## CHAP 2. Managed C++ 언어에 대한 소개

### 이 장을 시작하기 전에

이 장에서는 Visual C++ 도구에서 .NET Framework 코드를 작성하기 위한 기본인 Managed C++ 확장에 대한 언어적인 측면을 소개한다. 기존 표준 C++ 언어에 대한 기본적인 이해가 있는 독자라면 이 장을 좀 더 쉽게 이해할 수 있을 것이라고 본다. 표준 C++ 언어에 대한 이해가 전혀 없는 독자라면 약간의 어려움이 있을 것이므로 표준 C++ 언어에 대한 참고 자료를 구하여 비교 검토하는 것이 좋겠다.
.NET의 실질적인 코드 작성에 대해 먼저 공부하고자 하는 독자라면 우선 이장을 생략하고 Part II의 내용을 먼저 살펴보는 것도 문제가 없을 것으로 본다. 후에 Managed C++ 확장 언어의 자세한 기술적 내용을 살펴보고자 할 때 다시 이 장을 참조하는 것도 좋은 진행 방법으로 생각된다.

### C# 이냐 C++ 이냐 그것이 문제로다

Microsoft에 의해 .NET이 발표된 이후 가장 관심의 대상이 되고 있는 용어는 " C# "이 아닌가 싶다. MS는 C# 개발 언어가 .NET 기반의 프로그램 개발을 위해 최적화된 솔루션이며 또한, 기존의 C++ 언어를 발전시킨 형태라고 발표하였다.
C++ 언어를 주요 개발 언어로 사용하던 개발자의 입장에서는 과연 .NET이 널리 사용될 앞으로의 개발 환경에서 C++ 언어를 계속 고집해야 하는 가를 반문하지 않을 수 없는 시기가 온 것이다.
Visual Studio .NET과 함께 제공된 Visual C++ 개발 툴은 기존의 STL, MFC와 ATL 라이브러리를 지속적으로 제공하고 있으며 이에 더불어 ATL Server라는 새로운 개발 라이브러리를 제공한다. 그러나, 이러한 라이브러리들은 .NET 환경에서 동작하는 Managed 코드를 생성하기 위한 라이브러리가 아니고 기존의 Unmanaged 코드를 생성하기 위한 라이브러리들이다. 그렇다면, VC++은 .NET 개발 언어로 사용이 불가능한가? 그렇지 않다.
Visual C++ .NET 개발툴은 Managed C++라는 확장된 C++ 언어를 새롭게 정의하여 .NET 환경에서 동작하는 Managed 코드를 작성할 수 있도록 하였다. 이에 더불어, Visual C++는 Visual Studio .NET 개발 툴 중 유일하게 Managed 코드와 Unmanaged 코드를 모두 생성할 수 있는 개발 툴이며 두 경계를 유연하게 넘나들 수 있는 고급 개발 환경이라고 말할 수 있다.
또한, Visual C++ 개발 팀은 ANSI C++ 언어 표준화 과정을 지속적이고도 가장 빠르게 지원할 것임을 발표하여 앞으로 Visual C++가 .NET 기반의 프로그램 개발 및 기존의 Windows 용 혹은 COM 프로그램 개발, Web Service 개발, 표준 C++ 및 STL을 이용한 포터블 한 코드 개발 등 각종 분야에 모두 활용될 수 있는 가장 강력한 개발 환경이 될 것으로 예상되어

진다.

### .NET 기반의 Hello World 코드

Visual C++ .NET의 새로운 프로젝트로 Managed C++ Application 형식을 선택하면 아래와 같은 간단한 소스 코드가 생성된다.

<코드 ##. Hello World >

```
// This is the main project file for VC++ application project
// generated using an Application Wizard.

#include "stdafx.h"

#using <mscorlib.dll>     // .NET 클래스 라이브러리를 사용할 수 있도록
#include <tchar.h>

// System Namespace에 속하는 클래스를 Prefix 없이 그대로 사용하도록
using namespace System;

// This is the entry point for this application
int _tmain(void)
{
    // TODO: Please replace the sample code below with your own.
    Console::WriteLine(S"Hello World");
    return 0;
}
```

기존의 Console 어플리케이션 코드와 구조는 크게 다르지 않으나 위 코드에서는 중요한 차이점이 존재한다. 우선, **#using** 전처리 문은 표준 C++ 문법에는 없는 구문으로 CLR (Common Language Runtime) 기반 DLL 혹은 OBJ 파일의 메타데이터(Metadata)를 가져오기 위하여 사용한다. 위 코드에서는 mscorlib.dll 파일을 지정하여 .NET Framework의 기본적인 클래스 라이브러리를 사용할 수 있도록 한다.
C++의 **using namespace** 구문을 그대로 사용하여 .NET 클래스 라이브러리의 System Namespace에 속하는 클래스를 별도의 구분 없이 그대로 사용하도록 하였다. 만일, using 문을 위와 같이 사용하지 않았다면 Console::WriteLine 코드는 System::Console::WriteLine 과 같이 작성되어야 할 것이다.

main 함수내부에는 System에 속하는 Console 클래스의 WriteLine 메쏘드를 호출하여 콘솔 화면에 문자열을 출력하도록 한다. Console 클래스가 제공하는 모든 멤버는 static으로 선언되어 있어 별도로 Console 클래스의 객체를 생성할 필요 없이 메쏘드를 직접 호출할 수 있다.
WriteLine 메쏘드의 인자로 전달된 문자열을 보면 문자열 앞에 S 첨두어를 사용하고 있다. 이는 문자열의 타입이 .NET 클래스 라이브러리에 정의된 System::String* 형임을 알리는 것이다. S 첨두어를 사용하여 문자열을 System::String* 형으로 지정하면 Managed 코드에서 더 좋은 수행 성능을 얻을 수 있으며 - 기존의 C++ 문자열을 그대로 사용하면 내부 적으로 변환이 일어난다. - 동일한 문자열 상수 값에 대해서는 하나의 메모리 공간만을 사용하는 장점이 있다.

<코드 ##. 동일한 문자열 상수에 대한 메모리 비교>

```
// 다른 코드는 Hello World 예제와 동일함. 생략.

String* a = S"This is a test";
String* b = S"This is a different string";
String* c = S"This is a test"; // 문자열이 동일하면 같은 메모리 공간을 사용

if ( a == b ) {
   System::Console::WriteLine(S"a == b"); // 이와 같이 전체 이름을 지정할 수도 있다.
}

if ( a == c) {
   Console::WriteLine(S"a == c");
}
```

즉, 위와 같은 코드를 수행하면 a == c 조건이 충족되어 콘솔 화면에 a == c 문자열이 출력된다. 만일 c 변수에 문자열을 대입시 S 첨두어를 생략한다면 a == c 조건이 성립되지 않아 화면에 아무런 문자열도 출력되지 않는다. 문자열 데이터에 대한 좀 더 자세한 내용은 CHAP5에서 살펴보도록 하겠다.

### Managed C++의 이점

앞에서 기본적인 C++ .NET 코드가 동작하는 것을 확인하였으니 이러한 .NET 솔루션인 Managed C++가 개발자들에게 어떠한 이점을 제공하는지 간단히 짚고 넘어가도록 하자.

1. 당연한 이야기겠지만 Managed C++를 이용하여 .NET Framework 어플리케이션을 개발 할 수 있다.
2. Managed C++를 이용하면 Managed 코드와 Unmanaged 코드를 하나의 어플리케이션 개발 과정에서 함께 사용하여 개발 할 수 있다.
3. 기존에 개발되어있는 C++ 컴퍼넌트를 손쉽게 Managed C++ 코드를 이용 .NET Framework 컴퍼넌트로 개발 할 수 있다.
4. 기존의 C++ 언어에 대한 투자 비용을 최대한 활용할 수 있다.
5. 적절하게 Managed 코드와 Unmanaged 코드를 동시에 사용함으로써 최대한의 효율성을 얻을 수 있다.
6. .NET Framework가 제공하는 방대한 클래스 라이브러리를 사용할 수 있고 프로그램 개발 효율을 높일 수 있다.

위와 같이 여러 가지 표현으로 Managed C++의 장점을 표현할 수 있겠으나 한마디로 단축하자면 “Visual C++ .NET은 Managed와 Unmanaged 코드 모두를 동시에 개발 할 수 있는 가장 강력한 Visual Studio 개발 툴이다.” 라고 말 할 수 있을 것이다.

### Visual C++의 Managed 형식의 프로젝트 종류

Visual C++의 Managed 형식의 프로젝트에는 다음과 같은 종류가 있다.

#### 1. Managed C++ Application

이 책의 주제와 가장 관계가 깊은 프로젝트 종류로 .NET 기반의 다양한 형태의 어플리케이션 프로그램을 작성하기 위한 프로젝트 형식이다.

#### 2. Managed C++ Class Library

C++ DLL 타입으로 .NET 어플리케이션에서 사용될 수 있는 컴퍼넌트를 작성하기 위한 프로젝트 형식이다.

#### 3. Managed C++ Empty Project

Managed 확장 형식으로 컴파일 및 링크를 할 수 있도록 프로젝트를 설정하고 별도의 파일은 생성하지 않는다. 기존의 C++ 코드를 .NET Framework 코드로 포팅하기 위해서 사용할 수 있을 것이다.

#### 4. Managed C++ Web Services

이 책에서는 Web Service에 대한 내용은 다루지 않는다. 이 프로젝트는 Managed C++

를 이용하여 Web Service코드를 작성하기 위하여 사용된다.

이와 같은 프로젝트 형식 중 이 책의 주요 주제인 Windows 프로그램을 작성하기 위해서 Managed C++ Application 형식의 프로젝트를 가장 많이 사용하게 될 것이다. 또한, 책의 후반부에 컴퍼넌트 프로그램 작성 부분에서는 Class Library 형식의 프로젝트도 살펴보게 될 것이다.

### Managed C++의 키워드

Visual C++는 .NET Framework 프로그램을 개발하기 위하여 기존의 C++ 키워드를 대부분 사용 할 수 있지만 .NET 코드 작성을 위해서 추가된 키워드를 제공하고 있다. 그 키워드의 종류는 아래와 같다.

<표 ##. Managed C++의 확장된 키워드>

| 키워드 | 간단한 설명 |
|---|---|
| __abstract | __abstract로 지정한 __gc 클래스는 다른 클래스의 부모 클래스의 역할만 할 수 있다. 즉, __abstract 클래스를 이용 직접 객체를 생성할 수 없다. __abstract로 지정한 클래스도 자신의 멤버함수를 정의할 수는 있다. |
| __box | __box를 이용하면 Unmanaged 형식의 값을 새롭게 생성된 Managed 객체에 복사하여 사용할 수 있다. .NET 용어로 말하자면 Boxing이 발생한다. |
| __delegate | __delegate 키워드는 새로운 __gc 클래스를 정의한다. 이 클래스는 System::MulticastDelegate에서 상속을 받는다. 실질적으로는 C++의 함수 포인터와 같은 용도로 사용될 수 있다. |
| __event | CLR의 이벤트 모델을 지원하기 위한 키워드 |
| __gc | __gc 형식을 정의한다. __gc 형식으로 정의할 수 있는 것에는 class, interface, struct, 배열, 포인터 등이 있다. __gc 형식 클래스의 객체는 .NET의 Managed Heap 메모리에 생성되어진다. |
| __identifier | C++ 키워드로 정의된 이름을 가지는 외부 클래스 등을 이용할 수 있게 한다. |
| __interface | 인터페이스를 정의한다. 인터페이스는 클래스와 달리 모든 멤버 함수가 기본적으로 완전한 가상 함수이다. |
| __nogc | __gc 형식을 지정할 수 있는 모든 곳에 지정이 가능하며 Unmanaged 형식임을 나타낸다. 예를 들어, __nogc로 선언한 클래스의 객체는 기존과 마찬가지로 C++ Heap에 생성된다. 즉, delete를 이용하여 메모리를 해지해주어야 한다. __gc 나 __nogc를 지정하지 않으면 기본적으로 __nogc 형식으로 정의된다. |

| | |
|---|---|
| __pin | Managed 형식 클래스의 객체가 Managed Heap 메모리 가베지 컬렉션(GC)을 하는 동안 제거되거나 주소가 변경되지 않게 지정한다. 이 키워드는 Managed 형식의 객체 포인터를 Unmanaged 형식의 함수 인자 값으로 전달 시 유용하게 사용될 수 있다. |
| __property | Managed 클래스의 프로퍼티를 선언한다. |
| __sealed | __sealed로 지정한 클래스는 다른 클래스의 부모 클래스가 될 수 없다. 또한, 멤버함수의 경우에는 overriding 할 수 없다. |
| __try_cast | 지정한 형 변환을 시도하고 실패 시 예외 처리를 발생시킨다. |
| __typeof | 지정한 변수의 System::Type 정보를 리턴 한다. |
| __value | __value 형의 클래스를 선언한다. |

위 표와 같이 Managed C++는 확장된 키워드를 제공하여 Visual C++를 이용하여 .NET Framework 코드를 작성할 수 있도록 한다. 위에서 보인 각 키워드에 대한 자세한 내용은 이 책의 여러 부분에서 필요 시 각각 설명하도록 하겠고 우선, 가장 중요한 __gc를 이용한 타입 선언에 대해서만 먼저 살펴보도록 하자.

**__gc 클래스에 대하여**

기존의 표준 C++ 언어에서 클래스의 객체는 C++ Heap 메모리에 new를 사용하여 생성하거나 혹은 Stack 메모리에 생성하여 사용하였다. 그러나, __gc로 선언된 클래스는 기존 C++ 클래스와 달리 반드시 CLR이 관리하는 Heap 메모리에 생성된다. 즉, __gc로 선언된 클래스의 경우 언제나 new (사실은 __gc new 임)을 사용하여 CLR의 Managed Heap에 생성해야 하며 Stack에 생성하도록 할 수 없다.

<코드 ##. __gc 클래스의 객체 생성 방법>

```
#using <mscorlib.dll>
#include <tchar.h>
#include <string>          // 표준 C++ string 클래스 헤더
#include <iostream>

using namespace std;     // 표준 C++ 클래스 라이브러리 사용을 위해
using namespace System;

// __gc 클래스 형
__gc class CPort {
```

```
public:

   bool open(String* name) {
      // System::String 클래스의 Concat 멤버함수를 호출하여 문자열을 합친다.
      Console::WriteLine(String::Concat(name, S" Opened"));
      return true;
   }

   bool close() {
      Console::WriteLine(S"Closed");
      return true;
   }

};

// __nogc 클래스 형. __nogc를 생략하여도 동일한 코드이다.
__nogc class CNOGCPort {
public:

   bool open(string& name) {
      cout << name << " opened" << endl;
      return true;
   }

   bool close() {
      cout << "closed" << endl;
      return true;
   }
};

int _tmain(void)
{
   // __gc 클래스 객체 생성 => CLR Managed Heap에 생성
   CPort* port1 = new CPort();      // 실제로는 __gc new 임. __gc 형 생성시는 생략가능
   port1->open(S"COM1");
   port1->close();
```

```
    CPort portTest; // C3149 컴파일 에러. __gc 클래스 객체는 Statck에 생성할 수 없다.

    // __nogc 클래스 객체는 Stack에 생성할 수 있다.
    CNOGCPort port2;
    port2.open(string("COM2"));
    port2.close();

    return 0;
}
```

또한, 기존 C++ 클래스 객체는 new를 이용하여 Heap에 생성한 후에 개발자가 적절한 시점 - 객체를 모두 사용하고 더 이상 필요가 없을 때 - delete를 호출하여 메모리를 해지하도록 하였다. 그러나, __gc 클래스 객체인 경우 CLR의 가베지 컬렉션(이하 GC)에 의해 적절한 시기에 메모리가 자동으로 해지 되어 개발자가 delete를 호출하여 메모리를 해지하는 것에 대해 고려할 필요가 없다.

<코드 ##. __gc 와 __nogc 클래스 객체의 메모리 해지 비교>

```
    // __nogc 클래스의 경우 사용 완료 후 delete를 호출하여 메모리 Free하여야 함
    CNOGCPort* port3 = new CNOGCPort();
    port3->open(string("COM3"));
    delete port3;

    // __gc 클래스는 Garbage Collector가 메모리를 자동으로 Free 함.
    CPort* port4 = __gc new CPort();   // 이렇게도 호출 가능
    port4->open(S"COM4");

    return 0;
```

__gc 클래스의 경우도 기존 클래스와 마찬가지로 생성자와 소멸자를 정의할 수 있다. __gc의 소멸자는 GC에 의해 __gc 객체가 소멸될 때 자동으로 호출되어지나 delete를 호출하여 강제적으로 실행되도록 할 수 있으며 혹은, 직접 소멸자 함수를 호출 할 수도 있다. 이처럼 강제적으로 소멸자 함수를 호출하는 이유로서는 GC가 정확히 어느 시점에 객체의 메모리를 정리할지 알 수 없고 그러므로, 소멸자가 불려지는 시점이 예상외로 늦어질 수 있으므로 개발자가 원하는 시점에 소멸자를 호출하도록 하는 것이다.

<코드 ##. __gc 클래스의 소멸자와 소멸자의 호출 관계>

```
__gc class CPort {
public:
  // 기본 생성자
  CPort() { m_portName = 0; }
  // 생성자
  CPort(String* name) { m_portName = name; }
  // 소멸자. __gc 클래스의 소멸자는 기본적으로 virtual 임.
  ~CPort() { close(); Console::WriteLine(S"~CPort"); }
  // 기타 다른 함수 구현은 동일하여 생략…
private:
  String* m_portName;
}

  // __gc 클래스 delete 호출
  CPort* port1 = __gc new CPort();
  port1->open(S"COM1");
  delete port1; // 이와 같이 강제적으로 소멸자 호출을 시킬 수 있다.
  // 혹은 port1->~CPort(); 와 같이 직접 소멸자 함수를 호출 할 수도 있다.

  // __gc 클래스는 Garbage Collector가 메모리를 자동으로 Free 함.
  // 이 경우 port4를 참조하는 코드가 없으면 GC에 의해 소멸자가 호출되고
  // 객체가 메모리에서 Free 됨.
  CPort* port4 = __gc new CPort();  // 이렇게도 호출 가능
  port4->open(S"COM4");
```

단, __gc 객체의 소멸자가 직접 호출이 되었더라도 그 시점에 객체가 CLR Heap 메모리에서 없어졌다고 단정할 수는 없으며 단지 소멸자 함수 호출에 의해 객체가 내부적으로 사용하는 자원에 대해서만 정리를 했다고 이해하여야 한다. 실제로, __gc의 소멸자를 작성한 경우 개념적으로는 아래 코드와 같이 구현이 이루어진다고 보면 된다.

<코드 ##. __gc의 소멸자와 실제 내부 코드 관계>

```
  // 주의 : 아래 코드는 실제 컴파일 되지는 않는다. 개념을 보이기 위한 코드 임
```

```
// Finalize 멤버 함수 from System::Object
// 이 함수는 Object가 GC에 의해 소멸되는 과정에 자동으로 호출되어
// 객체의 내부적인 자원을 해지할 수 있는 기회를 제공한다.
CPort::Finalize() {
  close();  // 포트 자원을 Free
}

virtual ~CPort() {
  // GC에 의해 다시 Finalize가 호출되는 것을 막는다.
  System::GC::SuppressFinalize(this);
  CPort::Finalize();      // Finalize 함수를 호출하여 내부 자원 정리
}
```

__gc 클래스는 모두 System::Object 클래스를 기본으로 하므로 이 클래스에 해당하는 멤버 함수를 직접 호출 할 수 있다. 또한, .NET Framework의 클래스의 멤버 함수에서 Object 형식의 인자를 요청하는 곳에는 __gc 형의 모든 클래스를 그대로 전달 할 수 있다.

<코드 ##. __gc 클래스와 System::Object 클래스의 관계>

```
// System::Object::GetType을 호출하여 클래스 형을 얻고
// 다시 System::Object::ToString을 호출하여 문자열로 화면에 출력
// 화면에는 port4의 클래스 명인 CPort 문자열이 출력됨
Console::WriteLine(port4->GetType()->ToString());
// BaseType은? => System::Object가 화면에 출력됨
Console::WriteLine(port4->GetType()->BaseType);
```

위 코드에서 **System::Object::GetType** 멤버함수를 호출하면 현재 객체의 형 정보를 **System::Type** 클래스의 인스턴스를 통해 확인할 수 있다. 단순히, port4 변수형을 문자열로 하여 화면에 출력하기 윙해서는 다시 **System::Object::ToString** 멤버함수를 호출하면 된다. 이처럼 __gc로 정의한 모든 클래스는 기본적으로 System::Object를 기본으로 한다는 것과 모든 .NET Framework 클래스들 역시 System::Object를 기본으로 한다는 것을 기억하도록 하자.

**__abstract**를 __gc 클래스에 선언하면 이 클래스는 다른 클래스의 부모 클래스 역할만을 할 수 있으며 스스로 객체를 생성할 수 없게 된다. 이는 인터페이스와 유사한 특징이나 인터페이스와는 달리 자신의 멤버 함수를 정의하거나 멤버 변수를 정의할 수 있다. 즉, 단순히 객체를 생성할 수 없는 클래스로 선언하는 것이다.

<코드 ##. __abstract 클래스와 기존 클래스의 차이>

```
// __abstract 클래스
__abstract __gc class CVirtualPort {
public:
   virtual bool open(String* name) = 0;       // pure virtual function
   virtual bool close() = 0;

   // 멤버 함수 정의가 가능하다.
   String* GetPortName() {
      return m_portName;
   }

protected:
   String* m_portName;   // 멤버 변수 정의가 가능하다.
};

// 이와 같이 __abstract 클래스는 부모 클래스로서의 역할 만 할 수 있다.
__gc class CPort : public CVirtualPort {
      // 코드 생략…
};

// C3622 컴파일 에러 발생, __abstract 클래스는 생성 할 수 없다.
CVirtualPort* virt = new CVirtualPort();

// 상위 __abstract 클래스가 정의한 멤버 함수 호출 가능
CPort* port4 = new CPort();
Console::WriteLine(port4->GetPortName());
```

__gc 클래스를 **__sealed**로 선언하면 __abstract와는 완전히 반대로 다른 클래스의 부모 클래스 역할을 할 수가 없게 된다. 즉, 다른 개발자가 자신의 클래스를 상속하여 기능을 확장하거나 하는 것을 원하지 않는다면 __sealed로 클래스를 지정하면 된다.

<코드 ##. __sealed 클래스는 다른 클래스에서 상속 불가능>

```
// __sealed 클래스
__sealed __gc class CCantDerive {
public:
   bool open() { return true; }
   bool close() { return false; }
};

// C3246 컴파일 에러. __sealed로 선언된 클래스는 상속될 수 없다.
__gc class CPort : public CCantDerive {
    // 코드 생략…
};
```

__gc 클래스의 멤버로서는 당연히 .NET Framework의 클래스나 __gc 형식의 멤버가 포함될 수 있을 것이다. 이에 더불어 Unmanaged 형도 __gc 클래스의 멤버로 포함될 수 있으나 Unmanaged형의 객체를 포함하고자 할 때는 제한이 따른다. 이에 대한 자세한 내용은 CHAP ##에서 살펴보도록 하고 아래에 간단한 코드 예제만 제시하겠다.

<코드 ##. __gc 형의 가능한/불가능한 클래스 멤버>

```
// 이러한 형은 __gc 클래스에 포함된 객체로 사용할 수 있다.
struct RECT {
   int x;
   int y;
   int cx;
   int cy;
};

// 생성자, 소멸자, Copy 생성자, Assignment 생성자, 가상 함수 등을 포함한 형은
// __gc 클래스에 포함된 객체로 사용할 수 없다.
struct RECT2 {
   int x;
   int y;
   int cx;
   int cy;

   // 이 클래스의 객체를 __gc 멤버로 사용하고자 할 때 발생한
```

```
    // 각 컴파일 오류의 원인을 VC++ 컴파일러가 출력한 결과

    // error C3633: cannot define 'm_member8' as a member of managed 'CMemberTest'
    // because of the presence of default constructor 'RECT2::RECT2' on struct 'RECT2'
    RECT2() { };

    // error C3633: cannot define 'm_member8' as a member of managed 'CMemberTest'
    // because of the presence of copy constructor 'RECT2::RECT2' on struct 'RECT2'
    RECT2(RECT2& copy) { };

    // error C3633: cannot define 'm_member8' as a member of managed 'CMemberTest'
    // because of the presence of destructor 'RECT2::~RECT2' on struct 'RECT2'
    ~RECT2() { };
};

// __gc 클래스의 멤버
__gc class CMemberTest {
private:
    int        m_member1;
    long*      m_member2;
    String*    m_member3;
    Int16      m_member4;

    // C3160 컴파일 에러(interior __gc 포인터를 __gc 멤버로 사용 불가. 추후에 설명)
    // Int16*   m_member5;            // Int16은 .NET Framework 클래스 라이브러리의 value
    // string   m_member6;   // C3633 컴파일 에러
    // RECT2    m_member9;   // C3633 컴파일 에러

    string* m_member7;       // 이 경우는 상관 없음. 객체가 아니므로
    RECT       m_member8;  // RECT 구조체는 객체 멤버로 포함해도 문제가 없다.
    RECT2*     m_member10;    // 이 경우는 상관 없음. 객체가 아니므로
};
```

지금까지 간단하게 __gc 클래스 형의 특징에 대해 살펴보았다. 다음 절에서는 기존 C++ 클래스에서는 가능했지만 __gc 클래스에서는 제한 되는 점들에 대해 살펴보도록 한다.

**__gc 클래스의 제한 사항**

__gc 클래스는 기존의 C++ 클래스와 거의 유사한 형태로 코드를 작성할 수 있으나 기존 C++ 클래스에서 약간의 제한 사항이 따른다. 이러한 제한 사항들은 __gc 클래스가 .NET CLR 환경에서 동작하도록 하기 위하여 수반되는 제한 사항들이다.
일단, __gc 클래스는 당연히 일반 C++ 클래스(즉, __nogc 클래스)를 부모 클래스로 두거나 부모 클래스로 사용될 수 없다. 즉, __gc 클래스는 __gc 클래스 간에만 계승을 하거나 상속받을 수 있다.

<코드 ##. 불가능한 클래스간 상속 관계>

```
// __nogc 클래스
class RasManager {
public:
   void Connect() {};
   void Disconnect() {};
   void GetConnectionState() {};
};

// __gc 클래스. __nogc 클래스의 기능을 상속 받기를 원함
// C3253 컴파일 에러 : Managed Type은 Unmanaged Type을 상속 받을 수 없다.
__gc class RasManager2 : public RasManager {
public:
   void CreateRasEntry() {};
};
```

만일, 반드시 기존 C++ 클래스를 부모 클래스로 사용해야 할 경우가 있다면 이 클래스를 감싸는 __gc 클래스를 새로 구현하여 문제를 해결할 수 있을 것이다.

<코드 ##. Wrapper를 이용한 __gc와 __nogc 클래스간 상속>

```
// 기존에 구현하였던 __nogc 클래스
class RasManager {
public:
   void Connect() {};
   void Disconnect() {};
```

```
  void GetConnectionState() {};
};

// 기존 __nogc 클래스를 __gc 타입으로 Wrapping 함.
__gc class RasManagerWrapper {
public:
  RasManagerWrapper() {
    m_rasManager = new RasManager();  // __nogc 클래스 객체 생성
  }

  ~RasManagerWrapper() {
    delete m_rasManager;                    // __nogc 클래스 객체 삭제
  }

  void Connect() { m_rasManager->Connect(); }   // __nogc의 함수를 그대로 호출한다.
  void Disconnect() { m_rasManager->Disconnect(); }
  void GetConnectionState() { m_rasManager->GetConnectionState(); }

private:
  RasManager* m_rasManager;   // __nogc 클래스에 대한 참조 변수를 가진다.
};

// __gc 클래스. __nogc 클래스의 기능을 계승 받기를 원함.
// 이와 같이 __gc로 다시 Wrapping한 클래스에서 계승 받을 수 있다.
__gc class RasManager2 : public RasManagerWrapper {
public:
  void CreateRasEntry() {};
};
```

__gc 클래스의 가장 큰 차이점이라면 __nogc 클래스와 달리 다중 상속을 받을 수 없다는 것이다. __gc 클래스는 부모 클래스로 단 하나의 클래스에서만 상속을 받을 수 있다. 만일 특정 부모 클래스를 지정하지 않는 경우에는 자동적으로 System::Object 클래스를 상속 받게 된다.

<코드 ##. __gc 클래스는 다중 상속 금지>

```
__gc class A {
};

__gc class B {
};

// C2890 컴파일 에러. __gc 클래스는 여러 개의 클래스를 상속 받을 수 없다.
__gc class C : public A, public B {
};
```

단, 클래스와는 달리 복수 개의 인터페이스를 상속 받아 구현할 수 있다.

<코드 ##. 복수개의 인터페이스는 상속 가능>

```
__gc class A {
};

__gc __interface B {
};

__gc __interface C {
};

// 하나의 클래스와 여러 개의 인터페이스 상속 가능
__gc class D : public A, public B, public C {
};
```

__gc 클래스에 개발자가 직접 구현한 소멸자가 없는 경우에는 해당 객체에 대하여 delete를 수행할 수 없다. __gc 클래스 객체의 경우네 delete를 호출하는 것은 __nogc 클래스와는 달리 객체를 메모리에서 삭제하는 것이 아니라 단순히 객체가 사용하는 내부 자원을 정리하기 위해 Finalize 과정을 실행하는 것이므로 소멸자가 정의되어 있는 __gc 클래스에 대해서만 호출이 가능하다.
Managed 형식의 클래스에는 sizeof 나 offsetof와 같은 연산자를 수행할 수 없다. .NET Framework의 클래스는 바이너리 형태로 제공될 수 있고 또한, 버전에 따라 그 크기가 계속 변경될 수 있으므로 클라이언트 코드에 클래스의 크기를 특정 값으로 한정하는 형태의 코드는 작성할 수 없다.

<코드 ##. sizeof 연산자 사용 불가>

```
__gc class A {
};

// C2847 컴파일 에러. __gc 형식의 클래스에는 sizeof 연산을 할 수 없다.
int a = sizeof(A);
```

또한, __gc 클래스의 멤버 함수는 const나 volatile등으로 지정할 수 없다.

<코드 ##. const, volatile 멤버 함수 지정 불가>

```
__gc class A {
public:
  A() { };
  // C3842 컴파일 에러. __gc 클래스 멤버 함수에 const 지정 불가
  void methodA() const { };
private:
  int a;
};
```

이와 같이 __gc 클래스는 기존 C++와 다른 제한 사항을 가지고 있으나 실제 프로그램 코드 개발에 문제를 가져올 제약 사항은 아니다. 이러한 제한 사항은 단순히 기존 C++ 코드 작성 습관을 약간 변경시키면 될 것이고 지속적으로 .NET Managed 코드를 개발하면서 자연히 습관적으로 익혀질 부분이라 생각된다. 이외에도 몇 가지 세부적인 제한 사항들이 더 있으나 자세한 내용은 살펴보지 않겠다. 이에 대해 확인해보고자 하는 독자 여러분들은 Managed Extensions for C++ Specification 문서를 참고하기 바란다.

**__gc 배열에 대하여**

__gc 배열도 __gc 클래스와 마찬가지로 .NET CRL에 의해 관리되는 Managed Heap에 생성된다. 기본적으로 모든 __gc 배열에 저장될 수 있는 요소는 __gc 클래스나 구조체에 대한 포인터 혹은 __value 형의 데이터, 기본 C++ 데이터 형 등이다.
Managed C++ 언어를 사용하여 배열을 정의하는 방법에는 크게 __nogc 배열과 __gc 배열로 구분할 수 있는데 __nogc 배열은 기존 C++ 배열과 동일하다고 보면 된다. __gc 배열은

실제로는 .NET System::Array 클래스형의 객체로 이 클래스가 제공하는 모든 멤버를 사용할 수 있다. 아래 코드는 C++ 기본 데이터 형인 int에 대하여 __nogc 배열의 경우와 __gc 배열의 경우를 비교하였다.

<코드 ##. int 형에 대한 __nogc 배열과 __gc 배열 비교>

```
// __nogc array. C++ 기본 형은 그대로 정의하면 __nogc 배열임
int iArr1[10];
// __gc array. C++ 기본 데이터 형은 __gc 배열로 사용할 수 있다.
int iArr2 __gc[] = new int __gc[10];

// 배열 범위를 벗어났지만 런타임 에러는 발생하지 않는다.
iArr1[10] = 20;
// 배열 범위를 벗어나면 'System.IndexOutOfRangeException' 런타임 에러 발생
iArr2[10] = 30;
```

위와 같이 기본 C++ 데이터형의 경우 __nogc 배열에 대해서는 별도의 코드 구분 없이 그대로 배열을 선언하면 되고 __gc 형의 배열을 정의하는 경우에만 __gc 키워드를 사용한다. __gc 배열 역시 __gc 클래스의 경우와 마찬가지로 별도로 delete를 호출하여 메모리 해지를 할 필요가 없다. (왜? GC가 알아서 처리하므로) 또한, __nogc 배열에 대해서는 배열의 범위를 벗어나는 동작에 대해서 코드 실행 중 확인이 되지 않지만 __gc 배열의 경우는 범위를 벗어난 동작에 대하여 System.IndexOutOfRangeException 런타임 에러가 발생한다.
__value 형의 경우도 __nogc 배열이나 __gc 배열 모두에 저장될 수 있으나 C++ 기본 데이터 형의 선언 방법과는 반대로 __nogc 배열로 선언하는 경우에만 __nogc 키워드를 지정해 주어야 한다.

<코드 ##. __value 형의 __nogc 배열과 __gc 배열 비교>

```
// value 타입의 경우는 __nogc를 지정해야 __nogc 배열로 선언 가능
Int16 iArr3 __nogc[10];
// 별도의 __gc 지정 없이도 __gc 배열임
Int16 iArr4[] = new Int16[10];

iArr3[11] = 30;   // 런타임 에러 발생하지 않는다.
iArr4[11] = 40;   // 런타임 에러 발생
```

__value 형을 이용한 __nogc, __gc 배열의 경우도 마찬가지로 __gc 배열을 사용하는 경우에만 배열 범위가 확인되어 런타임 에러가 발생한다.
__gc 클래스나 구조체에 대해서는 반드시 __gc 배열을 이용하여야 한다. __gc 클래스는 언제나 CLR의 Managed Heap에 생성되어야 하므로 __nogc 배열과 같이 CLR Managed Heap이 아닌 메모리에 생성될 수 없다. 즉,

```
// __nogc 배열은 __gc 타입 클래스를 사용할 수 없다.
String*  strArray __nogc[10];
```

위와 같은 코드는 컴파일 오류 C2728을 발생시켜 코드가 생성되지 않는다. __gc 형의 클래스는 별도의 키워드 지정을 하지 않아도 __gc 배열로 정의된다.

<코드 ##. String __gc 클래스에 대한 __gc 배열 처리 및 데이터 정렬>

```
// __gc 클래스 포인터에 대한 __gc 배열
String* gcArr2[] = new String*[5];

// 각 배열 요소에 실제 String 객체 생성
gcArr2[0] = new String(S"Test1");
gcArr2[1] = new String(S"Test3");
gcArr2[2] = new String(S"Test2");
gcArr2[3] = new String(S"Test5");
gcArr2[4] = new String(S"Test4");
// Array는 Sort 멤버함수로 정렬이 가능하다.
Array::Sort(gcArr2);

// Array의 Length 멤버로 배열의 요소가 몇 개인지 확인가능 하다.
for (int i=0; i < gcArr2->Length; i++) {
  // 출력해보면 Test1부터 Test5까지 순서대로 문자열이 출력된다.
  Console::WriteLine(gcArr2[i]);
}
```

위 코드에서 볼 수 있듯이 __gc 배열은 System::Array의 객체이기 때문에 Array의 모든 멤버함수 및 멤버를 이용할 수 있다. __gc 클래스를 직접 정의하는 경우 내부에 C++ 기본 형의 배열을 선언하고자 한다면 반드시 __gc 혹은 __nogc 배열임을 지정해주어야 한다.

<코드 ##. __gc 클래스 내부의 C++ 기본 형 배열 선언>

```
__gc class CA {
public:
  CA() {
    m_value = 0;
    m_arr1 = new int __gc[10];
  }

  CA(int val) { m_value = val; }

  int GetValue() {
    return m_value;
  }

private:
  int m_value;
  int m_arr1 __gc[];        // __gc 클래스 내부에서 C++의 기본 형에 대한 배열 선언 시
  int m_arr2 __nogc[10];   // 반드시, __gc 혹은 __nogc로 지정해주어야 한다.
};
```

__gc 배열을 다차원 배열로 선언하고자 할 때는 기존 C++ 문법과 달리 아래와 같은 문법을 사용한다.

<코드 ##. __gc 배열의 다차원 배열 선언>

```
// , (콤마)를 이용하여 다차원 배열 선언
char   cArr1 __gc[,] = new char __gc[2,2];

cArr1[0,0] = 'A';
cArr1[0,1] = 'B';
cArr1[1,0] = 'C';
cArr1[1,1] = 'D';
```

위에서 언급한 바와 같이 __gc 배열에는 __value 형과, __gc 클래스에 대한 포인터, C++ 기본 데이터 형을 사용할 수 있다. 그러므로, 아래와 같은 코드들은 모두 컴파일 오류가 발생

하는 경우이다.

<코드 ##. __gc 배열 혹은 __nogc 배열로 사용할 수 없는 경우>

```
// __nogc class
__nogc class CB {
};

// C2728 컴파일 에러. __gc 클래스를 __nogc 배열로 사용할 수 없다.
String*   strArray __nogc[10];
// C2691 컴파일 에러. C++ 기본형에 대한 포인터 배열을 __gc로 할 수 없다.
char* cArr3 __gc[ ] = new char* __gc[10];
// C2691 컴파일 에러. 포인터 형을 __gc 포인터로 변경하여도 마찬가지로 불가.
char __gc* cArr3 __gc[ ] = {"test1","test2"};
// C2691 컴파일 에러. __nogc 클래스는 __gc 배열로 정의할 수 없다.
CB nogcArr2 __gc[ ] = new CB __gc[10];
```

**__gc 포인터에 대하여**

.NET CLR 환경에서 사용되는 메모리는 크게 나누어 CLR Managed Heap과 Unmanaged 메모리로 구분할 수 있다. 이 때 Managed Heap 메모리에 존재하는 데이터에 대한 포인터는 기존 C++의 포인터로 참조할 경우 CLR의 GC이 정확하게 메모리 해지 작업을 수행할 수 없으므로 새로운 포인터 타입인 __gc 포인터가 정의되었다. __gc 포인터는 CLR Managed Heap상에 위치하는 메모리를 참조하기 위해 사용되며 GC에 의해 메모리 해지를 정확히 수행하기 위한 자료로 활용될 수 있다. 기존의 C++ 일반적인 포인터를 나타내기 위해 __nogc 포인터라고 하고 이 포인터는 Stack이나 C++ Heap 메모리와 같이 CRL에 관리되지 않는 일반적인 메모리 영역을 참조하기 위해 사용된다.

<코드 ##. __gc 포인터와 __nogc 포인터의 예>

```
// __value 형 구조체
__value struct Score {
   int Math;
   int Physics;
   int English;
};
```

```
// __gc 클래스
__gc class Student {
public:
   Student() { StudentName = 0x0; }
   Student(String* name) { StudentName = name; }

   Score     SubjectScore;
   String* StudentName;   // __gc 클래스에 대한 포인터는 기본적으로 __gc 포인터
};

// main 함수 내부 코드…

Student* student1 = new Student(S"Bob"); // __gc 포인터
Score __gc* pScore1 = &student1->SubjectScore; // __gc 포인터
// C2440 에러. __gc 클래스 내부의 __value는 CLR Heap에 존재하므로
// __nogc 포인터로 지정 불가능
Score __nogc* score1 = &student1->SubjectScore;

Score score2; // __value 타입의 객체는 Stack 메모리에 생성 가능하다
Score __nogc* pScore2 = &score2;   // Stack 메모리에 대한 __nogc 포인터
// CLR Heap 메모리에 생성되지 않은 __value 타입도 __gc 포인터로 지정가능.
// 단, CLR GC 과정에서 CLR Heap 메모리에 위치하지 않는 __gc 포인터에 대해서는
무시함.
// 즉, __nogc* 형은 __gc* 로 변환가능. 단, 반대의 변환은 __pin 키워드 필요.
Score __gc* pScore2_1 = &score2;
```

__gc 혹은 __nogc를 포인터 선언 시마다 꼭 지정할 필요는 없다. 선언하는 데이터의 형에 따라 자동적으로 __gc 혹은 __nogc 포인터가 사용되어진다. 즉, __gc 클래스에 대한 포인터는 기본적으로 __gc 포인터이고 int와 같은 기본 C++ 형에 대한 포인터는 __nogc 포인터가 되는 것이다.

<코드 ##. __gc 와 __nogc 포인터의 기본 지정 방법>

```
int i1;
int __nogc* pI1 = &i1;
```

```
int* pI1_1 = &i1; // 기본적으로 __nogc 포인터 임

String __gc* pStr1 = __gc new String(S"Test");
Object* pObj1 = pStr1;   // 기본적으로 __gc 포인터 임
```

__gc 포인터는 크게 **Whole** 포인터와 **Interior** (내부 포인터정도로 해석) 포인터로 구분할 수 있다. 이렇게 구분을 짓는 이유는 GC에 의해 Heap 메모리를 정리할 때 현재 참조되고 있는 메모리 영역을 정확히 구분하기 위함으로 볼 수 있다. Whole 포인터는 지금까지 계속 살펴본 형태로 __gc 타입의 (전체) 객체를 직접 참조하는 포인터이다. 이와 달리, Interior 포인터는 __gc 객체의 내부 객체를 참조하는 포인터나 혹은 __value 타입, C++ 기본 데이터형을 가리키는 포인터 등이다. Interior 포인터에 대한 참조를 정확히 유지하는 데에는 GC 과정에 부하를 많이 줄 수 있으므로 다음과 같은 범위에서만 Interior 포인터를 정의할 수 있도록 한정 짓고 있다.

1. 지역 변수
2. 함수의 인자
3. 함수의 리턴 값

즉, 위의 경우가 아닌 위치에 Interior 포인터를 선언할 수 없다.

<코드 ##. Interior 포인터를 사용할 수 있는 위치>

```
// 함수의 인자로 Interior 포인터 선언 가능 혹은 함수의 리턴 타입으로도 가능
int __gc* InteriorTestFunc(int __gc* iParam) {
  return iParam;
}

int _tmain(void)
{
  // 코드 생략…
  int __gc* pMath3 = &score2.Math;   // 로컬 변수로 Interior 포인터 선언 가능
}
```

앞에서도 말한 바와 같이 CLR Managed Heap 메모리를 관리하는 GC는 모든 __gc 포인터에 대해 정확히 관리를 해주어야 하므로 __gc 포인터에 대한 다른 포인터 형으로의 변환 과정에는 기존 C++ 포인터 형 변환과 다른 제한 사항이 따른다. 특히, __gc 포인터는

__nogc 타입의 포인터로 변환될 수 없다. 만일, __gc 포인터를 __nogc 포인터로 변환이 가능하여 __nogc 포인터에 의해 CLR Heap 메모리가 참조되어진다면 GC는 __nogc 포인터의 참조에 대한 정보가 전혀 없으므로 어느 시점에 자동적으로 이 메모리 영역을 정리하게 될 것이다. 그러면, __nogc로 이 메모리를 참조하고 있던 코드는 엉뚱한 메모리를 참조하게 되어 예측할 수 없는 동작이 일어날 것이다.

<그림 ##. __gc 포인터를 __nogc로 변환 불 가능한 이유>

Managed C++의 형 변환을 위한 연산자는 기존 표준 C++가 제공하던 형 변환 연산자를 그대로 이용한다. 그러나, Managed C++의 포인터 형 변환은 사용 시 주의해야 할 제약 사항이 따른다. 우선 **dynamic_cast**의 경우를 보면 기존 C++와 마찬가지로 상위 클래스 형을 상속 받은 하위 클래스 형으로 변환 시 사용된다. 형 변환이 성공하면 해당 클래스 객체에 대한 포인터 참조를 얻을 수 있고 실패하면 NULL 값을 얻는다. 단, dynamic_cast는 __gc 타입 포인터에만 사용할 수 있고 __value 타입의 포인터에는 사용할 수 없다.

<코드 ##. dynamic_cast 예제 코드>

```
void DynamicCastTest(Object* param)
{
  // 원 형으로 변환 정상적인 경우
  Student* student1 = dynamic_cast<Student*>(param);
  // 전혀 관계없는 형으로 변환 비정상적인 경우
  String*  str1 = dynamic_cast<String*>(param);

  // 변환 후 값이 null 이면 잘못된 변환이다.
  if ( str1 == 0 ) {
    System::Console::WriteLine("Invalid dynamic_cast");
  }
}
```

**__try_cast**는 기존 C++에는 없던 연산자로 dynamic_cast와 유사하게 동작하나 형 변환이 잘못된 경우 **System::InvalidCastException** 예외사항(Exception)을 발생시킨다. 즉, 형 변환의 정상 실행 여부를 포인터 값이 NULL인지를 확인하는 것이 아니라 try, catch 문을 이용하여 예외 사항 처리를 해주면 된다. 마찬가지로 __gc 타입 포인터에만 사용할 수 있고 __value 타입의 포인터에는 사용할 수 없다.

<코드 ##. __try_cast 예제 코드>

```
void DynamicCastTest(Object* param)
{
  Student* student2;
  String* str2;

  try {
    student2 = __try_cast<Student*>(param); // 원 형으로 변환. 정상적인 경우
    str2 = __try_cast<String*>(param); // 잘못된 형 변환. Exception 발생
  } catch (System::InvalidCastException* e) {
    System::Console::WriteLine(e->ToString());
  }
}
```

**static_cast**를 이용하면 dynamic_cast나 __try_cast와 달리 실행 시 변환되는 형에 대한 체크 없이 포인터 형 변환을 할 수 있다. 이 변환 과정은 런타임 체크 과정이 없기 때문에 다른 변환에 비하여 수행 속도가 빠른 장점이 있다. 예를 들어, 컬렉션에 많은 수의 Object* 참조가 유지되고 있을 때 모든 참조 포인터를 다시 원래 형으로 변환하기 위하여 static_cast를 호출하면 성능 상의 이점이 클 것이다. 단, static_cast를 사용하는 경우 런타임 체크과정이 생략되어 __gc 포인터의 형을 엉뚱한 형으로 변환이 가능해지고 이러한 경우 GC이 잘못된 동작을 수행할 수 있는 위험성이 있다. 그러므로, 코드 개발 단계에서는 __try_cast와 같은 런타임 체크 기능이 있는 코드를 사용하고 개발 완료단계에 static_cast로 변경하여 안정성과 수행 성능을 동시에 충족하도록 할 수 있다.

<코드 ##. static_cast 예제 코드>

```
__gc class Derived1 {
};

__gc class Derived2 : public Derived1 {
};

// static_cast 테스트
Derived2* derived2 = new Derived2();
StaticCastTest(derived2);
```

```
// static_cast 예제
void StaticCastTest(Object* param)
{
  // 실제로 관련 있는 형으로의 변환은 문제가 없다.
  Derived2* derive2 = static_cast<Derived2*>(param);
  Derived1* derive1 = static_cast<Derived1*>(param);
  // 완전히 잘못된 경우. 이 경우 __try_cast를 사용하면 Exception 발생.
  // 기존 C++ 코드에서 static_cast 같은 경우 전혀 관계없는 형간의 변환은
  // 컴파일 오류가 발생했지만 지금 같은 경우 Object* 형은 모든 .NET 클래스의
  // 기본 Base가 되므로 어떤 형으로든 static_cast를 써서 변환이 가능하다.
  String* str1 = static_cast<String*>(param);
}
```

reinterpret_cast는 상호 관계가 전혀 없는 두 포인터 형간의 변환에 이용될 수 있으며 __gc 포인터에는 반드시 필요한 경우가 아니라면 사용하지 않는 것이 좋다. 일반적으로 단순한 형태의 클래스 포인터간 변환이나 __value 타입 포인터의 변환을 위해 사용한다.

<코드 ##. reinterpret_cast 예제 코드>

```
// __value 형 구조체
__value struct Score {
  int Math;
  int Physics;
  int English;
};

Score score3;
score3.Math = 90;
score3.English = 80;
score3.Physics = 70;
ReinterpretCastTest(&score3);

// reinterpret_cast 예제
// __value 타입의 포인터는 System::Void* 형 변환이 가능하다.
void ReinterpretCastTest(System::Void* param)
```

```
{
  Score* score1 = reinterpret_cast<Score*>(param);   // 가능.
  Score* score2 = static_cast<Score*>(param); // 가능.
  int __gc* iScore3 = reinterpret_cast<int __gc*>(param); // 심지어 이것도 가능.

  // 차례대로 90, 70, 80 이 출력됨
  // NOTE: Interior 포인터의 경우는 + 혹은 - 연산이 가능하다.
  // __box는 기본 C++ 형을 System::Object 객체로 Copy 하여 사용한다.
  System::Console::WriteLine("{0}", __box(*iScore3));
  System::Console::WriteLine("{0}", __box(*(iScore3 + 1)));
  System::Console::WriteLine("{0}", __box(*(iScore3 + 2)));
}
```

**const_cast**는 기존 C++의 변환과 마찬가지로 const형 혹은 volatile형 포인터를 그렇지 않은 형으로 변환 시 사용된다.

<코드 ##. const_cast 예제 코드>

```
int iVal = 10;
const int __gc* iPtr1 = &iVal;
*iPtr1 = 20;   // C2166 에러. const 형 값을 변경할 수 없다.

int __gc* iPtr2 = const_cast<int __gc*>(iPtr1);
*iPtr2 = 20;
System::Console::Write("{0}", __box(iVal)); // 변경된 20 이 출력됨
```

지금까지 __gc 포인터를 설명하면서 __gc 포인터는 __nogc 포인터로 변환이 불가능하다고 설명하였다. 그러나, 사실은 변환을 할 수 있는 방법이 있으면 이러한 방법이 있어야 Managed 코드와 Unmanaged 코드를 동시에 사용하는 프로그램 작성이 실질적으로 의미가 있을 것이다. (Unmanaged 함수에서 __nogc 포인터를 인자로 받아야 하는 경우 이 인자로 __gc 포인터 값을 사용해야 할 수도 있다.)
__gc 포인터를 __nogc 포인터로 변환하기 위해서는 GC에 의해 __gc 포인터가 가리키는 메모리 영역이 정리되지 않도록 해주어야 하는데 **__pin** 키워드에 의해 이러한 효과를 얻을 수 있다. __gc 포인터를 __pin과 함께 선언하면 그 메모리 영역은 pinned(고정)되었다라고 하며 이 포인터에 대해서는 __nogc 포인터로의 변환을 할 수 있게 된다.

<코드 ##. __pin을 이용한 __gc 포인터 __nogc 포인터로의 변환>

```
// __pin 테스트. iParam은 __nogc 포인터 임
void PinningTest(int* iParam) {
   cout << *iParam << endl;
}

// C2664 에러. __gc 포인터를 __nogc 포인터로 변환 불가능
PinningTest(&student1->SubjectScore.Math);

// __pin에 의해 __gc 포인터가 참조하는 메모리를 고정
// 객체의 일부분을 __pin 하였지만 전체 객체가 메모리에서 고정 됨
int __pin* pMath4 = &student1->SubjectScore.Math;
PinningTest(pMath4);      // __nogc 포인터로 변환 가능
pMath4 = 0;               // 메모리의 __pin의 고정 속성 제거
```

__pin에 의해 고정된 메모리는 __pin으로 선언한 포인터 변수가 선언 범위를 벗어나거나 혹은 0 값을 지정함으로써 고정된 속성이 사라지고 다시 GC에 의해 메모리 정리가 가능해진다. __pin의 고정된 속성이 사라진 후에는 Unmanaged 코드에서 이 포인터를 사용하는 동작을 수행하지 말아야 한다.
Managed 객체의 일부분을 __pin으로 고정시키면 전체 객체의 메모리가 고정되는 효과를 얻을 수 있다. 즉, 클래스 객체의 특정 멤버를 고정시킨다거나 혹은 배열의 한 요소를 고정시키게 되면 전체 클래스 객체 혹은 전체 배열의 메모리를 고정시키게 된다.

<코드 ##. 배열의 일부분을 __pin하는 경우 전체 메모리가 고정됨>

```
#include <cstdlib>            // printf 함수

// Array __pin 테스트
Byte arr[] = new Byte[4]; // Managed 배열
arr[0] = 'C';
arr[1] = '+';
arr[2] = '+';
arr[3] = '0';
Byte __pin * p = &arr[1]; // 전체 배열이 고정 됨
unsigned char * cp = p;    // __nogc 포인터로 변환 가능
```

```
printf("%s₩n", cp);          // Unmanaged 함수 인자로 사용 가능
```

Managed C++의 __gc 포인터는 지금까지 살펴본 것처럼 GC의 메모리 관리 기능을 지원하기 위해 존재하며 기존의 __nogc 포인터 다른 특성이 존재한다. __gc 포인터를 정확히 이해하기 위해서는 .NET CLR의 GC 과정 전체를 이해하는 것이 중요하나 이 장에서는 이에 대한 자세한 내부적 동작을 다루지는 않고 단순히 코드 작성의 관점에서면 살펴보았다. 뒤에 CHAP ##는 .NET CLR의 내부적인 구조에 대해 설명하면서 GC가 어떻게 동작하는지를 살펴보므로 이 장을 참조하여 __gc 타입의 의미에 대해 좀 더 고찰해볼 기회를 얻을 수 있을 것이다.

**정리**

Managed C++는 기존 C++의 확장된 언어로 .NET Framework에서 동작하는 코드를 작성하기 위하여 Microsoft가 새롭게 정의한 것이다. Managed C++가 제공하는 여러 키워드 중에 가장 중요한 의미를 가지는 __gc 키워드는 .NET CLR GC에 의해 관리되는 데이터 형을 선언하기 위해 사용되며 __gc 클래스, __gc 구조체, __gc 배열, __gc 포인터 등 기존 C++가 제공하는 기본적인 데이터형 마다 선언할 수 있다.
__gc 클래스 객체는 CLR Heap 메모리에 생성되며 메모리 관리가 GC에 의해 이루어진다. 기존 C++ 클래스와 비슷한 구조를 가지나 몇 가지 제약사항이 따르므로 코드 작성 시 이에 대한 주의가 필요하다. __gc 배열의 경우 역시 .NET CLR Heap 메모리에 생성되며 GC에 의해 관리된다. 기존 배열과 달리 실행 시 경계 조건이 체크되어 배열의 경계를 벗어나는 경우에는 런타임 에러가 발생한다. __gc 포인터는 CLR Heap 메모리 내부에 존재하는 객체를 참조하기 위해 사용되며 __gc 포인터에 대한 정보를 알고 있으므로 GC에 의한 메모리 정리가 정확히 이루어질 수 있게 된다. __gc 포인터는 기본적으로는 __nogc 포인터형으로 변환이 불가능하나 __pin을 이용하여 특정 메모리를 고정시킬 수 있고 이 경우 __nogc 포인터로의 형 변환이 가능해진다.
이 장에서 살펴본 Managed C++ 내용은 실제 .NET 프로그램을 작성하기 위한 언어적인 기본 개념이며 계속적으로 .NET 코드를 작성하면서 자연스럽게 그 구조를 익히게 될 것이다. 이 장의 내용이 완전히 이해가 가지 않는 독자들은 우선 Part II의 Windows 프로그램 작성 부분을 먼저 살펴보면서 실질적인 코드 작성을 먼저 익히고 후에 이 장의 내용을 다시 확인하면 좀 더 이해가 쉬울 것으로 생각된다.

**실전 연습**

1. 3명 학생의 과학, 수학, 영어 과목에 대한 점수를 Console에서 입력 받은 후 학생별 평균 점수를 출력하는 프로그램을 작성한다. 이 때 학생에 대한 클래스를 __gc

클래스로 정의하고 학생 클래스는 과목별 점수를 나타내는 int 형의 배열을 가지도록 설계한다. 또한, 학생 클래스에 GetAverage라는 멤버 함수를 정의하여 학생 별 평균치를 알아올 수 있도록 한다. 학생 3명에 대한 데이터를 관리하기 위해 마찬가지로 학생 클래스에 대한 배열을 이용한다.

2. 특정 텍스트 문서 파일을 파일에서 읽어 문자열을 단어 단위로 나누어 각 단어가 몇 개나 파일에 존재하는지 출력하는 프로그램을 작성한다. 각 단어를 관리하기 위한 __gc 클래스를 디자인하고 배열을 이용 모든 데이터를 관리하도록 한다. 그리고, 아직 File I/O에 대한 .NET 클래스 라이브러리를 살펴보지 않았으므로 파일 처리는 표준 C++ 라이브러리를 활용하도록 한다.

**실전 연습에 대한 힌트**

1. 1번 문제에 대한 힌트

우선 Student 클래스의 내부에 3개 과목 점수에 대한 값을 저장해 둘 __gc 배열을 선언한다. 그리고, 각 배열에 값을 지정할 멤버함수와 평균값을 얻어오는 멤버 함수를 구현한다. 예제 코드를 제시한다면 아래와 같은 형태일 것이다.

```
// 학생 클래스
__gc class Student {
public:
  Student() {
    // __gc 배열 생성. __gc 배열은 new를 사용 CLR Heap 메모리에 생성 됨.
    m_scores = new int __gc[3];
  }
  void SetScores(int index, int score);
  int GetAverage();       // 3과목에 대한 평균 점수를 얻는다.

private:
  int m_scores __gc[]; // 3과목에 대한 점수를 저장하는 배열
};

// 점수를 설정
void Student::SetScores(int index, int score)
{
```

```
    // range check
    if (index >= m_scores->Length && index < 0) return;
    m_scores[index] = score;
}

// 평균 점수를 얻는다.
int Student::GetAverage()
{
    int sum = 0;

    for (int i = 0; i < m_scores->Length; i++) {
        sum += m_scores[i];
    }

    return (sum / m_scores->Length);
}
```

3 학생에 대한 각 과목별 점수를 입력 받아야 하므로 Student 클래스에 대한 배열을 선언하여 사용한다.

```
// __gc 배열 생성. __gc 배열은 new를 사용 CLR Heap 메모리에 생성 됨.
// 기본적으로 __gc 타입의 경우 별도로 __gc를 지정하지 않아도 __gc 배열 임.
Student* students[] = new Student* [3];
```

콘솔에서 각 점수를 입력 받기 위해서는 **Console::ReadLine** 멤버 함수를 이용한다. 이 때 입력 받는 데이터 형이 문자열이므로 이 값을 int 형으로 변환이 필요할 것이다. String 클래스는 **IConvertible** 인터페이스를 구현하고 있으므로 **ToInt32** 멤버함수를 제공한다.

```
String* sScore = Console::ReadLine();
// 문자열을 int 형으로 변환
NumberFormatInfo* numFmt = new NumberFormatInfo();
int iScore = sScore->ToInt32(numFmt);
// Student 객체에 점수 값을 지정한다.
students[i]->SetScores(j, iScore);
```

ToInt32 멤버함수는 변환 시 포맷 정보를 제공하기 위하여 **IFormatProvider** 인터페이스를

구현하는 객체를 인자로 받는다. 위 코드에서는 NumberFormatInfo 클래스의 객체를 전달하였다. 정상적으로 각 학생의 과목별 점수를 입력 받았으면 GetAverage 함수를 호출하여 최종 평균 값을 출력하면 된다.

```
// 평균치를 출력한다.
for (int i = 0; i < students->Length; i++) {
    Console::WriteLine("Students Number [{0}] : {1}",
        __box(i), __box(students[i]->GetAverage()));
}
```

## 2. 2번 문제에 대한 힌트

우선 WORD 개수와 WORD를 저장하는 클래스를 디자인해보도록 한다. 각 데이터를 저장하는 자료 구조는 간단히 배열을 사용하는 것으로 한다. 예제 코드 클래스 선언은 아래와 같다.

```
// 특정 단어별 개수 정보를 관리하는 클래스
__gc class WordCounter {

  // 내부 constant 값. 배열의 크기를 증가할 필요가 있을 때 증가치 값
  static const int array_inc_size = 20;

public:

  WordCounter() {
    m_lastIndex = 0;
    m_words = new String* [array_inc_size];
    m_counts = new int __gc[array_inc_size];
  }

  void AddWord(String* word);        // 단어를 저장한다.
  int GetWordsCount();               // 현재 저장된 단어의 총 개수를 얻는다.
  String* GetWord(int index);        // 특정 인덱스의 단어 문자열을 얻는다.
  int GetWordFrequency(int index); // 특정 인덱스의 단어 빈도수를 얻는다.
```

```
private:
  String* m_words[];          // 각 단어의 문자열을 저장해두는 배열
  int     m_counts __gc[];    // 각 단어별 빈도 수를 저장해두는 배열
  int     m_lastIndex;        // 현재 데이터를 저장할 배열의 인덱스
};
```

위 클래스에서 가장 주의를 기울여야 할 부분은 AddWord 멤버함수의 구현이다. 현재 단어에 대한 정보를 관리하기 위해 초기 크기가 20인 배열을 사용하고 있는데 어떤 문서에 포함된 총 단어의 개수가 얼마나 될지 알 수 없으므로 배열의 크기를 동적으로 증가시켜야 한다. 그러나, System::Array 클래스는 배열의 동적 증가에 관련된 기능을 제공하지 않으므로 개발자가 직접 이에 관련된 코드를 작성해주어야 한다. 가장 간단한 방법으로 동일한 크기의 임시 배열을 생성하여 이 배열에 값을 모두 복사하여 두고 원래 배열의 크기를 증가시킨 후 다시 임시 배열의 값을 원래 배열로 복사하는 방법이 있을 수 있다. 이러한 동작은 수행 성능에 영향을 많은 영향을 끼치므로 배열의 초기 크기 및 증가 크기를 적당한 값으로 설정하는 것이 좋겠다. 예제 코드는 아래와 같다.

```
// 특정 단어를 배열에 추가한다.
void WordCounter::AddWord(String* word)
{
  // 현재 존재하는 단어이면 개수를 하나 증가한다.
  // Array::IndexOf 멤버함수는 2번째 인자가 1번째 인자인 배열의 어느 위치에
  // 있는가를 알려준다. 만일 배열에 그러한 데이터가 없다면 -1을 리턴.
  int index = Array::IndexOf(m_words, word);
  if ( index != -1 ) {
    ++m_counts[index];
  } else {

    // 배열의 크기가 작다. 공간을 더 확보해야 함.
    if ( m_lastIndex == m_words->Length ) {
      // 일단 현재 데이터를 복사해 둘 버퍼 생성. 현재 배열의 크기와 동일하게.
      String* m_tmp1[] = new String* [m_lastIndex];
      int m_tmp2 __gc[] = new int __gc[m_lastIndex];
      // 배열의 데이터 복사 (from 현재 데이터 버퍼 to 임시 버퍼)
      Array::Copy(m_words, m_tmp1, m_lastIndex);
      Array::Copy(m_counts, m_tmp2, m_lastIndex);
      // 배열 확장 후 다시 데이터 복사 (from 임시 버퍼 to 확장된 데이터 버퍼)
```

```
        m_words = new String* [m_lastIndex + array_inc_size];
        m_counts = new int __gc[m_lastIndex + array_inc_size];
        Array::Copy(m_tmp1, m_words, m_lastIndex);
        Array::Copy(m_tmp2, m_counts, m_lastIndex);
      }

      m_words[m_lastIndex] = word;
      ++m_counts[m_lastIndex];
      ++m_lastIndex;
    }
}
```

표준 C++ 라이브러리의 ifstream 클래스를 이용하여 특정 파일의 내용을 읽어와 단어별로 WordCounter 클래스의 객체에 저장하는 코드를 작성한다. 일단, 사용자에게 파일명을 입력 받아 파일을 연다.

```
#include <fstream>

using namespace std;
using namespace System::Runtime::InteropServices;

// 코드 생략…

// 파일 명을 입력 받는다.
Console::WriteLine(S"Input file name : ");
String* fileName = Console::ReadLine();

// 파일명을 이용하여 파일을 Open한다.
ifstream inFile;
// String 객체를 ANSI 문자열의 형태로 Native Heap 메모리에 복사한다.
IntPtr strHglbl = Marshal::StringToHGlobalAnsi(fileName);
inFile.open(static_cast<char*>(strHglbl.ToPointer()));
Marshal::FreeHGlobal(strHglbl); // 할당한 Native Heap 메모리 삭제
```

위 코드에서 입력 받은 문자열은 String 객체이고 ifstream 클래스의 open 멤버함수에 필요한 인자는 const char* 형이므로 그대로 사용하지는 못하고 변환 방법이 필요하다. 위 코드

에서는 .NET 클래스 라이브러리가 제공하는 System::Runtime::InteropServices::Marshal 클래스의 StringToHGlobalAnsi 함수를 호출하여 원하는 결과를 얻는다. 이 함수는 인자로 받은 String 객체의 내용을 Native Heap 메모리(CLR의 Managed Heap이 아닌 기존 C++ Heap 메모리)에 복사하고 이 Heap 메모리에 대한 IntPtr을 리턴 한다. IntPtr의 ToPointer 멤버함수를 호출하면 void* 타입을 얻을 수 있으므로 이를 다시 static_cast<char*> 하여 open 함수에 전달한다. Native Heap 메모리에 복사된 데이터를 다 사용하고 나서는 Marshal::FreeHGlobal 함수를 호출하여 Native Heap에 할당된 메모리를 해지한다.
파일에서 문자열을 읽어와 단어별로 저장하는 코드는 아래와 같은 형태로 작성할 수 있을 것이다.

```
#include <iostream>
#include <string>          // 표준 C++의 string 클래스 라이브러리

// 코드 생략…

// WordCounter 객체 생성
WordCounter* counter = new WordCounter();

// 파일이 Open 되었으면 처리한다.
if ( inFile.is_open() ) {
   string lines;
   // string 객체에서 문자열의 길이, 위치 등의 정보를 나타내기 위한 타입
   string::size_type rpos = 0, lpos = 0;

   // 파일에서 라인단위로 문자열을 읽어온다.
   while ( getline(inFile, lines) ) {
      // 한 라인 문자열에서 단어들을 찾아내어 저장하는 Loop
      while ( true ) {
         // 공백 문자열을 찾는다.
         lpos = lines.find(' ', lpos);
         // 더 이상 공백이 없는 경우 Loop를 빠져나간다.
         // find 함수는 지정한 문자열을 찾지 못하면 string::npos 값을 리턴한다.
         if ( lpos == string::npos ) {
            // 마지막 WORD 추가
            if ( rpos < lines.length() ) {
               String* word = new String(lines.substr(rpos).c_str());
```

```
            counter->AddWord(word);
          }
          break;
        }
        // 공백 사이의 단어를 String 객체로 만들어 WordCounter 객체에 추가한다.
        // substr 함수를 호출하여 특정 부분의 문자열을 얻을 수 있다.
        // c_str() 함수는 string 객체의 문자열 값을 const char* 형으로 제공한다.
        String* word = new String(lines.substr(rpos, lpos - rpos).c_str());
        counter->AddWord(word);
        rpos = ++lpos;
      }
    }
    // 파일을 닫는다.
    inFile.close();
}
```

마지막으로 WordCounter에 저장된 모든 단어의 종류와 그 개수를 출력하기 위해 아래와 같은 코드를 작성할 수 있다.

```
// 화면에 단어별 개수 출력
for ( int i = 0; i < counter->GetWordsCount(); ++i ) {
    Console::WriteLine("{0} : {1}", counter->GetWord(i),
            __box(counter->GetWordFrequency(i)));
}
```

이 연습 문제를 해결하기 위해 표준 C++의 ifstream, string 클래스 등을 사용하였으나 뒤에 .NET 클래스 라이브러리의 파일 입출력을 활용하면 굳이 표준 C++ 라이브러리를 사용할 필요가 없을 것이다. 이에 대한 내용은 다른 장에서 살펴보도록 하고 관심 있는 독자는 위 예제 코드를 모두 .NET 클래스 라이브러리를 사용한 버전으로 변경해보기 바란다.