

# Введение

**Ассемблер** - машинно-ориентированный язык программирования низкого уровня. Иными словами, это язык управления битами в регистрах и оперативной памяти. Исполнитель Ассемблера - микропроцессор, реже высокоуровневая прослойка операционной системы. Ассемблер - язык старый, датируется 1947 годом, содержит множество диалектов, описать которые можно как множество частных случаев реализации идеи машинно-ориентированного языка. Литература встречается, вся она, скорее всего, пролежала несколько десятилетий на полке библиотеки, прежде чем попасть в руки. Поэтому примеры из книг часто не работают или требуют адаптации под современные реалии.

Чтобы что-нибудь написать, приходится пройти через поиск на просторах Интернета. При этом то, что Вам нужно, попадает только каждый 10 или 20 раз. Периодически встречаются посты с заведомо ложной информацией, распознать которую новичку не под силу. Программные коды, взятые из литературы, обладают высокой скоростью роста сложности и потому не пригодны для начинающих.

Все это создает высокий входной порог сложности материала данной темы.

## Эзотерические языки программирования

Мы постараемся описать траекторию, по которой удастся преодолеть большую часть подобных трудностей.

Материалы изложены в порядке возрастания сложности.

Сразу писать на Ассемблере не удастся. Смиритесь. Для начала требуется познакомиться с чем-то попроще. Эзотерические языки - это языки программирования, созданные ради шутки или для проведения эксперимента по расширению программистских возможностей. Изучение более простых, чем Ассемблер, представителей этой языковой группы позволит упростить последующее понимание Ассемблера.

## BrainF

Познакомьтесь с младшим братом Ассемблера: BrainF. [BrainF Developer](#).

В Интернете пишут, что **BF** - это язык программирования низкого уровня, который обладает высокой производительностью. Все это позволяет его запускать вне зависимости от платформы, лишь бы был интерпретатор языка. На JavaScript, Python или VBA - не важно. Интерпретатор, как правило, крайне легковесный. [Интерпретатор BF размером 160 байт](#).

Периодически задаются вопросы: "А есть ли в таком языке какой-то практический смысл?"

Ответы на подобные вопросы, как ни странно, замалчивают. Постараемся, как можем, на них ответить.

Начнем с анекдота.

"На просторах социальной сети ВКонтакте была высказана мысль: "В **BF** проще войти, чем в Ассемблер. Поэтому можно писать на framework, который компилирует из **BF** в конкретный

Ассемблер. Такой путь от собеседования до первого сданного проекта намного короче."

В этот момент, где-то в далекой-далекой галактике появилась вакансия машинно-ориентированного программиста на **Java/BF**."

Почему именно **BF**?

- Максимальная простота: **BF** прост, учить в нем, в принципе, нечего, в нем всего 8 команд.
- Дает возможность поработать со стеком вызова функций и понять - что же это такое.
- Дает сильный стимул для развития алгоритмического мышления и неожиданно упрощает последующее понимание Ассемблера.
- Язык крайне недооценен, так как он очень прост в схемотехнической реализации. В теории, работающие на нем микропроцессоры будут ещё компактнее и проще.

Посмотрим - как выглядит **BF** :

- +- - инкремент декремент,
- >< - смещение на ячейку выше, ниже,
- [] - цикл открыть, закрыть,
- „ - ввод,вывод одного символа.

Все! Можно начинать "ваять"! Если по каким-то причинам нас не устраивает [BrainF Developer](#)

, то всегда можно попробовать написать свой интерпретатор языка. Предупреждаю, существуют ASCII и Unicode модели символов, программы с их учетом могут различаться.

В Интернете можно встретить Вызов/Чёллендж/Challenge, брошенный другим программистам: "Напиши интерпретатор **BF** на каждом языке программирования, который ты знаешь".

Встречаются программные коды на C++ и C#. Реже на Java. Пополняю копилку **BF** интерпретаторов на языке Julia. Не стоит воспринимать это, как сумасшествие. Это всего лишь одна из задач на ночь, которые, ради тренировки, исполняют кодеры.

[Try Jupyter with Julia](#). Вот программа, язык Julia:

```
In [7]: function BrainF(
    _Code::Base.String=
    """+++++
    +++++
    +++++
    +++++
    +++++
    +++++
    """, _IsNeedListing::Core.Bool=false
    ,_IsNeedFinListing::Core.Bool=false
    )
    ###Исправляем ошибку индексирования строки с разным типом Code.Char###
    ###Ошибка исправляется как ни странно преобразованием в массив.
    ###Потрачено 4 часа.
    _Code=_Code|>a->(_arr=Core.Char[];foreach((b::Core.Char)->push!(_arr,b),a);_arr;)
    #####
    _StrIn="";_StrOut="";
    ###Def Code###
    #_Code::Base.String;
    _CodeId=1; _CodeIdMax=_Code|>length;
    ###Def Mem###
```

```

_MemId=1; _Mem=Core.UInt32[]; push!(_Mem,0);
###LoopStack###
_LoopStack=Core.Any[]
###Listing###
function BrainFackListing()
    "###BrainFListing###TargetOperatorWasExecuted###"|>println;
    "###Code###"|>println;
    for i=1:_CodeIdMax;if(i==_CodeId);print("{*_Code[i]*}");else;print(_Code[i]);end
    "###Memory###"|>println;
    _MemIdMax=_Mem|>length;
    for i=1:_MemIdMax; _p="";if(i==_MemId);_p="else;_p="    "end;
        println("$(_p)_Mem[$(i)]==$( _Mem[i])<=$(convert(Core.Char, _Mem[i]))>");
    end;
    if(length(_StrOut)!=0);println("###Out###\n"*_StrOut);end;
    if(length(_StrIn)!=0);println("###In###\n"*_StrIn);end;
    if(length(_LoopStack)!=0);println("###LoopStack###");for i in _LoopStack; println
end;
###Work Part###
while (_CodeId <= _CodeIdMax)
    _char =convert(Core.Char, _Code[_CodeId]);
    if(
        (_char=='+'||(_char=='-')||(_char=='>')||(_char=='<')
        ||(_char=='.')||( _char==',')||( _char=='[')||( _char==']')
    );
        if(_char=='+');_Mem[_MemId]=_Mem[_MemId]+1;end;
        if(_char=='-');_Mem[_MemId]=_Mem[_MemId]-1;end;
        if(_char=='>');_MemId=_MemId+1;while(_MemId>length(_Mem));push!(_Mem,0);end;e
        if(_char=='<');_MemId=_MemId-1;if(_MemId<1);_MemId=length(_Mem) end;end;
        if(_char=='.');
            _charPrint=convert(Core.Char, _Mem[_MemId])
            print(_charPrint);
            _StrOut=_StrOut*_charPrint;
        end;
        if(_char==',');
            while(length(_StrIn)==0);_StrIn=readline();end;
            _Mem[_MemId]=convert(Core.UInt32, _StrIn[1])
            _StrIn=_StrIn[2:length(_StrIn)]
        end;
        if(_char=='[');push!(_LoopStack,(_CodeId=_CodeId,_MemId=_MemId))end;
        if(_char==']');
            var=pop!(_LoopStack)
            if(_Mem[var._MemId]!=0);push!(_LoopStack,var);_CodeId=var._CodeId;end;
        end;
        ###Listing###
        if(!false);if(_IsNeedListing);BrainFackListing();end;end;
    end;
    _CodeId=_CodeId+1;
end;
if(!false);if(_IsNeedFinListing);BrainFackListing();end;end;
end;
BrainF("">+++++++[-<+++++]<+++++++.....+++++
+++-----.....+++++-----
[-]+++++++.....")

```

HelloWorld!!!

Как видите, все символы, кроме тех 8 команд игнорируются. Давайте разберемся с синтаксисом. У BF, как правило, есть режим пошаговой отладки и вывод конечного снимка памяти. В нашем случае это 2 флага, соответственно.

Для упрощения интерпретации снимка памяти выводятся все её ячейки в виде индекса,

числового содержимого и соответствующего этому коду символа.

В случае нехватки ячеек памяти осуществляется динамическое их выделение. Ячейки не высвобождаются!!! Это заложено в стандартах языка.

Текущая ячейка помечается либо звёздочкой, либо обратным отступом.

In [9]: `BrainF(""" """, false, true)`

```
###BrainFListing###TargetOperatorWasExecuted###
###Code###
```

```
###Memory###
_Mem[1]==0==<>
```

Вот полный листинг программы занесения двух единиц в ячейку с последующим обнулением при помощи цикла.

In [11]: `BrainF("""++[-]""", !false, true)`

```
###BrainFListing###TargetOperatorWasExecuted###
###Code###
```

```
{+}+[-]
```

```
###Memory###
```

```
_Mem[1]==1==<0>
```

```
###BrainFListing###TargetOperatorWasExecuted###
###Code###
```

```
+{+}[-]
```

```
###Memory###
```

```
_Mem[1]==2==<0>
```

```
###BrainFListing###TargetOperatorWasExecuted###
###Code###
```

```
++{[]-]
```

```
###Memory###
```

```
_Mem[1]==2==<0>
```

```
###LoopStack###
```

```
(_CodeId = 3, _MemId = 1)
```

```
###BrainFListing###TargetOperatorWasExecuted###
###Code###
```

```
++[{ -}]
```

```
###Memory###
```

```
_Mem[1]==1==<0>
```

```
###LoopStack###
```

```
(_CodeId = 3, _MemId = 1)
```

```
###BrainFListing###TargetOperatorWasExecuted###
###Code###
```

```
++{[]-]
```

```
###Memory###
```

```
_Mem[1]==1==<0>
```

```
###LoopStack###
```

```
(_CodeId = 3, _MemId = 1)
```

```
###BrainFListing###TargetOperatorWasExecuted###
###Code###
```

```
++[{ -}]
```

```
###Memory###
```

```
_Mem[1]==0==<>
```

```
###LoopStack###
```

```
(_CodeId = 3, _MemId = 1)
```

```
###BrainFListing###TargetOperatorWasExecuted###
###Code###
```

```
++[-{ }]
```

```
###Memory###
```

```
_Mem[1]==0==<>
```

```
###BrainFListing###TargetOperatorWasExecuted###
###Code###
```

```

++[-]
###Memory###
_Mem[1]==0==<>

```

Пример задачи, копирование числа через цикл.

```

In [18]: BrainF("""+++[->+>+<<]""", false, true)

###BrainFListing###TargetOperatorWasExecuted###
###Code###
+++[->+>+<<]
###Memory###
_Mem[1]==0==<>
_Mem[2]==3==<2>
_Mem[3]==3==<2>

```

Обратите внимание, что текущий экземпляр числа при копировании уничтожается. Поэтому при копировании нужно создавать минимум две копии, одну из которых позже возвращать на исходное место.

```

In [19]: BrainF("""+++[->+>+<<]>>[-<<+>>]""", false, true)

###BrainFListing###TargetOperatorWasExecuted###
###Code###
+++[->+>+<<]>>[-<<+>>]
###Memory###
_Mem[1]==3==<2>
_Mem[2]==3==<2>
_Mem[3]==0==<>

```

Обратите внимание, что данный цикл - это цикл с предусловием. Он же используется вместо ветвления. Блок else у ветвления не предусмотрен.

Следует отметить, что есть отдельная группа часто встречающихся задач, которая требует умения перемножать числа.

Пример, получение в ячейке числа 15. Какой из вариантов короче?

```

In [12]: BrainF(""">+++[-<+++++>]<""", false, true)

###BrainFListing###TargetOperatorWasExecuted###
###Code###
>+++[-<+++++>]<
###Memory###
_Mem[1]==10==<
>
_Mem[2]==0==<>

```

```

In [13]: BrainF("""+++++ +++++ +++++""", false, true)

###BrainFListing###TargetOperatorWasExecuted###
###Code###
+++++ +++++ +++++
###Memory###
_Mem[1]==15==<2>

```

Таким образом, Вы можете довольно компактным программным кодом вычислить  $2 * 3 * 5 * 7$ , но для этого потребуется соответствующее число ячеек памяти.

```

In [32]: BrainF("""+[->+++[->+++++[->++++++<]<]<]""", false, true)

###BrainFListing###TargetOperatorWasExecuted###
###Code###
+[->+++[->+++++[->++++++<]<]<]

```

```

###Memory###
_Mem[1]==0==<>
_Mem[2]==0==<>
_Mem[3]==0==<>
_Mem[4]==210==<0>

```

In [34]:

```

BrainF("""++[->+++[->+++++[->++++++<]<]<]
>>>[-<<<+>>>]<<<
""",false,true)

```

```

###BrainFListing###TargetOperatorWasExecuted###
###Code###
++[->+++[->+++++[->++++++<]<]<]
>>>[-<<<+>>>]<<<

```

```

###Memory###
_Mem[1]==210==<0>
_Mem[2]==0==<>
_Mem[3]==0==<>
_Mem[4]==0==<>

```

Ещё одна задача. Требуется найти в тексте первую букву русского алфавита.

Есть несколько подходов к её решению. Подход первый: метод простого перебора.

Формируем окно для просмотра из "+.". Предварительно при помощи крупных множителей прибавляем к точке старта.

Процесс сильно ускоряется, если заранее посмотреть искомое значение по таблице, например, ASCII кодов.

In [70]:

```

BrainF("""
Предварительно с помощью крупных множителей компактно приближаемся к искомому числу
+++++++[->+++++++[->+++++++<]<]
>>[-<<+>>]<<
Осуществляем приближение с небольшим шагом
>++++[-<+++++++>]<
Даем мелкую доводку без циклов

Выводим окно для просмотра результата
.+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
.+.+.+.+.+.+.
""",false,true)

```

```

АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдезийклмнопрстуфхцчшщъыьэяё###BrainFListing###Target
OperatorWasExecuted###

```

```

###Code###

```

```

Предварительно с помощью крупных множителей компактно приближаемся к искомому числу
+++++++[->+++++++[->+++++++<]<]
>>[-<<+>>]<<
Осуществляем приближение с небольшим шагом
>++++[-<+++++++>]<
Даем мелкую доводку без циклов

```

```

Выводим окно для просмотра результата
.+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
+.+.+.+.+.+.+.+.+.+.+.+.+.+.+.
.+.+.+.+.+.+.

```

```

###Memory###

```

В подобных случаях нередко прибегают к макросам, которые выполняют часть работы за нас. Напишем макрос для получения русского символа А, код 1040 = 10813.

```
####BrainFListing###TargetOperatorWasExecuted###  
####Code###  
BF_Get_RussianA(10+++++++[->8+++++[->13++++++<>]<])ВозвратВНачало>>[-<<+>>]<<)  
####Memory###  
_Mem[1]==1040==<A>  
    _Mem[2]==0==<>  
        _Mem[3]==0==<>
```

Можно написать разложение каждой разницы относительно предыдущего символа на простые множители. И сделать вывод на экран. Только займет это неразумно много времени...

```
Привет мир!!!###BrainFListing###TargetOperatorWasExecuted###  
###Code###
```

```
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++  
+++++. [-]Π  
+++++  
+++++  
+++++  
+++++  
+++++
```

+++++ .[-]p  
+++++ .[-]и  
+++++ .[-]в  
+++++ .[-]е  
+++++ .[-]т  
+++++ .[-]



```
###Memory###
_Mem[1]==0==<>
###Out###
Привет мир!!!
```

```
In [138]: function Get_TXT(_str)
    _last=0;_next=0;
    _codeBF="Get_TXT(\n";
    foreach(
        (x::Core.Char)->(
            rez="";
            _next=convert(Core.Int64, x);
            _Diff=_next-_last;
            while (_Diff>0);rez=rez*"+";_Diff=_Diff-1;end;
            while (_Diff<0);rez=rez*"-";_Diff=_Diff+1;end;
            _codeBF=_codeBF*rez*".$(x)";
            _last=_next;
        )
    ,_str
    );
    return _codeBF*"[ - ]\n)\n";
end;
BrainF("""$(Get_TXT("Привет мир!!!"))""",false,true)
```

```
###Memory###
_Mem[1]==0==<>
###Out###
Привет мир!!!
```

Теперь можно приступить к написанию программы вывода "ФИО Номер группы" в консоль. Рекомендуется сначала дать это задание и посмотреть - насколько легко учащиеся с ним справятся.

Затем итеративно помогать им, предоставляя все новый и новый материал для раздумий.

```
BrainF("""$(Get_TXT("Калашников Сергей ЗИСИП_519"))""",false,true)
```

