

Hanbit eBook

Realtime 45

Thinking About

C++11 STL

프로그래밍

최흥배 지음

 한빛미디어
Hanbit Media, Inc.

Thinking About

C++11 STL 프로그래밍

Thinking About: C++11 STL 프로그래밍

초판발행 2013년 10월 31일

지은이 최흥배 / 펴낸이 김태현

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 978-89-6848-650-0 15000 / 정가 11,000원

책임편집 배용석 / 기획 김병희 / 편집 안선화

디자인 표지 여동일, 내지 스튜디오 [림], 조판 김현미

마케팅 박상용, 서은옥

이 책에 대한 의견이나 오타자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주세요.

한빛미디어 홈페이지 www.hanb.co.kr / 이메일 ask@hanb.co.kr

Published by HANBIT Media, Inc. Printed in Korea

Copyright © 2013 최흥배 & HANBIT Media, Inc.

이 책의 저작권은 최흥배와 한빛미디어(주)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanb.co.kr)로 보내주세요.

한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_ **최흥배**

2003년부터 현재에 이르기까지 PC 보드 게임부터 MMORPG, 모바일 게임을 아우르는 다양한 온라인 게임 서버 프로그램을 만들어온 개발자다. 게임 개발자로서 프로그래밍 언어 중 C++을 주 언어로, C#을 보조 언어로 사용하고 있다. 요즘은 C++11 프로그래밍과 Linux 플랫폼 프로그래밍, 심도 있는 .NET 기술, Scala에 대해 공부하고 있다. 기술이나 경험을 공유하는 것을 좋아하여 게임 개발자 커뮤니티나 세미나 강연을 통해 다른 프로그래머와 활발히 교류하고 있다. 웹이 대중화되기 전부터 프로그래밍 공부를 해 와서 그런지 여전히 새로운 기술을 배울 때는 책을 선호하여, 지금도 매달 새로운 프로그래밍 관련 책을 읽으며 연구하고 있다. 현재 T3엔터테인먼트(<http://www.t3.co.kr>)모바일 1팀에서 모바일 게임 서버를 개발 중이다.

- 블로그 : <http://jacking.tistory.com/>
- 트위터 : [@jacking75](#)

저자 서문

곧 있으면 C++11 표준이 나온 지 만 2년이 된다. 그러나 아직 현업이나 학교에서는 C++03 표준으로 C++ 프로그래밍을 하는 곳이 더 많은 것 같다. 이유는 한국에서 가장 많이 사용되고 있는 컴파일러인 Visual C++가 아직 완전하게 C++11을 지원하지 않고(우분투Ubuntu를 비롯한 리눅스에서 기본으로 제공하는 컴파일러는 최신 버전이 아니라서 윈도우Windows와 상황이 비슷하다), C++ 프로그래머 대부분이 아직 C++03 표준 문법만 알고 있기 때문이라고 생각한다.

다행히 최신의 C++ 컴파일러들은 C++11 표준을 모두 다 구현하고 있으며 Visual C++도 점차 지원을 늘리는 추세다. 또한, 새로 나오는 C++ 책들도 얼마 전부터는 C++11 표준에 대해 설명해주고 있다. 이 도서 역시 C++11과 관련된 책으로, C++11 중에서도 STL에 대한 부분을 집중적으로 다루고 있다.

C++로 프로그래밍을 하면 STL을 자주 사용하는데, C++11 문법을 다 모르더라도 이 책을 통해서 배운 기능만 사용한다면 기존보다 더 편하고 강력하게 C++ 프로그래밍을 할 수 있을 것이다.

이 책은 필자의 전작인 『[Thinking About C++ STL 프로그래밍](#)』(한빛미디어, 2012년)을 이은 책으로 전작에서 다루지 않은 부분을 다루고 있으며, 문법보다 활용에 초점을 맞추고 있다. 다만, 깊이 있는 이론보다는 활용에 중심을 두고 있어서 STL의 깊은 부분이나 세세한 부분까지는 다루지 않는다. 하지만 이 책을 보고 배운 것은 실제 프로그램을 개발하는 데 바로 사용할 수 있어서 독자에게는 실용적인 책이 될 것으로 생각한다.

이 책을 통해서 C++11의 새로운 STL 기능을 배우고, C++ 프로그래밍을 이전보다 더 쉽고, 강력하게 할 수 있기를 바란다.

끝으로 회사의 팀장님과 동료들, 오래된 게임 개발자 동료인 조진현 군, 한빛미디어의 김병희님 및 관계자 분들에게 감사의 마음을 전한다. 또 내가 집에서 프로그래밍 공부나 책 집필에 집중할 수 있도록 건강을 챙겨주고 배려해 준 아내 선민 씨에게, 집안일을 많이 도와주지 못해서 미안하다는 말과 함께 언제나 고맙고 사랑한다고 전하고 싶다.

집필을 마무리하며

최흥배

대상 독자 및 예제 파일

초급

초중급

중급

중고급

고급

이 책은 C++ 문법을 처음부터 설명해주는 책이 아니다. 그러므로 이 책을 보기 위해서는 기본적인 C++ 문법은 알고 있어야 한다. 그리고 이 책에서 사용하는 컴파일러인 Visual C++를 사용하여 C++ 프로젝트를 만들고 빌드하며 디버깅할 수 있어야 한다.

C++11을 알고 있다면 아주 좋지만 아직 모르고 있더라도 이 책을 보는 데 문제는 되지 않는다. 이 책을 이해하기 위해 필요한 C++11 기능은 책 초반에서 자세히 설명한다. 또 이 책은 필자의 전작인 『Thinking About C++ STL 프로그래밍』(한빛미디어, 2012년 12월)에서 설명하는 내용을 알고 있는 사람을 기준으로 하고 있으니, STL에 대해 잘 알지 못하는 분들은 이 책을 보기 전에 꼭 『Thinking About C++ STL 프로그래밍』(비매품, 무료로 제공)을 먼저 읽어보길 바란다.

책에서 사용한 예제 파일은 다음의 웹 사이트에서 다운받을 수 있다.

- <https://www.hanb.co.kr/exam/2650>

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

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

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

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

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

2. 무료로 업데이트되는, 전자책 전용 서비스입니다.

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

3. 독자의 편의를 위하여 DRM-Free로 제공합니다.

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

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

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

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

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

다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횡수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오타자 교정이나 내용의 수정·보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

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

차례

01	C++11 기초	1
	1.1 auto	1
	1.2 람다	5
	1.3 range base for	14
	1.4 enum	21
	1.5 nullptr	25
02	메모리 관리	28
	2.1 shared_ptr	29
	2.2 unique_ptr	43
03	tuple	54
	3.1 tuple이 필요할 때	54
	3.2 tuple 사용하기	55
	3.3 더 간단하게 tuple 정의하기	58
	3.4 tuple에 저장한 데이터 개수 알기	59
	3.5 tuple의 요소를 한꺼번에 다른 변수에 담기	60
	3.6 tuple과 tuple 합치기	62
04	array	64
	4.1 array 사용하기	64
	4.2 요소 접근	67

4.3	array의 크기 알기	68
4.4	fill을 사용하여 특정 값 채우기	68
05	forward_list	69
<hr/>		
5.1	forward_list를 만든 이유	69
5.2	forward_list 사용하기	70
5.3	데이터 추가하기	72
5.4	데이터 삭제하기	75
5.5	정렬	78
5.6	중복 제거	80
5.7	합치기	83
06	unordered_map	91
<hr/>		
6.1	unordered_map 사용하기	91
6.2	데이터 추가와 삭제	96
6.3	검색	97
6.4	검색 - 없으면 추가하기	101
6.5	클래스(혹은 구조체)를 key로 사용하기	103
07	chrono	107
<hr/>		
7.1	chrono 사용하기	107
7.2	시간 단위	109

7.3	시간 연산	111
7.4	clock 클래스	114

08	std::thread	117
-----------	--------------------	------------

8.1	스레드 만들기	117
8.2	스레드가 종료 될 때까지 기다리기	120
8.3	스레드 식별자와 스레드 교환	124
8.4	스레드 떼어내기, 스레드 종료 대기 가능 조사	127
8.5	하드웨어 스레드 개수 알기	130
8.6	스레드 일시 중지 및 양보하기	131
8.7	thread 객체를 join, detach하지 않고 파괴했을 때	133

09	동기화 객체	136
-----------	---------------	------------

9.1	공유 객체 동기화하기	136
9.2	자동으로 락 풀기	141
9.3	반복하여 락 걸기	144
9.4	지정한 시간 동안만 락 시도하기	146
9.5	unique_lock	148
9.6	복수의 mutex 객체를 동시에 락 걸기.....	149
9.7	프로그램에서 딱 한번만 실행하기.....	151
9.8	스레드마다 데이터 저장하기.....	153
9.9	이벤트를 사용한 동기화 객체	156
9.10	스레드 대기	159

10	std::atomic	160
<hr/>		
10.1	lock-free로 공유 자원 조작하기	160
10.2	초기화, 읽고 쓰기	164
10.3	연산 조작	166
10.4	바꾸기, 비교 후 바꾸기	170
11	async/future	174
<hr/>		
11.1	함수를 비동기로 실행하기	174
11.2	std::thread에서 비동기로 함수 실행	177
11.3	비동기 task	179

1 | C++11 기초

C++11에는 C++ 표준에 비해 문법상 많은 기능이 추가되었다. 그중 auto, lambda⁰¹, range base for, enum, nullptr은 STL 프로그래밍에도 자주 사용되는 C++11의 대표적인 기능으로, 이 책의 주제인 STL을 다루기 전에 먼저 이들 기능에 대해 간단히 살펴볼 것이다. 관련 문법을 알고 있다면 2장부터 학습하면 된다.

1.1 auto

1.1.1 컴파일 타임 때 타입을 정하는 ‘auto’

C++11에서는 ‘auto’라는 키워드가 새로 생겼다. auto를 사용하면 변수를 정의할 때 명시적으로 타입을 지정하지 않아도 된다. auto는 변수를 초기화할 때 초기화 값에 따라서 자동으로 타입을 결정해주기 때문이다. 즉, auto는 ‘변수를 정의할 때 명시적으로 타입을 지정하지 않고 컴파일 타임 때 자동으로 결정하게 해주는 키워드’라고 기억하면 된다. auto는 지역 변수에서만 사용할 수 있어서 클래스의 멤버 변수나 전역 변수, 함수의 인자로는 사용할 수 없다.

여기서 잠깐_ 정적 언어와 동적 언어의 차이

근래에 JavaScript, Ruby, Python 같은 스크립트 언어가 많은 인기를 얻고 있다. 이런 스크립트 언어를 ‘동적 언어’라고 한다. 이와 반대로 현재 가장 많이 사용되고 있는 언어인 C, C++, C#, Java, Objective-C는 ‘정적 언어’라고 한다.

정적 언어와 동적 언어의 가장 큰 차이는, 정적 언어는 변수 타입^{type}, 형을 선언하거나 정의할 때 명시적으로 지정해야 하지만 동적 언어는 변수 타입을 명시적으로 지정하지 않아도 된다는 점이다.

01 lambda는 주로 ‘람다’라 읽고 표기한다. 이후부터는 ‘람다’라고 표기한다.

C/C++에서의 지역 변수 정의 (정적 언어)

```
int Money = 500;
```

Ruby에서의 지역 변수 정의 (동적 언어)

```
def BuyItem
  ...
  Money = 500;
  .....
end
```

1.1.2 auto를 사용한 예제

auto를 사용하여 기본 타입의 문자열과 정수를 담은 변수를 정의하면 다음과 같다.

예제 1-1 문자열과 정수 변수에 auto 사용 (예제 파일: auto_01)

```
#include <iostream>

int main()
{
    // char*
    auto NPCName = "BugKing";
    std::cout << NPCName << std::endl;

    // int
    auto Number = 1;
    std::cout << Number << std::endl;

    return 0;
}
```

```
}
```

예제 1-1의 실행 결과

```
BugKing
```

```
1
```

auto는 당연히 포인터, 참조, const에도 사용할 수 있다.

예제 1-2 auto를 포인터, 참조, const에 사용한 예 (예제 파일: auto_02)

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    int UserMode = 4;
```

```
    auto* pUserMode = &UserMode;
```

```
    std::cout << "pUserMode : Value - " << *pUserMode << ", address : "
```

```
        << pUserMode << std::endl;
```

```
    auto& refUserMode = UserMode;
```

```
    refUserMode = 5;
```

```
    std::cout << "UserMode : Value - " << UserMode
```

```
        << " | refUserMode : Value - " << refUserMode << std::endl;
```

```
    return 0;
```

```
}
```

예제 1-2의 실행 결과

```
pUserMode : Value - 4, address : 003FF7D4
```

```
UserMode : Value - 5 | refUserMode : Value - 5
```


또 '사용자 정의 타입'인 클래스를 사용할 때도 auto를 사용할 수 있다.

```
struct CharacterInvenInfo
{
    int SlotNum;
    int ItemCode;
};

.....
auto* CharInven = new CharacterInvenInfo();
.....
```

템플릿 프로그래밍이나 STL을 사용할 때 auto를 사용하면 프로그래밍이 이전보다 훨씬 더 간편해짐을 알 수 있을 것이다. STL의 컨테이너를 사용할 때 가장 귀찮은 부분 중의 하나가 반복자를 정의할 때 변수 타입을 길게 적어야 한다는 점이다. 이런 경우가 자주 있다면 어쩔 수 없이 typedef를 사용하여 불편함을 조금이라도 감소시켜야 했지만, auto를 사용하면서는 이런 불편함을 깔끔하게 없앨 수 있다. 이 때문에 좀 과장하자면 auto는 STL 프로그래밍을 위해서 나온 것 같다고 느낄 정도다.

```
typedef std::list<MCommand*> LIST_COMMAND;
LIST_COMMAND::iterator iter = m_listCommand.begin();
```

위 코드를 다음처럼 바꿀 수 있다.

```
auto iter = m_listCommand.begin();
```

auto는 개념이 정말 단순하며 사용 방법이 아주아주 쉬워 공부하지 않아도 바로 사

용할 수 있어서 모든 C++11 프로그래머가 애용하는 기능 중 하나가 될 것으로 생각한다.

1.2 람다

람다^{lambda}는 ‘람다 함수’ 또는 ‘이름 없는 함수’라고 부르며, 그 성질은 함수 객체와 같다. C++ 규격에서 람다는 특별한 타입을 가지고 있다. 그렇지만 ‘decltype’ (C++11에서 새로 생긴 것)이나 ‘sizeof’에서는 사용할 수 없다.

람다 덕분에 C++의 표현력이 이전보다 훨씬 더 증대되었고 특히 STL의 알고리즘을 사용할 때 아주 유용하다.

1.2.1 C++에서 STL 알고리즘을 사용할 때 불편했던 점

기존 C++에서 STL의 find_if, sort 등의 알고리즘을 사용할 때 특정 조건자를 사용하려면 함수 객체를 정의해야 했다. 그런데 STL 알고리즘 함수에서만 사용하려고 따로 함수 객체를 만들려니 귀찮을 수밖에 없었다. 때문에 보통은 함수를 따로 만들어서 구현하거나, 함수를 정의하는 것도 귀찮아서 그냥 STL 알고리즘 사용을 포기하고 직접 컨테이너를 다루었다. 람다^{lambda} 덕분에 이제는 이런 수고를 할 필요가 없다.

1.2.2 람다 사용 방법

람다의 기본 문법은 다음과 같다.

```
int main()
{
    [] // lambda capture
    () // 함수의 인수 정의
    {} // 함수의 본체
    (); // 함수 호출
}
```

다음은 람다를 사용한 아주 간단한 예다.

```
int main()
{
    []{ std::cout << "Hello, TechDay!" << std::endl; }();
}
```

1.2.3 람다 사용 예

auto를 사용하면 람다를 변수에 대입할 수 있다.

예제 1-3 람다를 변수에 대입 (예제 파일: lambda_01)

```
#include <iostream>

int main()
{
    auto func = [] { std::cout << "Hello, TechDay!" << std::endl; };
    func();
    return 0;
}
```

예제 1-3의 실행 결과

```
| Hello, TechDay!
```

람다는 일반 함수처럼 파라미터를 정의할 수도 있다.

예제 1-4 파라미터 사용 (예제 파일: lambda_02)

```
#include <iostream>

int main()
```

```

{
    auto func = [](int n) { std::cout << "Number : " << n << std::endl; };

    func(333);
    func(7777);

    return 0;
}

```

예제 1-4의 실행 결과

```

| Number : 333
| Number : 7777

```

람다는 반환 값을 넘길 수도 있는데, 반환 값의 타입은 명시적으로 지정할 수도 있고 암묵적으로 추론할 수도 있다.

예제 1-5 반환 값 사용

```

int main()
{
    auto func1 = [] { return 3.14; };
    auto func2 = [] (float f) { return f; };
    auto func3 = [] () -> float{ return 3.14; };

    float f1 = func1();
    float f2 = func2(3.14f);
    float f3 = func3();

    return 0;
}

```

1.2.4 STL의 find_if에서 람다 사용

앞에서 “STL의 알고리즘을 사용할 때 람다를 사용하면 아주 유용하다”라고 했는데, 다음 예제를 보면 얼마나 편해지는지 한눈에 알 수 있다.

예제 1-6 find_if 알고리즘에서 람다 사용 (예제 파일: lambda_03)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

class User
{
public:
    User() : m_bDie(false) {}
    ~User() {}

    void SetIndex(int index) { m_Index = index; }
    int GetIndex() { return m_Index; }
    void SetDie() { m_bDie = true; }
    bool IsDie() { return m_bDie; }

private:
    int m_Index;
    bool m_bDie;
};

int main()
{
    vector< User > Users;
    User tUser1;
```

```

tUser1.SetIndex(1);
Users.push_back(tUser1);

User tUser2;
tUser2.SetIndex(2);
tUser2.SetDie();
Users.push_back(tUser2);

User tUser3;
tUser3.SetIndex(3);
Users.push_back(tUser3);

// 죽은 유저 찾기
auto Iter = find_if(Users.begin(), Users.end(),
    [](User& tUser) -> bool { return true == tUser.IsDie(); }
);

cout << "found Die User Index : " << Iter->GetIndex() << endl;
return 0;
}

```

예제 1-6의 실행 결과

```
| found Die User Index : 2
```

C++11 이전의 C++ 표준에서는 find_if 알고리즘을 사용하여 '죽은 유저'를 찾기 위해서 다음과 같은 함수 객체를 정의한 후 사용해야 했다.

```

struct FindDieUser
{
    bool operator()(User& tUser) const

```

```

    {
        return tUser.IsDie();
    }
};

```

```
Iter = find_if(Users.begin(), Users.end(), FindDieUser());
```

이에 반해 람다를 사용한 예제 1-6에서는 한 줄로 간단하게 끝내버렸다. 정말 간단하지 않은가!

1.2.5 캡처

람다를 사용할 때 람다 외부에 정의되어 있는 변수를 람다 내부에서 사용하고 싶으면 그 변수를 캡처capture하면 된다. 캡처는 참조나 복사로 전달이 가능하다. 참조를 사용하는 경우는 ‘&’을 사용하고 복사로 전달할 때는 ‘변수 이름’을 기술한다.

람다 표현의 “[(파라미터) { 식 } ”에서 앞의 ‘[’ 사이에 캡처할 변수를 기술한다.

참조로 캡처

예제 1-7 람다에서 캡처 사용 (예제 파일: lambda_04)

```

int main()
{
    vector< int > Moneys;
    Moneys.push_back(100);
    Moneys.push_back(4000);
    Moneys.push_back(50);
    Moneys.push_back(7);

    int TotalMoney1 = 0;
    for_each(Moneys.begin(), Moneys.end(), [&TotalMoney1](int Money) {
        TotalMoney1 += Money;
    });
}

```

```
});  
cout << "Total Money 1 : " << TotalMoney1 << endl;  
return 0;  
}
```

예제 1-7의 실행 결과

```
| Total Money 1 : 4157
```

람다 식이 외부에 있는 TotalMoney1 변수를 참조로 캡처하여 Moneys에 있는 값을 모두 더한다.

복사로 캡처

TotalMoney1 변수를 값으로 전달하면 어떻게 될까? 다음 코드를 살펴보자.

```
for_each(Moneys.begin(), Moneys.end(), [TotalMoney1](int Money) {  
    TotalMoney1 += Money;  
});
```

위의 코드를 컴파일하면 다음과 같은 컴파일 에러가 발생한다.

```
“error C3491: ‘TotalMoney1’: a by-value capture cannot be modified in a  
non-mutable lambda”
```

만약 복사로 캡처한 변수를 꼭 람다 내부에서 변경해야 한다면 mutable 키워드를 사용하여 에러 없이 컴파일할 수 있다.

```
[=]() mutable { std::cout << x << std::endl; x = 200; }();
```

컴파일은 되지만 람다 내부에서 변경한 외부 변수의 값은 람다를 벗어날 경우 람다 내부에서 변경하기 전 원래의 값이 된다.

복수의 변수 캡처

예제 1-7에서는 변수를 하나만 캡처했지만 복수의 변수도 캡처할 수 있다. '[' 사이에 캡처할 변수를 선언하면 된다.

```
[ &Numb1, &Numb2 ]
```

그럼 '['로 하면 어떻게 될까? 이렇게 하면 람다 식을 정의한 범위 내에 있는 모든 변수를 캡처할 수 있다. 또 람다 외부의 모든 변수를 복사하여 캡처할 때는 '=' 을 사용하면 된다.

예제 1-8 람다 외부의 모든 변수를 참조로 캡처하기 (예제 파일: lambda_05)

```
int main()
{
    vector< int > Moneys;
    Moneys.push_back(100);
    Moneys.push_back(4000);
    Moneys.push_back(1001);
    Moneys.push_back(7);

    int TotalMoney1 = 0;
    int TotalBigMoney = 0;

    // Money가 1000보다 크면 TotalBigMoney에 누적합니다.
    for_each(Moneys.begin(), Moneys.end(), [&](int Money) {
        TotalMoney1 += Money;
        if(Money > 1000) TotalBigMoney += Money;
    });
}
```

```

});

cout << "Total Money 1 : " << TotalMoney1 << endl;
cout << "Total Big Money : " << TotalBigMoney << endl;

return 0;
}

```

예제 1-8의 실행 결과

```

| Total Money 1 : 5108
| Total Big Money : 5001

```

default 캡처

람다 외부의 모든 변수를 참조(또는 복사)로 참조하고 일부는 복사(또는 참조)로 캡처할 수도 있다. 그러나 같은 변수를 캡처하거나 default 캡처한 일부 변수를 같은 방식(참조 또는 복사)으로 캡처할 수는 없다.

```

int main()
{
    int n1, n2, n3, n4, n5;

    [&, n1, n2] {}; // n3, n4, n5는 참조, n1, n2는 복사
    [=, &n1, &n2] {}; // n3, n4, n5는 복사, n1, n2는 참조
    [n1, n1] {}; // Error! 같은 변수를 사용
    [&, &n1] {}; // Error! n1을 이미 default 참조로 사용
    [=, n1] {}; // Error! n1을 이미 default 복사로 사용

    return 0;
}

```

1.3 range base for

이번에 설명할 'range base for'는 'auto'와 더불어 C++11의 기능 중 유용하면 서도 사용하기 쉽고 자주 사용하게 되는 기능이다. range base for를 사용하면 반복문을 아주 쉽고 안전하게 사용할 수 있다.

range base for 기능은 마이크로소프트사의 Visual C++(VC)의 특화 기능인 'for each'문과 비슷하다. 때문에 이미 for each문을 사용하고 있는 개발자라면 단지 range base for문으로 바꾸어서 사용하면 된다.

예제를 통해 일반적인 for문과 VC의 for each문, range base for문의 차이를 살펴 보겠다.

예제 1-9 for문, VC의 for each문, range base for문 비교 (예제 파일: range_01)

```
#include <iostream>

int main()
{
    int NumberList[5] = { 1, 2, 3, 4, 5 };
    std::cout << "일반적인 for문" << std::endl;

    for(int i = 0; i < 5; ++i)
    {
        std::cout << i << std::endl;
    }

    std::cout << "VC++ 특화의 for each문" << std::endl;

    for each(int i in NumberList)
    {
        std::cout << i << std::endl;
    }
}
```

```

    }

    std::cout << "range base for문" << std::endl;

    for(auto i : NumberList)
    {
        std::cout << i << std::endl;
    }

    return 0;
}

```

예제 1-9의 실행 결과

일반적인 for문

```

0
1
2
3
4

```

VC++ 특화의 for each문

```

1
2
4
5

```

range base for문

```

1
2
3
4
5

```

예제 1-9를 보면 일반적인 for문은 다음과 같이 시작 조건, 종료 조건, 증가 값이라는 3개의 조건에 의해서 반복된다는 것을 알 수 있다.

```
for(int i = 0; i < 5; ++i)
```

그러나 range base for문은 VC만의 반복문인 for each문과 비슷하게 데이터셋 변수와 이 데이터셋 요소의 타입을 선언하면 된다.

```
for(auto i : NumberList)
```

기존의 for문에 비해서(또는 for each문보다) 간편하지 않은가! 그뿐만 아니라, for each가 C++ 표준이 아닌 VC만의 기능인 데 비해 range base for문은 C++ 표준 기능이다.

range base for문의 표준 문법을 다시 한 번 살펴보자.

```
for (for-range-declaration : expression) statement
```

range base for를 사용함으로써 반복문 사용이 편해졌을뿐더러 for문을 사용할 때 종료 조건이 잘못되어 메모리를 침범하게 되는 위험도 피할 수 있다. 너무 당연하지만 range base for문은 배열뿐만이 아니라 STL의 컨테이너도 사용할 수 있다.

다음 예제를 통해 range base for문에서 STL 컨테이너를 어떻게 사용하는지 살펴보자.

예제 1-10 range base for문에서 STL 컨테이너 사용 (예제 파일: range_02)

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <string>

int main()
{
    std::cout << "range base for - vector" << std::endl;

    std::vector<int> NumberList;
    NumberList.push_back(1);
    NumberList.push_back(2);
    NumberList.push_back(3);

    for(auto i : NumberList)
    {
        std::cout << i << std::endl;
    }
    std::cout << std::endl;
    std::cout << "range base for - unordered_map" << std::endl;

    std::unordered_map<int, std::string> NumString;
    NumString.insert(std::make_pair<int, std::string>(1, "1"));
    NumString.insert(std::make_pair<int, std::string>(2, "2"));
    NumString.insert(std::make_pair<int, std::string>(3, "3"));

    for(auto i : NumString)
    {
        std::cout << "key : " << i.first << ", value : " << i.second << std::endl;
    }
}
```

```
std::cout << std::endl;

return 0;
}
```

예제 1-10의 실행 결과

```
range base for - vector
1
2
3

range base for - unordered_map
key : 1, value : 1
key : 2, value : 2
key : 3, value : 3
```

기본적으로 STL의 ‘이터레이터⁰²iterator’⁰²를 지원하는 컨테이너라면 range base for 문을 문제없이 사용할 수 있다. 그러므로 프로그래머가 자신만의 컨테이너를 만들 때 STL에서 정의한 이터레이터의 기능을 구현하면 range base for문을 사용할 수 있다.

range base for문에서 데이터셋의 요소 변경

예제 1-9에서는 다음과 같이 for문을 사용했는데 이런 경우 i의 값을 for문 안에서 변경할 수 있지만, for문을 나오면 NumberList의 요소에는 적용되지 않는다.

```
for(auto i : NumberList)
```

02 배열, 리스트, 큐 등의 다양한 컨테이너에서 같은 연산으로 동일한 동작을 할 수 있도록 한다.

만약 요소의 값을 변경하고 싶다면 참조를 사용하면 된다.

```
for(auto &i : NumberList)
```

참조를 사용하면 for문을 나와도 NumberList의 요소에는 변경이 적용되어 있다. 그런데 만약 for문에서 요소 값을 변경하지 못하도록 하려면 어떻게 해야 할까? 이럴 땐 const를 사용한다.

```
for(auto const i : NumberList)
```

또 for문에서 데이터셋 요소에 접근할 때는 임시 변수를 만드니, 이 비용을 줄이고 싶다면 참조를 사용하는 게 좋다. 만약 요소의 값을 변경하지 못하도록 하고 싶다면 const 참조를 사용한다.

```
for(auto const &i : NumberList)
```

예제를 통해서 배운 내용을 적용해보자.

예제 1-11 참조 사용하기 (예제 파일: range_03)

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> NumberList;
    NumberList.push_back(1);
    NumberList.push_back(2);
```



```

NumberList.push_back(3);

for(auto i : NumberList)
{
    std::cout << i << " * 10 : ";
    i *= 10;
    std::cout << i << std::endl;
}

for(auto i : NumberList)
{
    std::cout << i << " ";
}

std::cout << std::endl << std::endl;

for(auto &i : NumberList)
{
    std::cout << i << " * 10 : ";
    i *= 10;
    std::cout << i << std::endl;
}

for(auto i : NumberList)
{
    std::cout << i << " ";
}

std::cout << std::endl;

return 0;
}

```

예제 1-11의 실행 결과

```
1 * 10 : 10
```

```
2 * 10 : 20
```

```
3 * 10 : 30
```

```
1 2 3
```

```
1 * 10 : 10
```

```
2 * 10 : 20
```

```
3 * 10 : 30
```

```
10 20 30
```

1.4 enum

enum은 C++에서 이미 사용하고 있는 키워드다. 그러나 C++11의 enum은 C++03 표준과는 달리 두 종류의 enum으로 바뀌었다. 바로 ‘unscoped enumeration’과 ‘scoped enumeration’이다.

1.4.1 unscoped enumeration

unscoped enumeration은 기존 (C++03) enum과 비슷한 것이라고 생각하면 된다. unscoped enumeration은 다음과 같이 정의한다.

```
enum ITEMTYPE : short
{
    WEAPON,
    EQUIPMENT,
    GEM = 10,
    DEFENSE,
};
```

그리고 다음과 같이 사용하면 된다.

```
short ItemType = WEAPON;
```

다음과 같이 사용해도 된다.

```
short ItemType = ITEMTYPE::WEAPON; // C++03에서는 에러
```

1.4.2 scoped enumeration

scoped enumeration은 다음과 같이 정의한다.

```
enum class CHARACTER_CLASS : short
{
    WARRIOR = 1,
    MONK,
    FIGHTER,
};
```

그리고 다음과 같이 사용하면 된다.

```
CHARACTER_CLASS CharClass = CHARACTER_CLASS::WARRIOR;
```

다만, 다음과 같이 사용하면 에러가 발생한다.

```
short CharClassType = FIGHTER; // 에러
```

scoped enumeration은 unscoped enumeration과 다르게 CHARACTER_CLASS를 생략하면 안 된다. 즉, WARRIOR나 MONK는 CHARACTER_CLASS의 범위 안에 있음을 가리킨다. 그리고 'enum class' 대신 'enum struct'을 사용해도 괜찮다. 또 타입을 지정하지 않으면 기본적으로 'int' 타입이 된다.

1.4.3 형 변환

unscoped enumeration은 기존과 같이 암묵적으로 정수로 변환할 수 있다.

```
int i = WEAPON;
```

그러나 scoped enumeration은 명시적으로 '타입캐스팅typecasting'해야 한다.

```
int i = static_cast<int>(CHARACTER_CLASS::WARRIOR);
```

예제 1-12 scoped enumeration을 unscoped enumeration으로 사용하기 (예제 파일: enum_01)

```
#include <iostream>

// unscoped enumeration
enum ITEMTYPE : short
{
    WEAPON,
    EQUIPMENT,
    GEM = 10,
    DEFENSE,
};

// scoped enumeration
enum class CHARACTER_CLASS : short
```