**목차**

[닷넷 3](#_Toc124018831)

[닷넷 3](#_Toc124018832)

[공통 중간 언어(CIL, Common Intermediate Language) 3](#_Toc124018833)

[공용 타입 시스템과 공용 언어 사양, 공용 언어 기반구조 3](#_Toc124018834)

[메타데이터, 어셈블리, 모듈, 매니페스트 4](#_Toc124018835)

[공용 언어 런타임과 닷넷 프레임워크 5](#_Toc124018836)

[C#과 닷넷 프레임워크, 닷넷 코어와 닷넷 표준, 닷넷5, 6 5](#_Toc124018837)

[C# 기초 6](#_Toc124018838)

[자료형 6](#_Toc124018839)

[기본 자료형 6](#_Toc124018840)

[Datatype.sln 6](#_Toc124018841)

[형변환 7](#_Toc124018842)

[HandleValue.sln 7](#_Toc124018843)

[기본 문법 요소 8](#_Toc124018844)

[예약어, 식별자 8](#_Toc124018845)

[리터럴, 변수, 상수 9](#_Toc124018846)

[연산자, 문장 부호 10](#_Toc124018847)

[배열 10](#_Toc124018848)

[배열 10](#_Toc124018849)

[다차원 배열 12](#_Toc124018850)

[가변 배열 12](#_Toc124018851)

[제어문 12](#_Toc124018852)

[관계 연산자, 논리 연산자 12](#_Toc124018853)

[ControlStatement.sln 12](#_Toc124018854)

[if문 13](#_Toc124018855)

[switch문 14](#_Toc124018856)

[증감, 복합 대입 연산자 15](#_Toc124018857)

[RepetitiveStatement.sln 15](#_Toc124018858)

[while문 15](#_Toc124018859)

[for문 16](#_Toc124018860)

[점프문 16](#_Toc124018861)

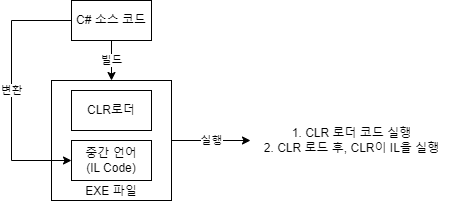
닷넷

## 닷넷

닷넷은 마이크로소프트에서 개발한 응용 프로그램 개발 환경이다. 일반적인 네이티브 언어와 다르게 닷넷 기반 프로그램은 닷넷이 설치된 환경에서만 실행된다.

닷넷을 설치하면 가상 머신 역할을 하는 CLR(Common Language Runtime)구성요소가 실행될 수 있는 환경이 마련된다. 닷넷 프로그램은 EXE/DLL로 기존 프로그램과 동일한 구조로 생겼지만, 내부적으로 CLR이 먼저 로드되어 실행되고 그 CLR이 함께 저장된 닷넷 코드(중간 언어)를 실행한다. 그러므로 C#을 C#컴파일러를 이용해 프로그램을 만들면 IL(Intermediate Language)이라 불리우는 중간 언어를 EXE/DLL 파일 내부에 생성하고, 프로그램이 시작하자마자 CLR을 로드하는 코드를 EXE파일에 추가한다.

CLR에 의해서 실행될 수 있는 중간 언어(IL, Intermediate Language)는 하나의 프로그래밍 언어에 종속된 것이 아니다. 그러므로 어떤 언어의 컴파일러이든지 결과물을 중간 언어로 생성한다면 CLR로 실행이 가능하다. 이처럼 중간 언어로 변역이 가능한 언어를 닷넷 호환 언어라고 하고, 이 중 마이크로소프트에서 공식적으로 제공되는 것은 C#, Visual Basic .Net, F#, C++/CLI가 있다. 또한 COBOL, Lisp, Python, PHP, Ruby 등의 언어도 중간 언어를 만들어내는 버전들이 있다. 닷넷 호환 언어들은 중간 코드 결과물을 공유하므로 상호간에 호출이 가능하다.



## 공통 중간 언어(CIL, Common Intermediate Language)

자바 VM에서의 중간 언어를 바이트코드(Bytecode)라고 하듯이, 닷넷의 CLR에서는 이를 CIL, IL, MSIL이라고 부른다. 모든 닷넷 호환 언어는 소스코드를 IL코드로 컴파일하고, CLR이 실행될 때 IL코드를 기계어로 최종 번역해 사용한다. 그러므로 닷넷 호환 언어는 CPU에 독립적인 결과를 낼 수 있다.

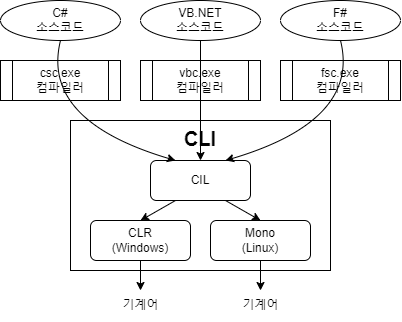
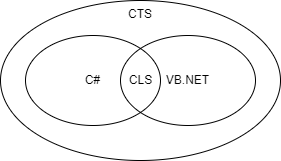
기계어와 어셈블리 언어가 대응하듯이, IL코드도 대응되는 IL언어가 있고, 이는 닷넷의 ilasm.exe를 통해서 컴파일할 수 있다. 그렇기 때문에 닷넷 호환 언어는 ‘언어 -> IL코드’ 또는 ‘언어 -> IL언어 -> ilasm.exe컴파일’와 같은 두 가지 방법을 이용해 주로 구현된다.

## 공용 타입 시스템과 공용 언어 사양, 공용 언어 기반구조

닷넷 호환 언어가 지켜야 하는 타입의 표준 규격을 정의한 것이 공용 타입 시스템(CTS, Common Type System)이다. 만약 새로운 언어를 이용해 닷넷 상에서 실행하고 싶다면 CTS 규약을 만족하는 내에서만 구현할 수 있다. 예를 들자면 CTS에서는 클래스의 단일 상속만을 하도록 정의하므로 언어가 클래스의 다중 상속을 지원하면 안 된다. 또한 이와 관련된 IL코드 또한 존재하지 않는다. 하지만 CTS에 정의된 모든 규격을 구현할 필요 또한 없으므로 필요하다고 판단되는 부분만 구현해도 괜찮다.

하지만 닷넷 호환 언어는 IL을 통해서 서로 호환이 가능해야 하므로 공용 언어 사양(CLS, Common Language Specification)에 정의된 내용만큼은 반드시 구현해야 한다.

공용 언어 기반구조(CLI, Common Language Infrastructure)는 마이크로소프트에서 ECMA 표준으로 제출한 공개 규약이다. CLI는 CTS를 포함하고, IL에 대한 정의, 메타데이터와 이들을 포함하는 이진 파일(binary file)의 구조까지 표준 사양으로 기술한다. CLI는 누구나 가져가서 임의로 구현할 수 있으며, CLI를 준수한 구현체에서 동작한 닷넷 파일은 또 다른 구현체에서도 실행하는 것이 가능하다. CLI를 마이크로소프트에서 구현한 실체가 바로 CLR이고, 그 밖에도 모노(Mono) 프레임워크, 닷넷코어(.Net Core)등이 오픈소스로 꾸준히 개발되고 있다.



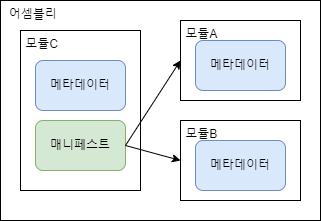
## 메타데이터, 어셈블리, 모듈, 매니페스트

메타데이터(Metadata)는 데이터를 기술하는 데이터이다. 예를 들면 그림 파일은 사진을 이루는 위치에 대한 컬러 값이 사진의 데이터이다. 반면에 이 데이터를 기술하기 위해서 너비, 높이, 해상도 등의 데이터를 표현할 수 있는데 이것이 메타데이터이다.

네이티브 언어들에서는 내가 작성한 코드에 대해서 해당 코드가 어떤 정보를 담고 있는지 코드 자체로는 알 수가 없다. 반면에 CLR에서 동작하는 실행 파일은 코드가 어떤 정보를 담고 있는지 자기 서술적인 메타데이터를 제공하고, 외부에서는 이를 리플렉션(Reflection)이라는 기술을 통해 사용할 수 있다. 그러므로 닷넷 호환 언어는 IL코드뿐 아니라 메타데이터도 함께 생성되게끔 해야 한다.

닷넷에서는 DLL, EXE파일과 같은 실행파일을 어셈블리(Assembly)라고 부른다. 이는 1개 이상의 모듈(Module)로 구성되고, 각 모듈들은 한 개의 파일이 대응된다. 만약 여러 개의 파일이 하나의 어셈블리를 구성하고 있으면, 이를 관리하는 데이터가 필요하고 모듈 중 하나는 반드시 해당 데이터를 가지고 있어야 한다. 이때의 데이터를 매니페스트(Manifest) 데이터라고 한다.

매니페스트를 포함하지 않는 모듈은 확장자가 netmodule이고, 매니페스트를 포함하는 모듈의 경우는 DLL, EXE이다. 어셈블리는 ‘참조 단위’이기도 하고 ‘배포 단위’이기도 하므로 다른 사람이 만든 어셈블리에 구현된 코드를 사용하고 싶다면 매니페스트가 포함된 모듈과 그와 관련된 모든 모듈을 가지고 있어야 한다.



## 공용 언어 런타임과 닷넷 프레임워크

공용 언어 런타임(CLR, Common Language Runtime)은 CLI사양을 따르는 가장 대표적인 VM이고, 마이크로소프트가 개발해 윈도우 운영체제용으로만 배포된다. CLR의 두 가지 큰 기능은 IL을 JIT 컴파일러를 이용해 기계어로 변환하는 것과 가비지 수집기(GC, Garbage Collector)를 제공해 동적 메모리 할당, 회수를 지원하는 것이다.

CLR은 프로세스 내에서 API에 의해 로드될 수 있다. 그러므로 윈도우 개발 환경에서 동작하는 기존 네이티브 프로그램에서 CLR을 내장해 필요할 때 닷넷을 활용하는 것이 가능하다. 그래서 CLR자체를 관리 환경(Managed Environment)라 하기도 하고, CLR이 로드되는 프로세스를 관리 프로세스(Managed Process)라고 부르기도 한다.

CLR에서 지원하는 기능만으로는 응용 프로그램을 개발하는데 불편함이 많이 생긴다. 그렇기 때문에 마이크로소프트에서는 이를 활용한 여러 가지 구성 요소를 함께 만들어 이를 하나의 패키지로 묶어 배포하는데, 이것이 ‘닷넷 프레임워크’이다. 즉, ‘CLR + 부가 구성 요소 = 닷넷 프레임워크’라고 볼 수 있으며, 부가 구성 요소는 BCL(Base Class Library), C#, VB.NET 컴파일러 및 각종 유틸리티 성격의 실행 파일, GAC(Global Assembly Cache) 등의 기능을 지원한다.

닷넷 응용 프로그램은 기존의 모든 윈도우 응용 프로그램에 대응해 만들 수 있다. 그리고 이를 만드는 과정에서 해야 하는 여러 작업들을 편하게 해주는 많은 기능들이 BCL에 구현되어 있다. 또한 새로운 버전의 닷넷 프레임워크가 출시될 때마다 기능이 추가됨으로써 BCL의 영역은 점점 커지고 있다.

## C#과 닷넷 프레임워크, 닷넷 코어와 닷넷 표준, 닷넷5, 6

위의 내용을 종합해보면 C#은 닷넷 프레임워크를 기반으로 IL을 생성하기 위한 도구이다. 즉, C#을 배우는 것은 닷넷 프레임워크를 배우는 것이나 마찬가지이다. 이때 다른 닷넷 호환 언어를 사용할 수도 있지만 C#은 마이크로소프트가 닷넷 프레임워크를 위해 만든 언어이기 때문에 C#을 사용하는 이점이 크다.

닷넷 코어(.NET Core)는 CLI의 다른 구현체이다. 그리고 이는 윈도우뿐 아니라 다른 플랫폼에서도 실행할 수 있도록 만들어진 것이 강점이다. 기존의 모노 프레임워크가 다중 플랫폼을 지원했지만 다중 플랫폼의 중요도가 올라가면서 마이크로소프트가 직접 나서서 만든 것이 닷넷 코어이다. 닷넷 코어는 CLI구현에 필요한 부분을 닷넷 프레임워크와 공유하는 형태로 개발됐으며, 소스코드가 공개되어 오픈소스로 개발되고 있다.

다중 플랫폼이 가능하지만 다르게 말하면 특정 플랫폼에서만 가능한 기능을 지원할 수 없다는 단점이 있다. 그렇기 때문에 닷넷 프레임워크와 닷넷 코어는 CIL에 해당하는 하부 구조는 동일하지만 이를 이용한 기반 라이브러리 단계부터는 서로 다른 라이브러리를 사용하게 되고 서로 호환되지 않아 라이브러리를 재사용할 수 없는 문제가 발생한다. 이를 해결하기 위해서 만들어진 호환 가능한 표준 규격을 닷넷 표준(.NET Standard) 라이브러리라고 한다.

2019년에 마이크로소프트는 닷넷 프레임워크 4.8이 마지막 버전이라고 발표했고, 이후 닷넷 코어 기반으로 이를 통합하기 시작했다. 그리고 2020년 11월 10일에 기존의 닷넷 프레임워크, 닷넷 코어라는 구분을 모두 없애고 .NET 5버전을 발표했으며 그로부터 1년 뒤 이를 개선한 .NET 6을 릴리스했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

C# 기초

# 자료형

## 기본 자료형

### Datatype.sln

자료형(Data Type)은 데이터를 담을 수 있는 일정한 형식을 말한다. 그 중에서 기본 자료형(Built-in Types)는 별도로 코드를 만들지 않아도 C#에서 자체적으로 제공하는 데이터 형식이다.

컴퓨터는 0과 1로 이루어진 bit로만 구성되어 있다. 그리고 우리는 이를 이용해 다양한 방식의 약속을 함으로써 비트의 모음이 어떤 의미를 가지고 있는지 해석한다. 이는 곧 이진수로 이루어진 숫자들을 통해서 자료형을 표현한다는 것을 의미한다. 또한 컴퓨터가 저장할 수 있는 비트의 개수는 한정되기 때문에 각 자료형 내에서는 해당 자료형에 정해진 비트의 개수로 표현할 수 있는 수의 개수만큼의 의미를 부여할 수 있다. 예를 들어 4비트를 이용해 자료형을 만들었다면 해당 자료형은 2^4 = 16개의 의미를 표현할 수 있을 것이다.

정수형 기본 타입은 정수를 표현하기 위해서 만들어진 자료형이다. 아래와 같은 자료형이 존재한다.

|  |  |  |
| --- | --- | --- |
| 자료형 | 범위 | 의미 |
| sbyte | -128 ~ 127 | 부호 있는 8비트 정수 |
| byte | 0 ~ 255 | 부호 없는 8비트 정수 |
| short | -32,768 ~ 32,767 | 부호 있는 16비트 정수 |
| ushort | 0 ~ 65,535 | 부호 없는 16비트 정수 |
| int | -2,147,483,648 ~ 2,147,483,647 | 부호 있는 32비트 정수 |
| uint | 0 ~ 4,294,967,295 | 부호 없는 32비트 정수 |
| long | -9,223,372,036,854,775,808 ~ -9,223,372,036,854,775,807 | 부호 있는 64비트 정수 |
| ulong | 0 ~ 18,446,744,073,709,551,615 | 부호 없는 64비트 정수 |

위와 같은 자료형이 다양하게 존재하는 이유는 필요한 정도의 범위를 표현하는 자료형을 사용함으로써 저장 공간을 효율적으로 사용하기 위해서이다. 특정 값을 저장하기 위해서는 ‘변수를 선언’해야 하며, int a;와 같은 문법을 이용해 선언할 수 있다. 또한 해당 선언한 변수에 값을 처음 넣는 행위를 ‘초기화’라고 하며, a = 5;와 같은 문법을 이용해 할 수 있다. 선언과 초기화는 int a = 5;와 같이 동시에 진행할 수도 있다.

정수형은 정수밖에 표현할 수 없기 때문에 우리 세계에 있는 수인 실수를 전부 표현할 수 없다. 그러므로 C#에서는 이를 표현할 수 있도록 해주는 실수형 기본 타입을 제공한다. 이는 아래와 같다.

|  |  |  |
| --- | --- | --- |
| 자료형 | 근사 범위 | 의미 |
| float |  | 4바이트 |
| double |  | 8바이트 |
| decimal |  | 16바이트 |

float과 double은 정밀도에서 차이가 나고, decimal은 반올림 오차가 허용되지 않는 회계 계산에 주로 이용한다. 또한 float을 수로 표현할 때는 float a = 5.5f;와 같이 수 뒤에 f를 붙여야 하고 decimal을 표현할 때는 decimal m = 2.2m;과 같이 m을 붙여야 한다.

약속만 되어 있다면 비트의 표현을 꼭 수로 할 필요는 없다. 그러므로 이를 활용해 우리가 자주 쓰는 문자를 표현하는 자료형 또한 C#에서 제공한다.

|  |  |  |
| --- | --- | --- |
| 자료형 | 범위 | 의미 |
| char | U+0000 ~ U+FFFF | 유니코드 16비트 문자 |
| string | 문자열 | 유니코드 문자열 |

char는 표현할 수 있는 범위의 수만 보면 ushort와 같은 범위를 가지지만 ushort는 수를 담기 위한 타입인 반면에 char는 문자를 담기 위한 타입이다. 또한 문자는 숫자와 다르게 표현할 때, char ch = ‘A’;와 같이 따옴표를 이용해 표현한다. 또한 문자와 문자는 연산을 하는 것이 논리적으로 맞지 않기 때문에 사칙연산을 이용한 계산이 불가능하다.

키보드 입력을 이용해 표현이 불가능한 문자들을 표현할 때는 이스케이프 시퀀스(escape sequence)를 이용할 수 있다. 예를 들어 ‘\n’와 같은 문자는 개행을 의미하고, ‘\t’은 탭을 의미한다. 또한 \나 ‘을 표현할 때는 ‘\\’, ‘\’‘와 같은 방식을 이용해 표현할 수 있다. 마지막으로 키보드로 입력할 수 없는 유니코드 문자들도 표현할 수 있는데 예를 들어 ▶와 같은 문자는 ‘\u2023’과 같은 방식을 이용해 표현할 수 있다.

문자열은 문자들을 여럿 모아서 표현할 때 사용한다. string자료형을 이용해 변수를 만들 수 있고, 문자열의 표현은 큰 따옴표를 이용해 문자들을 묶어서 string str = “Hello”;와 같이 사용할 수 있다. 위에서 말한 이스케이프 시퀀스 또한 문자열에 포함해서 표현할 수 있다. 이때 문자열이 시작하기 전에 @을 붙이면 \을 무시하는 문자열을 만들 수 있다. 문자와는 다르게 +연산을 지원하며 이는 문자열을 연결하는데 사용한다.

이 외의 자료형으로 bool이라는 자료형이 존재하는데, 이는 오직 true/false를 나타내는데 이용된다. 이는 나중에 배울 if, while문 등에서 평가식으로 이용되거나 참/거짓만이 존재하는 자료를 표현할 때 이용한다.

## 형변환

### HandleValue.sln

각 자료형은 서로 형변환하는 것이 가능하다. 이때 특정한 표시 없이 자연스럽게 되는 경우를 암시적 변환(Implicit conversion)이라고 하고, 형변환을 위해서 개발자가 명시적으로 표현하는 경우는 명시적 변환(Explicit conversion)이라고 한다.

만약 1바이트 크기를 가진 byte형 데이터가 2바이트 크기의 short형 데이터에 들어간다면 더 큰 범위를 표현하는 수에 들어가게 되므로 아무런 문제가 발생하지 않을 것이다. 이런 경우에는 자동으로 암시적 변환이 일어나서 값을 전달할 수 있다. 반대로 더 큰 데이터를 작은 데이터에 넣으려고 시도하면 암시적 변환이 불가능하기 때문에 컴파일러가 오류를 내보낸다.

반면에 2바이트 크기의 ushort형 데이터가 2바이트 크기의 char형 데이터에 전달되는 경우에는 크기가 같으므로 암시적 변환이 일어날 수 있어야할 것 같지만 문자는 숫자와 완전히 다른 의미를 표현하고 있으므로 암시적 변환을 할 수 없다. 하지만 개발자가 이를 인지하고 사용했다는 의미로 괄호를 이용해 u = (ushort)c;나 c = (char)u와 같이 명시적 변환을 하게 되면 정상적으로 데이터 형변환이 일어나게 된다. 또한 마찬가지로 크기가 큰 자료형에서 작은 자료형으로 형변환을 할 때도 명시적 형변환을 이용하면 형변환을 할 수 있다. 하지만 전달하려는 자료형의 값을 벗어나는 경우가 발생할 수 있으므로 값이 항상 정상적으로 값이 전달된다는 보장은 없다.

위와 같은 형변환을 다른 말로 캐스팅(casting)이라고 한다. 그리고 명시적 형변환을 할 때 사용하는 괄호 연산자를 형변환 연산자(cast operator)라고 한다.

# 기본 문법 요소

## 예약어, 식별자

예약어 또는 키워드(reserved word)는 C#에서 문법을 표현할 때 사용하기 위해서 미리 예약된 단어를 의미한다. 예를 들어 위에서 사용한 자료형들은 전부 예약어이고, 실제로는 아래와 같이 각각 대응하는 닷넷 형식을 편하게 나타낼 수 있도록 해주는 것이다.

|  |  |  |  |
| --- | --- | --- | --- |
| C# | 닷넷 형식 | C# | 닷넷 형식 |
| sbyte | System.Sbyte | float | System.Single |
| byte | System.Byte | double | System.Double |
| short | System.Int16 | decimal | System.Decimal |
| ushort | System.UInt16 | char | System.Char |
| int | System.Int32 | string | System.String |
| uint | System.UInt32 | bool | System.Boolean |
| long | System.Int64 |  |  |
| ulong | System.UInt64 |  |  |

식별자는 프로그래밍을 하면서 임의로 선택해서 이름 지을 수 있는 단어들을 의미한다. 식별자는 아래의 조건에 걸리지 않는다면 자유롭게 이름을 바꿀 수 있다.

* 식별자의 시작 문자는 숫자로 시작할 수 없고, 반드시 문자이어야 한다.
* 특수 문자 중에서 유일하게 \_문자만을 시작 문자로 사용할 수 있다.
* 유니코드 범위의 문자가 허용된다.(한글 식별자도 가능하다.)
* 예약어를 식별자로 사용할 수 없다. 만약 이를 사용해야 한다면 ‘@’를 접두어로 붙여 사용할 수 있다.
* 이스케이프 시퀀스도 식별자로 사용할 수 있다.

## 리터럴, 변수, 상수

리터럴(literal)은 한글로 ‘문자 상의, 문자 그대로의 같은’ 의미를 가지고 있다. 이는 소스코드에 포함된 값을 나타내는 용어로 string text = “Hello”;, int n = 5;, char ch = ‘N’;, bool result = true; 와 같은 코드에서 “Hello”, 5, ‘N’, true와 같이 그 자체로 값을 나타내는 코드를 리터럴이라고 한다.

변수(variable)은 자료형에 알맞은 저장소를 만들어 사용자가 이용할 수 있게 해준다. 사용자가 이를 직접 선언하고 사용하기 때문에 식별자라고 볼 수 있다. 예를 들어 int n = 5;와 같이 사용하면 int형 변수 n을 만들어 5를 대입한다는 의미이다. 그리고 n = 10;이라고 한다면 기존에 만든 변수 n에 다시 10을 대입한다는 의미이다.

닷넷에서는 크게 두 종류의 변수가 있는데, 값 형식(Value Type)을 가리키는 변수와 참조 형식(Reference Type)을 가리키는 변수가 있다.

둘의 차이를 이해하기 위해서는 스택(Stack)과 힙(Heap)이라고 불리는 메모리 저장소에 대해서 알 필요가 있다. 프로그램은 기본적으로 스레드라는 단위를 이용해 프로그램에 필요한 자원들을 관리한다. 그리고 개별 스레드마다 해당 스레드 전용의 저장소가 메모리에 할당되는데 이 공간을 스택이라고 한다. 반면에 힙은 프로그램에서 해당 저장소를 사용하겠다고 운영체제에 직접 요청해 할당 받은 공간을 힙이라고 한다. 이때 닷넷에서는 힙이 필요한 경우라면 CLR이 이를 자동으로 처리하고 관리한다.

위와 같은 메모리들은 사용하고 나서 더 이상 필요가 없어지면 다른 자원이 해당 메모리를 다시 활용할 수 있도록 할당 해제를 요청해야 한다. 스택 같은 경우는 컴파일러가 이를 자동으로 관리해주므로 스택의 메모리가 부족할 정도로 많이 사용하지만 않으면 문제가 되지 않는다. 반면에 힙은 사용자가 직접 요청해서 할당받는 공간이므로 네이티브 언어에서는 사용자가 이를 할당, 해제하는 과정을 직접 해줄 필요가 있었다. 하지만 닷넷에서는 힙공간에 대한 해제를 사용자가 하지 않아도 해당 공간의 사용 정도를 파악해 불필요한 경우에는 자동으로 해주기 때문에 사용자는 직접 할당하기만 하면 된다. 이러한 기능을 하는 닷넷의 요소를 가비지 수집기라고 한다.

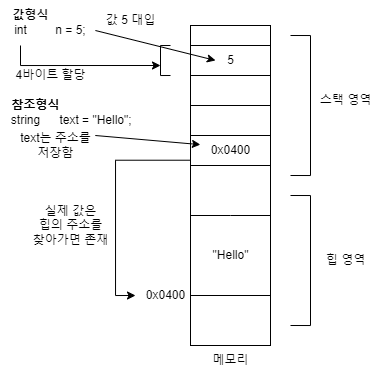
값 형식을 가리키는 변수는 변수가 가리키는 값 그 자체가 스택에 저장이 된다. 이러한 값 형식에 해당하는 자료형은 앞에서 배운 기본 자료형에서 string을 제외한 모두가 해당한다.

참조 형식을 가리키는 변수는 변수는 실제 값은 힙에 저장하고, 해당 값이 들어있는 힙 내의 주소를 스택에 저장한다. 이에 해당하는 자료형은 값 형식 자료형을 제외한 모두가 해당한다.

즉, 값 형식과 참조 형식은 실제 데이터의 저장장소가 다른 차이점이 있고, 뒤에서 소개할 특성들은 모두 이에 기인한다.

값 형식의 경우에는 초기화를 하지 않으면 초기값으로 0이나 false를 가지고 있다. 반면에 참조 형식은 초기화를 하지 않으면 아예 힙에 할당된 데이터가 없다. 또한 주소를 가리켜야 하는 스택에 0을 저장할 수는 없으므로 대신에 null이 저장되어 있다.

값 형식은 대입을 시도하면 스택에 있는 기존의 값에 새로운 값을 덮어씌운다. 마찬가지로 참조 형식은 대입을 시도하면 스택에 있는 기존의 주소에 새로운 주소를 덮어씌운다. 즉, 대입은 모두 같은 방식으로 작동한다. 하지만 참조 형식은 실제 데이터를 힙에 가지고 있으므로, 결과적으로는 힙에 있는 데이터들은 아무런 변화가 없고, 변수가 가리키는 데이터가 달라졌을 뿐이다. 또한 기존에 가리키던 힙의 데이터는 방치된 채로 있다가 가리키는 변수가 1개도 없으면 나중에 가비지 수집기가 자동으로 처리하게 된다.



변수가 값을 바꿀 수 있는 저장소에 대한 식별자였다면 상수(constant)는 값을 처음에 한 번만 설정 가능하고, 그 이후로는 다른 값을 대입해 바꿀 수 없는 저장소에 대한 식별자이다. 이는 변수를 선언하듯 선언하되 const int n = 5;와 같이 앞에 const 예약어를 붙이기만 하면된다. 이렇게 한 번 만들어준 상수는 n = 3;과 같이 다시 대입하는 것이 불가능하다. 또한 상수는 반드시 컴파일 단계에서 값이 이미 결정되어 있는 경우에만 값을 설정해줄 수 있다. 예를 들어 더 큰 값을 골라내는 Math.Max 메소드를 활용해 const int n = Math.Max(0, 5);를 한다면 사용자가 봤을 때는 당연히 5가 정답이므로 상수로 사용할 수 있을 것 같지만, 컴파일러의 입장에서는 해당 메소드를 들어가 연산을 거친 뒤에 결과가 나오게 되므로 상수로 사용할 수 없다고 오류를 띄우게 된다. 반면에 const int n = 5 \* 100;과 같이 사용한 경우에는 컴파일러가 바로 계산이 가능하므로 사용이 가능하다.

## 연산자, 문장 부호

C#을 통해 원하는 연산을 하기 위해서는 연산자(operator)와 문장부호(punctuator)를 통해 문법을 지키면서 원하는 명령을 만들어내야 한다. 예를 들어 { }는 C#의 각종 문법을 위해 코드를 묶어야 할 때 사용하고, ;은 명령을 내리는 단위인 구문의 끝을 알릴 때 사용한다. 또한 대표적인 연산자로 대입 연산자(assignment operator) ‘=’이 있고, 이는 오른쪽의 값을 왼쪽으로 넣어줄 때 사용한다. 또한 우리가 흔히 알고 있는 사칙연산은 ‘+’, ‘-‘, ‘\*’, ‘/’에 해당하는 산술 연산자(arithmetic operator)를 이용해 사용할 수 있다. 또한 특이한 연산자로 나머지 연산을 하는데 사용하는 ‘%’ 연산이 있다. 이는 말그대로 나눗셈을 하고 나서 나온 나머지를 구하는데 이용한다. 예를 들어 5 % 3을 하게 된다면 결과로 2가 나오게 될 것이다.

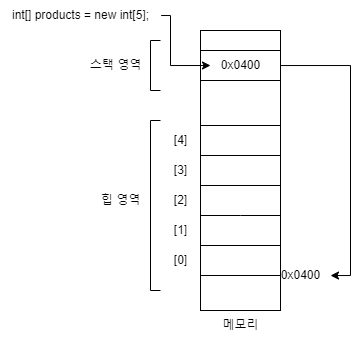
이 외에도 여러 연산자와 문장부호가 존재하는데 이는 뒤쪽에서 다른 개념을 배우면서 천천히 익히게 된다.

# 배열

## 배열

프로그램을 만들다 보면 같은 자료형의 변수 여러 개를 한 번에 묶어서 사용해야할 때가 많이 발생한다. 이런 경우에 필요한 변수를 모두 만들기도 힘들고, 이를 활용하기도 어렵다. 이런 경우를 위해서 C#에서는 배열(array)이라는 개념이 존재한다. 배열은 대괄호 문장부호와 new예약어를 이용해 int[] products = new int[5];와 같은 문법으로 사용한다. 해당 코드는 5개의 연속으로 나열된 int형 데이터를 만들어주고 이를 products라는 변수를 통해서 접근할 수 있게 만든다는 의미와 마찬가지이다. 각 개별데이터는 대괄호 문장부호를 이용해서 products[0], products[1], … , products[4]와 같이 변수명[숫자]에 해당하는 문법을 사용함으로써 접근할 수 있다. 이때 숫자에 해당하는 부분을 색인, 인덱스(index)라고 부른다.

우리가 배열을 만들 때 사용하는 new 예약어의 실제 기능은 힙 영역에 데이터를 할당하는 것이다. 또한 배열은 기본 자료형을 나열해서 접근할 수 있게 해주는 일종의 자료형이나 마찬가지이므로 기본 자료형이 아니란 것 또한 알 수 있다. 그러므로 배열을 통해 만들어진 변수는 참조 형식 변수임을 알 수 있고, 실제 변수가 저장하는 것은 힙에 대한 주소이라는 것도 알 수 있다. 이때 배열은 나열된 데이터를 접근할 수 있는 방법이 있어야 하므로 힙에 저장된 나열된 데이터 중 첫 번째 데이터의 주소를 변수에 저장하게 된다. 그리고 이를 바탕으로 인덱스를 이용해 적절한 주소를 찾아서 원하는 데이터에 접근하게 되는 것이다.



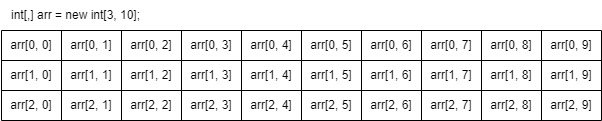
배열도 다른 자료형을 사용할 때와 마찬가지로 초기화를 할 수 있다. 초기화는 중괄호를 이용해 하는데, int[] products = new int[5] {4, 2, 3, 10, 5};와 같이 배열의 크기를 명시하거나, int[] products = new int[] {3, 2, 15};와 같이 배열의 크기를 명시하지 않는 2가지 방법으로 할 수 있다. 배열의 크기를 명시하지 않는 경우에는 초기화시에 들어간 값의 개수로 배열의 크기가 정해지게 된다. 반대로 배열의 크기를 명시했지만 아직 초기화 하지 않았다면 배열의 각 자료형에 알맞은 초기값이 들어가게 된다.

그리고 생성된 배열은 크기가 한 번 정해지면 더 이상 바꿀 수 없다. 하지만 배열은 참조 변수이기 때문에 실제로는 주소값을 저장하고 있는 것이고, 이것은 기존의 변수를 이용해 크기가 다른 배열을 새로 생성하거나 다른 배열이 가지고 있는 데이터를 가리키는 것을 가능케 해준다.

우리가 앞에서 배운 string자료형은 배열과 마찬가지로 char형 데이터를 연속적으로 나열한 것이고, 배열처럼 대괄호 연산자와 인덱스를 이용해 데이터에 접근하는 것 또한 가능하다. 하지만 이는 string에 관련된 내부 기능에서 대괄호 연산에 관련된 기능을 구현했기 때문에 가능한 것이며 string형이 실제 배열인 것은 아니다.

## 다차원 배열

지금까지의 배열들은 1차원으로만 저장했지만 2차원 이상의 배열도 만들 수 있다. 이러한 배열을 다차원 배열(multi-dimensional array)라고 한다. 다차원 배열의 사용은 기본적으로 일반 배열과 같지만 차원 분리자 , 를 추가로 이용해 int[,] arr2 = new int[10, 5];나 int[,,] arr3 = new int[8, 3, 10];과 같이 선언할 수 있다. 각 데이터 또한 일반 배열과 같이 힙을 이용해 저장하고, 각 값에 대한 접근도 마찬가지로 대괄호를 이용한다.



그리고 초기화는 아래와 같이 중괄호를 이용해 배열을 다시 배열하는 방식으로 한다.

Int[,] arr2 = new int[2, 3] {  
 {1, 2, 3},  
 {4, 5, 6}  
 };

## 가변 배열

C#에서는 다차원 배열 외에 가변 배열(jagged array)을 지원하는데, 이는 배열의 배열이라고 불리기도 한다. 이는 int[][] arr = new int[3][]; arr[0] = new int[10]; arr[1] = new int[5]; arr[2] = new int[2]; 와 같이 사용하게 된다. 즉, 다차원 배열은 콤마로 차수를 구별하기 때문에 서로 묶여 있는 구조인 반면에 가변 배열은 각 차수마다 새로운 배열이 들어가는 구조가 된다. 이는 2개의 차원을 가지더라도 안쪽 배열의 크기가 정해지지 않았기 때문에 다차원 배열처럼 공간이 할당됐는데 실제로 쓰지 않게 되는 공간이 많아질 경우에는 공간의 효율성을 높일 수 있게 된다. 하지만 사용 편의성과 유지보수성 때문에 그런 방식으로 자주 이용되기보다는 다차원 배열과 같은 형태로 m \* n개가 사용되는 경우가 더 일반적이다.

# 제어문

## 관계 연산자, 논리 연산자

### ControlStatement.sln

프로그램을 만들다 보면 각종 조건에 따라 다른 명령을 내려야하는 경우가 많이 생긴다. 이를 처리하려면 먼저 각종 조건을 평가하고 평가한 상황에 알맞게 프로그램이 흘러가도록 만들어야 한다. C#에서는 이를 true, false의 결과로 나오게 되는 bool자료형을 이용해서 처리한다. 그리고 이러한 처리를 위해 조건의 평가를 돕는 연산자가 필요한데, 그런 연산자로는 관계 연산자(relational operator)와 논리 연산자(logical operator)가 있다.

그 중 관계 연산자는 피연산자 2개를 비교해 해당 연산이 참인지 거짓인지에 따라 bool형 값의 결과를 내놓는다. 그리고 이에 해당하는 연산자는 아래와 같다.

|  |  |
| --- | --- |
| 관계 연산자 | 평가방식 |
| > | 좌측이 더 크면 참, 아니면 거짓 |
| < | 우측이 더 크면 참, 아니면 거짓 |
| >= | 좌측이 크거나 서로 같으면 참, 아니면 거짓 |
| <= | 우측이 크거나 서로 같으면 참, 아니면 거짓 |
| == | 같으면 참, 아니면 거짓 |
| != | 다르면 참 아니면 거짓 |

또한 우리가 원하는 조건이 꼭 하나의 비교로만 일어나는 것이 아닐 수 있다. 이런 경우에는 논리 연산자를 이용해 우리가 원하는 상황인지 확인할 수 있다. 이 때 논리 연산자는 피연산자로 bool형 값을 받게 되는데, 이를 계산하기 위해서 우리는 부울 대수(Boolean algebra)의 기본 연산을 하는 것과 같은 연산을 하게 되고, 논리 연산자는 이들을 기호로 나타낸 것이라고 생각할 수 있다. 논리 연산자는 &&(AND, 논리곱), ||(OR, 논리합), !(NOT, 부정), ^(XOR, 배타적 논리합)가 존재한다.

|  |  |  |
| --- | --- | --- |
| **&&(AND)** | **T** | **F** |
| **T** | T | F |
| **F** | F | F |
| **||(OR)** | **T** | **F** |
| **T** | T | T |
| **F** | T | F |
| **^(XOR)** | **T** | **F** |
| **T** | F | T |
| **F** | T | F |
| **!(NOT)** | **T** | **F** |
|  | F | T |

이와 같은 논리 연산자는 사용할 대 주의할 점이 있는데, 사용하는 과정에서 불필요하다고 판단되는 특정 계산을 생략하는 경우가 생길 수 있다는 것이다.

예를 들어 (n1 % 3 == 0) || (n2 % 3 == 0)을 계산한다고 할 때, n1이 3이라면 n1 % 3 == 0의 결과는 true가 될 것이다. 이때, OR연산의 특성상 둘 중 하나만 true이면 무조건 결과가 true이기 때문에 앞의 n1 % 3 == 0만 계산하고 뒤쪽의 n2 % 3 == 0은 계산을 생략해버리게 된다. 이와 같이 프로그램 실행 과정에서 뒤의 조건은 아예 실행조차 하지 않는 것을 단락 계산 또는 단축 평가(short-circuit evaluation)라고 한다. 예시의 OR연산과 마찬가지로 AND연산은 둘 중 하나만 false만 되면 결과는 무조건 false이기 때문에, 앞의 식이 false가 되는 경우에 단축 평가가 일어나게 된다.

## if문

특정 조건에 대한 선택을 위한 if문의 문법은 다음과 같다.

if (조건식)  
 구문;

이 경우에 조건식이 참이면 구문에 해당하는 코드를 실행하고, 거짓이면 이를 건너뛰게 된다. 이 때 if문을 이용해 실행할 코드가 여러 줄이라면 중괄호 문장부호를 사용해 표현하는 블록(block)을 이용해 해당 코드들을 묶어줄 필요가 있다.

또한 if문의 조건이 거짓인 경우에 특정 코드를 따로 실행하고 싶다면 else 예약어를 이용해 if문을 확장할 수 있다. 이는 아래와 같은 방법으로 사용한다.

if (조건식)  
 구문;  
else  
 구문;

또한 else 뒤에 바로 새로운 if문을 쓴다면 else if (조건식)과 같이 사용할 수 있으며, 이를 이용해 다양한 조건에서의 코드 실행을 구현할 수 있다.

만약 if문의 결과로 간단한 식을 이용해야 하는 상황이라면 이를 줄여 쓸 수 있도록 만들어진 연산자를 사용할 수 있다. 이를 조건 연산자라고 하며 피연산자를 3개 사용한다 해서, 삼항 연산자라고 하기도 한다. 이는 아래와 같은 문법으로 이용한다.

(조건식) ? 표현식1 : 표현식2

이 연산자는 조건식이 참이라면 표현식1을 거짓이라면 표현식2를 사용하게 된다.

## switch문

if문 중에서 조건 판단의 기준이 되는 식이 상수라면 switch문으로 다시 표현할 수 있다. switch문의 문법은 아래와 같다.

switch(인스턴스)  
case 상수식1:  
 구문;  
 break;  
 …  
case 상수식n:  
 구문;  
 break;  
default:  
 구문;  
 break;

위 식을 if-else문으로 바꾼다면 아래와 같을 것이다.

if (인스턴스 == 상수1)  
 구문;  
else if (인스턴스 == 상수2)  
 구문;   
…  
else if (인스턴스 == 상수n)  
 구문;   
else  
 구문;

switch문을 사용할 때는 case문의 마지막에 반드시 break;를 항상 적어 해당 case가 끝났다는 것을 알려야 한다. 하지만 case문에 아무런 구문을 사용하지 않고 case만 적게 된다면, 해당 case 다음에 있는 case와 코드를 공유하게 된다. 그리고 마지막에 사용하는 default의 경우에는 필수가 아니니 상황에 따라 필요가 없다면 쓰지 않아도 된다.

## 증감, 복합 대입 연산자

### RepetitiveStatement.sln

증감 연산자(increment & decrement operator)는 값을 1씩 증가시키거나 감소시킬 때 사용하는 연산자이다. 연산자는 ++, -- 두 종류가 있으며 각각 증가, 감소를 의미한다. 이를 사용할 때는 n++, ++n, n--, --n과 같이 특정 변수의 앞이나 뒤에 붙여서 사용한다.

연산자를 앞에 쓴 경우에는 연산자를 전위 표기했다 하고, 뒤에 쓴 경우에는 후위 표기했다고 한다. 각각은 코드 실행 순서와 관련이 있다.

구문에서 증감 연산자를 전위 표기한 경우에는 연산자가 쓰인 코드에서 해당 변수의 값을 이미 증감한 것으로 판단하고 그 변수를 사용한다. 반면에 후위 표기한 경우에는 해당 변수는 일단 사용하고 연산자가 있는 코드가 넘어가면서 변수의 값을 증감하게 된다.

복합 대입 연산자(compound assignment operator)는 사칙연산을 이용한 계산을 하는 식에서 연산의 피연산자 중 일부가 대입을 받는 변수 자신이라면 사용하게 된다. 이에 해당하는 연산자로는 +=, -=, \*=, /=, %=이 있고, 사용은 n += 5;와 같은 문법으로 사용한다. 이는 n = n + 5;와 완전히 같은 의미를 가진다.

## while문

반복문은 정해진 코드를 여러 번 반복해서 실행하기 위해서 존재한다. 코드를 계속 반복하면 프로그램이 끝나지 않기 때문에 코드를 실행할 조건을 정한 뒤, 그 조건을 만족시킬 때 코드를 실행하도록 해주는 문법이 while문이다. 이는 다음과 같은 문법을 이용한다.

while(조건식)  
 구문;

위의 코드는 조건식이 참이면 구문을 실행한 후, 해당 구문이 끝나면 다시 조건식을 평가하는 것을 반복한다. 그러다가 조건식이 거짓이라면 while문을 벗어나게 된다. 이와 같은 방법을 통해서 반복적인 코드 실행을 할 수 있다.

또한 비슷하게 do/while문을 사용할 수 있다. do/while문은 아래와 같은 문법을 이용한다.

do  
 구문;  
while(조건식);

이와 같은 문법은 일단 구문을 한 번 실행한 뒤, 조건식을 평가하게 된다. 이때 while의 마지막에 세미콜론을 적는 것을 잊지 않아야 한다.

## for문

while문을 이용하다 보면 while문 내에서 간단하지만 핵심이 되는 식이 존재하기 마련이다. 이를 좀더 명시적으로 표현하기 위해서는 for문을 이용할 수 있다. for문의 문법은 아래와 같다.

for (초기화; 조건식; 반복식)  
 구문;

초기화 식은 for문에 진입할 때, 처음 한번 실행되는 식이다. 그리고 조건식은 while문에서의 조건식과 같다. 마지막으로 반복식은 for문이 끝나고 나서 새로운 반복이 일어나기 직전에 실행되는 식이다. 즉, 초기화식 -> 조건식 -> 구문 -> 반복식 -> 조건식 -> 구문 -> … -> 반복식 -> 조건식 과 같은 순서로 식이 실행되고, 조건식이 거짓이 되기 전까지 이를 반복한다. 즉, 근본적으로 while문과 아무런 차이도 없고 단순히 표현의 방식이 다른 것이다.

## 점프문

반복문을 사용하다 보면 중간에 이를 벗어나고 다른 코드로 넘어가야 하는 경우가 발생한다.

점프문에 해당하는 것으로는 break, continue, goto가 있다.

break문은 while, for, foreach, do, switch에서 사용할 수 있다. switch문에서는 break문을 사용해 case를 끝내는데 사용하고, 반복문에서는 break문을 만나면 해당 반복문을 벗어나게 된다.

break문은 반복문의 반복을 끝내 버리는 반면에 continue문을 만나게 되면 반복문의 처음으로 돌아가 조건 비교부터 새로 반복문을 계속 이어 나가게 된다.

goto문은 사용하는 곳에 제한이 없다. goto문의 사용을 위해선 먼저 레이블 문이 있어야 한다. 레이블 문은 식별자: 과 같은 방법을 이용해 사용하고, goto 식별자; 와 같은 코드를 이용하면 실행이 이동해 레이블 문의 다음 코드부터 이어지게 된다. 이는 중첩된 반복문에서 유용하게 이용할 수 있다.