 생각되어 편집에 참여하게 되었습니다. 편집을 진행하는 중에는 저자의 의도와 역량을 제대로 전달하기 위해 노력을 했지만, 의외로 고려할 많은 것이 있다는 것을 알 수 있었습니다. 우리가 읽는 한 권의 책이 나오기 위해서는 저자의 역할과 노력이 많다는 것은 알고 있었지만, 그것 말고도 편집의 어려움과 많은 작업을 직접 경험해 볼 수 있었습니다.

앞으로 책을 읽을 때, 책이 전하는 내용뿐만 아니라 한 권의 책이 나오는데 노력했던 편집자를 포함한 많은 분들의 노고를 한 번쯤 되새기면서 책을 읽게 되지 않을까 생각합니다. 책이 만들어지는 과정을 통해 책의 소중함을 다시 한번 생각할 수 있었던 계기가 되었습니다.

함께 작업했었던 여러 분들(김제룡 님, 박민근 님, 조진희 님, 한상곤 님)과 좋은 기회를 제공해 주신 김병희 대리님과 김창수 팀장님께 감사 드립니다.

편집자 대표

이정재

대상 독자

초급	초중급	중급	중고급	고급

이 책은 STL의 기본적인 개념을 이해하고 사용법을 알고 싶은 개발자를 대상으로 한다. 때문에 독자가 C++의 기본적인 문법과 컴파일러를 사용하는 방법에 익숙하다고 가정한다. 즉, C++의 문법과 컴파일러를 통해 소스코드를 빌드하여 결과를 보는 방법은 논하지 않고 오직 STL의 기본적인 개념을 이해시키고, STL의 사용법을 보여주는 데만 중점을 둘 것이다.

한빛전자책 알림

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 준비 중이며, 조만간 선보일 예정입니다.

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.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 계실 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실

구입하신 도서는 사본을 보관하지 않는다는 조건으로 다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전되지 않습니다.

차례

10 알고리즘 179

1 C++ STL 소개

1.1 STL이 무엇인지 알고 있는가?

C++를 주 프로그래밍 언어로 사용하고 있다면 알고 있으리라 생각한다. STL은 C++ 언어의 '표준 템플릿 라이브러리Standard Template Library'의 약자다. STL을 간단하게 말하자면 일반적으로 많이 사용되는 자료구조와 알고리즘을 모은 라이브러리다.

STL은 C++ 언어가 처음 만들어질 때부터 있었던 것이 아니라 C++ 표준이 정해지기 전인 1993년 말 무렵에 알렉스 스테파노브Alex Stepanov가 C++ 언어의 창시자인 비얀 스트로스트룹Bjarne Stroustrup에게 제안 후 준비 기간을 걸쳐서 1994년에 표준 위원회에서 초안이 통과되었다(참고로 C++ 표준(C++98)은 1989년에 시작되어 1998년 9월에 마무리되었다).

1.2 STL은 어떻게 만들었을까?

물음에 대한 답은 STL의 실제 이름에 포함되어 있다. 좀 더 힌트를 준다면, STL을 구성하는 세 개의 단어 중에서 중간에 있는 단어를 잘 살펴보면 된다. 짐작이 가는가?

정답은 중간에 있는 템플릿Template이다. STL을 이해하려면, STL을 구성하고 있는 C++의 템플릿을 반드시 이해해야 한다. 템플릿은 C++를 더욱 강력하게 사용하는 데 꼭 필요하다.

1.3 언어를 공부한 사람은 템플릿에 대해 잘 알고 있을까?

예전에 C++을 잠시 공부했거나 지금 C++을 공부하고 있더라도 C++ 관련 서적을 한 권 혹은 두 권 정도 읽어본 경우라면 템플릿이라는 단어가 생소할 수 있다.

위에서 언급했듯이 템플릿은 C++이 세상에 나오면서 같이 나온 것이 아니라 1994년 무렵에야 세상에 조금씩 소개되다가 1998년에 C++ 표준이 정립된 이후 C++ 언어의 한 부분으로서 인정되었다.

1994년까지는 템플릿을 지원하는 C++ 컴파일러가 없었고, Microsoft의 C++ 툴로 유명한 Visual C++도 버전 6에서도 템플릿을 완벽하게 지원하지 못했다. 템플릿은 Visual Studio .NET 2003에서부터 제대로 지원되었다(아직도 템플릿 기능을 100% 완벽하게 지원하지는 못한다).

2000년 이전에 나온 C++ 입문서를 보면 템플릿에 대하여 빠뜨린 것이 꽤 많다. 요즘 나오는 입문서도 템플릿 부분이 빠져 있기도 하다. 템플릿은 일반 C++ 입문서에서는 가장 뒷부분에 나오다 보니 공부를 하다가 중간에 포기하게 되면 클래스라는 것은 알아도 템플릿은 잘 모를 수 있다.

개인적으로 C 언어를 생각하면 포인터가 떠오르고, C++ 언어를 생각하면 클래스와 템플릿이 떠오른다. 이유는 C 언어나 C++ 언어를 배울 때 정확하게 이해하기 가장 어려웠던 것이고 필자가 배웠던 다른 언어들에 비해 크게 달랐던 것이기 때문이다.

특히 C언어의 포인터는 처음 배울 때 문법적인 사용방법이 잘 이해가 안 돼서 어려웠지만, C++의 클래스나 템플릿은 문법적인 사용방법이 어려운 것이 아니고 프로그램 설계와 관련해서 클래스와 템플릿을 어떻게 활용해야 하는지 이해하기 어려웠다.

1.4 객체 지향 프로그래밍(OOP) C++

C++ 언어를 소개할 때 가장 먼저 이야기하는 것이 객체지향이라는 것이다. 현대의 많은 프로그래밍 언어들은 객체 지향 프로그래밍OOP(Object-Oriented Programming)을 지원합니다.

C 언어와 C++ 언어는 이름이 비슷하듯이 유사한 부분도 많다. C 언어로 프로그래밍 할 때는 절차 지향 프로그래밍을 하게 된다. C++도 절차 지향 프로그래밍을 할 수 있다. 그러나 제대로 된 C++ 프로그래밍을 하려면 객체 지향 프로그래밍을 해야 한다. 보통 C 언어를 배운 후 바로 이어서 C++를 배울 때는 객체 지향 프로그래밍에 대한 이해가 부족하다. 그래서 C 언어로 프로그래밍 할 때와 같은 절차 지향 프로그래밍을 하여 이른바 'C++를 가장한 C 프로그래밍'을 한다는 소리를 듣기도 한다. C++ 언어로 객체 지향 프로그래밍을 할 수 있는 것은 C 언어에는 없는 클래스가 있기 때문이다.

질문: C++로 할 수 있는 프로그래밍 스타일은 절차적 프로그래밍, 객체 지향 프로그래밍만 있을까?

답변: 아니다. Generic Programming 도 가능하다.

1.5 Generic Programming이라는 것을 들어 보았는가?

필자가 프로그래밍을 배울 때는 일반적으로 C++ 언어를 배우기 전에 C 언어를 공부했다. C 언어를 처음 공부했던 시기가 대략 1994년 즈음 이다. 그 당시의 다른 초보 프로그래머들처럼 포인터의 벽에 부딪혀 좌절하고, 도망(?)가서 3D Studio라는 그래픽 툴을 공부하다가 내가 할 것이 아니라는 생각에 포기하고, 1995년에 다시 C 언어를 공부하였고 이후 바로 C++ 언어를 공부했다.

이때도 OOP라는 단어는 무척 자주 들었고 C++로 프로그래밍을 잘한다는 것은 OOP를 잘한다는 것과 같은 뜻이었다.

대학을 다닐 때부터 내 용돈의 많은 부분은 프로그래밍 책을 사는 데 사용되었다. 그중에서 C++ 언어 책을 꽤 많이 구매하여 보았다(다만, 제대로 이해한 책은 별로 없었다).

책에서는 언제나 OOP라는 단어는 무수히 많이 보았지만, Generic Programming이라는 단어를 그 당시에 본 기억이 없다. 필자가 Generic Programming이라는 단어를 알게 된 것은 2001년 무렵이다. C++ 언어를 공부한지 거의 6년이 될 무렵에 알게 되었다.

아마 지금 공부하는 분들도 Generic Programming이라는 단어는 좀 생소할 것이다. Generic Programming은 한국에서는 보통 '일반적 프로그래밍'이라고 이야기 한다. 필자도 처음에는 그렇게 들었다.

그러나 이것은 잘못된 표현이라 생각한다. 영어 사전을 보면 Generic 이라는 것은 '총칭(總稱)적인' 이라는 뜻도 있는데 이것이 '일반적'이라는 단어보다 더 확실하며 필자가 2004년에 일본에서 구입한 『C++ 설계와 진화』(Bjarne Stroustrup 저)라는 일본 번역서에도 Generic은 총칭으로 표기하고 있다.

그럼 Generic Programming은 무엇일까? 네이버 사전에서 Generic이라는 단어를 검색하면, 다음과 같은 부분을 볼 수 있다.

【문법】총칭적인
the generic singular 총칭 단수《보기:The cow is an animal.》

보기의 영문을 필자의 짧은 영어 실력으로 번역을 하면 '암소는 동물이다'이다. 소는 분명히 고양이나 개와는 다르지만 '동물'이라는 것으로 „총칭"할 수 있다.

1.6 대체 C++언어에서 무엇을 '총칭'화할까?

필자가 만드는 프로그램은 Windows 플랫폼에서 실행되는 '온라인 게임 서버' 프로그램이다. 온라인 게임 서버를 만들 때는 „어떤 기능이 있어야 되는가?"를 결정한 후 클래스를 만든다. 클래스는 아는 바와 같이 멤버 변수와 멤버 함수로 이루어져 있다. 그리고 멤버 함수도 그 내용은 필자의 생각에 의해 변수들을 조작하는 내용으로 되어 있다.

'암소는 동물이다'라는 식으로 C++ 언어에서 총칭을 하는 것은 변수의 타입^{type}을 총칭화 하는 것이다. 변수의 타입을 총칭화하면 다음과 같은 장점이 있다.

- 템플릿을 이용하면 총칭화된 타입을 사용하는 클래스와 함수를 만들 수 있다.
- 템플릿을 사용하면 타입에 제약을 받지 않는 로직을 기술할 수 있다.

그리고 Generic Programming을 하기 위해서는 템플릿이 꼭 필요하다. 그런데 STL이 무엇으로 만들어졌는가? 바로, 템플릿으로 만들어졌다. STL은 Generic Programming으로 만들어진 가장 대표적인 예다.

필자 나름대로 템플릿을 이해하는 데 도움이 되었으면 해서 이런저런 이야기를 했는데 과연 도움이 되었는지 모르겠다. 아마 설명만 듣고서는 템플릿에 대해 명확하게 이해를 하지 못하리라 생각한다. 우리 프로그래머들은 정확하게 이해하려면 코드를 봐야겠죠? 템플릿은 크게 함수 템플릿과 클래스 템플릿으로 나눌 수 있다.

2 함수 템플릿

2.1 두 값을 비교하는 함수를 만들어야 한다

앞서 필자가 하는 일을 이야기했다. 온라인 게임을 만들고 있다. 게임에서 구현해야 되는 것에는 캐릭터 간에 체력(HP)을 비교하는 것이 필요하다. 그래서 두 개의 int 타입을 비교하는 Max라는 이름의 함수를 하나 만들었다.

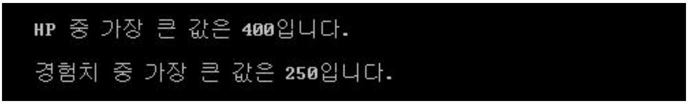
```
int Max( int a, int b );
```

일을 다 끝낸 후 다음 기획서를 보니 캐릭터와 NPC$^{\text{Non Player Character}}$가 전투를 하는 것을 구현해야 하는데 여기에는 경험치를 비교하는 기능이 필요하다. 구현해야 하는 것은 위에서 만든 Max 함수와 같다. 그래서 그것을 사용했다.

[리스트 2-1]

```
#include <iostream>
using namespace std;

int Max( int a, int b )
{
   return a > b ? a : b;
}

void main()
{
  int Char1_HP = 300;
  int Char2_HP = 400;
  int MaxCharHP = Max( Char1_HP, Char2_HP );
  cout << "HP 중 가장 큰 값은 " << MaxCharHP << "입니다." << endl << endl;

  float Char1_Exp = 250.0f;
  float Char2_Exp = 250.57f;
  float MaxCharExp = Max( Char1_Exp, Char2_Exp );
  cout << "경험치 중 가장 큰 값은 " << MaxCharExp << "입니다." << endl << endl;
}
```

앗, 체력(HP)을 저장하는 변수의 타입은 int인데, 경험치를 저장하는 변수의 타입은 int가 아닌 float 타입이다.



```
HP 중 가장 큰 값은 400입니다.
경험치 중 가장 큰 값은 250입니다.
```

당연하게 경험치를 비교하는 부분은 버그가 있다. 앞에 만들었던 Max와는 다르게 비교하는 변수의 타입이 float인 것이 필요하여 새로 만들었다.

[리스트 2-2]

```
float Max( float a, float b )
{
  return a > b ? a : b;
}
```

함수 오버로딩에 의해 경험치를 비교할 때는 int 타입의 Max가 아닌 의 float 타입을 비교하는 Max가 호출되어 버그가 사라지게 되었다.

이제 경험치 비교는 끝나서 다음 기획서에 있는 것을 구현해야 한다. 이번에는 돈을 비교하는 것이 있다. 그런데 돈을 저장하는 변수의 타입은 __int64입니다. __int64는 비주얼 C++에서만 사용할 수 있는 64비트 정수 타입이다. __int64 타입을 비교하는 것은 앞에서 만든 int 타입의 Max나 float 타입의 Max로 할 수 없다. 함수에서 사용하는 변수의 타입만 다를 뿐 똑같은 것을 또 만들어야 한다.

```
__int64 Max(__int64 a, __int64 b )
{
  return a > b ? a : b;
}
```

현재까지만 하더라도 이미 똑같은 로직으로 구현된 함수를 3개나 만들었는데, 게임에서 사용하는 캐릭터의 정보는 HP, 경험치, 돈 이외에도 더 많다. 저는 앞으로 Max 함수를 몇 개 더 만들어야 할지 모른다. Max 함수의 구현을 고쳐야 한다면 모든 Max 함수를 찾아야 한다. 함수 오버로딩은 문제를 해결하지만, 코

드가 커지고 유지보수는 어렵게 만든다.

프로그래밍에서 유지보수는 아주 중요하다. 왜냐하면, 프로그래밍은 언제나 변경이 가해지기 때문이다. 유지보수를 편하게 하는 가장 간단한 방법은 유지보수할 것을 줄이는 것이다.

2.2 Max 함수를 하나로 만들고 싶다. 어떻게 해야 할까?

앗, 혹시 모른다고요? 필자가 이 앞에 템플릿에 대해 설명할 때 이런 말을 하지 않았나요?

> '템플릿을 사용하면 타입에 제약을 받지 않는 로직을 기술할 수 있다'

즉, 템플릿을 사용하면 된다.

2.2.1 함수 템플릿 Max를 만들자

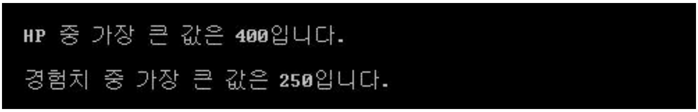
다음 코드는 템플릿을 사용하여 Max 함수를 구현한 것이다.

[리스트 2-3]

```
#include <iostream>
using namespace std;

// 템플릿으로 만든 값을 비교하는 Max 함수
template <typename T> T Max(T a, T b )
{
  return a > b ? a : b;
}

void main()
{
  int Char1_HP = 300;
  int Char2_HP = 400;
  int MaxCharHP = Max( Char1_HP, Char2_HP );
  cout << "HP 중 가장 큰 값은" << MaxCharHP << "입니다." << endl << endl;

  float Char1_Exp = 250.0f;
```

```
  float Char2_Exp = 250.57f;
  float MaxCharExp = Max( Char1_Exp, Char2_Exp );
  cout << "경험치 중 가장 큰 값은" << MaxCharExp << "입니다." << endl << endl;
}
```

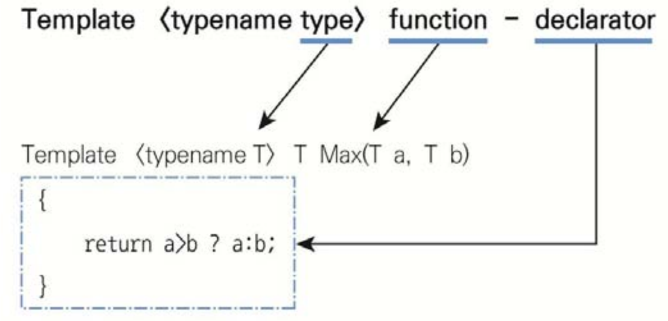
[리스트 2-3]의 실행 결과는 다음과 같다.



```
HP 중 가장 큰 값은 400입니다.
경험치 중 가장 큰 값은 250입니다.
```

이번에는 경험치 비교가 정확하게 이루어졌다. 템플릿을 사용하게 되어 이제는 불필요한 Max 함수를 만들지 않아도 된다. [리스트 2-3] 코드에서 template으로 만든 함수를 '함수 템플릿'이라고 한다. 함수 템플릿을 정의하는 방법은 다음과 같다.

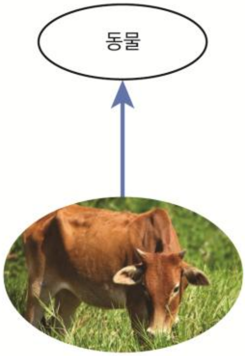
그림 2-1 함수 템플릿 정의



```
Template 〈typename T〉 T Max(T a, T b)
{
    return a>b ? a:b;
}
```

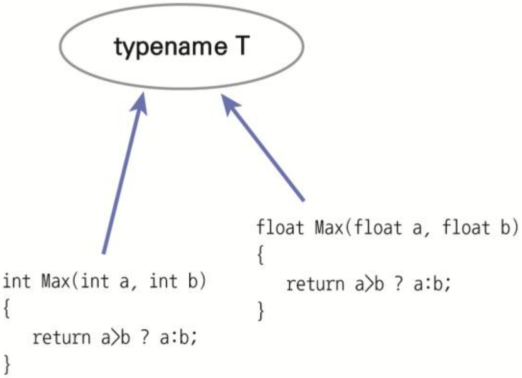
‘템플릿을 사용하면 Generic Programming을 할 수 있다“ 라고 앞서 이야기 했는데 위의 Max 함수 템플릿을 보고 좀 이해를 할 수 있었는가? 혹시나 해서 그림 2-2로 조금만 더 설명한다.

그림 2-2 Max 함수 템플릿



암소를 총칭$^{\text{Generic}}$화하면 동물이라고 할 수 있다. Max 함수 템플릿에서는 함수의 반환 값과 함수 인자인 a 와 b의 타입인 int나 float를 T로 Generic화했다.

그림 2-3 그림 제목



```
int Max(int a, int b)
{
    return a>b ? a:b;
}
```

```
float Max(float a, float b)
{
    return a>b ? a:b;
}
```

2.3 함수 템플릿과 컴파일

하나의 Max 함수 템플릿을 만들었는데 어떻게 int 타입의 Max와 float 타입의 Max를 사용할 수 있을까? 비밀은 컴파일하는 과정에 있다. 컴파일할 때 템플릿

으로 만든 것은 템플릿으로 만든 함수를 호출하는 부분에서 평가한다. 가상 함수처럼 실행시간에 평가하는 것이 아니다.

컴파일을 할 때, 함수 템플릿을 평가하므로 프로그램의 성능을 떨어뜨리지 않는다. 컴파일할 때 평가를 하면서 문법적으로 에러가 없는지 검사한다. 만약 에러가 있다면 컴파일 에러를 출력한다. 에러가 없다면 관련 코드를 내부적으로 생성한다.

[리스트 2-3]을 예로 들면, void main()의 다음 부분을 컴파일하면 Max를 호출할 때 사용한 인자의 변수의 타입이 Max에서 정의 한 문법에 틀리지 않는지 체크한 후 int 타입을 사용하는 Max 함수의 코드를 만든다.

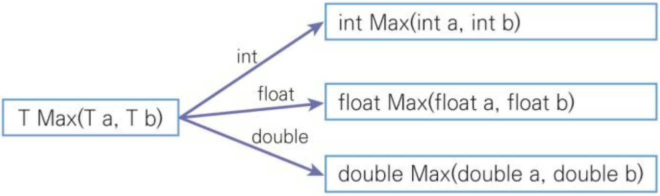
```
int MaxCharHP = Max( Char1_HP, Char2_HP );
```

이후 다음 부분에서 Max를 만나면 이번에도 위의 int 때와 같이 문법 체크를 한 후 에러가 없다면 float를 사용하는 Max 함수 코드를 만든다.

```
float MaxCharExp = Max( Char1_Exp, Char2_Exp );
```

Max가 만들어지는 과정을 나타내면 아래와 같다. 모든 타입에 대해 Max 함수를 만드는 것은 아니다. 코드에서 사용한 타입에 대해서만 Max 함수가 만들어진다.

그림 2-4 그림 제목



참고로 이렇게 만들어지는 코드는 소스 코드에 만들어지는 것이 아니고 프로그램의 코드 영역에 만들어진다. 컴파일 타임에 함수 템플릿을 평가하고 관련 코드를 만들기 때문에 템플릿을 많이 사용하면 컴파일 시간이 길어질 수 있으며, 각 타입에 맞는 코드를 만들어내므로 실행 파일의 크기도 커질 수 있다.

2.4 Max 함수 템플릿에 개선점이 없을까?

힌트를 준다면 Max의 두 인자 값은 함수 내부에서 변경되지 않는다. 그리고 인자의 타입은 C++의 기본형뿐만이 아닌 크기가 큰 타입을 사용할 수도 있다.

생각이 났는가? C++ 기초 공부를 차근차근 쌓아 올렸다면 알아차렸으리라 생각한다.

정답은 Max 함수 템플릿을 만들 때 템플릿의 인자에 const와 참조를 사용하는 것이다. Max 함수는 함수의 내부에서 함수의 인자를 변경하지 않는다. 그러니 함수에 const를 사용하여 내부에서 변경하는 것을 명시적으로 막고 Max 함수를 사용하는 사람에게 알리는 역할을 한다.

C++에서 함수 인자의 전달을 빠르게 하는 방법은 참조로 전달하는 것이다. 위의 Max 함수는 int나 float 같은 크기가 작은 타입을 사용하였기 때문에 참조로 전달하는 것이 큰 의미는 없지만, 만약 구조체나 클래스로 만들어진 크기가 큰 변수를 사용할 때는 참조로 전달하는 것이 훨씬 빠르다. 앞에 만든 Max 함수 템플릿을 const와 참조를 사용하는 것으로 바꾸어 보았다.

[리스트 2-4]

```
template <typename T> const T& Max(const T& a, const T& b )
{
  return a > b ? a : b;
}
```

2.5 class T 라는 것을 본적이 있나요?

함수 템플릿을 만들 때 'typename'을 사용했다. 그러나 좀 오래된 C++ 책에서 템플릿에 대한 글을 본 적이 있다면 'class'를 사용한 것도 본 적이 있을 것이다.

[리스트 2-5]

```
template <class T> const T& Max(const T& a, const T& b )
{
  return a > b ? a : b;
}
```

typename과 class는 기능적으로 다른 것이 아니다. 템플릿이 표준이 되기 전에는 'class'를 사용했다. 그래서 표준화 이전이나 조금 지난 뒤에 나온 책에서는 'class'로 표기했다. 그리고 예전에 만들어진 C++ 컴파일러도 템플릿 인자 선언으로 'class'만 지원했다. 만약, C++ 표준화 전후에 만들어진 컴파일러에서는 'class'를 사용해야 한다.

현재의 컴파일러에서는 'class', 'typename' 둘 다 지원한다. 하지만, 'class'보다 프로그래머에게 '타입'을 추상화한 것이라는 의미 전달을 명확하게 하는 typename을 사용한다. class만 지원하는 오래된 C++ 컴파일러에서 컴파일 해야 하는 것이 아니면 꼭 'typename'을 사용하도록 한다.

2.6 이제 Max 함수 템플릿에는 문제가 없을까?

위에서 Max 함수 템플릿에 대해서 const와 참조로 개선을 했는데 이제 문제가 없을까? 그럼 다음 코드는 문제가 없이 컴파일이 잘 될까?

[리스트 2-6]

```
// [리스트 2-3]의 Max 함수 템플릿을 사용한다.
void main()
{
  int Char1_MP = 300;
  double Char1_SP = 400.25;
  double MaxValue1 = Max( Char1_MP, Char1_SP );
  cout << "MP와 SP 중 가장 큰값은" << MaxValue1 << "입니다." << endl << endl;

  double MaxValue2 = Max( Char1_SP, Char1_MP );
  cout << "MP와 SP 중 가장 큰값은" << MaxValue2 << "입니다." << endl << endl;
}
```

[리스트 2-6]을 컴파일 하면 다음과 같은 에러가 출력된다.

```
max.cpp
max.cpp(16) : error C2782: 'const T &Max(const T &,const T &)'
        : 템플릿 매개 변수 'T'이(가) 모호합니다.
        max.cpp(6) : 'Max' 선언을 참조하십시오.
        'double'일 수 있습니다.
```

```
        또는      'int'
max.cpp(19) : error C2782: 'const T &Max(const T &,const T &)'
        : 템플릿 매개 변수 'T'이(가) 모호합니다.
        max.cpp(6) : 'Max' 선언을 참조하십시오.
        'int'일 수 있습니다.
        또는      'double'
```

이유는 컴파일러는 사람이 아니어서 서로 다른 타입의 인자가 들어오면 템플릿의 파라메터 T를 사용한 함수의 인자 a와 b의 타입을 int로 해야 할지, double로 해야 할지 판단할 수가 없기 때문이다. 이 문제는 어떻게 해결 해야 할까?

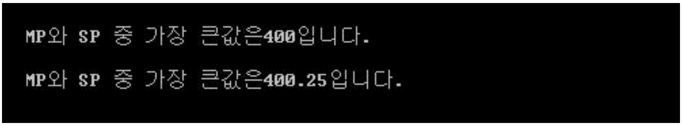
2.7 typename을 하나가 아닌 복수 개 사용하면 된다

위의 문제는 Max 함수를 정의할 때 typename을 하나만 사용해서 타입을 하나만 선언했다. 이제 typename을 여러 개 사용하면 위의 문제를 풀 수 있다.

[리스트 2-7]

```
template <typename T1, typename T2> const T1& Max(const T1& a, const T2& b )
{
  return a > b ? a : b;
}
```

[리스트 2-7]의 함수 템플릿을 사용하면 Max 함수의 인자 타입을 int와 double 혹은 double과 int 타입을 사용해도 컴파일이 잘된다. 그럼 제대로 실행되는지 실행해보자.



```
MP와 SP 중 가장 큰값은400입니다.
MP와 SP 중 가장 큰값은400.25입니다.
```

앗, 실행 결과에 오류가 있다.

```
int Char1_MP = 300;
double Char1_SP = 400.25;
double MaxValue1 = Max( Char1_MP, Char1_SP );
```

이 코드는 300과 400.25를 비교한다. 결과는 400.25가 나와야 하는데 400이 나와버렸다.

이유는 [리스트 2-7]의 함수 템플릿의 반환 값으로 T1을 선언했기 때문에 int 타입과 double 타입을 순서대로 함수 인자에 사용하면 반환 값의 타입이 int형으로 되어 버리기 때문이다. 이렇게 서로 다른 타입을 사용하는 경우에는 반환 값을 아주 조심해야 한다. 그리고 위의 예에서는 함수 템플릿의 파라메터로 typename을 2개 사용했지만 그 이상도 사용할 수 있다.

위의 Max 함수 템플릿 만족스러운가? 필자는 왠지 아직도 좀 불 만족스럽다.

Max(int, double);

여기서는 실수를 하면 찾기 힘든 버그가 발생할 확률이 높다. 이것을 어떻게 풀어야 될까?

2.8 함수 템플릿의 전문화 라는 것이 있다.

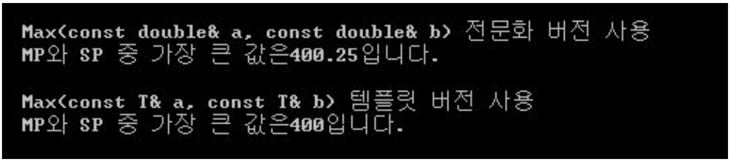
Max(int, double)을 사용하면 Max 함수 템플릿이 아닌 이것에 맞는, 특별하게 만든 함수를 사용하도록 한다. 함수 템플릿의 전문화Specialization라는 특별한 상황에 맞는 함수를 만들면 함수 오버로드와 같이 컴파일러가 상황에 맞는 함수를 선택하도록 한다.

[리스트 2-8]

```
#include <iostream>
using namespace std;

// 템플릿으로만든값을비교하는Max 함수
template <typename T1, typename T2> const T1& Max(const T1& a, const T2& b )
{
  cout << "Max(const T& a, const T& b) 템플릿 버전 사용" << endl;
  return a > b ? a : b;
}

// 전문화시킨Max 함수
template <> const double& Max(const double& a, const double& b)
```

```
{
  cout << "Max(const double& a, const double& b) 전문화 버전 사용" << endl;
  return a > b ? a : b;
}

void main()
{
  double Char1_MP = 300;
  double Char1_SP = 400.25;
  double MaxValue1 = Max( Char1_MP, Char1_SP );
  cout << "MP와 SP 중 가장 큰 값은" << MaxValue1 << "입니다." << endl << endl;

  int Char2_MP = 300;
  double Char2_SP = 400.25;
  double MaxValue2 = Max( Char2_MP, Char2_SP );
  cout << "MP와 SP 중 가장 큰 값은" << MaxValue2 << "입니다." << endl << endl;
}
```

[리스트 2-8]의 코드를 실행한 결과는 다음과 같다.



```
Max(const double& a, const double& b) 전문화 버전 사용
MP와 SP 중 가장 큰 값은400.25입니다.

Max(const T& a, const T& b) 템플릿 버전 사용
MP와 SP 중 가장 큰 값은400입니다.
```

컴파일러는 프로그래머의 생각을 완전히 이해하지는 않는다. 그래서 컴파일러가 어떠한 것을 선택할지 이해하고 있어야 된다. [리스트 2-8]은 double에 전문화된 Max 함수를 만든 예다.

질문_ Max(10.1, 20.4)를 호출한다면 Max(T, T)가 호출 될까? 아님 Max(double, double)가 호출 될까?

답을 빨리 알고 싶을 테니 뜸 들이지 않고 결과를 바로 보자.

```
Max<const double& a, const double& b> 전문화 버전 사용
```

전문화 버전이 호출 되었다. 이유는 호출 순서에 규칙이 있기 때문이다(최선에서 최악으로). 호출 순서는 다음과 같다.

1. 전문화된 함수와 맞는지 검사한다.
2. 템플릿 함수와 맞는지 검사한다.
3. 일반 함수와 맞는지 검사한다.

위의 순서를 잘 기억하고 전문화 함수를 만들어야 한다. 잘못하면 찾기 힘든 버그를 만들 수가 있다. 이제 함수 템플릿에 대한 이야기는 거의 다 끝난 것 같다.

아... 하나 더 있다. 지금까지 살펴본 것은 타입만을 템플릿 파라메터로 사용했는데 꼭 타입만 함수 템플릿에 사용할 수 있는 것은 아니다.

2.9 난-타입(non-type) 함수 템플릿

온라인 게임에서는 특정한 이벤트가 있을 때는 캐릭터의 HP, 경험치, 돈을 이벤트 기념으로 주는 경우가 있다. HP와 경험치, 돈의 타입은 다르지만 추가 되는 값은 int 상수로 정해져 있다. 위와 같이 타입은 다르지만 상수를 더 한 값을 얻는 함수를 만들려면 어떻게 해야 할까?

이런 문제도 함수 템플릿으로 해결할 수 있다.

함수 템플릿의 파라메터로 typename만이 아닌 값을 파라메터로 사용할 수도 있다. 아래의 코드는 캐릭터의 HP, 경험치, 돈을 이벤트에서 정해진 값만큼 더 해주는 것을 보여준다.

[리스트 2-9]

```
#include <iostream>
using namespace std;
```

```
// 지정된 값만큼 더해준다.
template <typename T, int VAL> T AddValue( T const& CurValue)
{
  return CurValue + VAL;
}

const int EVENT_ADD_HP_VALUE  = 50;    // 이벤트에 의해 추가 될 HP 값
const int EVENT_ADD_EXP_VALUE  = 30;    // 이벤트에 의해 추가 될 경험치
const int EVENT_ADD_MONEY_VALUE  = 10000;    // 이벤트에 의해 추가 될 돈

void main()
{
  int Char_HP = 250;
  cout << Char_HP <<"에서 이벤트에 의해" << AddValue<int,
       EVENT_ADD_HP_VALUE>(Char_HP) << " 로 변경" <<endl;

  float Char_EXP = 378.89f;
  cout << Char_EXP <<"에서 이벤트에 의해" << AddValue<float,
       EVENT_ADD_EXP_VALUE>(Char_EXP) << " 로 변경" <<endl;

  __int64 Char_MONEY = 34567890;
  cout << Char_MONEY <<"에서 이벤트에 의해" << AddValue<__int64,
       EVENT_ADD_MONEY_VALUE>(Char_MONEY) << " 로 변경" <<endl;
}
```

실행 결과는 다음과 같다.

```
250에서 이벤트에 의해300 로 변경
378.89에서 이벤트에 의해408.89 로 변경
34567890에서 이벤트에 의해34577890 로 변경
```

앞에서 사용했던 함수 템플릿 사용방법과 조금 달라서 생소할 수도 있을 것이다.

필자가 위에 든 예는 난-타입 함수 템플릿을 사용해야 되는 당위성이 좀 떨어질 수 있다고 생각한다. 하지만 간단한 예제를 사용해서 좀 더 쉽게 설명하기 위한 방법이라 변명해 본다.

난-타입을 사용하는 템플릿은 다음 장에 나오는 클래스 템플릿에서 또 다시 이야기 할 예정이니 잘 기억하고 있기를 바란다. 또, 난-타입을 잘 사용하면 템플릿 메타 프로그래밍을 할 때 큰 도움이 된다. 템플릿 메타 프로그래밍에 대해서는 다음에 설명할 것이다.

3 클래스 템플릿

2장에서 함수템플릿에 대해 설명을 했으니 이번에는 클래스 템플릿에 대해서 설명하려고 한다. 클래스 템플릿을 아주 간단하게 말하면 함수 템플릿이 함수에 템플릿을 사용한 것처럼 클래스 템플릿은 클래스에 템플릿을 사용한 것이다.

그러므로 함수 템플릿에 대해서 잘 모른다면 „2장. 함수 템플릿“을 먼저 보고 이 장을 보는 것이 이해하기에 좋다.

3.1 경험치 변경 이력 저장

기획팀에서 유저들이 게임에 접속하여 다른 유저들과 100번의 게임을 했을 때 유저들의 경험치가 변경 되는 이력을 볼 수 있기를 요청했다.

기획팀의 요구를 들어주기 위해서 저는 게임이 끝날 때마다 경험치를 저장한다. 또 경험치 이력 내역을 출력할 때 가장 최신에서 가장 오랜 된 것을 보여줘야 되기 때문에 스택stack이라는 자료 구조를 사용한다.

스택은 자료 구조 중의 하나로 가장 마지막에 들어 온 것을 가장 먼저 꺼내는 LIFO(Last In First Out) 형식으로 되어 있다. 데이터를 넣을 때를 push, 빼낼 때는 pop이라는 이름을 일반적으로 사용한다.

경험치 이력을 저장하는 클래스의 구현과 이것을 사용하는 것은 다음과 같다.

[리스트 3-1]

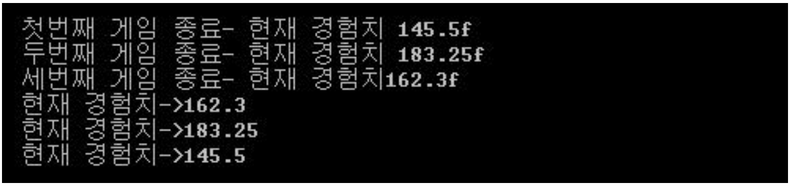
```
#include <iostream>

// 경험치를 저장할 수 있는 최대 개수
const int MAX_EXP_COUNT = 100;

// 경험치 저장 스택 클래스
class ExpStack
{
public:
  ExpStack()
  {
```

```
  Clear();
}

// 초기화 한다.
void Clear()
{
  m_Count = 0;
}

// 스택에 저장된 개수
int Count()
{
  return m_Count;
}

// 저장된 데이터가 없는가?
bool IsEmpty()
{
  return 0 == m_Count ? true : false;
}

// 경험치를 저장한다.
bool push( float Exp )
{
  // 저장할 수 있는 개수를 넘는지 조사한다.
  if( m_Count >= MAX_EXP_COUNT )
  {
    return false;
  }

  // 경험치를 저장 후 개수를 하나 늘린다.
  m_aData[ m_Count ] = Exp;
  ++m_Count;

  return true;
}

// 스택에서 경험치를 빼낸다.
float pop()
```

```
  {
    // 저장된 것이 없다면 0.0f를 반환한다.
    if( m_Count  < 1 )
    {
      return 0.0f;
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
  }

private:
  float  m_aData[MAX_EXP_COUNT];
  int    m_Count;
};

#include <iostream>
using namespace std;

void main()
{
  ExpStack kExpStack;

  cout << "첫번째 게임 종료- 현재 경험치 145.5f" << endl;
  kExpStack.push( 145.5f );

  cout << "두번째 게임 종료- 현재 경험치 183.25f" << endl;
  kExpStack.push( 183.25f );

  cout << "세번째 게임 종료- 현재 경험치162.3f" << endl;
  kExpStack.push( 162.3f );


  int Count = kExpStack.Count();
  for( int i = 0; i < Count; ++i )
  {
    cout << "현재 경험치->" << kExpStack.pop() << endl;
  }
```

```
}
```

[리스트 3-1]의 실행 결과는 다음과 같다.



```
첫번째 게임 종료- 현재 경험치 145.5f
두번째 게임 종료- 현재 경험치 183.25f
세번째 게임 종료- 현재 경험치162.3f
현재 경험치->162.3
현재 경험치->183.25
현재 경험치->145.5
```

실행 결과를 보면 알 수 있듯이 스택 자료구조를 사용하였기 때문에 제일 뒤에 넣은 데이터가 가장 먼저 출력되었다.

3.2 게임 돈 변경 이력도 저장해 주세요

경험치 변경 이력을 저장하고 출력하는 기능을 만들어서 기획팀에 보여주니 이번에는 게임 돈의변경 이력도 보고 싶다고 말한다.

위에서 경험치 변경 이력 저장 기능을 만들어 보았으니 금방 할 수 있는 것이다. 그래서 이번에는 이전 보다 훨씬 더 빨리 만들었다.

[리스트 3-2]

```
#include <iostream>

// 돈을 저장할 수 있는 최대 개수
const int MAX_MONEY_COUNT = 100;

// 돈 저장 스택 클래스
class MoneyStack
{
public:
  MoneyStack()
  {
    Clear();
  }
```

```
// 초기화 한다.
void Clear()
{
  m_Count = 0;
}

// 스택에 저장된 개수
int Count()
{
  return m_Count;
}

// 저장된 데이터가없는가?
bool IsEmpty()
{
  return 0 == m_Count ? true : false;
}

// 돈을 저장한다.
bool push( __int64 Money )
{
  // 저장 할 수 있는 개수를 넘는지 조사한다.
  if( m_Count >= MAX_MONEY_COUNT )
  {
    return false;
  }

  // 저장후 개수를 하나 늘린다.
  m_aData[ m_Count ] = Money;
  ++m_Count;

  return true;
}

// 스택에서 돈을 빼낸다.
__int64 pop()
{
  // 저장된 것이 없다면 0을 반환한다.
```

```
    if( m_Count  < 1 )
    {
      return 0;
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
  }

private:
  __int64  m_aData[MAX_MONEY_COUNT];
  int  m_Count;
};
```

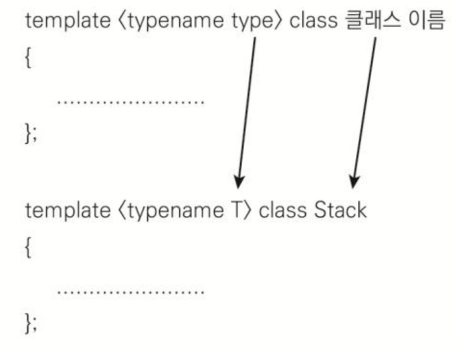
ExpStack 클래스와 MoneyStack 클래스가 비슷하다

게임 돈 변경 이력 저장 기능을 가지고 있는 MoneyStack 클래스를 만들고 보니 앞에 만든 ExpStack와 거의 같다. 저장하는 데이터의 자료형만 다를 뿐이지 모든 것이 같다. 그리고 기획팀에서는 게임 캐릭터의 Level 변경 이력도 저장하여 보여주기를 바라는 것 같다. 이미 거의 똑같은 클래스를 두 개 만들었고 앞으로도 기획팀에서 요청이 있으면 더 만들 것 같다. 이렇게 자료형만 다른 클래스를 어떻게 하면 하나의 클래스로 정의 할 수 있을까? 이와 비슷한 문제를 이전의 "함수 템플릿"에서도 나타나지 않았나? 그때 어떻게 해결할 수 있을까?(생각나지 않으면 앞 2장의 "함수 템플릿"을 다시 한번 확인한다)

템플릿으로 하면 된다. 기능은 같지만 변수의 자료형만 다른 함수를 템플릿을 사용하여 하나의 함수로 정의했듯이 이번에는 템플릿을 사용하여 클래스를 정의한다. 클래스에서 템플릿을 사용하면 이것을 클래스 템플릿이라고 한다. 클래스 템플릿을 사용하면 위에서 중복된 클래스를 하나의 클래스로 만들 수 있다.

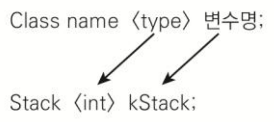
3.3 클래스 템플릿을 사용하는 방법

클래스 템플릿을 정의하는 문법은 다음과 같다.



```
template 〈typename type〉 class 클래스 이름
{
    ........................
};

template 〈typename T〉 class Stack
{
    ........................
};
```

정의한 클래스 템플릿을 사용하는 방법은 다음과 같다.



```
Class name 〈type〉 변수명;

Stack 〈int〉 kStack;
```

3.4 Stack 템플릿 클래스

지금까지 만들었던 ExpStack 과 MoneyStack을 클래스 템플릿으로 만든 코드는 다음과 같다.

[리스트 3-3]

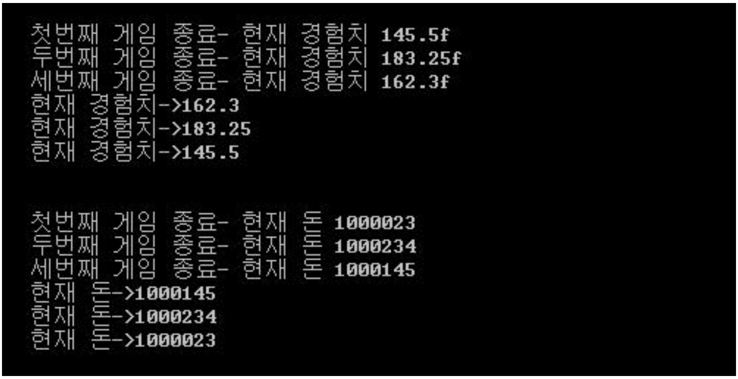
```
#include <iostream>

const int MAX_COUNT = 100;

template<typename T>
class Stack
{
public:
```

```
Stack()
{
  Clear();
}

// 초기화 한다.
void Clear()
{
  m_Count = 0;
}

// 스택에 저장된 개수
int Count()
{
  return m_Count;
}

// 저장된 데이터가 없는가?
bool IsEmpty()
{
  return 0 == m_Count ? true : false;
}

// 데이터를 저장한다.
bool push( T data )
{
  // 저장 할수 있는 개수를 넘는지 조사한다.
  if( m_Count >= MAX_COUNT )
  {
    return false;
  }

  // 저장후 개수를 하나 늘린다.
  m_aData[ m_Count ] = data;
  ++m_Count;

  return true;
}
```

```
  // 스택에서 빼낸다.
  T pop()
  {
    // 저장된 것이 없다면 0을 반환한다.
    if( m_Count  < 1 )
    {
      return 0;
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
  }

private:
  T  m_aData[MAX_COUNT];
  int    m_Count;
};

#include <iostream>
using namespace std;

void main()
{
  Stack<double> kStackExp;

  cout << "첫번째 게임 종료- 현재 경험치 145.5f" << endl;
  kStackExp.push( 145.5f );

  cout << "두번째 게임 종료- 현재 경험치 183.25f" << endl;
  kStackExp.push( 183.25f );

  cout << "세번째 게임 종료- 현재 경험치 162.3f" << endl;
  kStackExp.push( 162.3f );


  int Count = kStackExp.Count();
  for( int i = 0; i < Count; ++i )
  {
```

```
    cout << "현재 경험치->" << kStackExp.pop() << endl;
  }

  cout << endl << endl;

  Stack<__int64> kStackMoney;

  cout << "첫번째 게임 종료- 현재 돈 1000023" << endl;
  kStackMoney.push( 1000023 );

  cout << "두번째 게임 종료- 현재 돈 1000234" << endl;
  kStackMoney.push( 1000234 );

  cout << "세번째 게임 종료- 현재 돈 1000145" << endl;
  kStackMoney.push( 1000145 );


  Count = kStackMoney.Count();
  for( int i = 0; i < Count; ++i )
  {
    cout << "현재 돈->" << kStackMoney.pop() << endl;
  }
}
```

[리스트 3-3]의 실행 결과는 다음과 같다.



```
첫번째 게임 종료- 현재 경험치 145.5f
두번째 게임 종료- 현재 경험치 183.25f
세번째 게임 종료- 현재 경험치 162.3f
현재 경험치->162.3
현재 경험치->183.25
현재 경험치->145.5

첫번째 게임 종료- 현재 돈 1000023
두번째 게임 종료- 현재 돈 1000234
세번째 게임 종료- 현재 돈 1000145
현재 돈->1000145
현재 돈->1000234
현재 돈->1000023
```

클래스 템플릿으로 Stack을 구현하여 앞으로 다양한 데이터를 사용할 수 있게 되었다. 그런데 위의 Stack 클래스는 부족한 부분이 있다. 앞으로 이 부족한 부분을 채워 나가면서 클래스 템플릿에 대해서 좀 더 알아보자.

3.5 클래스 템플릿에서 non-type 파라메터 사용

위에서 만든 Stack 클래스는 데이터를 저장할 수 있는 공간이 100개로 정해져 있다. Stack의 크기는 사용하는 곳에 따라서 변할 수 있어야 사용하기에 적합하다.

함수 템플릿을 설명할 때도 non-type이 나왔는데 사용방법이 거의 같다. 템플릿 파라메터를 기본 데이터 형으로 한다. 다음 사용 예를 참고한다.

[리스트 3-4]

```
// 템플릿 파라메터중 int Size가 non-type  파라메터이다.
template <typename T, int Size>
class Stack
{
public:
  Stack()
  {
    Clear();
  }
```

```
// 초기화 한다.
void Clear()
{
  m_Count = 0;
}

// 스택에 저장된 개수
int Count()
{
  return m_Count;
}

// 저장된 데이터가 없는가?
bool IsEmpty()
{
  return 0 == m_Count ? true : false;
}

// 데이터를 담을수 있는 최대 개수
int GetStackSize()
{
  return Size;
}

// 데이터를 저장한다.
bool push( T data )
{
  // 저장할 수 있는 개수를 넘는지 조사한다.
  if( m_Count >= Size )
  {
    return false;
  }

  // 저장 후 개수를 하나 늘린다.
  m_aData[ m_Count ] = data;
  ++m_Count;

  return true;
```

```
  }

  // 스택에서 빼낸다.
  T pop()
  {
    // 저장된 것이 없다면 0을 반환한다.
    if( m_Count  < 1 )
    {
      return 0;
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
  }

private:
  T  m_aData[Size];
  int  m_Count;
};

#include <iostream>
using namespace std;

void main()
{
  Stack<int, 100> kStack1;
  cout << "스택의 크기는?" << kStack1.GetStackSize() << endl;

  Stack<double, 60> kStack2;
  cout << "스택의 크기는?" << kStack2.GetStackSize() << endl;
}
```

[리스트 3-4]의 실행 결과는 다음과 같다.

```
스택의 크기는?100
스택의 크기는?60
```

3.6 템플릿 파라메터 디폴트 값 사용

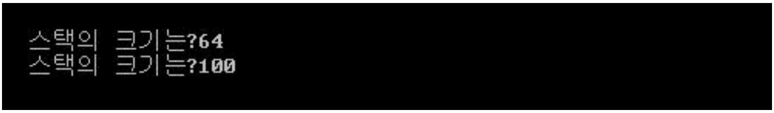
일반 함수에서 함수 인자의 디폴트 값을 지정하듯이 클래스 템플릿의 파라메터도 디폴트 값으로 할 수 있다.

[리스트 3-5]

```
// 템플릿 파라메터 중 int Size가 non-type 파라메터다.
// Size의 디폴트 값을 100으로 한다.
#include <iostream>

template<typename T, int Size=100>
class Stack
{
   ….. 생략
}

void main()
{
  Stack<int, 64> kStack1;
  cout << "스택의 크기는?" << kStack1.GetStackSize() << endl;

  Stack<double>  kStack2;
  cout << "스택의 크기는?" << kStack2.GetStackSize() << endl;
}
```

[리스트 3-5]에서 템플릿 파라메터 중 Size의 값을 디폴트 100으로 한다. 클래스를 생성할 때 두 번째 파라메터 값을 지정하지 않으면 디폴트 값을 사용한다.

[리스트 3-5]의 실행 결과는 다음과 같다.



```
스택의 크기는?64
스택의 크기는?100
```

3.7 스택 클래스의 크기를 클래스 생성자에서 지정

클래스 템플릿에 대한 설명을 계속 하기 위해 현재까지 만든 스택 클래스를 변경한다. 스택의 크기를 클래스 템플릿 파라메터가 아닌 생성자에서 지정하도록 변경할 것이다.

[리스트 3-6]

```
template<typename T, int Size=100> class Stack
{
public:
  explicit Stack( int size )
  {
    m_Size = size;
    m_aData = new T[m_Size];

    Clear();
  }

  ~Stack()
  {
    delete[] m_aData;
  }

  // 초기화 한다.
  void Clear()
  {
    m_Count = 0;
  }

  // 스택에 저장된 개수
  int Count()
```

```
{
  return m_Count;
}

// 저장된 데이터가 없는가?
bool IsEmpty()
{
  return 0 == m_Count ? true : false;
}

// 데이터를 담을 수 있는 최대 개수
int GetStackSize()
{
  return m_Size;
}

// 데이터를 저장한다.
bool push( T data )
{
  // 저장할 수 있는 개수를 넘는지 조사한다.
  if( m_Count >= m_Size )
  {
    return false;
  }

  // 저장 후 개수를 하나 늘린다.
  m_aData[ m_Count ] = data;
  ++m_Count;

  return true;
}

// 스택에서 빼낸다.
T pop()
{
  // 저장된 것이 없다면 0을 반환한다.
  if( m_Count  < 1 )
  {
    return 0;
```

```
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
  }

private:
  T*  m_aData;
  int  m_Count;

  int m_Size;
};

#include  <iostream>
using namespace std;

void main()
{
  Stack<int> kStack1(64);
  cout << "스택의 크기는? " << kStack1.GetStackSize() << endl;
}
```

[리스트 3-6]의 실행 결과는 다음과 같다.

```
스택의 크기는? 64
```

[리스트 3-6]의 코드에서 잘 보지 못한 키워드가 있을 것이다. 바로 explicit 이다. explicit 키워드로 규정된 생성자는 암시적인 형 변환을 할 수 없다. 그래서 [리스트 3-6]의 void main()에서 다음과 같이 클래스를 생성하면 컴파일 에러가 발생합니다.

```
 Stack kStack1 = 64;
```

3.8 클래스 템플릿 전문화

기획팀에서 새로운 요구가 들어왔다. 이번에는 게임을 할 때 같이 게임을 했던 유저의 아이디를 저장하여 보여주기를 원한다. 지금까지 만든 Stack 클래스는 기본 자료형을 사용하는 것을 전제로 했는데 유저의 아이디를 저장하려면 문자열이 저장되어야 하므로 사용할 수가 없다.

기본 자료형으로 하지 않고 문자열을 사용한다는 것만 다르지 작동은 비슷하므로 Stack이라는 이름의 클래스를 사용하고 싶다. 기존의 Stack 클래스 템플릿과 클래스의 이름만 같지 행동은 다른 Stack 클래스를 구현 하려고 한다. 이때 필요한 것인 클래스 템플릿의 전문화라는 것이다. 클래스 템플릿 전문화는 기존에 구현한 클래스 템플릿과 비교해서 이름과 파라메터 개수는 같지만 파라메터를 특정한 것으로 지정합니다.

전문화된 클래스 템플릿 정의는 다음과 같은 형태를 가진다.

```
template <> class 클래스 이름<지정된 타입>
{
   ………………….
};
```

[리스트 3-7]

```
// ID 문자열의 최대 길이(null 문자포함)
const int MAX_ID_LENGTH = 21;

template<typename T> class Stack
{
public:
  explicit Stack( int size )
  {
    m_Size = size;
    m_aData = new T[m_Size];

    Clear();
  }
```

```
~Stack()
{
  delete[] m_aData;
}

// 초기화 한다.
void Clear()
{
  m_Count = 0;
}

// 스택에 저장된 개수
int Count()
{
  return m_Count;
}

// 저장된 데이터가 없는가?
bool IsEmpty()
{
  return 0 == m_Count ? true : false;
}

// 데이터를 담을 수 있는 최대 개수
int GetStackSize()
{
  return m_Size;
}

// 데이터를 저장한다.
bool push( T data )
{
  // 저장할 수 있는 개수를 넘는지 조사한다.
  if( m_Count >= m_Size )
  {
    return false;
  }

  // 저장 후 개수를 하나 늘린다.
```

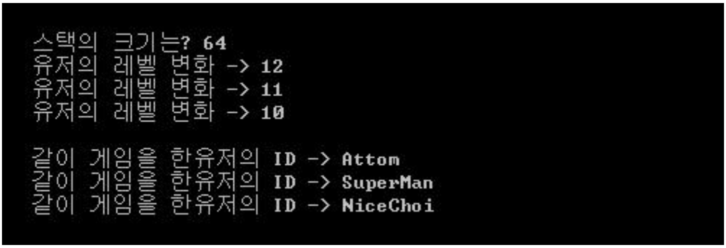
```
    m_aData[ m_Count ] = data;
    ++m_Count;

    return true;
  }

  // 스택에서 빼낸다.
  T pop()
  {
    // 저장된 것이 없다면 0을 반환한다.
    if( m_Count  < 1 )
    {
      return 0;
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
  }

private:
  T*  m_aData;
  int  m_Count;

  int m_Size;
};

#include <string>
// 아래의 코드는 문자열을 저장하기 위해 char* 으로 전문화한 Stack 클래스입니다.
template<> class Stack<char*>
{
public:
   explicit Stack( int size )
   {
      m_Size = size;

      m_ppData = new char *[m_Size];
      for( int i = 0; i < m_Size; ++i )
      {
```

```
            m_ppData[i] = new char[MAX_ID_LENGTH];
        }

        Clear();
    }

    ~Stack()
    {
        for( int i = 0; i < m_Size; ++i )
        {
            delete[] m_ppData[i];
        }

        delete[] m_ppData;
    }

    // 초기화한다.
    void Clear()
    {
        m_Count = 0;
    }

    // 스택에 저장된 개수
    int Count()
    {
        return m_Count;
    }

    // 저장된 데이터가 없는가?
    bool IsEmpty()
    {
        return 0 == m_Count ? true : false;
    }

    // 데이터를 담을 수 있는 최대 개수
    int GetStackSize()
    {
        return m_Size;
    }
```

```
    // 데이터를 저장한다.
    bool push( char* pID )
    {
        // 저장할 수 있는 개수를 넘는지 조사한다.
        if( m_Count >= m_Size )
        {
            return false;
        }

        // 저장 후 개수를 하나 늘린다.
        strncpy_s( m_ppData[m_Count], MAX_ID_LENGTH, pID, MAX_ID_LENGTH - 1);
        m_ppData[m_Count][MAX_ID_LENGTH - 1] = '\0';

        ++m_Count;

        return true;
    }

    // 스택에서 빼낸다.
    char* pop()
    {
        // 저장된 것이 없다면 0을 반환한다.
        if( m_Count  < 1 )
        {
            return 0;
        }

        // 개수를 하나 감소 후 반환한다.
        --m_Count;
        return m_ppData[ m_Count ];
    }

private:
    char** m_ppData;
    int  m_Count;

    int m_Size;
};
```

```
#include <iostream>
using namespace std;

void main()
{
   Stack<int> kStack1(64);
   cout << "스택의 크기는? " << kStack1.GetStackSize() << endl;
   kStack1.push(10);
   kStack1.push(11);
   kStack1.push(12);

   int Count1 = kStack1.Count();
   for( int i = 0; i < Count1; ++i )
   {
      cout << "유저의 레벨 변화 -> " << kStack1.pop() << endl;
   }

   cout << endl;

   char GameID1[MAX_ID_LENGTH] = "NiceChoi";
   char GameID2[MAX_ID_LENGTH] = "SuperMan";
   char GameID3[MAX_ID_LENGTH] = "Attom";

   // Stack 클래스 템플릿의 char*  전문화 버전을 생성한다.
   Stack<char*> kStack2(64);
   kStack2.push(GameID1);
   kStack2.push(GameID2);
   kStack2.push(GameID3);

   int Count2 = kStack2.Count();
   for(int i = 0; i < Count2; ++i)
   {
   cout << "같이 게임을 한유저의 ID -> " << kStack2.pop() << endl;
   }

}
```

[리스트 3-7]의 실행 결과는 다음과 같다.



```
스택의 크기는? 64
유저의 레벨 변화 -> 12
유저의 레벨 변화 -> 11
유저의 레벨 변화 -> 10

같이 게임을 한유저의 ID -> Attom
같이 게임을 한유저의 ID -> SuperMan
같이 게임을 한유저의 ID -> NiceChoi
```

3.9 클래스 템플릿 부분 전문화

클래스 템플릿은 템플릿 파라메터 중 일부를 구체적인 형(type)을 사용, 또는 템플릿 파라메터를 포인터나 참조를 사용하여 부분 전문화를 할 수 있다.

구체적인 형 사용에 의한 부분 전문화

template< typename T1, typename T2 > class Test { .... };

의 T2를 float로 구체화 하여 부분 전문화를 하면 다음과 같다.

template< typename T1 > class Test { ..... };

코드는 다음과 같다.

[리스트 3-8]

```
template< typename T1, typename T2 >
class Test
{
public:
  T1 Add( T1 a, T2 b )
  {
    cout << "일반 템플릿을 사용했습니다." << endl;
    return a;
  }
};
```

```
// T2를 float로 구체화한 Test의 부분 전문화 템플릿
template< typename T1 >
class Test<T1,float>
{
public:
  T1 Add( T1 a, float b )
  {
    cout << "부분 전문화 템플릿을 사용했습니다." << endl;
    return a;
  }
};

#include <iostream>
using namespace std;

void main()
{
  Test<int, int> test1;
  test1.Add( 2, 3 );

  Test<int, float> test2;
  test2.Add( 2, 5.8f );
}
```

위의 예에서는 템플릿 파라메터 2개 중 일부를 구체화하여 부분 전문화를 했지만, 2개 이상도 가능하다.

template< typename T1, typename T2, typename T3 > class Test { .... };

의 부분 전문화 템플릿은

template< typename T1, typename T2 > class Test { ..... };

- 포인터의 부분 전문화

template< typename T > class TestP { .... };

의 T의 T* 부분 전문화를 하는 다음과 같다.

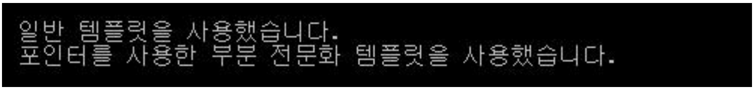
```
template< typename T > class TestP {  ...... };
```

코드는 다음과 같다.

[리스트 3-9]

```
template< typename T >
class TestP
{
public:
  void Add()
  {
    cout << "일반 템플릿을 사용했습니다." << endl;
  }
};

// T를 T*로 부분 전문화
template< typename T >
class TestP<T*>
{
public:
  void Add()
  {
    cout << "포인터를 사용한 부분 전문화 템플릿을 사용했습니다." << endl;
  }
};

#include <iostream>
using namespace std;

void main()
{
  TestP<int> test1;
  test1.Add();

  TestP<int*> test2;
  test2.Add();
}
```

[리스트 3-9]의 실행 결과는 다음과 같다.



```
일반 템플릿을 사용했습니다.
포인터를 사용한 부분 전문화 템플릿을 사용했습니다.
```

3.10 싱글톤 템플릿 클래스

클래스 상속을 할 때 템플릿 클래스를 상속 받음으로 상속 받는 클래스의 기능을 확장할 수 있다. 필자의 경우 현업에서 클래스 템플릿을 가장 많이 사용하는 경우가 클래스 템플릿을 사용한 싱글톤 클래스 템플릿을 사용하는 것이다.

어떠한 객체가 꼭 하나만 있어야 되는 경우 싱글톤으로 정의한 클래스 템플릿을 상속 받도록 한다.

싱글톤은 싱글톤 패턴을 말하는 것으로 어떤 클래스의 인스턴스가 꼭 하나만 생성되도록 하며, 전역적인 접근이 가능하도록 한다. 어떤 클래스를 전역으로 사용하는 경우 복수개의 인스턴스가 생성되지 않도록 싱글톤 패턴으로 생성하는 것을 권장한다.

사용하는 방법은 베이스 클래스를 템플릿을 사용하여 만든다. 그리고 이것을 상속 받는 클래스에서 베이스 클래스의 템플릿 파라메터에 해당 클래스를 사용한다. 즉 싱글톤 클래스 템플릿은 이것을 상속 받는 클래스를 싱글톤으로 만들어준다.

위에서 설명한 클래스 템플릿에 대하여 이해를 했다면 다음의 코드를 보면 더 잘 이해를 할 수 있으리라 생각한다. 싱글톤 클래스 템플릿은 직접 생성을 하지 않으므로 주 멤버들을 static로 만들어준다. 그리고 생성자를 통해서 _Singleton를 생성하지 않고 GetSingleton()을 통해서만 생성하도록 한다.

[리스트 3-10]

```
// 파라메터 T를 싱글톤이 되도록 정의한다.
#include <iostream>
using namespace std;

template <typename T>
```

```
class MySingleton
{
public:
    MySingleton() {}
    virtual ~MySingleton() {}

    // 이 멤버를 통해서만 생성이 가능하다.
    static T* GetSingleton()
    {
        // 아직 생성이 되어 있지 않으면 생성한다.
        if( NULL == _Singleton ) {
            _Singleton = new T;
        }

       return ( _Singleton );
    }

    static void Release()
    {
        delete _Singleton;
        _Singleton = NULL;
    }

private:
    static T* _Singleton;
};

template <typename T> T* MySingleton<T> ::_Singleton = NULL;

// 싱글톤 클래스 템플릿을 상속 받으면서 파라메터에 본 클래스를 넘긴다.
class MyObject : public MySingleton<MyObject>
{
public:
MyObject() : _nValue(10) {}

void SetValue( int Value ) { _nValue = Value;}
int GetValue() { return _nValue; }

private :
```

```
int _nValue;
};

void main()
{
    MyObject* MyObj1 = MyObject::GetSingleton();

    cout << MyObj1->GetValue() << endl;

    // MyObj2는 Myobj1과 동일한 객체이다.
    MyObject* MyObj2 = MyObject::GetSingleton();
    MyObj2->SetValue(20);

    cout << MyObj1->GetValue() << endl;
    cout << MyObj2->GetValue() << endl;
}
```

[리스트 3-10]의 실행 결과는 다음과 같다.

```
10
20
20
```

3.11 클래스 템플릿 코딩 스타일 개선

위에서 예제로 구현한 다양한 클래스 템플릿의 코딩 스타일은 클래스 선언 안에서 각 멤버들의 정의를 구현하고 있다. 클래스의 코드 길이가 크지 않은 경우는 코드를 보는데 불편하지 않지만 코드 길이가 길어지는 경우 클래스의 전체적인 윤곽을 바로 알아보기가 쉽지 않다.

긴 코드를 가지는 클래스 템플릿의 경우는 클래스의 선언과 정의를 분리하는 것이 좋다. 위에서 예제로 나온 클래스 템플릿 중 의 Stack 클래스 템플릿을 선언과 정의를 분리하면 다음과 같다.

[리스트 3-11]

```
#include <iostream>
```

```
using namespace std;

template<typename T> class Stack
{
public:
   explicit Stack( int size );

   ~Stack();

   // 초기화 한다.
   void Clear();

   // 스택에 저장된 개수
   int Count();

   // 저장된 데이터가 없는가?
   bool IsEmpty();

   // 데이터를 담을 수 있는 최대 개수
   int GetStackSize();

   // 데이터를 저장한다.
   bool push( T data );

   // 스택에서 빼낸다.
   T pop();

private:
   T*  m_aData;
   int  m_Count;

   int m_Size;
};

template < typename T > Stack<T>::Stack( int size )
{
   m_Size = size;
   m_aData = new T[m_Size];
```

```
    Clear();
}

template < typename T > Stack<T>::~Stack()
{
    delete[] m_aData;
}

template < typename T > void Stack<T>::Clear()
{
    m_Count = 0;
}

template < typename T > int Stack<T>::Count()
{
    return m_Count;
}

template < typename T > bool Stack<T>::IsEmpty()
{
    return 0 == m_Count ? true : false;
}

template < typename T > int Stack<T>::GetStackSize()
{
    return m_Size;
}

template < typename T > bool Stack<T>::push(T data )
{
    // 저장할 수 있는 개수를 넘는지 조사한다.
    if( m_Count >= m_Size )
    {
        return false;
    }

    // 저장 후 개수를 하나 늘린다.
    m_aData[ m_Count ] = data;
    ++m_Count;
```

```
   return true;
}

template < typename T > T Stack<T>::pop()
{
   // 저장된 것이 없다면 0을 반환한다.
   if( m_Count  < 1 )
   {
      return 0;
   }

   // 개수를 하나 감소 후 반환한다.
   --m_Count;
   return m_aData[ m_Count ];
}

void main()
{
  Stack<int> kStack1(7);
  cout << "스택의 크기는?" << kStack1.GetStackSize() << endl;
}
```

[리스트 3-11]의 코드를 보면 알 수 있듯이 클래스 안에 정의를 했던 것과의 차이점은 클래스 멤버 정의를 할 때 템플릿 선언하고 클래스 이름에 템플릿 파라메터를 적어 준다.

3.12 클래스 선언과 정의를 각각 다른 파일에 하려면

일반적인 클래스의 경우 크기가 작은 경우를 제외하면 클래스의 선언과 정의를 서로 다른 파일에 한다.

클래스 템플릿의 경우는 일반적인 방법으로는 그렇게 할 수 없다. 클래스 멤버 정의를 선언과 다른 파일에 하려면 멤버 정의를 할 때 'export'라는 키워드를 사용한다. [리스트 3-11]의 GetStackSize()에 export를 사용하면 다음과 같이 된다.

```
template < typename T >
export int Stack::GetStackSize()
{
  return m_Size;
}
```

그러나 export라는 키워드를 사용하면 컴파일 에러가 발생한다. 이유는 현재 대부분의 C++ 컴파일러에서는 export라는 키워드를 지원하지 않는다. 왜냐하면, 이것을 지원하기 위해 필요로 하는 노력은 컴파일러를 새로 만들 정도의 노력을 필요로 할 정도로 어렵다고 한다. 현재까지도 대부분의 컴파일러 개발자들은 구현 계획을 세우지도 않고 있으며 일부에서는 구현에 반대하는 의견도 있다고 한다.

그럼 클래스 템플릿의 선언과 정의를 서로 다른 파일에 할 수 있는 방법은 없을까? 약간 편법을 사용하면 가능하다.

inline이라는 의미를 가지고 있는 '.inl' 확장자 파일에 클래스 구현하고 이 .inl 파일을 헤더 파일에서 포함한다. (참고로 .inl 파일을 사용하는 것은 일반적인 방식은 아니고 일부 라이브러리나 상용 3D 엔진에서 간혹 사용하는 것을 볼 수 있다).

[리스트 3-11]의 Stack 클래스 템플릿의 선언과 정의를 다른 파일로 하는 예의 일부를 다음에 구현했다.

[리스트 3-12]

```
// stack.h 파일
template<typename T>
class Stack
{
public:

  void Clear();

};

#include <iostream>
```

```
#include "stack.hpp"

// stack.inl 파일

template < typename T >
void Stack<T>::Clear()
{
  std::cout << "Clear() "<< std::endl;
}

// stack.cpp 파일
#include "stack.h"

void main()
{
   Stack<int> k;
   k.Clear();
}
```

이것으로 클래스 템플릿에 대한 설명을 마쳤다. 함수 템플릿에 대한 글을 봤다면 템플릿에 대한 어느 정도 이해를 가지고 있을 테니 어렵지 않게 이해를 할 수 있으리라 생각하지만 필자의 부족한 글 때문에 이해가 어렵지 않았을까라는 걱정도 조금한다.

글만 보고 넘기지 말고 직접 코딩 해 보기를 권장한다. 본문에 나오는 예제들은 모두 코드 길이가 짧은 것이라서 직접 코딩을 하더라도 긴 시간은 걸리지 않을 것이다.

다음 장부터는 본격적으로 STL에 대한 설명에 들어간다. 앞에서 이야기 했듯 STL은 템플릿으로 만들어진 것이다. 아직 템플릿의 유용성을 느끼지 못한 독자들은 STL에 대해서 알게 되면 템플릿의 뛰어남을 알게 되리라 생각한다.