

Microsoft®



Microsoft®
Visual Studio®

Visual C++ 10과 C++0x

Contents

0. 서두	5
0.1 C++의 문제점	5
0.2 C++는 시대에 뒤 떨어진 프로그래밍 언어일까요?	6
1. C++0x에 대해서	7
1.1 C++의 새로운 표준 C++0x	7
1.2 Visual C++의 이전 버전과의 호환성	7
1.3 Visual C++ 9와 Visual C++ 10	8
2. auto	9
2.1 정적 언어와 동적 언어의 차이	9
2.2 C#의 var	9
2.3 C++에서 STL(표준 템플릿 라이브러리)를 사용할 때 불편한 점	10
2.4 컴파일 타임 때 타입을 정하는 'auto'	10
2.5 auto를 사용한 예제	11
2.5.1 지역 변수로 사용	11
2.5.2 클래스 정의에 사용	12
2.5.3 STL에서 사용	12
2.6 핵심 요약	14
3. static_assert	15
3.1 assert와 #error	15
3.2 assert와 #error를 사용할 수 없을 때	15
3.3 static_assert의 형식	15
3.4 static_assert 사용 예	16
3.4.1 프리프로세서 지시어 대체	16
3.4.2 템플릿에서 사용	16
3.4.3 변수의 크기 조사	17
3.5 핵심 요약	17
4. 우측 값 참조 (RValue Reference)	18
4.1 C++98/03에서의 실수	18
4.2 좌측 값(LValue)과 우측 값(RValue)	19

Contents

4.3 좌측 값 참조와 우측 값 참조	20
4.4 Move semantics	21
4.5 Move 생성자와 Move 대입 연산자	22
4.6 Move semantics에 의한 성능 향상	23
4.7 move 생성자와 move 대입 연산자 사용 예	24
4.8 std::move	25
4.9 VC 10에서의 STL과 우측 값 참조	26
4.10 우측 값 참조를 사용할 때 주의할 점	28
4.11 우측 값 참조와 좌측 값 참조의 함수 오버로드	29
4.12 우측 값 참조는 우측 값이 아니다	30
4.13 Perfect Forwarding	30
4.14 핵심 요약	34
5. 람다 (lambda)	35
5.1 C#과 람다	35
5.2 C++에서 STL 알고리즘을 사용할 때 불편했던 점	36
5.3 C++에서의 람다 사용 법	37
5.3.1 문법	37
5.3.2 람다를 변수에 대입	37
5.3.3 람다를 함수의 인자로 사용하기	38
5.3.4 람다의 파라미터	38
5.3.5 반환 값 넘기기	39
5.4 캡처	39
5.5 클래스에서 람다 사용	43
5.6 STL의 find_if에서 람다 사용	44
5.7 람다 식을 STL 컨테이너에 저장	45
5.8 람다를 반환하는 함수	46
5.9 람다에서의 재귀	46
5.10 핵심 요약	47
6. decltype	48
6.1 템플릿 프로그래밍 시 문제	48
6.2 decltype 사용하기	48
7. nullptr	49
7.1 nullptr이 필요한 이유	49
7.2 nullptr 구현 안	50
7.3 nullptr 사용 방법	50
7.4 nullptr의 올바른 사용과 틀린 사용 예	50
8. 참고	52

0. 서두

0.1 C++의 문제점

C++가 처음 나왔을 때는 다양한 하드웨어로의 이식성이 높고, 하드웨어를 잘 활용하여 고성능 프로그램을 만들 수 있으며 객체지향 프로그래밍을 할 수 있는 구조와 표준 템플릿 라이브러리(STL) 덕분에 C 언어 보다 더 좋은 생산성을 얻을 수 있어서 단숨에 C 언어를 제치고 프로그래머에게 가장 인기 있는 프로그래밍 언어가 되었습니다. 그러나 지금은 다른 언어들에게 최고의 자리를 내어 주었습니다.

C++ 언어가 나왔을 때는 하드웨어 성능이 지금보다 빈약한 시기로 프로그램이 작은 메모리와 낮은 CPU 점유율을 가지도록 성능에 대해 지금보다 더 많이 신경을 쓰면서 프로그램을 만들었지만, 지금은 멀티 코어 CPU와 기가 바이트 단위의 메모리를 가진 컴퓨터가 일반화된 시대로 바뀌어 생산성을 더 중요시 하고 있습니다.

C++가 예전만큼의 인기를 얻지 못하는 것은 근래에 나온 언어에 비해서 사용하기가 쉽지 않고 생산성이 떨어지기 때문이라고 생각합니다.

“왜 C++이 어려울까요?”라고 트위터에 질문을 올려보았습니다. 이 질문에

@whatthepaul님은

“사용자가 알아야 할 것이 너무 많아서가 아닐까요? 그만큼 속도에서는 이득이 있는 거겠지만 생산성은 떨어지겠죠. 점점 생산성이 중요해지는 추세인데 좀 반대에 있는 느낌이지 -_-”,

@birdkr님은

“문법이나 포인터 개념도 어렵지만, 저는 지원해주는 기본 라이브러리가 턱없이 부족한 것도 한 원인이라 생각합니다. 요즘 인기 있는 언어들처럼 뭔가 똑딱똑딱하면 그럴 듯 한 것이 나와야 하는데 C++은 간단한 것도 쉽게 만들 수 없으니까요.”

라는 의견을 주셨습니다. 두 분의 의견은 제 주위의 C++ 개발자들이나 인터넷의 개발자 커뮤니티나 블로그 등에 있는 C++에 대한 불만과 비슷합니다.

처음 C++이 만들어질 때에 비해서 지금은 세상도 많이 변했고, 컴퓨터 하드웨어 성능도 훨씬 더 좋아졌습니다. 이전 C++도 변화해야 할 때입니다. 그리고 다른 프로그래밍 언어들처럼 생산성을 향상 시켜야 합니다.

0.2 C++는 시대에 뒤 떨어진 프로그래밍 언어일까요?

요즘은 컴퓨터 하드웨어 성능이 예전에 비해서 훨씬 좋아져서 프로그램을 만들 때 성능보다는 높은 품질로 빨리 만드는 것이 중요하다고 했습니다. 그러나 정말 성능을 신경 쓰지 않아도 될까요?

어떤 프로그램이나에 따라서 답변이 틀릴 것이라고 생각합니다. 예전에 비하면 성능에 크게 신경 쓰지 않아도 괜찮은 분야가 더 늘었지만 아직도 몇몇 분야에서는 성능에 신경을 써야 합니다. 제가 알고 있는 분야 중 임베디드와 게임은 아직도 성능에 신경을 써야 하는 분야입니다.

특히 제가 일하고 있는 게임 업계는 언제나 하드웨어의 발전 속도보다 게이머들이 이전보다 더 좋은 고화질의 그래픽, 더 사람 같은 AI를 원하고 있기 때문에 프로그래머들은 언제나 성능에 많은 신경을 써야 합니다. 또 HD급의 그래픽을 사용하는 게임이 아니라면 낮은 사양의 CPU와 작은 메모리를 가진 하드웨어에서도 게임을 실행되게 하기 위해 최적화에 많은 시간을 들입니다.

아직은 C#이나 Java로는 C++과 같은 높은 성능을 필요로 하는 게임을 만들기가 어렵기 때문에 주류급 게임을 만들 때는 사용할 수 없습니다.

아무리 하드웨어 사양이 높아지고 있더라도 하드웨어를 최대한 활용하여 높은 성능을 필요로 하는 프로그램을 만들어야 하는 분야는 여전히 있습니다. 이런 곳에서는 C#이나 Java로는 턱 없이 부족합니다. C++가 적격입니다.

C++의 필요성은 충분히 알고 있지만 C++의 학습의 어려움, 낮은 생산성 등 부족한 부분을 빨리 누군가 메꾸워 주기를 바라는 분들이 많이 있으리라 생각합니다. 바로 이런 부족한 부분을 메꾸기 위해서 C++의 새로운 표준이 현재 만들어지고 있습니다. 그리고 기쁜 소식은 표준 작업이 끝나기 이전에 새로운 C++의 기능을 새로 나올 Visual C++을 통해서 사용할 수 있습니다

1. C++0x에 대해서

1.1 C++의 새로운 표준 C++0x

C++은 처음 표준을 만들 때 다양 사람들의 의견을 반영하여 완성도 있게 만들어졌고, 이미 C++로 만들어진 프로그램이 많이 있어서 C#이나 Java처럼 새로운 표준이 자주 만들어지지 않아서 어떤 분들은 C++은 이제 과거의 언어로 더 이상 발전하지 않는 언어라고 생각하시는 분도 있으리라 생각합니다. C++은 결코 과거에 머물러 있는 언어가 아닙니다. 그 증거로 몇 년전부터 새로운 C++의 새로운 표준이 만들어지고 있습니다.

새로운 표준이 될 개정안을 임시적으로 C++0x라고 부르고 있습니다(C++0x 이전에는 C++98과 C++03 라고 부르는 표준안이 있었습니다).

표준 위원회는 2009년 안에 표준을 확정하기 위해서 2006년까지 받았던 제안을 중심으로 표준 작업을 하고 있다고 합니다. 하지만 C++0x라고 가치를 붙이면서까지 2009년 안에 표준 작업을 끝내려고 했지만 결과적으로 2010년을 넘겼습니다. 2009년을 넘겼으므로 C++0x라는 가치의 이름도 바뀌어야 하지만 가치를 바꾸면 혼란이 생길 수 있으므로 C++0x라는 가치를 그대로 사용하기로 했습니다.

C++0x는 코어 언어의 기능 추가와 표준 C++ 라이브러리(STL)를 확장이라는 크게 두 개의 분류로 나누어서 작업을 하고 있습니다.

1.2 Visual C++의 이전 버전과의 호환성

C++의 창시자이자 C++ 표준 위원회의 멤버인 Bjarne Stroustrup는 새로운 C++ 표준은 이전의 표준과 100% 호환성을 가진다고 합니다. 즉 이전에 만들었던 C++ 코드들이 표준을 준수했다면 새로운 표준에서 컴파일 하는데 아무런 문제가 없습니다.

예전에 Visual C++ 6(이하 VC6)에서 Visual C++ 7(이하 VC 7)로 넘어갈 때 VC 6에서 만들었던 코드가 VC 7에서는 에러와 경고를 내서 순조롭게 넘어가지 못한 경우가 있었습니다. 지금도 이것 때문에 아직도 VC 6을 사용하고 있는 곳도 있는 것으로 알고 있습니다.

VC 6에서 만들었던 코드가 VC 7에서 문제가 된 이유는 VC 6는 C++98 표준이 확립되기 전에 나와서 VC 6에서 만들었던 코드 중에는 표준에 맞지 않는 코드가 있기 때문입니다. 그러나 VC 7은 표준을 거의 완벽하게 준수하고 있어서 VC 6에서 만들었던 코드 중 표준에 맞지 않는 코드는 에러나 경고를 발생시킵니다.

VC 7부터는 표준을 거의 100% 준수하고 있으므로(참고로 100% C++ 표준을 준수한 상업용 컴파일러는 거의 없습니다) VC 7에서부터 만들었던 코드는 Visual C++ 10(이하 VC 10)에도 문제 없이 컴파일 할 수 있습니다.

즉 VC 7, VC 8, VC 9에서 컴파일에 문제가 없는 코드라면 VC 10에서도 수정 없이 그대로 사용할 수 있습니다.

1.3 Visual C++ 9와 Visual C++ 10

C++ 표준위원회에서 C++0x에서 정책적으로 라이브러리 부분을 언어적인 부분보다 우선 시 하자고 정했기 때문에 `tr1`이라는 라이브러리가 VC 9가 나올 무렵에 나와서 VC 9가 출시 된 이후 Visual Studio Service Pack을 통해서 VC 9에서 사용할 수 있게 되었습니다.

VC 10에서는 VC 9에 추가된 `tr1`과 달리 언어적인 부분에서의 새로운 기능이 구현 되었습니다.

언어적인 부분의 개선에 의해서 C++ 언어 표현력이 증대되었고 성능적 측면에서도 개선이 이루어졌습니다.

VC 10에 구현된 C++0x의 새로운 기능은 VC 9의 `tr1`과 달리 언어적 측면에서의 새로운 기능이라서 새롭게 바뀐 C++을 이전에 비해 훨씬 더 강하게 체감할 수 있습니다.

VC 10에 구현된 C++0x의 새로운 기능은 `auto`, `static_assert`, RValue Reference, `lambda`, `decltype`, `nullptr`입니다.

다음 장부터는 새로운 기능들에 대해서 설명할 테니 이전 보다 더 편리하고 강력해진 C++의 새로운 기능을 직접 느껴보시기 바랍니다. 보통의 C++ 프로그래머가 아닌 새로운 C++ 표준을 사용하는 C++0x 프로그래머가 되시기 바랍니다.

2. auto

2.1 정적 언어와 동적 언어의 차이

근래에 Ruby나 Python과 같은 스크립트 언어가 많은 인기를 얻고 있습니다. Ruby나 Python 같은 언어를 '동적 언어'라고 합니다. 반대로 제가 주로 사용하는 언어인 C++는 '정적 언어'라고 합니다.

정적 언어와 동적 언어는 여러가지 차이점이 있습니다. 그중 정적 언어는 변수 type을 선언이나 정의 할 때 명시적으로 지정해야 합니다. 그러나 동적 언어인 Ruby, Python은 변수 type을 명시적으로 지정할 필요가 없어서 편리합니다.

〈 코드 2-1. C/C++에서의 지역 변수 정의 〉

```

<코드>
void BuyItem()
{
    int Money = 500;
    .....
}
</코드>

```

〈 코드 2-2. Ruby에서의 지역 변수 정의 〉

```

<코드>
def BuyItem
    Money = 500;
    .....
end
</코드>

```

2.2 C#의 var

C#은 C++과 같은 정적 언어이지만 var라는 키워드를 사용하여 변수 type을 명시적으로 지정하지 않아도 됩니다. 동적 언어와 차이점은 변수의 type을 실행할 때가 아닌 컴파일 할 때 결정합니다.

C#의 var 키워드를 사용하면 코딩 시 번거롭거나 코드의 가독성을 나쁘게 하는 코드를 없애 수 있어서 아주 유용합니다. var는 특히 C#에서 LINQ를 사용할 때 자주 사용합니다.

〈 코드 2-3. LINQ에서 var를 사용할 때와 사용하지 않을 때 비교〉

```

<코드>
// LINQ에서 var를 사용하지 않는 경우
IEnumerable<IGrouping<string, Student>> studentQuery3 =
    from student in students group student by student.Last;

```

```
// LINQ에서 var를 사용한 경우
var studentQuery3 = from student in students group student by student.Last;
</코드>
```

<코드 2-3>에서 var 키워드를 사용하여 LINQ를 사용하면 코딩 해야 할 량도 줄어 들고 코드 가독성도 var를 사용하지 않을 때에 비해 훨씬 더 좋아진 것을 알 수 있습니다.

2.3 C++에서 STL(표준 템플릿 라이브러리)를 사용할 때 불편한 점

C++에서 STL을 사용할 때 코딩이 불편할 때가 있습니다. 예를 들면 list와 같은 STL 컨테이너에서 컨테이너에 있는 모든 요소를 순회할 때입니다.

< 코드 2-4. C++에서 list 사용 >

```
<코드>
list<int> NumList;
NumList.push_back( 10 );

for( list<int>::iterator iter = NumList.begin();
    iter != NumList.end();
    ++iter )
{
    .....
}
</코드>
```

list 컨테이너의 첫 번째 요소를 얻기 위해서 반복자를 정의할 때 꽤 길게 코딩 해야 됩니다(map과 같은 연관 컨테이너는 훨씬 더 깁니다). 위의 경우는 그나마 좀 짧은 편이고 템플릿을 사용한 클래스를 list의 type으로 사용하는 경우는 아주 길어집니다.

<코드 2-4>를 보면 앞선 <코드 2-3>에서 var 키워드를 사용하지 않을 때와 비슷한 불편함이 있다는 것을 알 수 있습니다. 만약 C++에도 C#과 같이 var 라는 것이 있으면 이 문제를 쉽게 해결할 수 있겠죠?

C++0x에서는 바로 var와 같은 것을 만들어서 <코드 2-4>과 같은 불편함을 해결 하였습니다. 해결 방법은 C++0x에서 새로 생긴 'auto' 라는 키워드를 사용하는 것입니다.

2.4 컴파일 타임 때 타입을 정하는 'auto'

auto를 사용하면 변수를 정의할 때 명시적으로 type을 지정하지 않아도 됩니다. auto로 정의한 변수를 초기화할 때 type을 결정합니다. 즉 C#의 var와 같이 컴파일 타임 때 변수의 type을 결정합니다. 하지만 클래스의 멤버 변수나 전역변수, 함수의 인자로는 auto를 사용할 수는 없습니다.

auto가 복잡한 개념은 아니지만 변수를 정의할 때는 언제나 type을 지정해야 한다고 아주 당연하게 생각하는 C++ 프로그래머로서는 조금 과장을 해서 auto는 아주 획기적이라고 느끼시는 분들도 있으리라 생각합니다(저는 C++0x를 공부하기 전에는 C++에서는 절대 auto 같은 기능은 지원하지 않고 이런 것은 C#과 같은 근래의 언어나 스크립트 언어에서만 가능한 것이라고 생각하고 있었습니다).

auto 키워드는 C#의 var와 비슷하고, 개념이나 사용 방법이 아주 간단합니다. '변수를 정의할 때 명시적으로 type을 지정하지 않고 컴파일 타임 때 결정' 하게 해주는 키워드라고 기억하시면 됩니다.

그럼 auto를 어떻게 사용하는지 예제 코드를 보겠습니다.

2.5 auto를 사용한 예제

2.5.1 지역 변수로 사용

문자열과 정수를 담은 변수를 auto를 사용하여 정의하면 아래와 같습니다.

〈 코드 2-5. 문자열과 정수 변수에 auto 사용 〉

〈코드〉

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

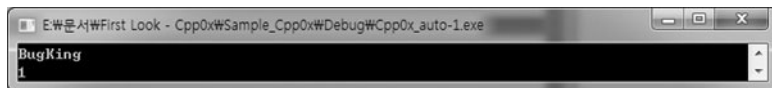
int main()
{
    // char*
    auto NPCName = "BugKing";
    cout << NPCName << endl;

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

    getchar();
    return 0;
}
```

〈/코드〉

〈 결과 2-5 〉



또 당연히 포인터나 참조, const도 사용할 수 있습니다.

〈 코드 2-6. 포인터, 참조, const에 사용 〉

〈코드〉

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

int main()
{
    int UserMode = 4;
    auto* pUserMode = &UserMode;
    cout << "pUserMode : Value - " << *pUserMode << ", address : " << pUserMode << endl;
}
```

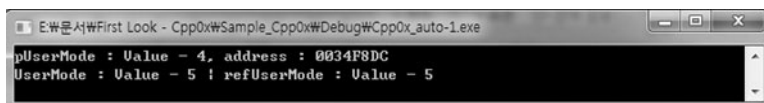
```

    auto& refUserMode = UserMode;
    refUserMode = 5;
    cout << "UserMode : Value - " << UserMode << " | refUserMode : Value - " <<
    refUserMode << endl;

    getchar();
    return 0;
}
</코드>

```

< 결과 2-6 >



2.5.2 클래스 정의에 사용

클래스를 정의할 때도 사용할 수 있습니다.

< 코드 2-7. 클래스 생성 때 auto 사용 >

```

<코드>
struct CharacterInvenInfo
{
    int SlotNum;
    int ItemCode;
};

.....
auto* CharInven = new CharacterInvenInfo();
.....
</코드>

```

2.5.3 STL에서 사용

auto 키워드는 템플릿 프로그래밍이나 STL을 사용할 때 진가를 더 발휘합니다.

auto가 없을 때는 STL의 컨테이너를 사용할 때 반복자를 정의 하기 위해 길게 코딩을 하던가 또는 typedef를 사용하였지만 auto가 생겨서 이런 불편함이 없어졌습니다.

```

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

```

를 아래처럼 바꿀 수 있습니다.

<코드>

```
auto iter = m_listCommand.begin();
</코드>
```

아래의 예제를 보면 auto를 사용하여 STL 사용이 얼마나 간단해졌는지 쉽게 알 수 있습니다.

< 코드 2-8. vector의 요소를 순회할 때 auto 사용 >

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

struct Item
{
    int ItemCode;
    int Money;
    int SkillCode;
};

int main()
{
    cout << "Use vector Iterator - 1" << endl;
    vector< int > ItemCodeList;
    ItemCodeList.push_back( 20 );
    ItemCodeList.push_back( 30 );
    ItemCodeList.push_back( 40 );

    for( auto IterPos = ItemCodeList.begin(); IterPos != ItemCodeList.end(); ++IterPos)
    {
        cout << "ItemCode : " << *IterPos << endl;
    }
    cout << endl;


    cout << "Use vector Iterator - 2" << endl;
    vector< Item > ItemList;
    Item item1;    item1.ItemCode = 1;    item1.Money = 100;    item1.SkillCode = 0;
    Item item2;    item2.ItemCode = 2;    item2.Money = 200;    item2.SkillCode = 10;
    Item item3;    item3.ItemCode = 3;    item3.Money = 300;    item3.SkillCode = 0;

    ItemList.push_back( item1 );
    ItemList.push_back( item2 );
    ItemList.push_back( item3 );

    for( auto IterPos = ItemList.begin(); IterPos != ItemList.end(); ++IterPos)
    {
        cout << "ItemCode : " << IterPos->ItemCode << ", Money : " << IterPos->Money << endl;
    }
    cout << endl;

    getchar();
    return 0;
}
</코드>
```

〈 결과 2-8 〉



```

E:\문서\First Look - Cpp0x\Sample_Cpp0x\Debug\Cpp0x_auto-1.exe
Use vector Iterator - 1
ItemCode : 20
ItemCode : 30
ItemCode : 40

Use vector Iterator - 2
ItemCode : 1, Money : 100
ItemCode : 2, Money : 200
ItemCode : 3, Money : 300

```

auto는 정말 단순한 개념과 사용 방법이면서 바로 C++ 프로그래머의 작업에 많은 도움을 주는 기능입니다. 아마 모든 C++ 프로그래머들이 많이 애용하는 기능 중의 하나가 될 것 이라고 생각합니다.

2.6 핵심 요약

auto를 사용하면

1. 지역 변수를 정의 때 명시적으로 type을 지정하지 않아도 됨
2. 컴파일 타임 때 type을 결정
3. 코딩이 간편해지고, 코드 가독성이 좋아짐

3. static_assert

3.1 assert와 #error

C++로 프로그래밍할 때 버그나 에러가 발생할 위험이 있는 곳에 경고를 발생시켜 문제를 빨리 발견하기 위해서 assert 매크로나 #error 프리프로세서 지시어를 사용합니다. (프리프로세서는 #ifdef 등을 사용할 때를 말합니다).

assert와 #error 중 보통 assert를 자주 사용하며 assert는 논리적인 오류 찾기, 작업 결과 확인, 처리해야 할 오류 조건 테스트를 할 때 사용합니다.

assert는 프로그램이 실행 될 때 평가 됩니다(디버그 모드에서 사용합니다).

〈 코드 3-1. assert 사용 예 〉

```
</코드>
PLAYER* pPlayer;
.....
assert( NULL != pPlayer )
</코드>
```

3.2 assert와 #error를 사용할 수 없을 때

assert는 실행 시에 사용하고, #error는 프리프로세서에 사용하기 때문에 템플릿 실체화 시(컴파일 타임)에는 이것들을 사용할 수 없습니다. 버그의 발견은 프로그램이 실행될 때 발견하는 것보다 컴파일 할 때 발견하면 버그를 잡는데 소요되는 시간이 짧아집니다.

C++0x에서 새로 생긴 static_assert는 컴파일 할 때 평가됩니다.

static_assert는 특히 컴파일 타임에서 실체화할 템플릿의 전제 조건을 조사할 때 사용하면 좋습니다.

예를 들면 Stack이라는 클래스 템플릿을 정의할 때 템플릿 파라미터로 type과 크기를 사용할 때 크기가 일정 크기 이상일 때만 컴파일 되기를 바란다면 static_assert를 사용하면 딱 좋습니다.

3.3 static_assert의 형식

static_assert의 형식은 다음과 같습니다.

원형

`static_assert ("constant-expression", "error-message");`

파라미터

"constant-expression" - 검사할 조건 식

"error-message" - 조건이 false일 경우 출력할 error 메시지

결과

constant-expression가 false일 경우 컴파일러는 에러 메시지를 출력합니다.

static_assert는 다음과 같은 경우에 사용하면 유용합니다.

1. 기본 타입(int, long 등)이나 유저 정의 타입(class, struct 등으로 만든 타입)의 크기를 확인하고 싶을 때
2. 어떤 타입의 최대 크기를 넘어서는지 확인하고 싶을 때

3.4 static_assert 사용 예

3.4.1 프리프로세서 지시어 대체

〈 코드 3-2. 상수 값의 크기 조사 〉

〈코드〉

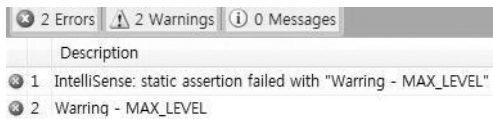
```
#include <iostream>
using namespace std;

const int MAX_LEVEL = 120;

int main()
{
    static_assert( MAX_LEVEL <= 100, "Warring - MAX_LEVEL" );
    return 0;
}
```

〈/코드〉

〈 그림 3-1. 〈코드 3-2〉의 컴파일 결과 〉



〈코드 3-2〉는 MAX_LEVEL의 값이 100을 넘으면 컴파일 할 때 에러를 출력합니다.

VC++ 10의 경우 편리한 IntelliSense가 컴파일 하기 전에 붉은 밑줄로 에러가 있음을 사전에 알려주기도 합니다.

〈 그림 3-2. 인텔리센스의 에러 통지 〉

```
int main()
{
    static_assert( MAX_LEVEL <= 100, "Warring - MAX_LEVEL" );
    return 0;
}
```

3.4.2 템플릿에서 사용

〈 코드 3-3. Stack 클래스 템플릿의 최소 스택 크기 조사 〉

〈코드〉

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



```

template< typename T1, int StackSize >
class MYSTACK
{
    static_assert( StackSize >= 10, "Stack Size Error" );
public :
    MYSTACK() : data( new T[StackSize] )
    {
    }

private:
    T1* data;
};

int main()
{
    MYSTACK< int, 5 > MyStack;
    return 0;
}
</코드>

```

〈 그림 3-3. 〈코드 3-3〉의 컴파일 결과 〉



Error List			
1 Error 2 Warnings 0 Messages			
	Description	File	Line Column
1	Stack Size Error	main.cpp	7 1

3.4.3 변수의 크기 조사

〈 코드 3-4. int 타입의 크기 조사 〉

```

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

int main()
{
    static_assert( sizeof(int) == 4, "not int size 4" );
    return 0;
}
</코드>

```

static_assert의 개념이나 사용 법이 간단하기 때문에 위의 예제 코드를 보면 어떻게 사용하고, 어디에 사용하면 좋을지 알 수 있으리라 생각합니다.

3.5 핵심 요약

1. assert와 비슷한 조건 조사를 할 수 있음
2. 컴파일 타임 때 사용하여 프로그램 실행 전에 문제를 찾을 수 있음
3. 템플릿 프로그래밍에 사용하면 특히 유용

4. 우측 값 참조 (RValue Reference)

4.1 C++98/03에서의 실수

C++98/03에서는 너무 과하다 싶을 정도로 추상화와 효율성을 같이 가져가려고 하다가 실수를 범했습니다. 이 실수는 추상화와 효율성의 조합을 위해 불필요한 복사를 초래 하였습니다.

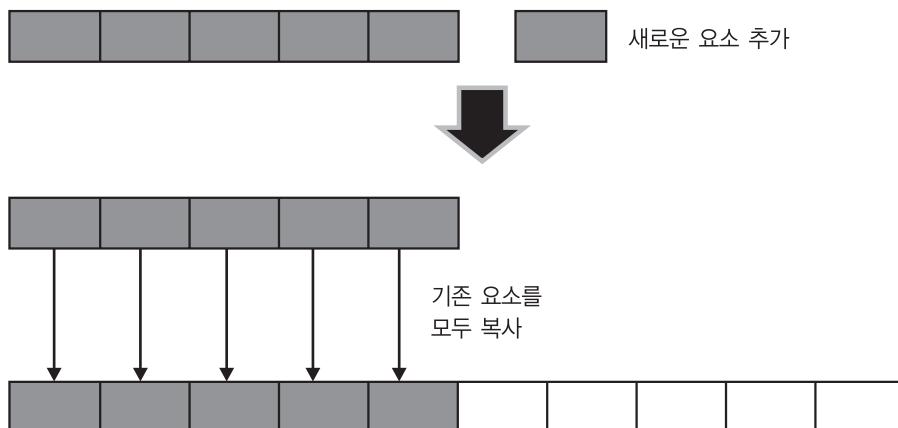
int와 같은 작은 것의 복사는 괜찮지만 중량급 오브젝트에서는 무시하기 힘듭니다. 다행히 RVO나 NRVO의 도움으로 불필요한 복사 생성자를 제외하여 어느 정도 문제를 경감 시켜주지만 이것 만으로는 모든 상황에서 과도한 복사 문제를 해결해 주지 못합니다.

의미 없는 복사

위에서 언급한 불필요한 복사의 예를 들면

- (a) vector에서 새로운 요소를 추가할 때 확장하는 경우,
- (a) string 객체간의 결합 등이 있습니다.

〈 그림 4-1. vector의 확장 〉

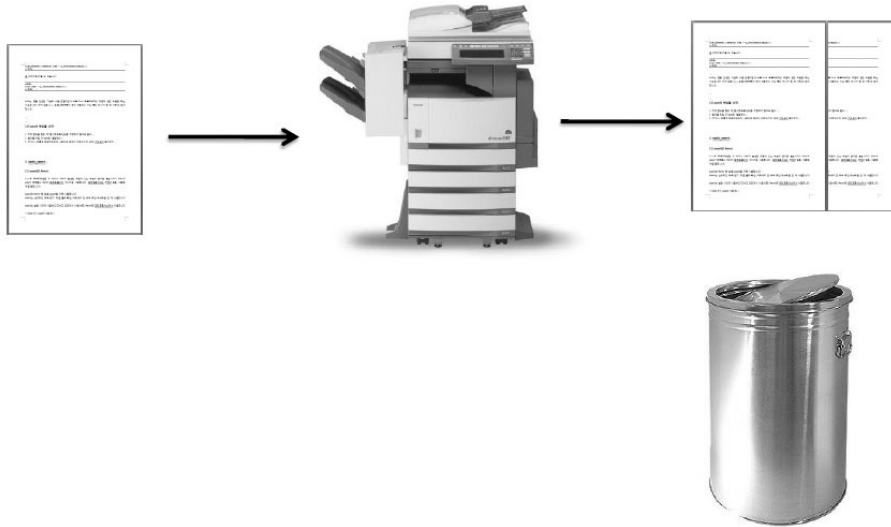


〈그림 4-1〉과 같이 vector를 확장할 때는 새로운 메모리 공간을 할당한 후 기존 요소를 하나씩 복사한 후 앞서 할당한 영역을 제거합니다. 이것은 영리한 방법은 아닙니다.

vector를 확장할 때 기존 요소를 그냥 그대로 사용할 수 있다면 기존 요소를 복사하지 않아도 되고, 기존에 사용하던 메모리 영역을 없애 필요도 없습니다. 바꾸어 말하면 “복사기로 문서 하나를 복사를 하는데 복사가 끝난 후 원본은 버린다”라는 경우와 비슷합니다. 원본을 버린다면 굳이 복사를 할 필요도 없습니다. 너무나 멍청한 짓입니다. 그런데 C++ 03까지는 위에서 예를 든 (a)와 (b)를 할 때 복사기로 복사를 하고 원본을 버리는 것과 같은 행동을 했습니다.

(참고로 (a)와 (b)에서는 의미 없는 복사 때문에 메모리 할당, 해제까지 덩(?)으로 합니다)

〈 그림 4-2. 원본을 버리는 복사 〉



C++0x에서는 이런 불필요한 복사를 방지하는 기능을 제공합니다. 그래서 이전보다 성능적인 측면에서의 개선이 이루어졌습니다. 불필요한 복사를 방지하여 성능 개선을 이루게 된 것은 “우측 값 참조 (RValue Reference)” 덕택입니다.

C++0x에서는 “우측 값 참조”에 의해서 복사가 아닌 메모리 상의 이동을 할 수 있어서 메모리 할당, 복사, 해제를 줄일 수 있어서 프로그램의 성능 향상을 얻을 수 있습니다.

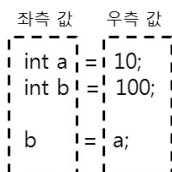
우측 값 참조가 간단한 개념은 아니라서 쉽게 이해가 안 될 수도 있지만 제가 처음부터 하나하나 자세하게 설명할테니 저의 설명을 지금부터 잘 따라오시면 쉽게 이해할 수 있습니다.

4.2 좌측 값(LValue)과 우측 값(RValue)

“우측 값 참조”를 간단하게 정의하면 우측 값의 참조라고 할 수 있습니다. 그래서 우측 값 참조를 알기 위해서는 필연적으로 “우측 값”이 무엇을 뜻하는지 알아야 합니다.

이 글을 보는 분들은 C++에서의 좌측 값과 우측 값에 대해서 어떻게 알고 계실지 궁금합니다. 아마 대부분 좌측 값은 “식의 왼쪽에 있는 것”, 우측 값은 “식의 오른쪽에 있는 것”이라고 생각하지 않을까 생각합니다. 이와 같이 우측 값과 좌측 값을 나누는 것은 C 언어라면 맞습니다.

〈 그림 4-3. C 언어에서의 좌측 값과 우측 값 〉



〈그림 4-3〉처럼 좌측 값과 우측 값을 나누는 것은 C 언어에서는 맞지만 C++에서는 틀립니다. C와 C++은 좌측 값과 우측 값 정의가 서로 다릅니다.

C++03의 사양서 3.10/1절에서 좌측 값과 우측 값에 대해서 “모든 식은 좌측 값 또는 우측 값이다. 중요한 것은 좌측 값 또는 우측 값이라고 말하는 성질은 식과 관련된 것으로 오브젝트에 대한 것이 아니다”라고 합니다.

예를 들면 *ptr, ptr[index], ++x 등은 모두 좌측 값입니다. 우측 값은 그것이 존재하는 완전식이 끝나는 시점에서(;이 있는 위치) 사라져 버리는 임시 값입니다. 우측 값은 예를 들면 1729, x+y, std::string("C++") 또한 x++ 등입니다.

즉, 식이 끝난 후 계속 존재하는 값은 좌측 값, 식이 끝나면 존재하지 않는 값은 임시 값은 우측 값입니다.

왠지 좀 확실하게 와닿지 않으시죠? 프로그래머에게는 글 보다는 코드가 가장 확실하죠. 아래의 코드로 표현한 그림을 보시면 확실하게 이해할 수 있을 것입니다. =

〈 그림 4-4. 좌측 값과 우측 값 예 〉

```
struct PLAYER{};
Int foo() { return 0; }

Int main()
{
    int nCount = 0;
    nCount: ← LValue
    0: ← RValue

    PLAYER Player;
    Player: ← LValue
    PLAYER(): ← RValue

    foo(): ← RValue
}
```

이제 좌측 값과 우측 값이라는 것에 대해서 구분할 수 있을 테니 이번 장의 주제인 우측 값 참조에 대해서 본격적으로 들어갑시다.

4.3 좌측 값 참조와 우측 값 참조

C++에는 참조라는 라는 것이 있습니다. 사용 방법은 ‘&’을 사용합니다.

〈 그림 4-5. refA는 변수 a를 참조 〉

```
.....
int a = 10;
int& refA = a;
.....
```

참조

refA는 변수 a를 참조합니다. 이후 refA의 값을 변경하면 refA가 참조하고 있는 변수 a의 값이 변경됩니다. 즉 refA는 변수 a를 가리키고 있는 것입니다.

이렇게 ‘&’을 사용한 참조를 정확하게는 ‘LValue Reference’라고 부릅니다. LValue Reference는 C++ 98 때부터 있는 것으로 지금 저희가 잘 알고 자주 사용하고 있는 것입니다.

C++0x에는 새롭게 RValue Reference라는 것이 생겼습니다.

사용 방법은 기존의 참조와 비슷하여 참조가 ‘&’을 사용했듯이 ‘&&’를 사용합니다.

```
<코드>
int&& RrefA = 119;
</코드>
```

우측 값 참조는 외견 상으로는 좌측 값 참조에 비해서 ‘&’을 하나 더 사용한다는 것만 다르지만 의미상 좌측 값 참조와 우측 값 참조는 다릅니다.

< 그림 4-6. 우측 값 참조와 좌측 값 참조 사용 예 >

```
int nCount;
int& lrefValue1 = nCount; ----- a) 문제 없음
int& lrefValue2 = 10; ----- b) 에러

int&& rrefValue1 = 10; ----- c) 문제 없음
int&& rrefValue2 = nCount; ----- d) 에러
```

<그림 4-6>에서 b)는 좌측 값 참조에 우측 값을 대입했기 때문에 에러가 발생하고, d)는 우측 값 참조에 좌측 값을 대입했기 때문에 에러가 발생합니다.

이렇게 좌측 값 참조는 좌측 값을 참조해야 하고, 우측 값 참조는 우측 값을 참조해야 합니다.

4.4 Move semantics

4장 첫 머리에서 우측 값 참조 덕택에 불필요한 복사를 없앨 수 있다고 했는데 불필요한 복사를 없앨 수 있는 것은 바로 우측 값 참조의 Move semantics 덕택입니다.

Move semantics에 의해서 C++0x에서는 기존에는 없는 ‘move 생성자’, ‘move 대입 연산자’라는 것이 생겼습니다. move 생성자와 move 대입 연산자는 ‘&&’를 사용합니다.

클래스를 정의할 때 move 생성자나 move 대입 연산자를 정의하면 어떤 오브젝트에서 다른 오브젝트로 리소스를 복사가 아닌 이동 시킬 수 있습니다. 이 이동이라는 것은 오브젝트를 다른 장소의 메모리로 이동 시키는 것입니다.

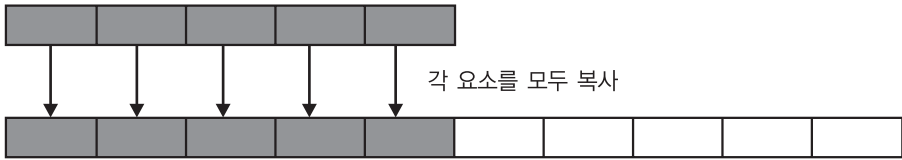
‘move 생성자’, ‘move 연산자’는 암묵적으로는 만들어지지 않으면 ‘복사 생성자’가 ‘move 생성자’보다 우선 순위가 높고, ‘대입 연산자’가 ‘move 대입 연산자’보다 우선 순위가 높습니다.

복사가 아닌 리소스의 이동은 STL의 vector의 크기를 키울 때 사용할 수 있습니다.

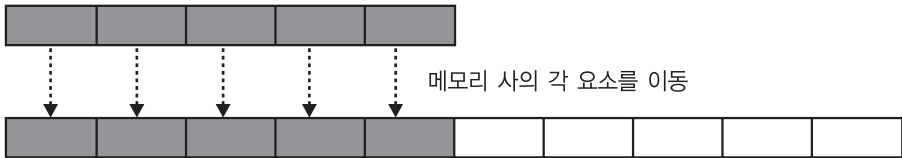
현재의 STL vector는 새로운 요소를 삽입할 때 빈 영역이 없으면 새로운 메모리를 확보한(보통 현재 할당된 메모리의 2배) 후 기존 요소를 복사한 후 새로운 요소를 삽입합니다. 그러나 C++0x에서는 Move semantics에 의해 성능 부하가 큰 복사가 아닌 메모리 상으로의 이동을 합니다.

〈 그림 4-7. C++98/03과 C++0x에서의 vector의 크기 확장 〉

C++98/03 - vector의 크기를 확장한다면



C++0x - vector의 크기를 확장한다면



C++0x의 모든 STL에는 Move semantics가 적용됩니다. 그래서 기존의 코드를 바꾸지 않고 C++0x를 사용하는 컴파일러만 사용해도 성능 향상이 이루어집니다. VC 10은 우측 값 참조를 지원하므로 모든 STL에 Move semantics가 적용 되었습니다.

4.5 Move 생성자와 Move 대입 연산자

앞서 C++0x에서는 우측 값 참조에 의해 'move 생성자'와 'move 대입 연산자'라는 것이 생겼다고 하였습니다. move 생성자와 move 대입 연산자 정의는 아래와 같습니다.

〈 코드 4-1. QuestInfo 클래스 〉

〈코드〉

```
class QuestInfo
{
public:
    // 복사 생성자
    QuestInfo(const QuestInfo& quest)
        : Name(new char[quest.NameLen]), NameLen(quest.NameLen)
    {
        memcpy(Name, quest.Name, quest.NameLen);
    }

    // 대입 연산자
    QuestInfo& operator=(const QuestInfo& quest)
    {
        if (this != &quest)
        {
            if (NameLen < quest.NameLen)
            {
                // 버퍼를 확보한다
            }

            NameLen = quest.NameLen;
            memcpy(Name, quest.Name, NameLen);
        }

        return *this;
    }
}
```

```

// move 생성자
QuestInfo(QuestInfo&& quest)
    : Name(quest.Name), NameLen(quest.NameLen)
{
    quest.Name = NULL;
    quest.NameLen = 0;
}

// move 대입 연산자
QuestInfo& operator=(QuestInfo&& quest)
{
    if( this != &quest )
    {
        delete Name;

        Name = quest.Name;
        NameLen = quest.NameLen;

        quest.Name = NULL;
        quest.NameLen = 0;
    }
    return *this;
}

private:
    char* Name;
    int NameLen;
};
</코드>

```

위의 <코드 4-1>에서 QuestInfo(QuestInfo&& quest)가 move 생성자, QuestInfo& operator=(QuestInfo&& quest)가 move 대입 연산자입니다.

외견 상으로 기존의 복사 생성자, 대입 연산자와의 차이는 함수 파라미터에서 ' & ' 가 아닌 ' && ' 을 사용하는 것입니다.

4.6 Move semantics에 의한 성능 향상

우측 값 참조를 설명할 때 이것 덕분에 프로그램의 성능이 좋아진다고 했습니다.

왜 그럴까요? <코드 4-1>에서 복사 생성자와 Move 생성자, 대입 연산자와 Move 대입 연산자의 구현을 다시 한번 잘 보시기 바랍니다. 잘 보시면 왜 성능이 좋은지 바로 아실 수 있을 것입니다. 이유를 찾으셨나요?

앞서 여러 번 우측 값 참조는 Move Semantics에 의해 복사가 아닌 메모리 상의 이동을 한다고 말했습니다. <코드 4-1>의 Move 생성자와 Move 대입 연산자는 넘겨 받은 인자를 복사하지 않고 메모리 상의 이동을 하고 있습니다.

< 코드 4-2. 복사 생성자와 Move 생성자 >

```

<코드>
// 복사 생성자
QuestInfo(const QuestInfo& quest)
    : Name(new char[quest.NameLen]), NameLen(quest.NameLen)
{
    memcpy(Name, quest.Name, quest.NameLen);
}

```

```
// move 생성자
QuestInfo(QuestInfo&& quest)
    : Name(quest.Name), NameLen(quest.NameLen)
{
    quest.Name = NULL;
    quest.NameLen = 0;
}
</코드>
```

<코드 4-2>의 코드에서 move 생성자 정의에서 굵게 표시한 코드를 보면 복사 생성자와 달리 메모리 주소를 대입한 것을 알 수 있을 것입니다. 이것이 바로 복사가 아닌 이동입니다. 메모리 이동을 한 후 인자로 넘겨진 객체가 사라지더라도 메모리 파괴가 일어나지 않도록 인자로 넘겨진 객체에는 NULL을 대입하고 있습니다.

이렇게 복사가 아닌 이동을 하는 것은 크기가 작은 오브젝트에서는 큰 의미가 없지만 크기가 큰 오브젝트에서는 그 차이가 무시할 수 없을 것입니다.

4.7 move 생성자와 move 대입 연산자 사용 예

move 생성자와 move 대입 연산자를 정의한 클래스는 어떻게 동작하는 예제를 통해서 보여드리겠습니다.

< 코드 4-3. 복사 생성자와 대입 연산자 정의 >

```
<코드>
#include <iostream>

using namespace std;

class NPC
{
public:
    int NPCCode;
    string Name;
    NPC() { cout << "기본생성자" << endl; }
    NPC( int _NPCCode, string _Name ) { cout << "인자있는생성자" << endl; }
    NPC(NPC& other) { cout << "복사생성자" << endl; }
    NPC& operator=(const NPC& npc) { cout << "대입연산자" << endl; return *this; }
    NPC(NPC&& other) { cout << "Move 생성자" << endl; }
    NPC& operator=(const NPC&& npc) { cout << "Move 연산자" << endl; return *this; }
};

int main()
{
    cout << "1" << endl;
    NPC npc1( NPC( 10,"Orge1" ) );

    cout << endl << "2" << endl;
    NPC npc2(11,"Orge2");
    NPC npc3 = npc2;

    cout << endl << "3" << endl;
    NPC npc4; NPC npc5;
    npc5 = npc4;
```



```

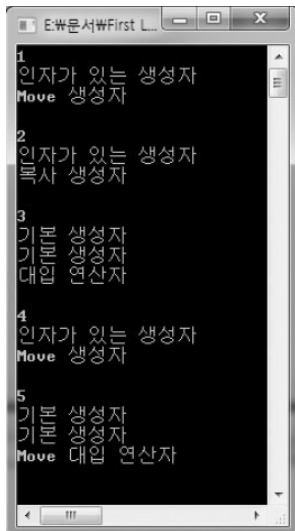
cout << endl << "4" << endl;
NPC npc6 = NPC(12, "Orge3");

cout << endl << "5" << endl;
NPC npc7; NPC npc8;
npc8 = std::move(npc7);

getchar();
return 0;
}
</코드>

```

< 결과 4-3 >



<코드 4-3>의 결과를 보면 move 생성자와 move 대입 연산자를 정의한 NPC 클래스는 우측 값을 사용했을 때는 move 생성자와 move 대입 연산자가 호출 됨을 알 수 있습니다. 그런데 <코드 4-3>에서 std::move라는 이전까지 보지 못했던 함수를 사용했습니다. move 라는 이름을 사용하는 걸로 봐서 우측 값 참조와 관련된 함수일 것 같지 않나요?

4.8 std::move

```

<코드>
cout << endl << "5" << endl;
NPC npc7; NPC npc8;
npc8 = std::move(npc7);
</코드>

```

위 코드는 <코드 4-3>에서 npc8에 npc7 이라는 좌측 값을 대입했는데 결과를 보면 move 대입 연산자가 사용 되었습니다. move 생성자나 move 대입 연산자는 인자가 우측 값일 때만 사용되므로 일반적인 대입 연산자가 호출 되어야합니다. 그러나 위 코드에서는 move 대입 연산자가 호출 되었습니다. 이것은 바로 std::move 라는 함수를 사용하였기 때문입니다.

std::move는 Move Semantics를 위해 이번에 새롭게 추가된 것입니다. 이것은 좌측 값(LValue)을 우측 값(RValue)으로 타입 캐스팅하기 위해 표준 라이브러리에서 제공하는 함수입니다.

std::move는 아래와 같이 구현되어 있습니다.

```
<코드>
namespace std {
    template <class T>
        inline typename remove_reference<T>::type&& move(T&& x)
        {
            return x;
        }
}
</코드>
```

위 코드를 보면 알 수 있듯이 std::move 라는 것이 뭔가 복잡한 기능을 가진 것이 아니고 단순히 타입 캐스팅을 해주는 함수라는 것을 알 수 있습니다.

4.9 VC 10에서의 STL과 우측 값 참조

앞에서 VC 10의 STL에는 우측 값 참조가 적용되었다고 말했습니다. 이 덕분에 기존의 코드를 수정하지 않고 VC 10으로 빌드만 해도 기존에 비해서 성능이 더 좋아집니다.

우측 값 참조가 적용된 VC 10의 STL vector

VC 10의 vector쪽 소스 코드를 보면 이전의 vector에는 없던 코드가 있습니다.

```
<코드>
.....
#ifdef _HAS_RVALUE_REFERENCES

        vector(_Myt&& _Right)
            : _Mybase(_Right._Alval)
        {
            // construct by moving _Right
            _Assign_rv(_STD forward<_Myt>(_Right));
        }

.....
</코드>
```

위 코드를 보면 바로 아시겠죠? 네 move 생성자가 정의 되어 있습니다.

< 코드 4-4. 우측 값 참조를 사용한 vector 사용 >

```
<코드>
vector<int> foo()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(2);

    return v;
}
```

```
int main()
{
    vector<int> v1 = foo();
    cout << v1[0] << endl;
    return 0;
}
</코드>
```

<코드 4-4>를 VC 9(visual Studio 2008)와 VC 10에서 디버깅 해보면 vector의 생성자에서 서로 다른 부분을 호출하는 것을 볼 수 있습니다.

VC 9에서 디버깅을 해보면 다음의 위치에서 브레이크 포인트가 걸립니다.

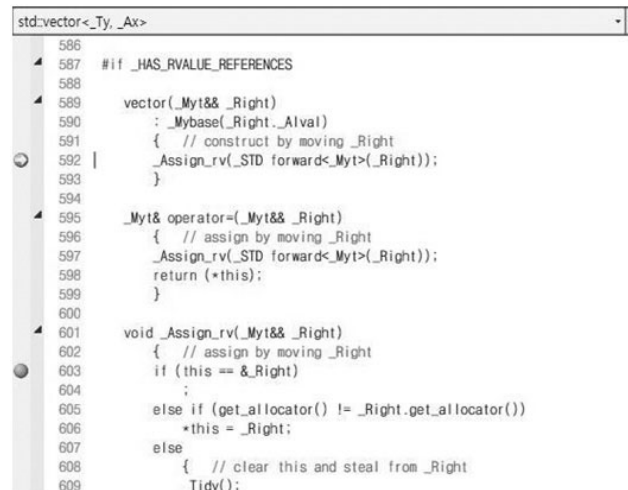
< 그림 4-8. VC 9의 vector >



```
std::vector<_Ty, _Ax>
{
    490 }
    491
    492 vector(size_type _Count, const _Ty& _Val, const _Alloc& _Al)
    493 : _Mybase(_Al)
    494 { // construct from _Count + _Val, with allocator
    495     _Construct_n(_Count, _Val);
    496 }
    497
    498 vector(const _Myt& _Right)
    499 : _Mybase(_Right._Alval)
    500 { // construct by copying _Right
    501     if (_Buy(_Right.size()))
    502         _TRY_BEGIN
    503             _Mytast = _Ucopy(_Right.begin(), _Right.end(), _Myfirst);
    504             _CATCH_ALL
    505             _Tidy();
    506             _RERAISE;
    507             _CATCH_END
    508         }
    509
    510 template<class _Iter>
    511 vector(_Iter _First, _Iter _Last)
    512 : _Mybase()
    513 { // construct from [_First, _Last)
    514     _Construct(_First, _Last, _Iter cat(_First));
    515 }
```

위 코드를 보면 복사 생성자에서 받은 vector의 크기를 비교하여 현재 공간이 인자로 받은 vector보다 작으면 재할당을 하고, vector에 있는 모든 요소를 복사합니다.

< 그림 4-9. VC 10의 vector >



```
std::vector<_Ty, _Ax>
{
    586
    587 #if _HAS_RVALUE_REFERENCES
    588
    589 vector(_Myt&& _Right)
    590 : _Mybase(_Right._Alval)
    591 { // construct by moving _Right
    592     _Assign_rv(_STD forward<_Myt>(_Right));
    593 }
    594
    595 _Myt& operator=(const _Myt&& _Right)
    596 { // assign by moving _Right
    597     _Assign_rv(_STD forward<_Myt>(_Right));
    598     return (*this);
    599 }
    600
    601 void _Assign_rv(_Myt&& _Right)
    602 { // assign by moving _Right
    603     if (this == &_Right)
    604         ;
    605     else if (get_allocator() != _Right.get_allocator())
    606         *this = _Right;
    607     else
    608     { // clear this and steal from _Right
    609         _Tidy();
    610     }
    611 }
```

보시는 바와 같이 VC 10에서는 move 생성자가 호출됩니다. 그리고 요소를 복사하지 않고 메모리 상의 이동을 합니다.

또한 vector에 할당된 공간을 다 사용한 상태에서 새로운 요소를 삽입하면 기존에는 새로운 공간을 할당 후 저장하고 있던 모든 요소를 복사를 하지만 C++0x에서는 Move Semantics에 의해 메모리 상의 이동을 합니다.

STL의 string

너무 당연하지만 string 클래스도 우측 값 참조가 잘 적용되어 있습니다.

```
<코드>
string msg1("Error");
string msg2("- Network");
string msg3(": Accept");

string Msg = msg1 + " " + msg2 + " " + msg3;
</코드>
```

string에 만약 우측 값 참조가 적용되지 않았다면 operator+()가 호출될 때마다 임시 문자열이 생성되어 동적 메모리 할당과 복사가 일어나지만 우측 값 참조를 적용하면 불필요한 동적 메모리 할당과 해제 그리고 복사를 제거할 수 있습니다.

4.10 우측 값 참조를 사용할 때 주의할 점

우측 값 참조를 사용하면 Move Semantics에 의해 의도하지 않은 버그가 발생할 수 있습니다.

< 그림 4-10. 좌측 값이 우측 값으로 사용된 이후 >

```
NPC npc7;
NPC npc8;
npc8 = std::move(npc7);
.....
.....
```

← 이후 npc7을 사용할 수 있을지 알 수 없다!!!

<그림 4-10>에서 npc7은 좌측 값이지만 std::move 함수에 의해서 우측 값으로 사용되었습니다. move 대입 연산자는 메모리 상의 이동을 하므로 npc7이 동적 메모리를 사용한 멤버를 가지고 있다면 std::move를 사용한 이후의 npc7의 값은 유효하지 않을 것입니다.

이런 문제는 STL에 우측 값 참조가 적용 되었으므로 vector에서도 이런 문제가 발생합니다.

< 코드 4-5. Vector의 대입 문제 >

```
<코드>
int main()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(12);
```

```

        vector<int> v2 = std::move(v1);

        cout << v1.size() << endl;
        cout << v2.size() << endl;

        return 0;
}
</코드>

```

〈 결과 4-5 〉



〈코드 4-5〉에서 v1을 v2에서 move 생성자를 사용하기 위해서 std::move()를 사용했습니다. std::move는 메모리 이동을 한다고 했습니다. 그래서

```

<코드>
vector<int> v2 = std::move(v1);
</코드>

```

이 후에는 v1의 크기는 0이 됩니다.

만약 우측 값 참조의 Move Semantics를 잘 이해하지 못한 상태에서 우측 값 참조를 사용한다면 아주 골치 아픈 버그를 만들 수 있습니다. 우측 값 참조에서 사용하는 우측 값은 임시 값이라는 것을 잘 기억하고 우측 값 참조를 사용해야 합니다.

4.11 우측 값 참조와 좌측 값 참조의 함수 오버로드

우측 값 참조와 좌측 값 참조는 타입이 다르므로 함수 오버로드를 적용할 수 있습니다.

〈 코드 4-6. 함수 오버로드 〉

```

<코드>
struct ITEM {};

void GetItem( ITEM& item )
{
    cout << "LValue 참조" << endl;
}

void GetItem( ITEM&& item )
{
    cout << "RValue 참조" << endl;
}

int main()
{
    ITEM item;
    GetItem( item );
    GetItem( ITEM() );
}

```

```

getchar();
return 0;
}
</코드>

```

< 결과 4-6 >



4.12 우측 값 참조는 우측 값이 아니다

우측 값은 임시 값입니다. 그러나 우측 값 참조는 임시 값이 아닙니다. 우측 값을 참조하여 우측 값 참조가 되었더라도 우측 값 참조는 우측 값이 아닙니다. 그래서 우측 값 참조는 우측 값 참조로 초기화 할 수 없습니다.

< 그림 4-11. 우측 값 참조의 성공과 실패 예 >

```

#include <iostream>

int main()
{
    int LValue1;

    int&& RValueRef1 = LValue1; Error

    int&& RValueRef2 = std::move(LValue1); OK

    int&& RValueRef3 = RValueRef2; Error

    return 0;
}

```

4.13 Perfect Forwarding

앞선 설명에서 우측 값 참조를 사용할 때 주의할 점을 이야기 했습니다. 좌측 값이 우측 값이 되어 우측 값 참조로 사용되면 그 좌측 값은 보증할 수 없게 됩니다.

템플릿 프로그래밍에서는 인자 추론이라는 것이 있는데 이 인자 추론에 의해서 좌측 값과 우측 값을 파라미터로 사용한 함수가 원하는대로 호출되지 않는 문제가 발생합니다.

〈 코드 4-7.〉

〈코드〉

```
#include <iostream>

struct X {};

void func( X&& t )
{
    std::cout << "RValue" << std::endl;
}

void func( X& t )
{
    std::cout << "LValue" << std::endl;
}

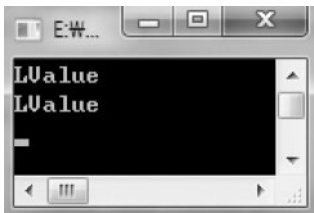
template<typename T>
void foo(T&& t)
{
    func( t );
}

int main()
{
    X x;
    foo(x);

    foo( X() );

    getchar();
    return 0;
}
</코드>
```

〈 결과 4-7〉



〈코드 4-7〉의 결과를 보면 인자 추론을 할 때 우선 순위에 의해서 좌측 값을 파라미터로 사용한 함수가 모두 호출 되었습니다.

이 문제를 해결하기 위해 foo 함수에서 std::move를 사용 하였습니다.

〈 코드 4-8〉

〈코드〉

```
#include <iostream>

struct X {};
```

```

void func( X&& t )
{
    std::cout << "RValue" << std::endl;
}

void func( X& t )
{
    std::cout << "LValue" << std::endl;
}

template<typename T>
void foo(T&& t)
{
    func( std::move(t) );
}

int main()
{
    X x;
    foo(x);

    foo( X() );

    getchar();
    return 0;
}
</코드>

```

< 결과 4-8 >



std::move를 사용하여 foo 함수에서 우측 값 함수를 호출하도록 하였습니다.

그러나 여기에서도 문제가 있습니다. std::move를 사용하는 바람에 foo 함수에 좌측 값을 전달해도 우측 값 참조로 사용됩니다. 앞서 이야기 했지만 좌측 값이 우측 값 참조로 사용되면 그 좌측 값을 보증하지 못합니다. <코드 4-8>의 foo 함수는 의도하지 않은 버그를 발생할 수도 있습니다.

이런 문제는 std::forward 라는 표준 라이브러리에서 제공하는 함수를 사용하여 해결할 수 있습니다. std::forward도 std::move와 같이 타입 캐스팅을 해주는 함수입니다. std::forward 함수는 좌측 값은 좌측 값으로, 우측 값은 우측 값으로 캐스팅 해 줍니다.

<코드 4-7>과 <코드 4-8>의 문제를 std::forward 함수를 사용하여 아래와 같이 해결했습니다.

< 코드 4-9. std::forward 사용 예 >

```

<코드>
#include <iostream>

struct X {};

void func( X&& t )

```



```

{
    std::cout << "RValue" << std::endl;
}

void func( X& t )
{
    std::cout << "LValue" << std::endl;
}

template<typename T>
void foo(T&& t)
{
    func( std::forward<T>(t) );
}

int main()
{
    X x;
    foo(x);

    foo( X() );

    getchar();
    return 0;
}
</코드>

```

< 결과 4-9 >



std::forward는 아래와 같이 구현 되어 있습니다.

```

<코드>
namespace std {
    template <class T, class U,
    class = typename enable_if<
    (is_lvalue_reference<T>::value ?
    is_lvalue_reference<U>::value :
    true) &&
    is_convertible<typename remove_reference<U>::type*,
    typename remove_reference<T>::type*>::value
    >::type>
    inline T&&
    forward(U&& u)
    {
        return static_cast<T&&>(u);
    }
}
</코드>

```

4.14 핵심 요약

1. 식이 끝난 후 계속 존재하는 값은 좌측 값, 식이 끝나면 존재하지 않는 값은 임시 값은 우측 값
2. ' & ' 을 사용한 참조를 정확하게는 ' LValue Reference ' 라고 부른다. 사용 방법은 기존의 참조와 비슷하여 참조가 ' & ' 을 사용했듯이 ' && ' 를 사용
3. 좌측 값 참조는 좌측 값을 참조하고, 우측 값 참조는 우측 값을 참조한다.
4. 불필요한 복사를 없앨 수 있는 것은 바로 우측 값 참조의 Move semantics 덕택. Move semantics에 의해서 C++0x에서는 기존에 없는 'move 생성자', 'move 대입 연산자' 라는 것이 생겼음
5. 'move 생성자', 'move 연산자' 는 암묵적으로는 만들어지지 않으면 '복사 생성자' 가 'move 생성자' 보다 우선 순위가 높고, '대입 연산자' 가 'move 대입 연산자' 보다 우선 순위가 높다.
6. 표준 라이브러리에서 제공하는 std::move를 사용하면 좌측 값을 우측 값으로 타입 캐스팅 할 수 있다.
7. 우측 값 참조와 좌측 값 참조는 타입이 다르므로 함수 오버로드를 적용할 수 있다.
8. 우측 값 참조는 우측 값이 아니다.
9. std::forward 함수는 좌측 값은 좌측 값으로, 우측 값은 우측 값으로 캐스팅 해 준다.

5. 람다 (lambda)

람다는 람다 함수 또는 이름 없는 함수라고 부르며 그 성질은 함수 오브젝트와 같습니다. 규격에서는 람다는 특별한 타입을 가지고 있다고 합니다. 그렇지만 decltype나 sizeof에서는 사용할 수 없습니다.

C++0x는 람다 덕택에 C++의 표현력이 이전보다 훨씬 더 증대 되었습니다.

5.1 C#과 람다

C#이나 동적 프로그래밍 언어를 공부 하신 분들은 람다에 대해서 들어보셨을 것입니다. 람다를 잘 모르는 분들을 위해서 현재 가장 쉽게 람다를 접할 수 있는 C#을 통해서 람다의 사용 예를 들어 보겠습니다.

C#에서의 람다

람다는 식과 문을 포함하여 대리자나 식 트리 형식을 만드는데 사용할 수 있는 익명함수입니다. 형식은 다음과 같습니다.

입력 매개 변수 => 식 or 문

람다는 주로 어떤 라이브러리의 식과 결합해서 사용할 식을 만들 때 사용합니다. 람다가 없다면 다른 식과 결합하기 위해서는 따로 함수를 만들어서 사용해야 하므로 거추장스러워 지는데 람다를 사용하면 아주 간단하게 구현할 수 있습니다.

〈 코드 5-1. 이름 중 문자 길이가 TextLength 보다 작은 이름 〉

〈코드〉

```
string[] MobNames = { "Babo", "Cat", "Orge", "Tester", "CEO" };
int TextLength = 4;
```

// 람다 식 사용

```
var ShortNames1 = MobNames.Where(MobName => MobName.Length < TextLength);
```

// foreach 사용

```
List<string> ShortNames2 = new List<string>();
foreach(string MobName in MobNames)
{
    if (MobName.Length < TextLength)
    {
        ShortNames2.Add(MobName);
    }
}
```

〈/코드〉

〈코드 5-1〉을 보면 람다를 사용하여 한 줄로 끝나는 것을 람다를 사용하지 않으면 List 컨테이너를 생성하고 foreach 구문을 사용하여 이름을 하나씩 조사하여 List 컨테이너에 추가해야 되는 코드가 필요해집니다.

C#에서 람다가 가장 유용하게 사용되는 부분은 LINQ 일겁니다.

만약 LINQ에서 람다를 사용하지 않게 된다면 LINQ를 사용하기가 꽤 힘들어질 것입니다.

5.2 C++에서 STL 알고리즘을 사용할 때 불편했던 점

기존의 C++에서 STL의 find_if, sort 등의 알고리즘을 사용할 때 특정 조건자를 사용하기 위해서는 펄터(functor)를 만들어야 합니다. 그런데 이것 때문에 따로 펄터를 만드는 것이 귀찮아서 보통 그냥 따로 함수를 만들어서 구현하기도 합니다.

〈 코드 5-2. 펄터를 사용한 find_if 알고리즘 〉

〈코드〉

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

using namespace std;

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

    void SetDie() { m_bDie = true; }
    bool IsDie() { return m_bDie; }

private:
    bool m_bDie;
};

struct FindDieUser
{
    bool operator() (User& tUser) const { return tUser.IsDie(); }
};

int main()
{
    vector< User > Users;
    User tUser1;      Users.push_back(tUser1);
    User tUser2;      tUser2.SetDie();    Users.push_back(tUser2);
    User tUser3;      Users.push_back(tUser3);

    vector< User >::iterator Iter;
    Iter = find_if( Users.begin(), Users.end(), FindDieUser() );

    return 0;
}
</코드>
```

〈코드 5-2〉는 유저들 중 죽은 유저를 찾기 위해 find_if 알고리즘을 사용했는데 이것 때문에 struct로 펄터를 만들어야 합니다. 그런데 펄터를 만들지 않고 그냥 간단하게 기술할 수 있으면 좋겠죠?

혹시 <코드 5-1>에서 C#에서는 람다 식으로 쉽게 구현했던 것을 보았는데 <코드 5-2>의 find_if 알고리즘에서도 이런 것을 사용하면 좋지 않을까요? C++0x에서 람다가 새로 생겼습니다.

5.3 C++에서의 람다 사용 법

5.3.1 문법

람다의 문법은 아래와 같습니다.

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

아래는 람다를 사용한 간단한 예입니다.

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

5.3.2 람다를 변수에 대입

auto를 사용하면 람다를 변수에 대입할 수 있습니다.

< 코드 5-3. 람다를 변수에 대입 >

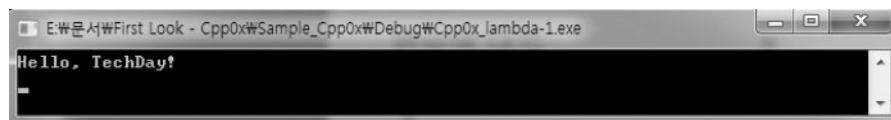
```
<코드>
#include <iostream>

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

    func();

    getchar();
    return 0;
}
</코드>
```

< 결과 5-3 >



5.3.3 람다를 함수의 인자로 사용하기

템플릿 프로그래밍에서 함수의 인자로 람다를 사용할 수 있습니다.

〈 코드 5-4. 함수의 인자로 사용 〉

```

<코드>
#include <iostream>

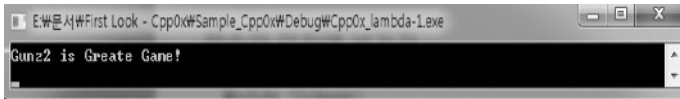
template< typename Func >
void Test( Func func )
{
    func();
}

int main()
{
    auto func = [] { std::cout << "Gunz2 is Greate Game!" << std::endl; };
    Test( func );

    getchar();
    return 0;
}
</코드>

```

〈 결과 5-4 〉



5.3.4 람다의 파라미터

람다는 일반 함수처럼 파라미터를 정의할 수 있습니다.

〈 코드 5-5. 파라미터 사용 〉

```

<코드>
#include <iostream>

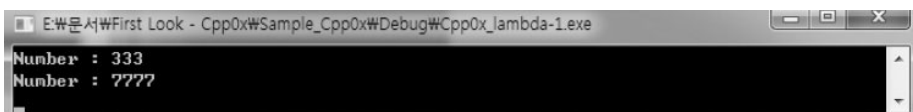
int main()
{
    auto func = []( int n ) { std::cout << "Number : " << n << std::endl; };

    func( 333 );
    func( 7777 );

    getchar();
    return 0;
}
</코드>

```

〈 결과 5-5 〉



5.3.5 반환 값 넘기기

람다는 당연히 반환 값을 넘길 수도 있습니다. 반환 값의 타입은 명시적으로 지정할 수도 있고, 암묵적으로 타입을 추론할 수 있습니다.

〈 코드 5-6. 반환 값 사용 〉

```

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

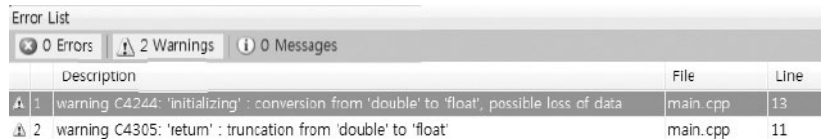
    float f1 = func1();
    float f2 = func2( 3.14f );
    float f3 = func3();

    return 0;
}
</코드>

```

〈코드 5-6〉을 컴파일하면 아래와 같은 경고가 표시 됩니다.

〈 그림 5-1. 〈코드 5-6〉의 컴파일 경고 〉



Error List			
0 Errors 2 Warnings 0 Messages			
	Description	File	Line
1	warning C4244: 'initializing' : conversion from 'double' to 'float', possible loss of data	main.cpp	13
2	warning C4305: 'return' : truncation from 'double' to 'float'	main.cpp	11

〈그림 5-1〉의 경고 내용을 보면 func1가 반환하는 3.14은 double 타입으로 추론되었고, func2는 float 타입의 파라미터를 반환하고 있어서 float로 추론되었습니다. func3는 반환 타입을 float로 명시적으로 지정하였기 때문에 3.14를 반환하지만 func1과 같이 double이 아닌 float 타입으로 반환 되었습니다.

반환 타입을 명시적으로 반환 할때는 〈코드 5-6〉의 func3 처럼

→ 반환타입

으로 지정합니다.

5.4 캡처

람다를 사용할 때 람다 외부에 정의되어 있는 변수를 람다 내부에서 사용하고 싶을 때는 그 변수를 캡처(Capture)합니다.

캡처는 참조나 복사로 전달이 가능합니다. 참조를 사용하는 경우는 '&'을 사용하고, 복사로 전달할 때는 그냥 변수 이름을 기술합니다.

람다 표현의

```
<코드>
[] (파라미터) { 식 }
</코드>
```

에서 앞의 '[]' 사이에 캡처 할 변수를 기술합니다.

참조로 캡처

<코드 5-7>은 람다 외부의 변수를 참조로 캡처하고 있습니다.

< 코드 5-7. lambda에서 캡처 사용 >

```
<코드>
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;
}
</코드>
```

< 결과 5-7 >



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

<코드 5-8>과 같이 포인터 변수를 전달할 수도 있습니다.

< 코드 5-8. 캡처로 포인터 전달 >

```
<코드>
.....
int TotalMoney2 = 0;
int* pTotalMoney2 = &TotalMoney2;
for_each(Moneys.begin(), Moneys.end(), [pTotalMoney2](int Money) {
    *pTotalMoney2 += Money;
});

cout << "Total Money 2 : " << TotalMoney2 << endl;
.....
```


</코드>

< 결과 5-8 >



복사로 캡처

그럼 TotalMoney1 변수를 값으로 전달하면 어떻게 될까요?

<코드>

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

</코드>

아래와 같은 컴파일 에러가 발생합니다.

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

만약 꼭 복사로 캡처한 변수를 람다 내부에서 변경을 해야한다면 mutable 키워드를 사용하면 에러 없이 컴파일 할 수 있습니다.

<코드>

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

</코드>

단 컴파일은 되지만 람다 내부에서 변경한 외부 변수의 값은 람다를 벗어나면 람다 내부에서 변경하기 전의 원래의 값이 됩니다.

복수의 변수 캡처

<코드 5-7>에서는 하나의 변수만을 캡처했지만 복수의 변수를 캡처하는 것도 가능할까요? 네 당연히 가능합니다.

[] 사이에 캡처 할 변수를 선언하면 됩니다.

<코드>

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

</코드>

그럼 [&] 로 하면 어떻게 될까요? 이렇게 하면 람다 식을 정의한 범위 내에 있는 모든 변수를 캡처할 수 있습니다. 또 람다 외부의 모든 변수를 복사하여 캡처할 때는 [=]을 사용합니다.

〈 코드 5-9. 람다 외부의 모든 변수를 참조로 캡처하기 〉

〈코드〉

```
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 1000 보다 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;
}

```

〈/코드〉

〈 결과 5-9 〉



default 캡처

람다 외부의 모든 변수를 참조(또는 복사)로 참조하고 일부는 복사(또는 참조)로 캡처할 수 있습니다. 그러나 같은 변수를 캡처하거나 default 캡처한 일부 변수를 같은 방식(참조 또는 복사)으로 캡처할 수 없습니다.

〈 그림 5-2. default 캡처 〉

```
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;
}

```

5.5 클래스에서 람다 사용

클래스의 멤버 함수 내에 람다 식을 정의하고, 이 람다 식에서 해당 클래스의 멤버를 호출 할 수 있습니다. 클래스 멤버 내의 람다 식은 해당 클래스에서는 friend로 인식하므로 람다 식에서 private 멤버의 접근도 할 수 있습니다. 그리고 클래스의 멤버를 호출할 때는 'this'를 캡처합니다.

〈 코드 5-10. 클래스 멤버 호출 〉

〈코드〉

```
class NetWork
{
public:
    NetWork()
    {
        SendPackets.push_back(10);
        SendPackets.push_back(32);
        SendPackets.push_back(24);
    }

    void AllSend() const
    {
        for_each(SendPackets.begin(), SendPackets.end(), [this](int i){ Send(i); });
    }

private:
    vector< int > SendPackets;

    void Send(int PacketIndex) const
    {
        cout << "Send Packet Index : " << PacketIndex << endl;
    }
};

int main()
{
    NetWork().AllSend();

    return 0;
}
</코드>
```

〈 결과 5-10 〉



〈코드 5-10〉의 NetWork 클래스의 멤버 함수 AllSend()에서 람다를 사용했습니다.

〈코드〉

```
for_each( SendPackets.begin(), SendPackets.end(),
[this](int i){ Send(i); });
</코드>
```

위와 같이 '[]' 사이에 'this'를 기술하였고, NetWork 클래스의 private 멤버 함수인 Send를 호출하고 있습니다. 당연히 멤버 변수도 호출 할 수 있습니다.

5.6 STL의 find_if에서 람다 사용

그럼 <코드 5-2>에서 평터를 정의하여 사용했던 것을 람다를 사용하는 것으로 바꾸어 보겠습니다.

< 코드 5-11. find_if 알고리즘에서 lambda 사용 >

<코드>

```
#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);

    vector< User >::iterator Iter;
    Iter = find_if( Users.begin(), Users.end(), [](User& tUser) -> bool { return
true == tUser.IsDie(); } );
    cout << "found Die User Index : " << Iter->GetIndex() << endl;

    return 0;
}
</코드>
```

< 결과 5-11 >



find_if 알고리즘을 사용하여 죽은 유저를 찾기 위해서 <코드 5-2>에서는 평터를 정의한 후 사용하였지만

<코드>

```
struct FindDieUser
{
    bool operator() (User& tUser) const { return tUser.IsDie(); }
};
```

```
Iter = find_if( Users.begin(), Users.end(), FindDieUser() );
</코드>
```

<코드 5-11>은 람다를 사용하여 한 줄로 간단하게 끝내버렸습니다.

```
<코드>
Iter = find_if( Users.begin(), Users.end(), [](User& tUser) -> bool {
return true == tUser.IsDie();
} );
</코드>
```

예전에는 알고리즘을 사용하려면 평터를 정의해야 되기 때문에 귀찮아서 알고리즘 사용이 꺼려졌는데 람다 덕분에 알고리즘 사용이 너무나 간편해졌습니다. 제 생각에 람다를 가장 자주 사용하는 부분이 바로 STL의 알고리즘을 사용할 때가 아닐까 생각합니다.

<참고>

STL의 알고리즘 사용과 성능

컨테이너의 요소를 검색을 할 때 STL의 알고리즘을 사용하는 것이 본인이 직접 for 문이나 while 문을 사용하여 순회해서 찾는 것 보다 성능이 더 좋습니다. 그런데 알고리즘을 사용하려면 평터를 정의해야 하기 때문에 귀찮아서 알고리즘을 사용하지 않는 경우가 적지 않습니다.

그러나 VC 10부터는 람다를 사용할 수 있으므로 STL의 알고리즘을 쉽게 사용할 수 있습니다. 람다를 사용하는 것은 평터를 사용하는 것과 같으므로 같은 이점을 얻을 수 있습니다.

티에프님의 Vector Container Iterating 속도 비교(<http://npteam.net/775>)

</참고>

5.7 람다 식을 STL 컨테이너에 저장

람다 식을 STL의 function을 사용하여 STL 컨테이너에 저장할 수 있습니다. 아래는 int를 반환하는 람다 식을 vector에 저장하여 사용하는 것입니다.

< 코드 5-12. 람다 식을 vector 컨테이너에 저장 >

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

using namespace std;

int main()
{
    vector<function<int()>> v;

    v.push_back( [] { return 100; } );
    v.push_back( [] { return 200; } );

    cout << v[0]() << endl;
    cout << v[1]() << endl;
}
</코드>
```

〈 결과 5-12 〉



5.8 람다를 반환하는 함수

5.7에서 람다를 vector에 저장하기 위해 STL의 function을 사용하였는데 이것을 또 다른 방법으로 사용하면 람다를 반환 값으로 사용하는 함수에서도 사용할 수 있습니다.

〈 코드 5-13. 람다를 반환하는 함수 〉

〈코드〉

```
#include <iostream>
#include <functional>
#include <string>

std::function< void() > f()
{
    std::string str("lambda");
    return [=] { std::cout << "Hello, " << str << std::endl; };
}

int main()
{
    auto func = f();
    func();
    f()();
}
```

〈/코드〉

5.9 람다에서의 재귀

람다는 재귀도 가능합니다.

〈 코드 5-14. 람다의 재귀 〉

〈코드〉

```
int main()
{
    function<int(int)> fact = [&fact](int x) {
        return x == 0 ? 1 : x * fact(x - 1);
    };

    cout << fact(3) << endl;
}
```

〈/코드〉

fact 변수를 참조로 넘겨서 재귀를 하고 있습니다.

지금까지 람다의 다양한 사용 방법을 설명 하였습니다. 람다를 사용하여 C++로 다양하게 표현할 수 있는 것을 보았을 것입니다. 람다 덕택에 C++의 표현력이 이전보다 훨씬 더 좋아졌습니다.

참고로 Visual C++ 10에서는 Concurrency Runtime(ConRT)라는 것이 새로 추가 되었습니다. 이것은 템플릿으로 만들어진 것으로 람다가 많이 사용 되었다고 합니다.

5.10 핵심 요약

1. 람다는 람다 함수 또는 이름 없는 함수라고 부르며 함수 오브젝트이다.
2. 규격에서 람다는 특별한 타입을 가지고 있다고 한다. 단 decltype나 sizeof에서는 사용 불가
3. 람다를 정의한 곳의 외부 변수를 람다 내부에서 사용하고 싶을 때는 캡처한다.
4. 외부 변수를 참조 또는 복사로 캡처할 수 있다.
5. 클래스에서도 람다를 사용할 수 있다. 클래스는 람다를 friend로 인식한다.
6. 람다 덕택에 C++의 표현력이 이전보다 훨씬 더 증대 되었다.

6. decltype

6.1 템플릿 프로그래밍 시 문제

템플릿 타입(type)이랑 템플릿 메타 프로그래밍 등에서 함수의 반환 형이 복잡하게 정의 되어 있는 경우 타입을 단순하게 기술할 수 없습니다.

C++0x에서는 이런 문제를 풀기 위해 auto와 decltype 두 개를 제공합니다.

decltype는 auto 처럼 식의 타입을 컴파일 할 때 결정할 수 있습니다.

6.2 decltype 사용하기

사용 예는 아래와 같습니다.

〈 코드 6-1. decltype 사용 예 〉

```
<코드>
int Hp;
decltype(Hp) NPCHp = 5;
decltype(Hp + NPCHp) TotalHp;
decltype(Hp*) pHp = &Hp;
</코드>
```

decltype(Hp) NPCHp = 5;
는 Hp가 int 타입이므로 int NPCHp = 5로 컴파일 할 때 결정됩니다.

또 decltype(Hp + NPCHp) TotalHp;
는 int와 int의 덧셈이므로 int TotalHp;로 됩니다.

decltype(Hp*) pHp = &Hp;는 int* pHp = &Hp가 됩니다.

또한 함수의 반환 타입으로도 사용할 수 있습니다.

〈 코드 6-2. 함수의 반환 타입으로 decltype 사용 〉

```
<코드>
int foo();
decltype(foo()) value;
</코드>
```

decltype(foo()) value;는 int value;로 됩니다.

7. nullptr

nullptr은 C++0x에서 추가된 키워드로 널 포인터(Null Pointer)를 나타냅니다.

7.1 nullptr이 필요한 이유

C++03까지는 널 포인터를 나타내기 위해서는 NULL 매크로나 상수 0을 사용하였습니다. 그러나 NULL 매크로나 상수 0을 사용하여 함수에 인자로 넘기는 경우 int 타입으로 추론되어 버리는 문제가 발생하기도 합니다.

〈 코드 7-1. 함수 인자 추론 문제 〉

```

<코드>
#include <iostream>

using namespace std;

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

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

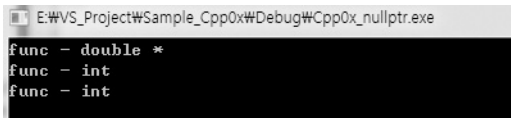
int main()
{
    func( static_cast<double*>(0) );

    func( 0 );
    func( NULL );

    getchar();
    return 0;
}
</코드>

```

〈 결과 7-1 〉



```

E:\VS_Project\Sample_Cpp0x\Debug\Cpp0x_nullptr.exe
func - double *
func - int
func - int

```

첫 번째 func 호출에서는 double* 로 캐스팅을 해서 의도했던 func이 호출 되었습니다. 그러나 두 번째와 세 번째 func 호출의 경우 func(double* p) 함수에 널 포인터를 파라미터로 넘기려고 했는데 의도하지 않게 컴파일러는 int로 추론하여 func(int a)가 호출 되었습니다.

바로 이와 같은 문제를 해결하기 위해서 nullptr 이라는 키워드가 생겼습니다.

7.2 nullptr 구현 안

C++0x에서 nullptr의 드래프트 문서를 보면 nullptr은 아래와 같은 형태로 구현 되어 있습니다.

〈 코드 7-2. nullptr 구현 클래스 〉

```

<코드>
const class {
public:
    template <class T>
    operator T*() const
    {
        return 0;
    }

    template <class C, class T>
    operator T C::*() const
    {
        return 0;
    }

private:
    void operator&() const;
} nullptr = {};
</코드>

```

7.3 nullptr 사용 방법

사용방법은 너무 너무 간단합니다. ^^

그냥 예전에 널 포인터로 0이나 NULL을 사용하던 것을 그대로 대체하면 됩니다.

```

<코드>
char* p = nullptr;
</코드>

```

〈코드 7-1〉에서 널 포인터를 파라미터로 넘겨서 func(double* p)가 호출하게 하기 위해서는

```

<코드>
func( nullptr );
</코드>

```

로 호출하면 됩니다.

7.4 nullptr의 올바른 사용과 틀린 사용 예

올바른 사용

```

<코드>
char* ch = nullptr; // ch에 널 포인터 대입.
sizeof( nullptr ); // 사용 할 수 있습니다. 참고로 크기는 4 입니다.

```

```
typeid( nullptr ); // 사용할 수 있습니다.
throw nullptr; // 사용할 수 있습니다.
</코드>
```

틀린 사용

```
<코드>
int n = nullptr; // int에는 숫자만 대입가능한데 nullptr은 클래스이므로 안됩니다.

int n2 = 0
if( n2 == nullptr ); // 에러

if( nullptr ); // 에러

if( nullptr == 0 ); // 에러

nullptr = 0; // 에러

nullptr + 2; // 에러
</코드>
```

VC++ 10에서는 예전처럼 널 포인터를 나타내기 위해서 0이나 NULL 매크로를 사용하지 말고 꼭 nullptr을 사용하여 함수나 템플릿에서 널 포인터 추론이 올바르게 되어 C++을 더 효율적으로 사용하기 바랍니다.

왜 nullptr 이라고 이름을 지었을까?

nullptr을 만들 때 기존의 라이브러리들과 이름 충돌을 최대한 피하기 위해서 구글로 검색을 해보니 nullptr로 검색 결과가 나오는 것이 별로 없어서 nullptr로 했다고 합니다.

제안자 중 한 명인 Herb Sutter은 현재 Microsoft에서 근무하고 있는데 그래서인지 C++/CLI에서는 이미 nullptr 키워드를 지원하고 있습니다.

8. 참고

C++0x

<http://en.wikipedia.org/wiki/C%2B%2B0x>

static_assert

http://en.wikipedia.org/wiki/C%2B%2B0x#Static_assertions

Rvalue Reference

<http://blogs.msdn.com/vcblog/archive/2009/02/03/rvalue-references-c-0x-features-in-vc10-part-2.aspx>

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html>

<http://www.artima.com/cppsource/rvalue.html>

http://cpplover.blogspot.com/2009/11/rvalue-reference_23.html

http://cpplover.blogspot.com/2009/11/rvalue-reference_25.html

lambda

<http://blogs.msdn.com/vcblog/archive/2008/10/28/lambda-auto-and-static-assert-c-0x-features-in-vc10-part-1.aspx>

<http://cpplover.blogspot.com/2009/11/lambda.html>

<http://cpplover.blogspot.com/2009/12/lambda.html>

decltype

<http://blogs.msdn.com/vcblog/archive/2009/04/22/decltype-c-0x-features-in-vc10-part-3.aspx>

nullptr

http://d.hatena.ne.jp/faith_and_brave/20071002/1191322319

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2431.pdf>

http://ja.wikibooks.org/wiki/More_C%2B%2B_idioms/nullptr

<http://d.hatena.ne.jp/KZR/20080328/p1>

저자 : 최홍배 마이크로소프트 Visual C++ MVP



현재 온라인 게임 개발 회사 마이에트 엔터테인먼트에서 온라인 게임의 서버 프로그램을 개발하고 있으며, Microsoft Visual C++ MVP로도 활동하고 있습니다. 프로그래밍 언어로 C++와 C#을 좋아하고 요즘은 병렬 프로그래밍과 클라우드 컴퓨팅, 스마트 폰 프로그래밍에 많은 관심을 가지고 있습니다. 블로그(<http://jacking.tistory.com/>)와 트위터(@jacking75)를 통해서 게임 이야기나 프로그래밍, 게임 개발에 대한 정보를 다른 개발자들과 공유하고 있습니다.

Microsoft®

한국마이크로소프트(유)

서울특별시 강남구 대치동 892번지 포스코 센터 서관 5층 (우) 135-777

고객지원센터 : 1577-9700, 제품홈페이지 : <http://www.microsoft.com/visualstudio>, 개발자포털 : <http://msdn.microsoft.com>