프로그래밍 기초 with C#

12. 함수와 모듈화

학습목표

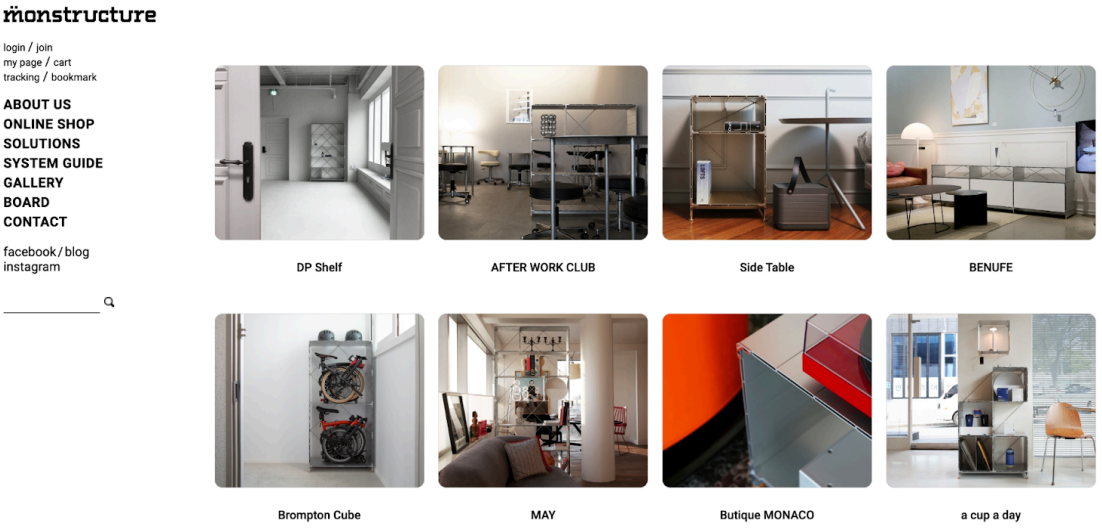
- 모듈화에 대해서 이해할 수 있다.
- 함수를 작성할 수 있다.
- 함수의 동작 원리를 이해할 수 있다.
- 객체의 속성에 대해서 이해할 수 있다.

개요

Sokoban을 구현하면서 별다른 기능이 없음에도 불구하고 점점 프로그램의 복잡성이 증가하고 있음을 느낄 수 있을 것이다. 이대로는 복잡성이 점점 증가하다가 결국 손도 댈 수 없는 프로그램이 된다. 복잡성을 느끼는 것은 한번에 너무 많은 정보와 문맥을 파악해야 하기 때문이다. 따라서 프로그램의 복잡성을 줄이려면 프로그램을 여러 개의 작은 단위(문맥)으로 쪼개야 한다. 이번 시간에서는 모듈화의 개념과 이를 달성시킬 수 있는 기능인 함수에 대해서 알아보도록 하자.

모듈화

혹시 모듈 가구에 대해서 들어본 적 있는가? 모듈 가구란 규격화된 부품을 이용해 내가 원하는 형태로 조립할 수 있는 가구를 뜻한다.



국내 모듈 가구 전문 브랜드 몬스트럭쳐

모듈(Module)은 쉽게 말하면 파츠라고 할 수 있다. 위의 가구를 보면 부품을 사서 내가 원하는대로 조립할 수 있는데 여기서 부품 하나하나를 모듈이라고 할 수 있다. 이와 비슷하게 프로그램에서 **모듈화**(Modularization)라는 것은 전체 프로그램을 여러 파트로 나누는 것을 말한다.

하지만 무엇을 기준으로 나눌 것인지가 중요하다. 모듈 하나하나는 논리적 또는 기능적으로 분리되어 독립적인 일을 수행할 수 있어야 한다. 어떻게 모듈화를 했는가에 따라 프로그램의 유지보수성이 상이해진다. 잘못된 모듈화는 프로그램의 복잡성이 줄어들기는 커녕 되려 복잡성을 증가시킬 수 있다. 다시 말해 파악해야 하는 문맥이 전혀 줄지 않았다는 것이다.

모듈화는 잘하는 방법은 결국 경험이다. 프로그램을 많이 작성하고, 여러 선배 프로그래머의 인사이트를 봐야 늘 수 있다. 또, 일단 구현력이 받춰줘야 모듈화를 할 수 있다. 구현이 안된다면 모듈화도 할 수 없음을 인지해두자.

프로그래밍 언어에서 모듈화를 위한 기능으로는 함수와 클래스가 있다. 클래스가 좀 더 고차원의 모듈이며, 우리는 우선 함수를 이용한 모듈화부터 배울 것이다.

함수

함수(Function)*는 일련의 과정에 이름을 붙인 것이다. 함수는 코드 중복을 제거해 프로그램의 **모듈성**(Modularity)을 높일 수 있다**. 함수에 () 연산자를 붙이면 함수를 호출할 수 있으며, 호출이 되면 함수에 작성된 코드가 전부 실행된다. 함수의 실행이 끝나면 다시 호출했던 지점으로 돌아오며, 반환값을 주기도 한다. 아래의 예시를 보자.

* 때에 따라 메소드(Method), 서브루틴(Subroutine), 프로시저(Procedure) 등으로 불린다.

** 프로그램을 프로시저(Procedure)로 나눠 작성하는 방식을 절차지향 프로그래밍(Procedural Programming)이라 한다.

```
string message = "Hello World!";
// 실행 흐름이 아래의 구문에 도달하면 String.Replace()가 실행되며
// String.Replace()가 끝난 후 다시 이 지점으로 실행 흐름이 돌아온다.
// 또, Replace()는 새로운 문자열을 반환한다.
string message2 = message.Replace("World", "Programming");
```

문법에 대해서 자세히 알아보도록 하자.

문법

함수는 **헤더**(Header)와 **바디**(Body)로 나뉜다.

```
<return-type> <identifier> <parameter-list> <body>
```

함수는 0개 이상의 **매개변수**(Parameter)를 가질 수 있으며, 매개변수는 함수의 바디에서 활용되는 변수다. 매개변수는 () 연산자를 사용할 때 **인수**(Argument)에 의해 초기화 된다. 또 return문을 사용해 호출자에게 결과값을 반환할 수 있다. 아래의 함수를 작성하고 입력해보자.

```
using System;

class Program
{
    static void Main()
    {
        // <return-type>은 int다.
        // <identifier>는 Add다.
        // <parameter-list>는 int 타입의 매개변수 2개로 구성되어 있다.
        int Add(int a, int b)
        {
            // return문은 호출자(Caller)에게 값을 전달한다.
            // 아래의 식이 int 타입이기 때문에 <return-type>이 int인 것이다.
            return a + b;
        }

        int c = Add(10, 20);    // 10과 20은 각각 a와 b에 대한 인수다.
        Console.WriteLine(c);
    }
}
```

하지만 모든 함수가 반환값이 있지는 않다. 그런 경우에 반환 타입은 void를 사용하면 된다.

```
using System;

class Program
{
    static void Main()
    {
        // 아래의 함수는 반환되는 데이터가 없다.
        void Print(string message)
```

```
    {
        if (string.IsNullOrWhiteSpace(message))
        {
            return; // 반환 타입이 void 라도 사용할 수 있다. 함수를 끝내는 데 활용한다.
        }

        Console.WriteLine(message);
    }

    Print("Hello");
  }
}
```

함수는 반드시 호출되고 있는 구문 전에 정의되어 있을 필요는 없다.

```
using System;

class Program
{
  static void Main()
  {
    Print("Hello");

    // 함수를 뒤에서 정의해도 동작한다.
    void Print(string message)
    {
        if (string.IsNullOrWhiteSpace(message))
        {
            return; // 반환 타입이 void 라도 사용할 수 있다. 함수를 끝내는 데 활용한다.
        }
```

```
        Console.WriteLine(message);
    }
  }
}
```

함수가 구문 하나일 경우에는 => 연산자로 식 본문 정의(Expression Body Definition)를 할 수 있다.

```
using System;

class Program
{
  static void Main()
  {
    // int Add(int a, int b)
    // {
    //     return a + b;
    // }
    // 위와 동일하다.
    int Add(int a, int b) => a + b;

    // void Print(string message)
    // {
    //     Console.WriteLine(message);
    // }
    // 위와 동일하다.
    void Print(string message) => Console.WriteLine(message);

    Print(Add(10, 20));
  }
}
```

가변 인수

이쯤에서 String.Format()을 다시 살펴보자. Format()은 인수를 다양하게 넘겨줄 수 있었다.

```
int num1 = 10;
int num2 = 20;
int num3 = 30;
string.Format("{0}", num1);                // 인자 2개
string.Format("{0} {1}", num1, num2);      // 인자 3개
string.Format("{0} {1} {2}", num1, num2, num3); // 인자 4개
```

어떻게 이게 가능할까? 이를 위한 기능인 **가변 인수**(Variadic Argument)라는 기능이 있다. 가변 인수를 사용하려면 params 라는 키워드를 사용하면 된다.

```
void Print(params int[] numbers)
{
  int length = numbers.Length;
  for (int i = 0; i < length; ++i)
  {
    Console.Write(numbers[i]);
  }
}

Print(1, 2, 3); // numbers: [1, 2, 3]
Print();      // 빈 것도 호출할 수 있다. numbers: []
```

스택 프레임

자, 그럼 함수의 동작 원리에 대해서 살펴보자. 어떻게 함수는 모든 작업이 끝났을 때, 다시 호출된 지점으로 돌아올 수 있을까? 함수는 프로세스 주소 공간 중 **스택**(Stack)을 사용하는 데, 스택에 함수를 동작하는 데 필요한 모든 데이터를 저장한다.* 이러한 데이터를 **스택 프레임**(Stack Frame)이라고 한다. 스택 프레임에 호출된 지점의 주소가 저장되어

있기에 함수가 끝나고 다시 호출 지점으로 돌아올 수 있는 것이다. 스택 프레임은 디버그를 위해서도 사용되는 데, 스택 프레임의 정보를 이용하면 함수가 어떤 순서로 호출되고 있는지** 확인할 수 있기 때문이다.

* 그래서 함수 내부에서 사용되는 모든 메모리의 양은 컴파일 시간에 계산되므로 정적 할당 영역이라고 하는 것이다.

** 이를 호출 스택(Call Stack)이라고 한다.

호출 규약

스택 프레임과 관련해 알아야 할 것이 바로 **호출 규약**(Calling Convention)이다. 호출 규약은 함수가 어떻게 호출자로부터 인수를 받을 것이며, 결과값을 호출자에게 어떻게 돌려줄 것인지 규정해 놓은 것이다. 즉, 스택 프레임에 어떤 데이터를 저장할 것인지 나타낸 것이라고 할 수 있다. 호출 규약은 프로세서마다 다르다. ARM은 여기에서 확인할 수 있으며, x86-64은 Microsoft 버전과 그 외의 OS에서 사용하는 System V AMD64가 있다.*

* 원래 32비트 아키텍처 시절만 해도 호출 규약이 굉장히 다양했으나, 64비트 아키텍처가 개발되면서부터는 거의 하나로 합쳐졌다.

```
            int c = Add(1, 2);
00007FF8A18D4961  mov          ecx,1
00007FF8A18D4966  mov          edx,2
00007FF8A18D496B  call         CLRStub[MethodDescPrestub]@7ff8a172e3f8 (07FF8A172E3F8h)
00007FF8A18D4970  mov          dword ptr [rbp+2Ch],eax
00007FF8A18D4973  mov          eax,dword ptr [rbp+2Ch]
00007FF8A18D4976  mov          dword ptr [rbp+44h],eax
```

빨간색 박스는 인수를 전달하는 것을 나타내고, 초록색 박스는 반환값이 전달되는 것을 나타낸다. 여기를 참고하여 비교해보라.

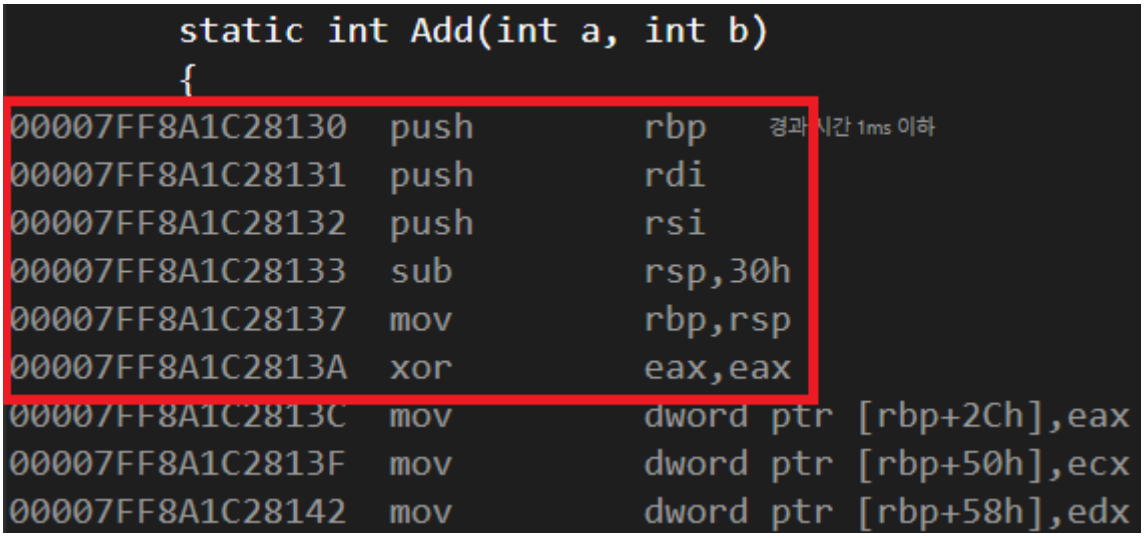
함수로 인한 오버헤드

같은 코드라도 함수로 만들어 호출하게 되면 함수 바디 실행 전 스택 프레임을 구성하는 절차*와 함수를 끝낸 후 스택 프레임을 정리하는 절차**가 추가되기 때문에 **오버헤드***(Overhead)가 생긴다. 다만 이러한 오버헤드를 줄이기 위해 컴파일러가 충분히 최적화를 진행하고 있으며, 실제 테스트 결과로 병목 지점이 되는 게 아닌 이상 걱정할 필요는 없다. 언제나 가독성이 우선임을 명심하자.

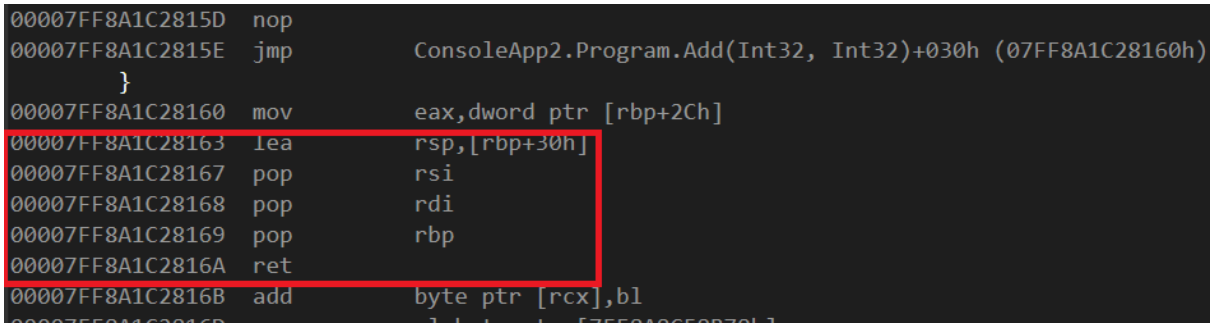
* 프롤로그(Prolog)라고 한다.

** 에필로그(Epilog)라고 한다.

*** 오버헤드란 명령어 실행 준비를 위해서 필요하지만, 실제 명령을 수행하는데는 쓸모없는 리소스(보통은 메모리)를 의미한다.



빨간색 박스 영역이 프롤로그로 스택 프레임을 구성하는 것을 나타낸다.



빨간색 박스 영역이 에필로그로 스택 프레임을 정리하는 것을 나타낸다.

객체의 속성

아래의 코드를 보자.

```
class Program
{
  static void Main()
  {
    int num = 10;
    int num = 20; // 오류!
  }
}
```

우리가 알고 있는 것처럼 같은 이름을 가진 객체를 중복하여 정의할 수 없다. 다시 아래의 예제를 보자.

```
class Program
{
  static void Main()
  {
    int num = 10;

    void Foo()
    {
      int num = 20; // 오류가 아님
      System.Console.WriteLine(num); // 누가 출력될까?
    }
  }
}
```

분명히 같은 이름을 썼음에도 불구하고, 컴파일 오류가 발생하지 않는다. 이를 이해하려면 객체의 속성에 대해서 이해해야 한다.

범위

먼저 **범위**(Scope)다. 범위는 객체가 유효한 영역을 말한다. 우리가 여지껏 사용한 객체는 모두 블록(Block) 범위를 가지고 있으며, 이를 **지역 변수**(Local Variable)라고 한다. 반대로 어디서든 접근할 수 있는 변수를 **전역 변수**(Global Variable)라고 한다.

```
class Program
{
  static void Main()
  {
    int num = 10; // 이 변수는 정의된 후부터 Main() 끝날 때까지 유효하다.

    {
      int num2 = 20; // 이 변수는 정의된 후부터 블록이 끝날 때까지 유효하다.
    }
    num2 = 30; // 오류!
```

```
    for (int i = 0; i < 10; ++i) // i는 for 끝날 때까지 유효하다.
    {
        System.Console.WriteLine("Hello");
    }

    do
    {
        int i = 20; // do 블록 끝날 때까지 유효하다.
    } while (i == 20); // 오류!

    void Foo()
    {
        int num = 20; // 이 변수는 정의된 후부터 Foo() 끝날 때까지 유효하다.
        System.Console.WriteLine(num);
    }
  }
}
```

수명

객체의 **수명**(Lifetime)은 객체가 유효한 시간이다. 지역 변수는 범위를 벗어나게 되면 바로 수명이 끝나게 된다. 스택 프레임에 의해서 알아서 정리가 될 것이기 때문이다. 다만, 프로세스 주소 공간 중 힙이나 데이터 영역에 저장되어 있는 객체는 프로그램이 끝날 때까지 수명이 유지될 수 있다.

인수 전달 방식

인수를 전달할 때는 2가지 방식이 있다. 객체의 복사본을 전달하거나, 객체에 접근할 수 있는 주소를 전달하는 것이다.* 프로그래머는 이 둘을 명확히 인지하고 있어야 하는데, 내가 쓴 코드가 원본에 영향을 주는지를 알아야 하기 때문이다. 객체의 복사본이 전달되는 경우 말 그대로 복사본이기 때문에 원본에 영향을 주지 않으나, 주소가 전달된 경우는 원본에 접근할 수 있기 때문에 영향을 주게 된다.

* 전자를 Pass By Value 혹은 Call By Value, 후자를 Pass By Reference 혹은 Call By Reference라고 한다.

기본적으로 인수 전달 방식은 복사본을 전달(Pass By Value)하는 것이다.

```
class Program
{
  static void Main()
  {
    int num = 10;
    Foo(num);

    void Foo(int x)
    {
      x = 20; // num에 영향을 주지 않는다.
    }

    int[] arr = { 1, 2, 3};
    Boo(arr);

    void Boo(int[] x)
    {
      x[0] = 10; // arr에 영향을 준다.
      x = new int[3]; // 지역 변수 x가 가리키는 배열이 달라진다.
      x[0] = 7;  // arr에 영향을 주지 않는다.
    }
  }
}
```

주소를 전달(Pass By Reference)하고 싶다면 ref를 사용할 수 있다.

```
class Program
{
  static void Main()
  {
    int num = 10;
    Foo(num);

    void Foo(ref int x)
    {
      x = 20; // num도 20이 된다.
```

```
    }

    int[] arr = { 1, 2, 3};
    Boo(arr);

    void Boo(ref int[] x)
    {
        x[0] = 10;     // arr에 영향을 준다.
        x = new int[3]; // arr이 가리키고 있는 배열이 바뀐다.
        x[0] = 7;
    }
  }
}
```

주소를 전달하되 의도를 명확하게 하기 위해 in과 out을 사용할 수 있다.

더해보기

1. 수업 내용을 복기하며 아래 내용을 정리해보세요.
    1.1. 모듈화란 무엇인가요?
    1.2. 스택 프레임이란 무엇인가요?
    1.3. Pass(Call) By Value와 Pass(Call) By Reference의 차이점은 무엇인가요?
    1.4. 객체의 속성에는 무엇이 있으며, 각각의 의미는 무엇인가요?
    1.5. 함수에는 어떤 오버헤드가 있나요?
2. 함수에서 정의된 매개변수의 개수와 인수의 개수가 다르면 어떤 일이 발생하나요?
3. 함수를 호출해서 인수를 적을 때 위치에 맞춰서 전달할 수도 있지만, 이름을 이용해서 전달하거나(Named Argument), 일부만 전달할 수도(Optional Argument) 있습니다. 여기를 참고해 둘의 사용 방법을 정리해보세요.
4. 주석에 관한 여러 규칙이 있듯, 함수도 잘 작성하기 위한 여러 인사이트가 있습니다. 아래 자료를 참고해서 정리해보세요.
    4.1. https://dkje.github.io/2020/08/03/CleanCodeSeries2-copy/
    4.2. https://youtu.be/th7n1rmlO4I?si=RT762NUkXSrmK69x
    4.3. https://www.steveonstuff.com/2022/01/27/no-such-thing-as-clean-code

    4.4. https://news.hada.io/topic?id=9002&utm_source=slack&utm_medium=bot&utm_campaign=T04DHQPSW7J
    4.5. https://jojoldu.tistory.com/697
    4.6. https://jojoldu.tistory.com/703
    4.7. https://jojoldu.tistory.com/721
5. Sokoban에서 함수로 모듈화를 해보세요.

참고자료

- C# docs - get started, tutorials, reference. | Microsoft Learn
- Assembly 2: Calling convention – CS 61 2018
- C# 개발자를 위한 Win32 DLL export 함수의 호출 규약 (3) - x64 환경의 __fastcall과 Name mangling
- Calling convention - Wikipedia
- Linux x64 Calling Convention: Stack Frame - Red Team Notes
- x64 calling convention | Microsoft Learn
- 모듈
- 버그가 생기는 이유와 기초 예방법