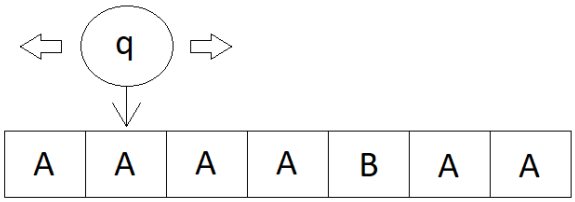
Jacek Matulewski, e-mail: *jacek@fizyka.umk.pl*
Strona wykładu: *http://www.fizyka.umk.pl/~jacek/dydaktyka/programowanie/*

Przykład: maszyna Turinga

Ostatni rozdział tej części poświęcony jest w całości jednej, nieco większej aplikacji, która ma być symulatorem maszyny Turinga. Przeznaczenie aplikacji, choć interesujące, jest tylko pretekstem do tego, żeby zaprezentować większy projekt. Proszę potraktować cały rozdział jak ćwiczenie i przeczytać go przy komputerze z uruchomionym Visual Studio.

Maszyna Turinga

Maszyna Turinga składa się z trzech elementów: taśmy, głowicy i programu. Program jest stały, będziemy go wczytywać z pliku, ale stan głowicy i zawartość taśmy będą się zmieniać. Przyjmijmy, że stan głowicy (nazywany rejestrem) kodowany jest małymi literami alfabetu łacińskiego *a* – *z*. Taśma jest natomiast zbiorem wartości kodowanych dużymi literami *A* – *Z* poza literami *R* i *L*, które są zarezerwowane dla oznaczenia komend przesuwających głowicę na taśmie. Głowica wskazuje na jedną wartość zapisaną na taśmie nazywaną wartością bieżącą. Stan maszyny będzie wobec tego kompletnie wyznaczana przez stan głowicy, jej pozycję oraz zawartość taśmy. Taśma, w naszym przypadku skończona, w oryginalnym sformułowaniu mogła być nieskończona (por. zadanie 3 poniżej).



Rysunek 10.1. Schemat maszyny Turinga

Kod programu składa się z uporządkowanych „czwórek”, w których na kolejnych pozycjach znajdują się: bieżący stan głowicy, bieżąca wartość taśmy, komenda lub nowa wartość taśmy oraz nowy stan głowicy. Program nie jest wykonywany linia po linii, a właściwa linia programu jest ustalana na podstawie bieżącego stanu głowicy (rejestru) i wartości we wskazywanym przez nią miejscu na taśmie. Jeżeli w programie jest obecna linia, której dwa pierwsze elementy odpowiadają rejestrowi i bieżącej wartości, to jest ona wykonywana. Jeżeli nie, działanie programu kończy się. Poprawny program nie może zawierać dwóch linii o tych samych dwóch pierwszych elementach –byłby wówczas niejednoznaczny.

Możliwe są trzy komendy umieszczone w trzecim elemencie „czwórek” zapisanych w liniach programu:

1. przesuń głowicę w lewo, co jest zakodowane przez literę *L*,
2. przesuń głowicę w prawo, czemu odpowiada litera *R*,
3. dla pozostałych dużych liter zmieniana jest wartość na taśmie w bieżącej pozycji głowicy na wartość z trzeciego miejsca czwórki.

Przykładowy program składający się tylko z trzech linii może wyglądać następująco:
`qASs` – jeżeli stan głowicy to *q*, a bieżąca wartość *A*, to zamień wartość z *A* na *S* i ustaw głowicę na *s*
`sSRq` – jeżeli stan głowicy to *s*, a bieżąca wartość *S*, to przesuń głowicę w prawo i zmień jej stan na *q*
`qBRb` – jeżeli stan głowicy to *q*, a bieżąca wartość *B*, to przesuń głowicę w prawo i zmień jej stan na *b*

Aby opisać stan maszyny jednym łańcuchem użyjemy prostego formatu, w którym do stanu taśmy (drukowane litery) dodajemy jedną małą literę oznaczającą stan głowicy, umieszczoną przed wartością na którą wskazuje

głowica. Dla przykładu stan maszyny widoczny na rysunku 10.1 w tym formacie jest zapisywany jako `AqAAABAA`. Oznacza to, że taśma zawiera wartości `AAAABAA`, głowica ustawiona jest na drugiej pozycji (indeks 1, bieżąca wartość to *A*), a jej stan równy jest *q*. W takim formacie powinien być także początkowy stan maszyny, który program będzie odczytywał z pliku.

Spróbujmy prześledzić działanie maszyny z powyższym programem dla stanu z rysunku 10.1 (por. tabela 10.1). Początkowy stan głowicy i wskazywana przez nią wartość (*q*, *A*) odpowiadają pierwszej linii programu `qASs`. Po jej wykonaniu stan maszyny zmieni się na `AsSAABAA`. To oznacza, że teraz odpowiednia jest druga linia programu `sSRq` (nie dlatego, że następuje po pierwszej, a dlatego, że wskazuje na nią stan głowicy i bieżąca wartość taśmy). Zmienia ona stan maszyny na `ASqAABAA`. Teraz ponownie właściwa jest pierwsza linia programu, która zmienia bieżącą wartość na *S*, a stan głowicy na *s*. To znowu powoduje wybranie linii, która przesuwa głowicę. Te dwie linie są powtarzane, aż głowica natknie się na wartość *B*. Wówczas wykonywana jest trzecia linia programu, która zmienia stan głowicy na *b* i przesuwa ją w prawo do wartości *A*. Zwróćmy jednak uwagę, że w programie nie ma linii rozpoczynającej się od *b* i *A*. To oznacza, że działanie programu się kończy i maszyna pozostaje w stanie `ASSSBbAA`. Cały przebieg programu widoczny jest w tabeli 10.1. Warto go prześledzić, bo to będzie też przykład, na którym będziemy testować działanie programu.

Tabela 10.1. Ewolucja stanu maszyny

Stan maszyny	Nast. polecenie
`AqAAABAA`	`qASs`
`AsSAABAA`	`sSRq`
`ASqAABAA`	`qASs`
`ASsSABAA`	`sSRq`
`ASSqABAA`	`qASs`
`ASSsSBAA`	`sSRq`
`ASSSqBAA`	`qBRb`
`ASSSBbAA`	Brak

Maszyny Turinga są interesujące z kilku względów. Dla informatyków są interesujące, gdyż udowodniono, że zwykłe komputery są równoważne maszynom Turinga, można więc je badań na prostym modelu. W filozofii i psychologii poznawczej – gdyż maszyna Turinga jest wygodnym narzędziem przy precyzowaniu pojęć i problemów procesu poznawania i sztucznej inteligencji.

Dodawanie plików tekstowych do projektu

Stwórzmy projekt aplikacji konsolowej .NET lub .NET Core. Do projektu dodajmy dwa pliki tekstowe[1] *program.txt* i *taśma.txt*, a następnie ustawmy ich własność *Kopiuj do katalogu wyjściowego* na *Zawsze kopiuj*, dzięki czemu pliki te będą podczas kompilacji kopiowane do katalogu z plikiem wykonywalnym programu.

W pliku *program.txt* umieść trzy linie z kodem programu:

```
qASs
sSRq
qBRb
```

natomiast w pliku *taśma.txt* umieść początkowy stan maszyny:

```
AqAAABAA
```

Następnie wróć do pliku *Program.cs* i dodaj do metody `Main` polecenia wczytujące oba pliki (listing 10.1).

Listing 10.1. Kod wczytujący i prezentujący zawartość dwóch plików tekstowych dodanych do projektu

```
static void Main(string[] args)
```

[1] Opis dodawania plików tekstowych do projektów jest w rozdziale 7. w podrozdziale „Pliki tekstowe”.

```
{
    string[] kodProgramu = File.ReadAllLines("program.txt");
    string łańcuchOpisującyStanMaszyny = File.ReadAllText("taśma.txt");

    Console.WriteLine("Początkowy stan maszyny: " + łańcuchOpisującyStanMaszyny);
    Console.WriteLine("Program:");
    foreach (string linia in kodProgramu) Console.WriteLine(linia);
}
```

Analiza zapisu taśmy

Musimy przygotować dwie metody pomocnicze: pierwsza będzie weryfikować i parsować kod programu do czwórek, które przechowamy w krotkach. Druga zanalizuje łańcuch z pliku *taśma.txt*, rozdzielając zawartość taśmy oraz stan i położenie głowicy. Zacznijmy od drugiej z tych metod. Na listingu 10.2 widoczna jest metoda `analizujOpisStanuMaszyny`, która zmienia łańcuch w opisanym wyżej formacie na trzy elementy (zwracane w krotce): zawartość taśmy, znak będący stanem głowicy oraz liczba całkowita z położeniem głowicy na taśmie. Główną częścią tej metody jest pętla `for`, która przebiega cały łańcuch w poszukiwaniu małej litery oznaczającej głowicę. Wszystkie pozostałe litery muszą być wielkie, dlatego aby to sprawdzić (wykluczyć obecność kolejnych małych liter), pętla jest kontynuowana także po znalezieniu pierwszej małej litery. Metodzie `analizujOpisStanuMaszyny` towarzyszy metoda `pobierzŁańcuchOpisującyStanMaszyny`, która działa odwrotnie do pierwszej. Z trzech elementów tj. łańcucha z zawartością taśmy, znaku stanu głowicy i jej położenia tworzy łańcuch opisujący stan maszyny. Wykorzystuje do tego metodę `Insert` z klasy `string`, wstawiającą jeden łańcuch w drugi.

Listing 10.2. Metody odpowiedzialne za konwersję stanu maszyny Turinga z i do łańcucha

```
class Program
{
    #region Łańcuch opisujący stan maszyny
    static (char[] taśma, char stanGłowicy, int położenieGłowicy)
analizujOpisStanuMaszyny (string łańcuchOpisującyStanMaszyny)
    {
        char początkowyStanGłowicy = ' ';
        int początkowePołożenieGłowicy = -1;
        for (int i = 0; i < łańcuchOpisującyStanMaszyny.Length; ++i)
        {
            if (łańcuchOpisującyStanMaszyny[i] == 'L' ||
                łańcuchOpisującyStanMaszyny[i] == 'R')
                throw new Exception("Taśma nie może zawierać wartości L lub R");
            char c = łańcuchOpisującyStanMaszyny[i];
            if (char.IsLower(c))
            {
                if (początkowePołożenieGłowicy != -1)
                    throw new Exception(
                        "Znaleziono więcej niż jeden znak oznaczający głowicę");
                początkowyStanGłowicy = c;
                początkowePołożenieGłowicy = i;
                łańcuchOpisującyStanMaszyny =
                    łańcuchOpisującyStanMaszyny.Remove(początkowePołożenieGłowicy, 1);
            }
            else
```

```
            {
                if (c < 'A' || c > 'Z')
                    throw new Exception(
                        "Niepoprawna wartość na taśmie (" + c + ") na pozycji " + i);
            }
        }
        if (początkowePołożenieGłowicy < 0)
            throw new Exception(
                "Nie znaleziono znaku oznaczającego głowicę");
        return (łańcuchOpisującyStanMaszyny.ToCharArray(),
                początkowyStanGłowicy,
                początkowePołożenieGłowicy);
    }

    static string pobierzŁańcuchOpisującyStanMaszyny(
        (char[] taśma, char stanGłowicy, int położenieGłowicy) stanMaszyny)
    {
        string s = new string(stanMaszyny.taśma);
        s = s.Insert(stanMaszyny.położenieGłowicy, stanMaszyny.stanGłowicy.ToString());
        return s;
    }
    #endregion
    ...
```

Wczytywanie i parsowanie kodu programu

Metoda `parsujProgram` (listing 10.3) jest drugą z zapowiedzianych metod. Metoda ta sprawdza kod programu i tworzy na jego podstawie zestaw czwórek. Wykorzystuje do tego niewielką metodę `czyCzwórkaPoprawna`, która sprawdza, czy w linii kodu małe litery są na pierwszej i czwartej pozycji, a duże – na drugiej i trzeciej, i czy są to litery z zakresu *a-z* lub *A-Z*. Ponadto metoda `parsujProgram` sprawdza, czy wśród czwórek nie ma dwóch, które zaczynają się od takich samych dwóch pierwszych znaków. Prawidłowe czwórki zapisywane są w słowniku `czwórki`. Zwróćmy uwagę, że przy deklaracji tego słownika jako zmiennej lokalnej w metodzie `parsujProgram` nie korzystamy jawnie z typu `SortedList<(char, char), (char, char)>`, a ze zdefiniowanego aliasu `Czwórki`. Kluczem i wartością w tym słowniku są pary znaków (krotki), w którym pierwszym elementem jest stan głowicy, a drugim wartość na taśmie. Pierwsza para oznacza bieżący stan maszyny, a druga – jej nowy stan zmieniony przez tę linię programu.

Listing 10.3. Kod odpowiedzialny za parsowanie kodu programu dla maszyny Turinga

```
using System;
using System.Collections.Generic;
using System.IO;

namespace MaszynaTuringa
{
    using Czwórki = SortedList<(char stanGłowicy, char wartośćNaTaśmie), (char
nowyStanGłowicy, char nowaWartośćNaTaśmie)>;

    class Program
    {
```

```
        #region Łańcuch opisujący stan maszyny
        ...
        #endregion

        #region Parsowanie kodu programu
        static bool czyCzwórkaPoprawna(string linia)
        {
            Func<char, bool> isLowerLetter = (char c) => c >= 'a' && c <= 'z';
            Func<char, bool> isUpperLetter = (char c) => c >= 'A' && c <= 'Z';
            return isLowerLetter(linia[0]) && isUpperLetter(linia[1]) &&
isUpperLetter(linia[2]) && isLowerLetter(linia[3]);
        }

        static Czwórki parsujProgram(string[] kodProgramu)
        {
            Czwórki czwórki = new Czwórki();
            foreach (string linia in kodProgramu)
            {
                if (string.IsNullOrWhiteSpace(linia)) continue;
                if (!czyCzwórkaPoprawna(linia)) throw new Exception("Niepoprawna linia
kodu (" + linia + ")");
                (char, char) stan = (linia[0], linia[1]);
                (char, char) nowyStan = (linia[3], linia[2]);
                if (czwórki.ContainsKey(stan)) throw new Exception("Program nie może
zawierać dwóch poleceń o takim stanie głowicy i wartości na taśmie");
                czwórki.Add(stan, nowyStan);
            }
            return czwórki;
        }
        #endregion
```

Wykonywanie programu

I wreszcie przejdźmy do kluczowej metody całego symulatora maszyny Turinga – metody `wykonajProgram` odpowiedzialnej za wykonywanie sparsowanego programu (zbioru czwórek). Widoczna jest na listingu 10.4. Metodzie tej towarzyszy metoda `znajdźPolecenie`, która na podstawie stanu głowicy i bieżącej wartości wybiera właściwą czwórkę programu. Metoda ta zwraca parę dwóch wartości: nową wartość głowicy oraz polecenie (*L* lub *R*) lub nową wartość taśmy. Jeżeli nie uda się znaleźć pasującej czwórki, metoda `znajdźPolecenie` zwraca wartość `null` (jest to możliwe, bo zwracana przez nią krotka jest opakowana w `Nullable`). Metoda `wykonajProgram` w warunku pętli `while` odczytuje wartość zwracaną przez metodę `znajdźPolecenie` i jednocześnie sprawdza, czy jest ona różna od `null` (por. komentarz do listingu 6.4 z rozdziału 6.). Jeżeli jest, pętla się kończy. W każdej iteracji pętli `while` wykonywana jest instrukcja `switch`, która sprawdza, czy trzeci znak czwórki zawiera nową wartość na taśmie, czy polecenia przesunięcia głowicy. Reaguje na te polecenia, zmieniając wcześniej stan głowicy. Kolejne stany maszyny zapisywane są w postaci odpowiednio sformatowanych łańcuchów przekonwertowanych metodą `pobierzŁańcuchOpisującyStanMaszyny` (listing 10.2) do listy łańcuchów `historia`. Ta lista zostanie zwrócona przez metodę `wykonajProgram` i po jej zakończeniu pokazana w metodzie `Main`.

Listing 10.4. Metoda wykonajProgram odpowiedzialna jest za działanie maszyny Turinga

```
static (char nowyStanGłowicy, char nowaWartośćLubPolecenie)? znajdźPolecenie(
    char stanGłowicy, char wartośćNaTaśmie, Czwórki program)
```

```
{
    (char stanGłowicy, char wartośćNaTaśmie) bieżącyStan = (stanGłowicy,
wartośćNaTaśmie);
    if (program.ContainsKey(bieżącyStan)) return program[bieżącyStan];
    else return null;
}

static List<string> wykonajProgram(
    (char[] taśma, char stanGłowicy, int położenieGłowicy) stanMaszyny, Czwórki program)
{
    List<string> historia = new List<string>();
    (char nowyStanGłowicy, char nowaWartośćLubPolecenie)? polecenie;
    while((polecenie = znajdźPolecenie(stanMaszyny.stanGłowicy,
stanMaszyny.taśma[stanMaszyny.położenieGłowicy], program)) != null)
    {
        stanMaszyny.stanGłowicy = polecenie.Value.nowyStanGłowicy;
        switch (polecenie.Value.nowaWartośćLubPolecenie)
        {
            case 'L':
                stanMaszyny.położenieGłowicy--;
                break;
            case 'R':
                stanMaszyny.położenieGłowicy++;
                break;
            default:
                stanMaszyny.taśma[stanMaszyny.położenieGłowicy] =
polecenie.Value.nowaWartośćLubPolecenie;
                break;
        }
        historia.Add(pobierzŁańcuchOpisującyStanMaszyny(stanMaszyny));
    }
    return historia;
}
```

Teraz możemy wrócić do metody `Main` i umieścić w niej wywołania powyższych metod otoczone sekcją `try..catch`. Zwróćmy uwagę, że tylko w tej metodzie znajdują się polecenia wyświetlające komunikaty w konsoli. To oznacza, że pozostała część kodu nie jest związana z żadnym sposobem komunikacji z użytkownikiem i łatwo byłoby ją przenieść np. do aplikacji okienkowej.

Listing 10.5. Metoda Main organizująca wczytywanie i analizowanie plików, a następnie uruchomienie programu maszyny Turinga

```
static void Main(string[] args)
{
    string[] kodProgramu = File.ReadAllLines("program.txt");
    string łańcuchOpisującyStanMaszyny = File.ReadAllText("taśma.txt");

    Console.WriteLine("Początkowy stan maszyny: " + łańcuchOpisującyStanMaszyny);
    Console.WriteLine("Program:");
    foreach (string linia in kodProgramu) Console.WriteLine(linia);
```

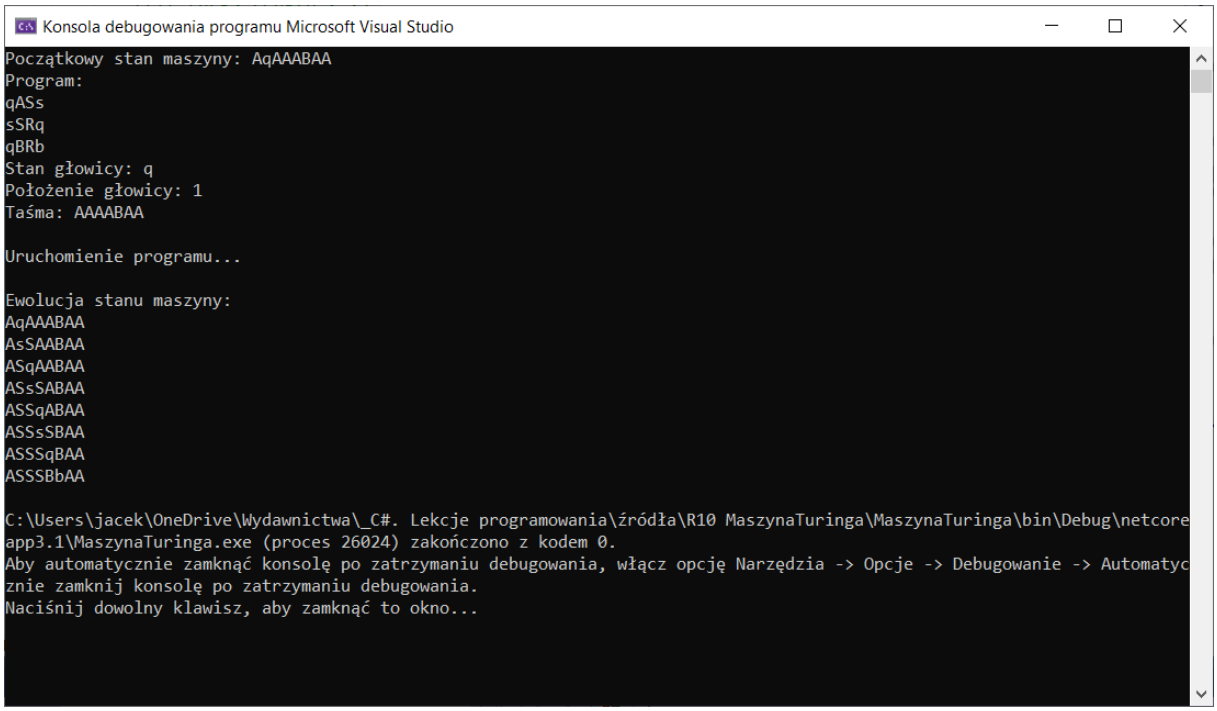
```
        try
        {
            //analiza taśmy
            var stanMaszyny = analizujOpisStanuMaszyny(łańcuchOpisującyStanMaszyny);
            Console.WriteLine("Stan głowicy: " + stanMaszyny.stanGłowicy);
            Console.WriteLine("Położenie głowicy: " + stanMaszyny.położenieGłowicy);
            Console.WriteLine("Taśma: " + new string(stanMaszyny.taśma));

            //parsowanie programu
            Czwórki program = parsujProgram(kodProgramu);

            Console.WriteLine("\nUruchomienie programu...");
            List<string> historia = wykonajProgram(stanMaszyny, program);

            Console.WriteLine("\nEwolucja stanu maszyny:");
            foreach (string linia in historia)
                Console.WriteLine(linia);
        }
        catch (Exception exc)
        {
            ConsoleColor color = Console.ForegroundColor;
            Console.ForegroundColor = ConsoleColor.Red;
            Console.Error.WriteLine("Błąd: " + exc.Message);
            Console.ForegroundColor = color;
        }
    }
```

I wreszcie uruchommy program (*F5*). Powinniśmy zobaczyć wczytane z pliku stan maszyny i program, a następnie listę kolejnych stanów, które powinny zgadzać się z tymi z tabeli 10.1. Działanie maszyny powinno zakończyć się po siedmiu krokach.



Rysunek 10.2. Ewolucja stanu maszyny Turinga w wyniku działania programu

Argumenty linii komend

W tej chwili program wczytuje stan maszyny Turinga z pliku *taśma.txt* i wykonuje program zapisany w pliku *program.txt*. Uogólnijmy projekt w taki sposób, żeby oba pliki były wskazywane z linii komend (listing 10.6). Argumenty linii komend są przekazywane do metody `Main` w tablicy łańcuchów `args`. Załóżmy, że pierwszy element tej tablicy będzie zawierał ścieżkę do pliku z kodem programu, a drugi – do pliku z łańcuchem opisującym stan maszyny. Zanim użyjemy podanych ścieżek, sprawdzamy czy wskazywane przez nie pliki w ogóle istnieją. Używamy w tym celu statycznej metody `File.Exists`. Jeżeli argumentów jest mniej niż dwa lub któryś z plików nie istnieje, wyświetlamy stosowny komunikat i kończymy działanie programu.

Listing 10.6. Ustalanie nazw plików z danymi wejściowymi na podstawie argumentów linii komend

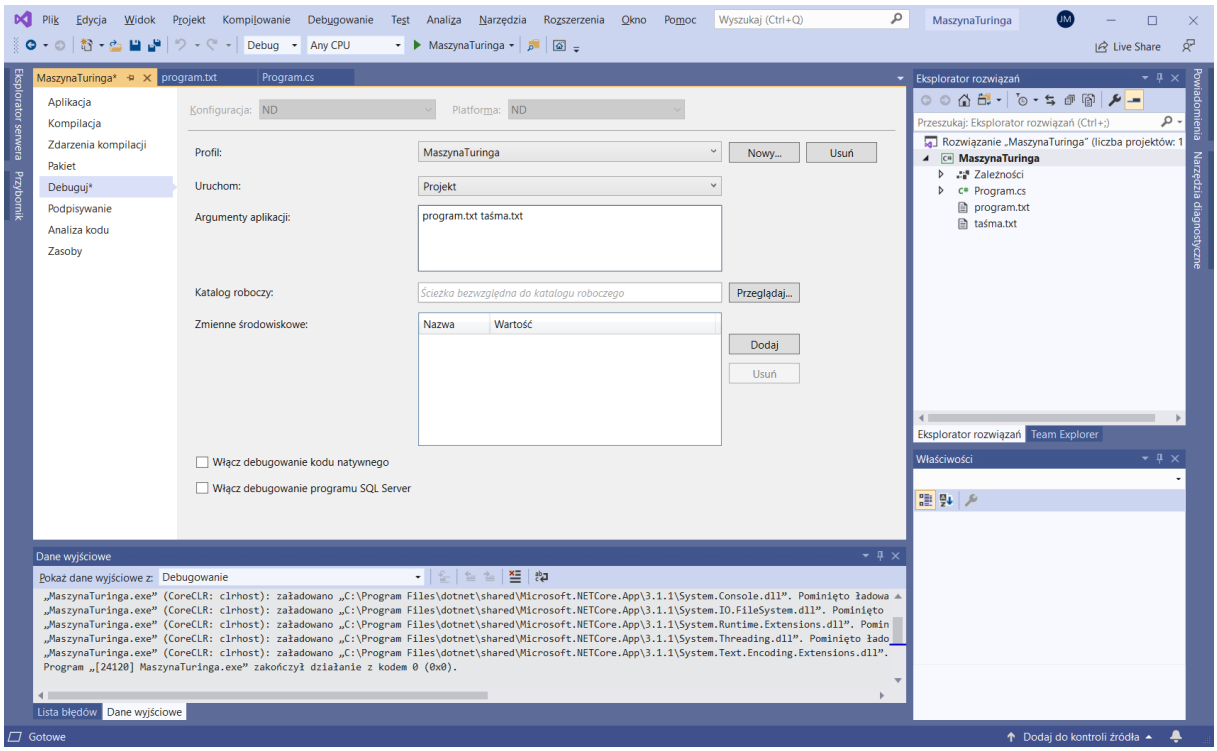
```
static void Main(string[] args)
{
    if (args.Length < 2)
    {
        Console.Error.WriteLine("Do uruchomienia program wymaga dwóch argumentów");
        return;
    }

    string ścieżkaPlikuProgramu = args[0];
    if (!File.Exists(ścieżkaPlikuProgramu))
    {
        Console.Error.WriteLine("Brak pliku z kodem programu " + ścieżkaPlikuProgramu);
        return;
    }
    else Console.WriteLine("Plik z kodem programu: " + ścieżkaPlikuProgramu);
    string[] kodProgramu = File.ReadAllLines(ścieżkaPlikuProgramu);

    string ścieżkaPlikuTaśmy = args[1];
    if (!File.Exists(ścieżkaPlikuTaśmy))
    {
        Console.Error.WriteLine("Brak pliku ze stanem maszyny " + ścieżkaPlikuTaśmy);
        return;
    }
    else Console.WriteLine("Plik ze stanem maszyny: " + ścieżkaPlikuTaśmy);
    string łańcuchOpisującyStanMaszyny = File.ReadAllText(ścieżkaPlikuTaśmy);

    Console.WriteLine("Początkowy stan maszyny: " + łańcuchOpisującyStanMaszyny);
    Console.WriteLine("Program:");
    foreach (string linia in kodProgramu) Console.WriteLine(linia);
```

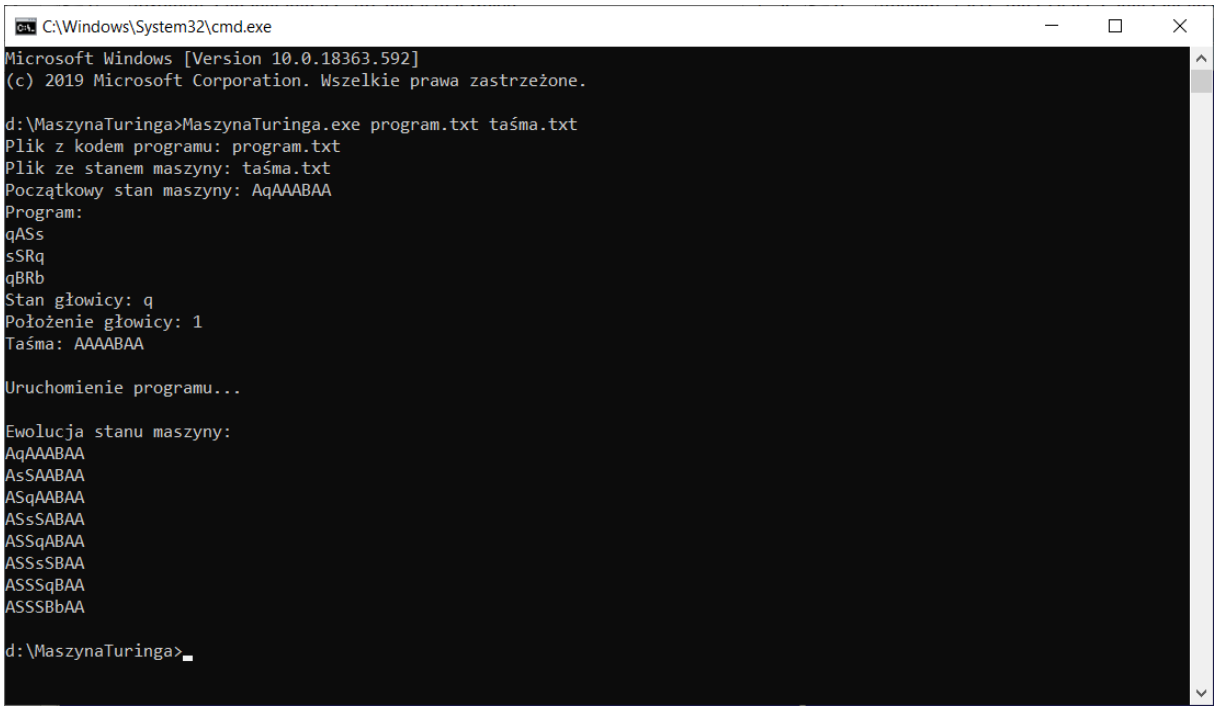
Po tych zmianach uruchomienie programu klawiszem *F5* lub kombinacją *Ctrl+F5* spowoduje wyświetlenie komunikatu „Do uruchomienia program wymaga dwóch argumentów”. Czy to oznacza, że z powodu braku argumentów linii komend nie będziemy mogli wygodnie testować programu? Oczywiście, nie. Możemy podać odpowiednie argumenty w ustawieniach projektu. W menu *Projekt* środowiska Visual Studio wybierzmy ostatnią pozycję *Właściwości MaszynaTuringa...* i przejdźmy na zakładkę *Debuguj* (rysunek 10.3). W widocznym na niej polu *Argumenty aplikacji* wpiszmy ścieżki do plików np. dotychczas używane *program.txt* i *nazwa.txt* (brak katalogu spowoduje, że pliki te będą szukane w katalogu aplikacji). Teraz możemy uruchomić program.



Rysunek 10.3. Podawanie argumentów linii komend używanych przy uruchamianiu aplikacji w środowisku VS

Dystrybucja programów

Projekt, który przygotowaliśmy możemy uruchomić na dowolnym komputerze z platformą .NET lub .NET Core (w zależności od wybranego typu projektu). Niezbędną do uruchomienia wersję platformy możemy odczytać z ustawień projektu z zakładki *Aplikacja*. Widoczna jest (i można ją zmienić) w rozwijanej liście *Docelowa struktura*. W przypadku projektu .NET wystarczy skopiować tylko plik z rozszerzeniem *.exe* i oczywiście dwa pliki tekstowe z programem i stanem maszyny. W przypadku platformy .NET Core plików jest więcej. Oprócz pliku *MaszynTuringa.exe* należy skopiować także plik *MaszynaTuringa.dll* oraz *MaszynaTuringa.runtimeconfig.json*. Na rysunku 10.4 widoczna jest konsola Windows po uruchomieniu aplikacji poza środowiskiem Visual Studio.



Rysunek 10.4. Działanie ostatecznej wersji programu uruchomionego poza środowiskiem Visual Studio