컴퓨터정보과

C# 프로그래밍

기본문법

자료형

프로그램에서 데이터를 표현하는 형식(크기, 해석방법, …)

기본 자료형

- **값형** (value type) : struct로 생성한 자료형
  - 숫자형
    - 정수형 (integer)
      - sbyte(1), **byte(1)**, short(2), ushort(2), **int(4)**, uint(4), **long(8)**, ulong(8)
    - 실수형
      - floating point number(부동소수점) - IEEE754
        - float(4), **double(8)**
      - fixed point number (고정소수점)
        - decimal(16)
  - 논리형 (boolean)
    - **bool**
  - 문자형 (character)
    - **char**

참고 : 값 형식 - C# 참조 - C# | Microsoft Learn

- **참조형** (reference type) : class로 생성한 자료형
  - 문자열형
    - **string**

참고 : 참조 형식 - C# 참조 - C# | Microsoft Learn

정수형 Integer Number

signed type	unsigned type	size (bytes)	리터럴 (접미사)
sbyte	**byte**	**1**	**없음, 없음**
short	ushort	2	없음, 없음
int	**uint**	**4**	**없음,100U**
long	ulong	8	100L, 100UL

기타: nint, nuint (시스템 가변 int)
진법 : 10진법, 16진법(0x, 0X 접두어), 2진법(0b, 0B 접두어) … 8진법 없음

실수형 Real Number

type	decimal place	size (bytes)	리터럴 (접미사)
float	6~9	4	3.14f
double	**15~17**	**8**	**3.14 ,3.14d**
decimal	28~29	16	3.14m

float, double : 부동소수점형 (IEEE754), decimal : 고정소수점형
실수계열은 unsigned가 없음

최대값: int.MaxValue, double.MaxVlaue
최소값: int.MinValue, double.MinValue

논리형 Boolean

type	size (bytes)	리터럴
bool	1	true / false

C언어처럼 0이나 1로 참과 거짓을 판단할 수 없음
주로 제어문과 함께 사용 됨

```
//C
while(1) { … }

//C#
while(true) { … }
```

문자형 Character

type	size (bytes)	리터럴	문자코드
char	2	‘1’, ‘a’, ‘A’, …	Unicode (ascii + α)

C언어는 ascii code를 기본으로 1바이트
내부적으로는 정수형

문자열형 String

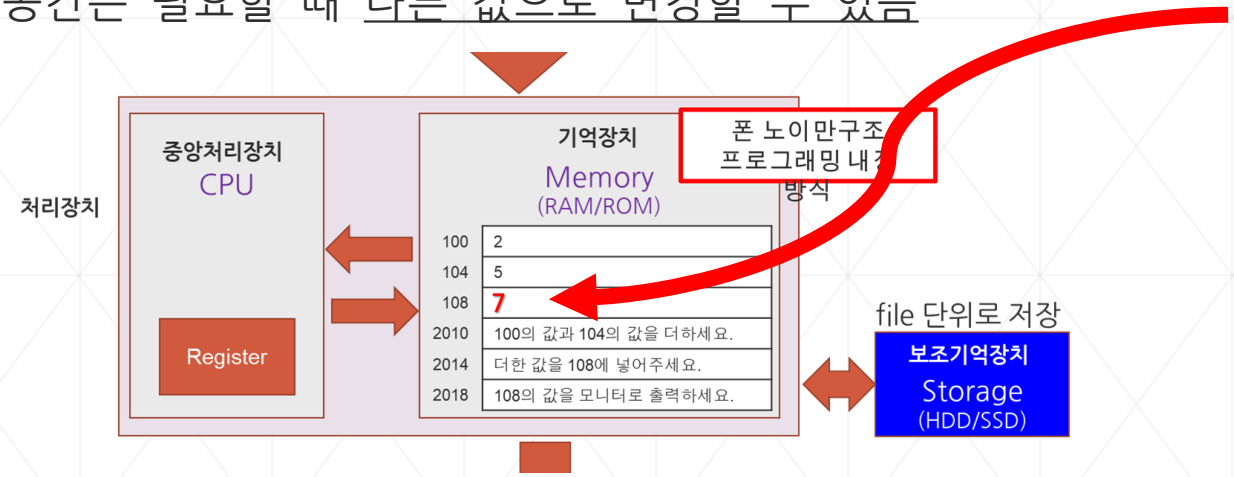
type	size (bytes)	리터럴
string	알 수 없음 (내부적으로 가변)	“ ”, ”a”, ”ABC”, …

C언어처럼 문자 n개가 나열되어 있다고 생각하면 됨.
앞의 다른 타입과 달리 단일 값이 여러 개가 모여있는 복합구조의 형태를 갖고 있음.

변수 & 상수

변수 Variable

- 메모리(RAM)에 프로그램 실행 시 필요한 값을 저장하기 위해 할당받은 공간
  해당 공간은 필요할 때 다른 값으로 변경할 수 있음



- 공간할당을 받기 위해서는 어떤 형태의 데이터를 저장할 것이며, 얼마만큼의 크기가 필요한지 정보를 제공해줘야 함.(자료형 정보가 필요함)

- 변수 선언 및 대입

```
double radius; //변수 공간 선언
radius = 45.5; //변수 공간에 값 대입
```

  - double 자료형 으로 변수 공간을 할당받고 해당 공간의 이름을 프로그램에서는 radius라고 부른다.
  - radius라는 변수 공간에 double 자료의 리터럴 45.5를 보관한다.

상수 Constant

- 메모리(RAM)에 프로그램 실행 시 필요한 값을 저장하기 위해 할당받은 공간
  해당 공간은 처음 초기화한 값을 다른 값으로 변경할 수 없음
- 기호(심볼릭, Symbolic) 상수 선언 및 대입

  const double radius = 45.5; //상수 공간 선언 후 동시에 초기화 해야함
  - double 자료형 으로 상수 공간을 할당받고 해당 공간의 이름을 프로그램에서는 radius라고 부른다.
  - radius라는 상수 공간에 double 자료의 리터럴 45.5를 보관한다.

  ~~radius = 55.5~~; //상수 공간에 초기값 대입 후 다른 값을 대입하면 에러가 발생함
  - 이후 radius의 공간에 있는 값은 변경이 불가능하다.

- 리터럴 (Literal) 상수
  - double radius = **50.5;**
  - 문자 그대로…
  - 50.5와 같이 ‘ 소스 코드에 고정된 값 ' 을 뜻하며, 해당 내용도 상수(constant)로 간주한다.

암시적 형식 지역 변수 var

- 자료형 키워드 : var
  - 가능 조건
    - 메소드 내의 지역변수만 가능
    - 변수 선언과 동시에 초기화
  - 예
    - `var i = 10ul;`
    - `var d = 10.2m;`
    - `var q = 20.1;`

자료형 변환

- 자동 (묵시적) 형변환
- 강제 (명시적) 형변환
  - 형변환 연산자 : ()
    - 값형 ↔ 값형, 참조형 ↔ 참조형
  - 형변환 메소드 : 자료형에 따라 다름
    - 값형 ↔ 참조형
    - 문자열 ↔ 정수, 실수, 불, …
    - 정수,실수, 불, … ↔ 문자열

강제 형변환 - 형변환 연산자 ()

- 큰 자료에서 작은 자료형으로 변환할 때
  - 강제 형변환

```
long a = 100000000;
byte b = (byte)a;
```

  - 자동 형변환

```
byte b = 10;
long a = b;
```

- 정수, 실수간 형변환

```
int a = 10;
double b = (double) a;
```

강제 형변환 - 형변환 메소드

- 문자열과 숫자간의 형변환
  - “ 문자열 ” → 숫자 : 자료형.Parse(“ 문자열값 “)
    - 예제

```
int a = int.Parse( “ 1 ” );
double b = double.Parse( “ 1.1 ” );
bool c = bool.Parse( “ true");
```

  - 숫자 → “ 문자열 ” : 숫자리터럴.ToString()
    - 예제

```
string d = a.ToString(); //1.ToString();  (1).ToString();
string e = b.ToString(); // (1.1).ToString();
string f = c.ToString(); //(true).ToString();
```

연산자와 수식

데이터를 가공(계산)하는 기능

연산자

- 산술연산자
- 문자열연산자
- 비교연산자
- 논리연산자
- 대입연산자
- 증감연산자
- sizeof 연산자
- 비트연산자
- &, ^, ¦, ~
- <<, >>
- as, is, (), typeof(), ?:, ??, =>, new, [], . , …

산술 연산자

연산자	설명	비교
+	덧셈	
-	뺄셈	
*	곱셈	
/	나눗셈	
%	나머지	실수에도 사용 가능

1. 정수간 연산 결과 : 정수
2. 실수간 연산 결과 : 실수
3. 실수와 정수간 연산 결과 : 실수

문자열 연산자

연산자	설명	비교
+	문자열 연결	
문자열[숫자]	문자 선택	

- 문자열 연결 연산자
  "1" + "3" ➔ "13"
  "1" + 3 ➔ "13"
  1 + 3 ➔ 4
  '가' + '힣' ➔ 99235

  (문자열 및 숫자 연산 결과 : 문자열 연결)

- 문자 선택 연산자

```
string name = "name";
char a = name[1];
```

조건 연산자 (삼항 연산)

연산자	설명	비교
?:	조건에 따른 선택	불_표현식 ? (true인 경우) : (false인 경우)

비교 연산자

연산자	설명	비교
==	같다	
!=	다르다	
>	왼쪽 피연산자가 크다	
>=	왼쪽 피연산자가 크거나 같다.	
<	오른쪽 피연산자가 같다.	
<=	오른쪽 피연산자가 크거나 같다.	

논리 연산자

연산자	설명	비교
!	논리 부정	!true
||	논리 합	true || false
&&	논리 곱	True && false

- 두 연산자의 결과 값은 항상 bool 형으로 나온다.

대입 연산자

연산자	설명	비교
=	오른쪽 피연산자를 왼쪽 변수에 대입	a = b
+=	오른쪽 피연산자와 왼쪽 피연산자를 연산하고 왼쪽 변수에 넣는다.	a += b
-=		a -= b
*=		a *= b
/=		a /= b
%=		a %= b

증감 연산자

연산자	설명	비교
[변수]++	기존 값에 1을 더함	후위 a++
++[변수]	기존 값에 1을 더함	전위 ++a
[변수]--	기존 값에 1을 뺌	후위 a--
--[변수]	기존 값에 1을 뺌	전위 --a

제어문

프로그래밍의 실행 순서를 조정

제어문

- 제어문에 사용하는 조건식의 결과는 C언어 처럼 0이냐 0이 아니냐로 판별 불가
- 반드시 bool 타입으로 조건식의 결과가 나와야 함.

C#

```
int a = 10;
if (a == 10)
{
    …
}
```

C

```
int a = 10;
if (a)
{
    …
}
```

- 분기문
  - 조건분기문 : if, switch-case
    - 조건연산자(삼항연산자)
    - 무조건분기문 : goto
- 반복문 :
  - while, do-while, for, foreach
    - break, continue

if

- 기본 : 괄호의 bool 표현식 결과가 true이면 실행, false이면 무시하는 구문
- if를 활용한 다양한 if 조건 문장
  - if
  - if – else
  - nested if
  - if - else if - else
- C언어와 다른 점: 조건문은 수치데이터가 아닌 bool형 데이터만 취급한다.

switch

- 비교 값을 이용하여 해당하는 case를 실행하는 조건문
- C언어와 다른 점.
  - 정수 데이터가 아닌 문자열 데이터도 비교 값으로 사용할 수 있다.
    - 최근 C#의 switch문법은 기본 문법 외에 많은 기능과 비교 값 타입이 추가되었다.
    - (해당 기능은 수업 시간에 사용하지 않음)
  - 빈 case문일때만 break를 생략할 수 있음.

while

- 가장 기본적인 반복문
  - 조건이 만족하면 계속 반복해서 실행
  - if와 유사 : if는 조건이 만족하면 한 번만 실행
- for문 대비, 반복의 횟수가 명확하지 않는 경우에 많이 사용.

do-while

- while문 변형
- 보통 반복문은 조건을 검사한 후 실행하는데 반해
  - do-while은 먼저 실행한 후 조건을 검사하여 다시 실행할지 결정
- 주로 console 프로그램에서 메뉴 출력 시 유용하게 사용함.

for

- 가장 구조화된 반복문
- for구문을 통해서 시작값, 조건, 횟수등을 유추할 수 있음.
- while 대비, 반복의 횟수가 명확한 경우 사용

foreach

- 제어문 중 C언어 없는 유일한 문법
- 향상된 for문이나 foreach라는 용어로 표현함.
  - python의 for문과 유사함.
- 많이 사용하는 for문의 패턴을 간편화한 문법
  - 패턴: 배열과 같은 연속적인 자료형 구조를 처음부터 끝까지 탐색하는 용도
- 문법 : foreach(자료형 변수 in 컬렉션) { … }

배열

동일한 자료형의 모임

배열 Array

- Array : 동일한 데이터 형식의 요소가 메모리에 순서대로 나열된 데이터 구조
  - 크기는 선언될 때 정적으로 정의되며, 실행 중에 크기를 변경할 수 없습니다(고정길이)

1차원 배열 Array

- 데이터형식[ ] 배열이름 = new 데이터형식[크기];
  - int[] array = new int[5];
  - 인덱스(index) :[0~ n-1]
    - array[1] = 2;
    - int a = array[3];
- 초기화
  - 초기화 하지 않으면, 기본 값으로 초기화 됨
    - 값타입(0, 0.0, false) vs. 참조타입(null)
  - int[] array = new int[3]{1,2,3};
  - int[] array = new int[]{1,2,3};
  - int[] array = {1,2,3};

예제

```
string[] arrName = new string[] { "1", "2", "3" };

Console.WriteLine("{0}차원", arrName.Rank);

for (int i = 0; i < arrName.Length; i++) {
    Console.WriteLine(arrName[i]);
}

foreach (var name in arrName) {
    Console.WriteLine(name);
}

for (int i = 0; i < arrName.GetLength(0); i++) {
    Console.WriteLine(arrName[i]);
}
```

다차원 배열 Array

- 데이터형식[,] 배열이름 = new 데이터형식[2차원길이(행), 1차원길이(열)];
  - int[,] array = new int[2,3];
  - 인덱스(index) : [0~ i-1 , 0~j-1]
    - array[0,1] = 2;
    - int a = array[1,2];
- 초기화
  - 초기화 하지 않으면, 기본 값으로 초기화 됨
    - 값타입(0, 0.0, false) vs. 참조타입(null)
  - int[,] array = new int[2,3]{ {1,2,3} , {4,5,6} };
  - int[,] array = new int[] { {1,2,3} , {4,5,6} };
  - int[,] array = { {1,2,3} , {4,5,6} };

예제

```
string[,] arrName = new string[,]
{
    { "1", "2", "3" },
    { "11", "12", "13" },
    { "21", "22", "23" }
};

Console.WriteLine("{0}차원", arrName.Rank);

//for (int i = 0; i < arrName.Length; i++) {
//    Console.WriteLine(arrName[i]);
//}

foreach (var name in arrName) {
    Console.WriteLine(name);
}

for (int i = 0; i < arrName.GetLength(0); i++) {
    for (int j = 0; j < arrName.GetLength(1); j++) {
        Console.WriteLine(arrName[i, j]);
    }
}
```

Array관련 메소드와 프로퍼티

- 정적 메소드
  - Clear() : 배열의 모든 요소 초기화
  - Sort() : 배열을 정렬
  - IndexOf() : 배열에서 특정 데이터의 인덱스를 반환
- 인스턴스 메소드
  - **GetLength**(n) : 배열에서 지정한 차원의 길이 반환, 다차원 배열에서 유리
    - 1차원 0 , 2차원 1, 3차원 2, …
- 인스턴스 프로퍼티
  - **Length** : 배열의 전체 길이 반환
  - Rank : 배열의 차원 반환
- 가변배열 (jagged array)
  - 배열이 요소인 배열
    - int[][] test = new int[2][];

Collection

같은 성격을 갖는 데이터의 모음

Collection

기본 Collection	Generic Collection
ArrayList	List<T>
Queue	Queue<T>
Stack	Stack<T>
Hashtable	Dictionary<TKey, TVvalue>
모든 자료형(object)을 담을 수 있다.	T형의 자료형만 담을 수 있다.

- 일반화 프로그래밍 (Generic Programming)
  - 특정한 데이터 유형을 결정하지 않고 일반화된 형태로 구현하는 프로그래밍 기법
  - 형식 매개변수를 이용해 사용하고자 하는 데이터의 유형을 결정한다.
- 일반화 콜렉션
  - 제네릭 프로그래밍 개념에 따라 구현된 컬렉션 자료 구조
  - 기본 콜렉션은 모든 데이터를 object로 형을 변환해서 저장한다.
    이를 다시 사용하려면 원래 형태로 형변환을 진행해야한다. 이에 따른 부하가 발생

List<T>

- ArrayList의 generic 버전
- 배열과 가장 닮은 콜렉션
  - 요소 접근시 [index] 이용
- 크기를 미리 지정할 필요가 없음
  - 가변길이 (Count)
- 주요 메소드
  - Add(), RemoveAt(), Insert()
- 생성 예제
  - List<int> listNumber = new List<int>();
  - T의 자리에 저장하고자 하는 데이터 형 int를 지정한다.
    - listNumber 리스트는 int의 데이터만 저장한다.
  - 나머지는 코드로 확인

Dictionary<TKey, TVal>

- Hashtable의 generic 버전
- 키(key)와 값(value)의 쌍으로 이루어진 데이터
  - key는 중복될 수 없다.
- 크기를 미리 지정할 필요가 없음
  - 가변길이 (Count)
- key를 기준으로 value를 찾을 수 있다.
  - 요소 접근시 [] 이용
  - []안에는 인덱스가 아니라 key를 사용한다.
- 주요 메소드
  - Add(), Remove()
- 생성 예제
  - Dictionary<string, int> dictNumber = new Dictionary<string, int>();
  - key는 string 형, value는 int형
  - 나머지는 코드로 확인