
Hanbit eBook
Realtime 69
Thinking About
C++11 STL
프로그래밍
개정2판
최흥배 지음
한빛미디어
Hanbit Media, Inc.

Thinking About

C++11 STL 프로그래밍

개정2판

Thinking About: C++11 STL 프로그래밍 개정 2판

초판발행 2013년 10월 31일
2판발행 2014년 06월 12일

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 978-89-6848-662-3 15000 / **정가** 12,000원

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

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

Published by HANBIT Media, Inc. Printed in Korea

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

저자 소개

지은이_ 최흥배

2003년부터 현재에 이르기까지 PC 보드 게임부터 MMORPG, 모바일 게임을 아우르는 다양한 온라인 게임 서버 프로그램을 만들어온 개발자다. 게임 개발자로서 프로그래밍 언어 중 C++를 주 언어로, C#을 보조 언어로 사용하고 있다(그러나 최근에는 모바일 게임 서버 개발에 C#을 더 많이 사용하고 있다). 요즘은 C++11/14 프로그래밍과 심도 있는 .NET 기술, 유명 백엔드 오픈 소스 라이브러리 및 프로그램, 프로그래밍 언어 Ruby와 Scala에 대해 공부하고 있다. 기술과 개발 경험을 여러 사람과 나누는 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 서버 플랫폼을 개발 중이다.

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

저자 서문

C++11이 나온 지 몇 년 되지 않았는데 올해는 새로운 표준인 C++14(마이너 업데이트)가 나올 예정이다. 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(우분투를 비롯한 리눅스에서 기본으로 제공하는 컴파일러는 최신 버전이 아니라서 윈도우와 비슷한 상황이다) C++ 프로그래머 대부분이 아직 C++03 표준 문법만 알고 있기 때문이라고 생각한다.

다행히 최신 C++ 컴파일러들은 C++11 표준을 모두 다 구현하고 있으며 Visual C++도 점차 지원을 늘리는 추세다(곧 나올 새로운 버전인 Visual C++13에서 대부분을 구현할 예정이다). 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++11 STL 프로그래밍』(한빛미디어, 2013년)의 내용을 더 보강하고, 문법보다 활용에 초점을 맞추고 있다. 다만, 깊이 있는 이론보다는 활용에 중심을 두고 있어서 STL의 깊은 부분이나 세세한 부분까지는 다루지 않는다. 하지만 이 책에서 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 게임 개발자 커뮤니티를 통해서 알게 된 임영기 님, 박주항 님 이지현 님, 이욱진 님을 비롯한 게임 프로그래머분들, 부산에서 게임 프로그래머가 되기 위해 같이 공부하고 지금은 같은 업계 동료가 된 조진현, 최우영 군, 한빛미디어의 정지연 님, 김병희 님 및 관계자분들에게 감사의 마음을 전한다.

그리고 집에서 프로그래밍 공부와 책 집필에 집중할 수 있도록 건강을 챙겨주고 배려해 준 아내 선민 씨에게 집안일을 많이 도와주지 못해서 미안하다는 말과 함께 언제나 고맙고 사랑한다고, 또한 올해 초에 태어난 귀여운 딸 지유에게 사랑하고 건강하게 자라길 바란다고 전하고 싶다.

집필을 마무리하며

최흥배

대상 독자 및 예제 파일

초급 **초중급** 중급 중고급 고급

이 책은 C++ 문법을 처음부터 설명해주는 책이 아니므로 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 없다. 이 책을 이해하는 데 필요한 C++11 기능은 책 초반에서 자세히 설명한다. 또한, 이 책은 필자의 전작인 『Thinking About C++ STL 프로그래밍』(한빛미디어, 2012년, 비매품, 무료 제공)에서 설명하는 내용을 알고 있다고 전제하고 있으므로 STL에 대해 잘 알지 못하는 분들은 이 책을 보기 전에 꼭 먼저 읽어보기를 바란다.

책에서 사용한 예제 파일은 다음 웹 사이트에서 내려받을 수 있다.

- https://www.hanbit.co.kr/exam/2662

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

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

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

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

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

2. **무료로 업데이트되는, 전자책 전용 서비스입니다.**

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

3. 독자의 편의를 위해 DRM-Free로 제공합니다.

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

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

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

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

1 | C++11 기초

C++11은 C++ 표준보다 문법상 많은 기능이 추가되었다. 그중 auto, lambda[01], range based for, enum, nullptr은 STL 프로그래밍에도 자주 사용되는 C++11의 대표적인 기능이므로 이 책의 주제인 STL을 다루기 전에 먼저 이들 기능에 대해 간단히 살펴본다. 관련 문법을 알고 있다면 2장부터 학습해도 된다.

1.1 auto

1.1.1 컴파일 때 형을 정하는 'auto'

C++11에는 'auto'라는 키워드가 새로 생겼다. auto를 사용하면 변수를 정의할 때 명시적으로 형Tpye을 지정하지 않아도 된다. auto는 변수를 초기화할 때 초기화 값에 따라서 자동으로 형을 결정해주기 때문이다. 즉, auto는 '변수를 정의할 때 명시적으로 형을 지정하지 않고 컴파일 때 자동으로 결정해주는 키워드'라고 기억하면 된다. auto는 지역 변수에서만 사용할 수 있으므로 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 언어는 변수형Type을 선언하거나 정의할 때 명시적으로 지정해야 하지만, 동적 언어는 변수형을 명시적으로 지정하지 않아도 되는 점이 가장 다르다.

01 lambda는 주로 '람다'라 읽고 표기한다. 이후부터는 '람다'라고 표기한다.

C/C++에서 지역 변수의 정의 (정적 언어)

```
int Money = 500;
```

Ruby에서 지역 변수의 정의 (동적 언어)

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

1.1.2 auto를 사용한 예제

auto를 사용하여 기본형의 문자열과 정수를 담을 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 1-1] 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 1-2] 실행 결과

```
pUserMode : Value - 4, address : 003FF7D4
UserMode : Value - 5 ¦ refUserMode : Value - 5
```

또한, '사용자 정의형'인 클래스에도 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 중 하나가 반복자를 정의할 때 변수형을 길게 적어야 한다는 점이다. 이런 경우가 자주 있다면 어쩔 수 없이 typedef를 사용하여 불편함을 조금이라도 감소시켜야 하지만, auto를 사용하면 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();
```

이 코드를 다음처럼 바꿀 수 있다.

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

auto는 개념이 매우 단순하고 사용 방법이 아주 쉬워서 공부하지 않아도 바로 사용할 수 있으므로 모든 C++11 프로그래머가 애용하는 기능 중 하나가 될 것으로 생각한다.

1.2 람다

람다lambda는 '람다 함수' 또는 '이름 없는 함수'라고 부르며, 그 성질은 함수 객체와 같다. C++ 규격에서 람다는 특별한 형을 가지고 있다. 그러나 'decltype'(C++11에서 새로 생김)과 'sizeof'에서는 사용할 수 없다.

람다 덕택에 C++의 표현력이 이전보다 훨씬 더 증대되었고, 특히 STL 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. 그 때문에 보통은 함수를 따로 만들어서 구현하거나, 함수를 정의하는 것도 귀찮아서 그냥 STL 알고리즘 사용을 포기하고 직접 컨테이너를 다루었다. 람다lamda 덕택에 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 1-3] 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 1-4] 실행 결과

```
Number : 333
Number : 7777
```

람다는 반환값을 넘길 수도 있는데, 반환값의 형은 명시적으로 지정할 수도 있고 암묵적으로 추론할 수도 있다.

[예제 1-5] 반환값 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 1-6] 실행 결과

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

C++11 이전의 C++ 표준에서는 find_if 알고리즘을 사용해서 '죽은 유저'를 찾으려면 다음과 같은 함수 객체를 정의해야 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 끝내버린다. 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. 참조로 전달할 때는 '&'을, 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()
{
  std::vector< int > Moneys;
  Moneys.push_back(100);
  Moneys.push_back(4000);
  Moneys.push_back(50);
  Moneys.push_back(7);

  int TotalMoney1 = 0;
  std::for_each(Moneys.begin(), Moneys.end(), [&TotalMoney1](int Money) {
    TotalMoney1 += Money;
  });
  std::cout << "Total Money 1 : " << TotalMoney1 << std::endl;
  return 0;
}
```

[예제 1-7] 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;
});
```

이 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 벗어나면 람다 내부에서 변경하기 전의 원래 값이 된다.

복수의 변수 캡처

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

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

그럼 '[&]'으로 하면 어떻게 될까? 이렇게 하면 람다 식을 정의한 범위 내에 있는

모든 변수를 캡처할 수 있다. 또한, 람다 외부의 모든 변수를 복사하여 캡처할 때는 '[=]'을 사용한다.

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

```
int main()
{
  std::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 누적한다.
  std::for_each(Moneys.begin(), Moneys.end(), [&](int Money) {
    TotalMoney1 += Money;
    if(Money > 1000) TotalBigMoney += Money;
  });

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

  return 0;
}
```

[예제 1-8] 실행 결과

```
Total Money 1 : 5108
ToTal Big Money : 5001
```

default 캡처

람다 외부의 모든 변수를 참조(또는 복사)로 캡쳐하고 일부는 복사(또는 참조)로 캡처할 수 있다. 하지만 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 defalut 참조로 사용
  [=, n1] {};        // Error! n1을 이미 defalut 복사로 사용

  return 0;
}
```

1.3 range based for

이번에 설명할 'range based for'는 'auto'와 더불어 C++11의 기능 중 유용하면서도 사용하기 쉽고 자주 사용하는 기능이다. range based for를 사용하면 반복문을 아주 쉽고 안전하게 사용할 수 있다.

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

예제를 통해 일반적인 for문과 VC의 for each문, range based for문의 차이를 살펴보자.

[예제 1-9] for문, VC의 for each문, range based 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 based for문" << std::endl;

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

   return 0;
}
```

[예제 1-9] 실행 결과

```
일반적인 for문
0
```

```
1
2
3
4
VC++ 특화의 for each문
1
2
4
5
range based for문
1
2
3
4
5
```

예제 1-9를 보면 일반적인 for문은 다음과 같이 시작 조건, 종료 조건, 증가값이라는 세 가지 조건에 의해서 반복된다는 것을 알 수 있다.

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

그러나 range based for문은 VC만의 반복문인 for each문과 비슷하게 데이터셋 변수와 이 데이터셋 요소의 형을 선언하면 된다.

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

기존 for문보다(또는 for each문보다) 간편하지 않은가! 그뿐만 아니라 for each가 C++ 표준이 아닌 VC만의 기능인 데 비해 range based for문은 C++ 표준 기능이다.

range based for문의 표준 문법을 다시 한 번 살펴보자.

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

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

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

[예제 1-10] range based for문에서 STL 컨테이너 사용 (예제 파일: range_02)

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

int main()
{
   std::cout << "range based 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 based 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 1-10] 실행 결과

```
range based for - vector
1
2
3

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

기본적으로 STL의 '이터레이터Iterator'[02]를 지원하는 컨테이너라면 range based

02 배열, 리스트, 큐 등의 다양한 컨테이너에서 같은 연산으로 동일하게 동작하도록 한다.

for 문을 문제없이 사용할 수 있다. 그러므로 프로그래머가 자신만의 컨테이너를 만들 때 STL에서 정의한 이터레이터의 기능을 구현하면 range based for문을 사용할 수 있다.

range based 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)
```

만약 요소의 값을 변경하고 싶다면 참조를 사용한다.

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

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

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

또한, for문에서 데이터셋 요소에 접근할 때 임시 변수를 만드므로 이 비용을 줄이고 싶다면 참조를 사용하는 게 좋다. 만약 요소 값을 변경하지 못하게 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 1-11] 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++11에서는 C++03 표준과 달리 'unscoped enumeration'과 'scoped enumeration' 두 종류의 enum으로 바뀌었다.

1.4.1 unscoped enumeration

unscoped enumeration은 기존(C++03) enum과 비슷하다. unscoped enumeration은 다음과 같이 정의한다.

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

사용 방법은 다음과 같다.

```
short ItemType = WEAPON;
```

또는 다음과 같은 방법으로 사용해도 된다.

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

1.4.2 scoped enumeration

scoped enumeration은 다음과 같이 정의한다.

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

사용 방법은 다음과 같다.

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

다만, 다음과 같이 사용하면 에러가 발생한다.

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

scoped enumeration은 unscoped enumeration과 다르게 CHARACTER_CLASS를 생략할 수 없다. 즉, WARRIOR나 MONK는 CHARACTER_CLASS의 범위 안에 있음을 가리킨다. 그리고 'enum class' 대신 'enum struct'을 사용해도 괜찮다. 또한, 형을 지정하지 않으면 기본적으로 'int'형이 된다.

1.4.3 형 변환

unscoped enumeration은 기존과 같이 암묵적으로 정수로 변환할 수 있다.

```
int i = WEAPON;
```

그러나 scoped enumeration은 명시적으로 '타입캐스팅Typecasting'을 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
{
  WARRIOR = 1,
  MONK,
  FIGHTER,
};

enum struct BATTLE_TYPE : short
{
  DEATH_MATCH = 1,
  TEAM,
};

int main()
{
  // unscoped enumeration
  std::cout << "ITEM WEAPON의 Type 번호 : " << ITEMTYPE::WEAPON << std::endl;

  short ItemType = EQUIPMENT;
  std::cout << "ITEM EQUIPMENT의 Type 번호 : " << ItemType << std::endl;

  // scoped enumerations
  short CharClassType3 = (short)CHARACTER_CLASS::FIGHTER;

  CHARACTER_CLASS CharClass = CHARACTER_CLASS::WARRIOR;

  // short   CharClassType = FIGHTER;                    // 에러
  // short   CharClassType2 = CHARACTER_CLASS::FIGHTER;  // 에러
```

```
    // CHARACTER_CLASS CharClass2  = WARRIOR;                  // 에러

    return 0;
}
```

[예제 1-12] 실행 결과

```
ITEM WEAPON의 Type 번호 : 0
ITEM EQUIPMENT의 Type 번호 : 1
```

1.5 nullptr

nullptr은 C++11에서 추가된 키워드로, '널 포인터Null Pointer'를 뜻한다.

1.5.1 nullptr이 필요한 이유

C++03까지는 널 포인터를 나타내기 위해 NULL 매크로나 상수 0을 사용했다. 그러나 NULL 매크로나 상수 0을 사용하여 함수에 인자로 넘기면 int형으로 추론되어버리는 문제가 발생하기도 한다.

[예제 1-13] 함수 인자 추론 문제 (예제 파일: nullptr_01)

```
#include <iostream>

using namespace std;

void func(int a)
{
    cout << "func - int " << endl;
}

void func(double *p)
{
    cout << "func - double * " << endl;
```

```
}

int main()
{
  func(static_cast<double*>(0));
  func(0);
  func(NULL);

  return 0;
}
```

[예제 1-13] 실행 결과

```
func - double *
func - int
func - int
```

첫 번째 func 호출에서는 'double*'로 캐스팅해서 의도했던 func이 호출되었다. 그러나 두 번째와 세 번째 func 호출은 'func(doube* p)' 함수에 널 포인터를 인자로 넘기려고 했지만, 의도하지 않게 컴파일러는 int로 추론하여 'func(int a)'가 호출되었다.

바로 이와 같은 문제를 해결하기 위해서 'nullptr'이라는 키워드가 생겼다.

1.5.2 nullptr 사용 방법

사용 방법은 아주 간단하다. 예전에 널 포인터로 '0'이나 'NULL'을 사용하던 것을 'nullptr'로 바꾸기만 하면 된다.

```
char* p = nullptr;
```

예제 1-13에서 널 포인트를 인자로 넘겨서 'func(double* p)'를 호출하려면 다음

과 같이 한다.

```
func(nullptr);
```

1.5.3 nullptr의 올바른 사용과 잘못된 사용 예

nullptr의 올바른 사용 예

```
char* ch = nullptr; // ch에 널 포인터를 대입한다.
sizeof(nullptr);    // 사용할 수 있다. 참고로 크기는 4다.
typeid(nullptr);    // 사용할 수 있다.
throw nullptr;      // 사용할 수 있다.
```

nullptr의 잘못된 사용 예

```
int n = nullptr; // int에는 숫자만 대입할 수 있는데 nullptr은 클래스이므로 안 된다.

int n2 = 0
if(n2 == nullptr);   // 에러

if(nullptr);         // 에러
if(nullptr == 0);    // 에러
nullptr = 0;         // 에러
nullptr + 2;         // 에러
```

2 | 메모리 관리

C/C++는 동적으로 할당한 메모리를 사용한 후 해제하지 않으면 '메모리 누수 Memory Leak'[01]가 발생하여 시간이 지난 후 프로그램 실행에 큰 문제를 일으킨다. 작은 프로그램에서는 프로그래머가 실수 없이 할당과 해제를 잘하므로 메모리 누수가 거의 발생하지 않는다. 하지만 동적 메모리 할당을 자주 사용하고 코드가 복잡한 대형 프로젝트에서는 필연적으로 발생한다(PC 온라인 게임의 클라이언트 프로그램은 메모리 누수 검출 라이브러리나 유틸리티 사용이 필수다).

그래서 C++ 이후에 나온 언어인 Java나 C#을 C++와 비교할 때, C++에 비해 장점으로 꼽는 것이 메모리 관리를 자동으로 한다는 점이다. 그러나 시스템에서 메모리 관리를 하는 것이 꼭 좋은 것만은 아니다. Java나 C#은 성능에 조그마한 문제가 발생해도 메모리 해제를 위한 가비지 컬렉션Garbage Collection, GC이 작동한다. CG는 당연히 리소스를 소비하므로 너무 자주 작동하면 성능상 좋지 않다.

C++로 만든 대부분의 프로그램은 성능을 중요시하기 때문에 아직 GC를 사용하기에는 적당하지 않다. 그렇다고 메모리 관리의 어려움을 계속 프로그래머에게만 맡겨놓을 수는 없다.

이런 문제를 해결하기 위해 C++11에서는 사용하지 않는 메모리를 자동으로 해제시켜 주는 'shared_ptr'와 'unique_ptr'라는 라이브러리가 새로 생겼다. 동적으로 할당한 메모리 해제를 자동으로 관리해주는 것을 '스마트 포인터Smart Pointer'라고 하는데, shared_ptr와 unique_ptr가 이에 해당한다. 사실 C++03에도 같은 기능을 하는 'auto_ptr'라는 것이 있지만, 제약이 있어서 거의 사용하지 않는다. shared_

01 컴퓨터 프로그램이 필요하지 않은 메모리를 계속 점유하고 있는 현상이다. 할당된 메모리를 사용한 다음 반환하지 않는 것이 누적되면 메모리가 낭비된다. [출처: 위키백과]

ptr와 unique_ptr는 auto_ptr의 문제를 해결한 스마트 포인터다.

스마트 포인터를 사용하면 혹시 성능에 좋지 않은 영향을 줄까 걱정하는 사람들도 있을 것이다. 그러나 성능 측정을 해보면 기존대로 그냥 메모리를 사용할 때와 비교해서 별로 차이가 나지 않는다. 즉, 스마트 포인터를 사용하면 성능 측면에서는 손해가 거의 없으면서도 메모리 관리의 편리함을 얻을 수 있으므로 C++11에서는 이것을 자주 사용하기 바란다.

여기서 잠깐_ auto_ptr 문제

반만 스마트한 포인터 auto_ptr

http://psychoria.blog.me/40155382971

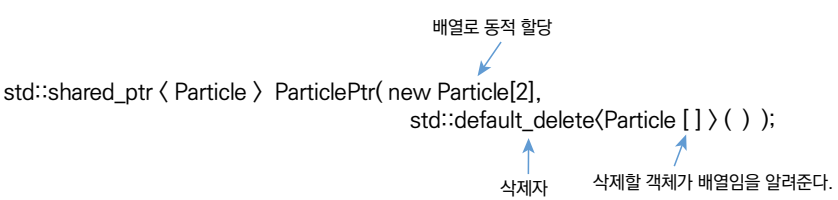
2.1 shared_ptr

2.1.1 shared_ptr 사용하기

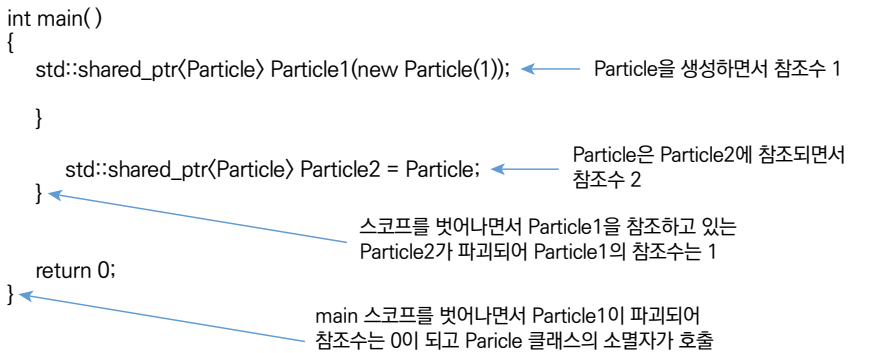
필요한 헤더 파일 선언

```
#include <memory>
```

shared_ptr 정의



shared_ptr의 메모리 관리(shared_ptr 참조수 증가와 감소)



shared_ptr은 생성될 때 참조수가 1이 되고 이 객체를 다른 곳에서 참조할 때마다 참조수를 1씩 증가시킨다. 그리고 소멸자가 호출될 때마다 참조 카운트를 -1씩 감소시켜서 참조수가 0이 될 때, 관리하고 있는 객체를 사용하지 않는 것으로 판단하고 파괴한다.

간단한 예제를 통해서 shared_ptr이 어떻게 동작하는지 살펴보자.

[예제 2-1] shared_ptr 사용 (예제 파일: shared_ptr_01)

```
#include <iostream>
#include <memory>

class Particle
{
public:
  Particle(int nID)
  {
     m_nID = nID;
     std::cout << "Particle " << m_nID << "번 생성. " << std::endl;
  }
```

```
    ~Particle()
    {
        std::cout << "Particle " << m_nID << "번 소멸. " << std::endl;
    }

    int m_nID;
};

int main()
{
    std::shared_ptr<Particle> Particle1(new Particle(1)); // 참조수 1
    std::cout << "Particle1의 참조수: " << Particle1.use_count() << std::endl;

    {
        std::shared_ptr<Particle> Particle2 = Particle1;    // 참조수 2
        std::cout << "Particle1의 참조수: " << Particle1.use_count() << std::endl;
        std::cout << "Particle2의 참조수: " << Particle2.use_count() << std::endl;
    }

    // Particle2는 스코프를 벗어나면서 파괴되어 Particle1의 참조수는 1
    std::cout << "Particle1의 참조수: " << Particle1.use_count() << std::endl;

    // Particle1가 파괴되어 참조수는 0이 되면서 자동으로 메모리 해제
    return 0;
}
```

[예제 2-1] 실행 결과

```
Particle 1번 생성.
Particle1의 참조수: 1
Particle1의 참조수: 2
Particle2의 참조수: 2
Particle1의 참조수: 1
Particle 1번 소멸.
```

2.1.2 shared_ptr로 관리하는 객체 사용

shared_ptr로 관리하는 객체의 포인터를 얻을 때는 'get()', 참조를 얻을 때는 'operator *', 인스턴스의 멤버에 접근할 때는 'operator ->'를 사용한다.

[예제 2-2] shared_ptr이 관리하는 인스턴스 사용하기 (예제 파일: shared_ptr_02)

```
#include <iostream>
#include <memory>

class Particle
{
public:
  Particle(int nID)
  {
     m_nID = nID;
     std::cout << "Particle " << m_nID << "번 생성. " << std::endl;
  }

  ~Particle()
  {
     std::cout << "Particle " << m_nID << "번 소멸. " << std::endl;
  }

  int m_nID;
};

Particle* GetParticlePointer(std::shared_ptr<Particle>& ParticlePtr)
{
  return ParticlePtr.get();
}

int main()
{
  std::shared_ptr<Particle> ParticlePtr(new Particle(10));
```

```
    // Particle 인스턴스의 참조를 얻어서 직접 멤버 변수 사용
    Particle& ParticleRef = *ParticlePtr;
    std::cout << "ParticleRef의 ID: " << ParticleRef.m_nID << std::endl;

    // Particle 인스턴스의 포인터를 얻어서 멤버 변수 사용
    Particle* pParticle = GetParticlePointer(ParticlePtr);
    std::cout << "pParticle의 ID: " << pParticle->m_nID << std::endl;

    // Particle1을 통해서 Particle 인스턴스의 멤버 접근
    std::cout << "ParticlePtr의 ID: " << ParticlePtr->m_nID << std::endl;

    return 0;
}
```

[예제 2-2] 실행 결과

```
Particle 10번 생성.
ParticleRef의 ID: 10
pParticle의 ID: 10
ParticlePtr의 ID: 10
Particle 10번 소멸.
```

2.1.3 명시적으로 관리하는 객체 삭제 및 다른 객체로 바꾸기

shared_ptr의 reset() 멤버 함수를 사용하면 기존에 관리하던 객체를 다른 객체로 교체할 수 있다(이때 기존 객체는 파괴된다). 또한, reset() 멤버 함수에 인자값을 사용하지 않으면 명시적으로 객체를 파괴할 수 있다.

[예제 2-3] reset()으로 관리하는 객체 삭제 및 바꾸기 (예제 파일: shared_ptr_03)

```
#include <iostream>
#include <memory>

class Particle
```

```
{
public:
    Particle(int nID)
    {
        m_nID = nID;
        std::cout << "Particle " << m_nID << "번 생성. " << std::endl;
    }

    ~Particle()
    {
        std::cout << "Particle " << m_nID << "번 소멸. " << std::endl;
    }

    int m_nID;
};

int main()
{
    std::shared_ptr<Particle> ParticlePtr(new Particle(10));
    std::cout << "ParticlePtr의 ID: " << ParticlePtr->m_nID << std::endl;

    std::cout << "reset - 다른 인스턴스로 교체" << std::endl;
    ParticlePtr.reset(new Particle(11));
    std::cout << "ParticlePtr의 ID: " << ParticlePtr->m_nID << std::endl;

    std::cout << "reset - 인스턴스 삭제" << std::endl;
    ParticlePtr.reset();
    std::cout << "ParticlePtr의 참조수: " << ParticlePtr.use_count() << std::endl;

    return 0;
}
```

[예제 2-3] 실행 결과

```
Particle 10번 생성.
ParticlePtr의 ID: 10
reset - 다른 인스턴스로 교체
Particle 11번 생성.
Particle 10번 소멸.
ParticlePtr의 ID: 11
reset - 인스턴스 삭제
Particle 11번 소멸.
ParticlePtr의 참조수: 0
```

2.1.4 두 개의 shared_ptr끼리 서로의 객체 교환하기

STL의 vector 같은 컨테이너들은 swap을 사용하여 가지고 있는 요소를 교환하는데, shared_ptr 역시 swap 기능을 지원한다. 간단한 예제를 통해서 swap 기능을 어떻게 사용하는지 살펴보자.

[예제 2-4] swap()으로 교환하기 (예제 파일: shared_ptr_04)

```
#include <iostream>
#include <memory>

class Particle
{
public:
  Particle(int nID) { m_nID = nID; }

  ~Particle() {}

  int m_nID;
};

int main()
{
```

```
    std::shared_ptr<Particle> ParticlePtr1(new Particle(10));
    std::cout << "ParticlePtr1의 ID: " << ParticlePtr1->m_nID << std::endl;

    std::shared_ptr<Particle> ParticlePtr2(new Particle(20));
    std::cout << "ParticlePtr2의 ID: " << ParticlePtr2->m_nID << std::endl;

    std::cout << "swap 사용" << std::endl;
    ParticlePtr1.swap(ParticlePtr2);

    std::cout << "ParticlePtr1의 ID: " << ParticlePtr1->m_nID << std::endl;
    std::cout << "ParticlePtr2의 ID: " << ParticlePtr2->m_nID << std::endl;

    return 0;
}
```

[예제 2-4] 실행 결과

```
ParticlePtr1의 ID: 10
ParticlePtr2의 ID: 20
swap 사용
ParticlePtr1의 ID: 20
ParticlePtr2의 ID: 10
```

2.1.5 배열 객체 다루기

많은 객체나 데이터를 다루다 보면 배열식으로 동적 할당을 해야 할 때가 자주 있다. shared_ptr도 당연히 배열로 할당한 것을 다룰 수 있다. 방법은 매우 간단하다. shared_ptr을 생성할 때 삭제자Deleter를 사용하여 배열임을 알려주면 된다.

[그림 2-1] 배열 객체 사용