

Зори С.А. "АК", 2019

# ВВЕДЕНИЕ В АССЕМБЛЕР

# 1. Языки высокого и низкого уровней

**Машинный код** - единственный язык, понятный процессору (это **команды!**).

Поскольку ЦП не может непосредственно исполнять операторы ЯВР, программы на этих языках должны предварительно переводиться в машинный код (*транслироваться* — компилироваться или интерпретироваться).

Программа  
на языке  
высокого  
уровня

```
CLS 0
LOCATE 5, 1
PRINT a$; n
k=n MOD 4
IF k<>1 THEN
IF k=3 THEN
G=2
ELSE
```

Транслятор



Машинные  
коды

```
FD 8C DB 53
81 C3 39 00
03 DA 8C CD
8B C2 80 E4
0F B1 04 8B
F2 D3 E6 D3
EA FE C6 8B
CE D1 E9 4E
```

Ассемблер строго говоря не является языком ни высокого, ни низкого уровня – он занимает некоторое промежуточное положение максимально приближенное к низкому (машинным кодам).

Основное отличие между языком ассемблера и ЯВР состоит в том, что операторы ЯВР переводятся в целые наборы машинных команд-кодов, а любой оператор ассемблера непосредственно преобразуются в соответствующую ему единственную машинную команду - код.

# Ассемблер – это язык символического кодирования машинных команд

- Язык низкого уровня
- Аппаратно – зависимый
- 1 инструкция программы компилируется в 1 машинную команду
- Несколько разных машинных инструкций могут иметь один псевдокод

Программы на языке ассемблера также переводятся в бинарный машинный код с помощью программы-транслятора, называемой тоже *ассемблером*.

Существуют свои достоинства у каждого языка, однако только на языке ассемблера можно писать программы, напрямую используя множество команд процессора и имея низкоуровневый доступ ко всем его ресурсам.

# Преимущества языка ассемблера:

1. *Низкоуровневый доступ* к компьютеру,
2. Достижение *максимальной скорости* за счет  
возможности полностью управлять  
процессором.
3. *Минимальный размер* исполняемого кода

# Недостатки языка ассемблера:

1. Повышенный риск совершения ошибок (зависание),
2. Непереносимость на системы с несовместимыми семействами МП
3. Отсутствие библиотек подпрограмм\* для выполнения стандартных операций вывода/вывода, чтения и пр.



# Язык assembler

- Программы практически не пишутся сегодня целиком на Assembler:
  - исключение: некоторые микроконтроллеры
- Практически применяется:
  - в системном ПО
  - низкоуровневому доступу к аппаратуре
  - для ускорения “бутылочных горлышек”
  - для отладки (и взлома) чужого кода

В рамках данного курса:  
Язык Assembler – не цель, а *средство*  
изучения архитектуры МП и  
механизмов работы с ресурсами  
компьютера!

# Стандарты языка

Assembler в содеpжит в себе два синтаксических стандарта языка ассемблера – **MASM** (Microsoft Assembler) и **Ideal**.

Программируем

- *для простоты* в виде ассемблерных вставок в C++ (VisualStudio или др. среды разработки)
- либо в MASM Win32 любых сред разработки

# Компиляторы языка

	Windows	DOS	Linux	BSD	QNX	MacOS, работающий на процессоре Intel/AMD
FASM	x	x	x	x		
GAS	x	x	x	x	x	x
GoAsm	x					
HLA	x		x			
MASM	x	x				
NASM	x	x	x	x	x	x
RosAsm	x					
TASM	x	x				

# Основные части программы на ассемблере

Программу можно разделить на пять основных частей: заголовок, макроопределения, данные, тело и заключение.

Заголовок содержит установочную информацию.

В макроопределениях определяются переменные, которым присваиваете значения различных выражений и констант.

В данных определяются переменные, которые будут храниться в памяти.

Тело содержит собственно код программы.

Заключение отмечает конец исходного текста.

# Основные части программы IDEAL (real mode)

%TITLE "..."

IDEAL

MODEL small

**DATASEG**

...

**CODESEG**

...

**END**

# Основные части программы MASM (real mode)

NAME “...”

386

model small

**DATA SEGMENT**

...

DATA ENDS

**CODE SEGMENT**

ASSUME CS: CODE, SS: STACK, DS: DATA

...

CODE ENDS

**END**

# Основные части программы MASM Win32

**.386** ; 32-разрядные приложения

**.model flat** ; в программах Win32 используется линейная модель памяти (flat)

**.bss**

; в этом сегменте описываются неиниц. данные

**.const**

; в этом сегменте описываются константы-макроопределения

**.data**

; в этом сегменте описываются данные

**.code**

; в этом сегменте описывается код программы

**end** ; конец модуля



# Основные части ASM-32/64 программы

## во вставках в C++ *Builder, Visual C++*

Системы программирования C++ Builder, Visual C++ позволяют вставлять в текст программы участки кода, написанные на ассемблере.

Ассемблерный текст заключается в блок:

```
_asm {...}
```

В ассемблерных вставках можно использовать ранее объявленные на C++ в основном модуле программы переменные, функции и другие идентификаторы.

***Фактически описываем только Тело ASM-программы (сегмент кода)***

## 1. Заголовок

Программа на языке ассемблера начинается с заголовка.

В нем содержатся *команды и директивы*, не приводящие к созданию машинного кода при трансляции.

Они указывают ассемблеру, как выполнить определенные действия, генерируя исполняемый файл.

## Заголовок IDEAL (real mode)

**%TITLE "Тестовая программа"**

**IDEAL**

**P386**

**MODEL small**

## Заголовок MASM (real mode)

**NAME "Тестовая программа"**

**.386**

**MODEL small**

Необязательная строка **%TITLE (NAME)**

описывают назначение программы ,

Директива **IDEAL** переводит Turbo Assembler в режим Ideal.

(Если программа написана на MASM, директиву Ideal нужно опустить).

Далее следует директива **MODEL**, выбирающая одну из нескольких моделей памяти - обычно - *small* (малая модель памяти).

## Модели памяти

Название	Код	Данные	Определение
tiny	near	near	Код, данные и стек содержатся в одно сегменте 64 Кбайт. Используется только для COM-программ
small,	near	near	Код и данные содержатся в различных сегментах, размером до 64 Кбайт. Используется для небольших и средних EXE-программ. Наилучшим образом подходит для большинства чисто ассемблерных программ
medium	far	near	Неограниченный размер кода. Под данные отводится один сегмент 64 Кбайт. Используется для написания больших программ с небольшим объемом данных
compact	near	far	Размер кода ограничен одним сегментом 64 Кбайт. Размер данных неограничен. Используется при написании малых и средних по размеру программ с большим количеством переменных
large	far	far	Размер кода и данных неограничен. Используется в больших программах. Размер переменной не может превышать 64 Кбайт
huge	far	far	Размер кода и данных неограничен. Аналогична большой модели памяти. (Введена для совместимости с языками высокого уровня)

## Заголовок в MASM Win32

**.386**

**.model flat, stdcall**

; в программах Win32 используется только линейная модель

**option casemap: none**

**include** \masm32\include\windows.inc

**include** \masm32\include\kernel32.inc

**includelib** \masm32\lib\kernel32.lib

**include** \masm32\include\user32.inc

**includelib** \masm32\lib\user32.lib

; подключаем необходимые модули с описаниями внешних  
макросов и функций, и библиотеки импорта MASM-32

## Заголовок в ASM-32/64 вставке C/C++ VS

**Не нужен, все инициализации препроцессора  
делает основная программа**



## 2. Макроопределения

После заголовка программы следуют различные *описания констант и переменных*. В языке ассемблера константы часто называют макроопределениями, использующими директиву **EQU**, которая связывает значение с идентификатором. Исключительно для числовых значений, помимо директивы **EQU**, можно применять знак равенства (=).

Макроопределения могут располагаться в любом месте программы, но «классически» — сразу после заголовка программы.

<b>Count</b>	<b>EQU</b>	<b>10</b>
<b>Element</b>	<b>EQU</b>	<b>5</b>
<b>Size</b>	<b>=</b>	<b>Count * Element</b>
<b>MyBoat</b>	<b>EQU</b>	<b>"Gypsy Venus"</b>
<b>Size</b>	<b>=</b>	<b>0</b>

- После описания имени константы с помощью директивы **EQU** вы не можете изменять его значение.
- Имена-идентификаторы, описанные с помощью “=“ - вы можете свободно изменять их значения (в любом месте программы).

- **EQU** может описывать все типы равенств, включая числа, выражения и символьные строки.  
“**=**” может описывать только равенства, состоящие из чисел либо числовых выражений -  $\text{Count} * \text{Size}$ ,  $\text{Address} + 2$  и т.п..
- Идентификаторы макроопределений не являются переменными - ни они, ни их значения не содержатся в сегменте данных программы.

- Команды ассемблера не могут изменить значения идентификатора в макроопределениях, независимо от того, были они описаны с помощью директивы EQU или знака =.
- Выражения, описанные с помощью **EQU**, вычисляются, когда соответствующее имя-идентификатор используется программой. Выражения, описанные через “=”, вычисляются непосредственно в месте определения (ассемблер сохраняет текст EQU-выражения, а для выражения, описанного с помощью знака равенства, - только его значение).

Например,

**LinesPerPage = 66**

**NumPages = 100**

**Total Lines = LinesPerPage \* NumPages**

Если где-либо в программе вы измените значение

NumPages (например, NumPages = 200),

значение TotalLines останется прежним.

Ситуация изменится, если вы опишете TotalLines

с помощью EQU:

**TotalLines EQU LinePerPage \* NumPages**

В этом случае Turbo Assembler сохранит не вычисляемое значение, а действительный текст выражения, следующего за директивой EQU (в нашем случае - *LinePerPage \* NumPages*). Позже в программе, когда вы будете использовать TotalLines, ассемблер вставит этот текст так, как будто вы набирали его в этом месте исходного текста, затем выражение будет вычислено и заменено конечным значением.

\*Сейчас макроопределения используется редко!

### 3. Сегмент данных

Сегмент данных программы предшествует командам программы.

Сегмент данных программы должен начинаться с директивы **DATASEG (DATA SEGMENT)** .

Она дает указание ассемблеру разместить в памяти *переменные*, указанные в сегменте данных программы.



Сегмент данных может содержать два типа переменных: **инициализированные** и **неинициализированные**.

**Инициализированные** переменные имеют определенные значения, которые вы определили в тексте программы, и содержатся в файле программного кода. Эти переменные автоматически загружаются в память и доступны для чтения при исполнении программы.

Неинициализированные переменные аналогичны инициализированным, за исключением того, что они не занимают пространства в исполняемом файле и, следовательно, имеют неопределенные значения при исполнении программы.

\* Обычно теперь их не разделяют.

# Сегмент данных IDEAL (real mode)

## DATASEG

numRows	DD	259
numColumns	DB	80
videoBase	DW	0B00h
zzzz	DD	?

Директива DATASEG информирует ассемблер о необходимости выделения пространства в памяти под сегмент данных программы. Затем определены переменные.

## Сегмент данных MASM (real mode)

### **DATA SEGMENT**

numRows	DD	259
numCols	DB	80
videoBase	DW	0B00h
zzzz	DD	?

### **DATA ENDS**

# Сегмент данных MASM Win32

**.data**

numRows	DD	259
numColumns	DB	80
videoBase	DW	0B00h
zzzz	DD	?

# Сегмент данных в ASM-32/64 вставках в C++ *Builder, Visual C++*

Не нужен.

Переменные описываем средствами определения данных C++ до вставки (соответствующими ASM-данным типами данных C++)

В отличие от языков высокого уровня, в которых расположение переменных в памяти обычно не является важным, в языке ассемблера переменные располагаются в памяти в порядке их описания последовательно в соответствии с объявленной длиной.

## Директивы выделения памяти:

Директива	Название	число выделяемых байтов	Характерное использование
db	Определить байт	1	Байты, строки
dw	Определить слово	2	Целые числа
dd	Определить двойное слово	4	Длинные целые
dp	Определить указатель	4	32-битовый указатель
df	Определить дальний указатель	6	48-битовый указатель
dq	Определить учетверенное слово	8	Вещественные числа
dt	Определить десять байтов	10	BSCD-числа



# Типы данных (основные)

обозначение	расшифровка	размер, байт
db	data byte	1
dw	data word	2
dd	data double word	4
dq	data quad word	8
dt	exTended precision	10

Знаковые и беззнаковые типы не различаются!

# Типы данных (основные) и их соответствие типам данных C++

Тип данных ASM	Тип данных C++	размер, байт
db	char	1
dw	short int, unsigned short int	2
dd	int, unsigned int, long int, unsigned long int, float	4
dq	long float, double	8

Имена переменных в ассемблере являются *метками-указателями*, которые указывают на выделяемые в памяти области под данные - на пространство, резервируемое под значения переменных.

Программа может обращаться к этому пространству, используя метку *как указатель* на соответствующее значение в памяти.

В ассемблерных программах метки преобразуются в адреса памяти (смещения в сегменте данных), по которым содержатся значения переменных, что позволяет обращаться к памяти по именам, а не по численным адресам.

Строка состоит из отдельных ASCII-символов, каждый из которых занимает один байт, следовательно, директива DB является в ассемблере простым инструментом для определения строк символов, которые представляют собой в памяти записанный последовательно набор ASCII-символов.

## **DATASEG**

**aTOm      DB    "ABCDEFGHJKLM"**

**nTOz      DB    "NOPQRSTUVWXYZ"**

Директивы выделения памяти можно использовать для определения массивов, разделяя элементы запятыми:

<b>perfectTen</b>	<b>DW</b>	<b>1, 2, 3, 4, 5, 6, 7, 8, 9, 10</b>
<b>M</b>	<b>DB</b>	<b>9, A, 0, 245, 7</b>
<b>theDate</b>	<b>DD</b>	<b>12, 11, 2019</b>

# Неинициализированная переменная в ASM

обозначается «?».

## Temp dw ?

Имеется возможность *дубликации* (размножения) значений для резервирования места или заполнения массива:

**Temp dw 16 DUP(?)**

**Ten\_Null db 10 DUP(0)**

**A db 10 DUP(1,2)**

**T dw 10 DUP(2 DUP(5))**

**B db 10 DUP(«a»)**

# Инициализация данных (примеры)

данные	пример
десятичное целое число	db 1 dd -999999
шестнадцатеричное целое число	db 0xff dw 0x1234 ; 0x34 0x12 или dd 56789765h
СИМВОЛ	db 'a' dw 'a' ; реальное значение 0x61 0x00

# Инициализация данных (примеры)

данные	пример
массив	dw 1, 2, 3 dw 100 dup (0)
строка	db 'abc' db 'hello', 13, 10, '\$'
вещественное число	dd 1.23e20, -1.23 dq 1.23e20, -1.23 dt 1.23e20, -1.23
неопределенное значение	dw ? dw 100 dup (?)



Данные в ASM- вставках в C++ *Builder, Visual C++*

Ориентируемся на это!

Переменные описываем типами данных C++  
соответствующими ASM- данным, при  
необходимости инициализируем их

## 4. Тело программы

После сегмента данных располагается тело программы, известное под названием *кодového сегмента* - раздела памяти, который содержит исполняемый код ассемблерной программы.

Внутри этой области можно выделить четыре колонки текста: *метки, мнемонику, операнды и комментарии*.

Количество пробелов между колонками в тексте программы произвольное.

**Метки** помечают места в программе, на которые могут ссылаться другие команды и директивы. Для строк без меток эта колонка не заполняется. В кодовом сегменте метка всегда заканчивается двоеточием (:).

Во второй колонке содержатся **мнемоники**. Под каждой мнемонической формулировкой в этой колонке скрывается одна **машинная команда**: *mov* - для *Move* (пересылка данных), *jmp* - для *Jump* (безусловный переход), и пр.

Третья колонка содержит **операнды**, которые обрабатываются командами. Некоторые команды не требуют операндов, в этом случае третья колонка остается пустой. Многие команды требуют двух операндов, другие - только одного. Ни одна из команд процессора не требует больше двух операндов.

Первый оператор обычно называется назначением (*dst*), второй — источником (*src*).

# Например

команда	dst	src
<b>mov</b> dst, src	регистр память	регистр память число*

\* Далее "число" =  
"непосредственный операнд"

Четвертая и последняя колонка являются необязательными, и если включаются в программу, то должны начинаться с точки с запятой (;). Turbo Assembler игнорирует все символы от точки с запятой до конца строки, предоставляя вам место для размещения короткого **комментария**, описывающего выполняемые в данной строке действия.

# Сегмент кода IDEAL (real mode)

## CODESEG

```
Start:      mov ax,@data
            mov ds,ax
            mov ax,[x]
            mov bx,[y]
            cmp bx,0      ;сравним bx и 0 ( y и 0)
            jge notabs    ;если bx >= 0
            neg bx        ;иначе меняем знак
notabs:     imul bx

exit:       mov [res], ax
END Start
```

## Сегмент кода MASM (real mode)

**CODESG SEGMENT**

ASSUME CS:CODESG, DS:DATASG

Start:           mov ax, DATA  
                  mov ds,ax  
                  mov ax,x  
                  mov bx,y  
                  cmp bx,0           ;сравним bx и 0 ( y и 0)  
                  jge notabs        ;если bx  $\geq$  0  
                  neg bx            ;иначе меняем знак

notabs:          imul bx

exit:            mov res, ax

**CODESG ENDS**

END Start



## Сегмент кода MASM Win32

**.code**

**\_WinMainCRTStartup:**

mov ax,x

mov bx,y

cmp bx,0 ;сравним bx и 0 ( y и 0)

jge notabs ;если bx >= 0

neg bx ;иначе меняем знак

notabs: imul bx

exit: mov res, ax

invoke ExitProcess, NULL ;выходим из проги

**end \_WinMainCRTStartup**

## Сегмент кода во вставках в C++ *Builder*, *Visual C++*

```
_asm {  
    mov ax,x  
    mov bx,y  
    cmp bx,0      ;сравним bx и 0 ( y и 0)  
    jge notabs    ;если bx >= 0  
    neg bx        ;иначе меняем знак  
notabs:  
    imul bx  
exit:  
    mov res, ax  
}
```

## 5. Заключение

Последней частью программы на языке ассемблера является заключение - одиночная строка, информирующая Turbo Assembler о достижении конца программы. В заключении используется единственная директива END.

**END**      **Start**      ; Конец программы / метка входа

Справа от директивы END вы должны определить метку, с которой вы хотите начать *выполнение программы.*

Обычно эта метка совпадает с меткой, указывающей на первую команду, следующую за директивой CODESEG.

Во вставках – не нужен!

# Простейшая программа как ASM- вставка в *Visual C++*

```
#include "pch.h"
#include <iostream>
using namespace std;
int main()
{
    setlocale(LC_ALL, "Rus");
    int a, b, t1, t2, t3, t4;

    cout << "Введите число a\n";
    cin >> a;
    cout << "Введите число b\n";
    cin >> b;
```

# Простейшая программа как ASM- вставка в *Visual C++*

```
_asm  
{  
  mov eax, [a]      ; ссылаемся на переменную «a» как [a]  
  mov ebx, [b]      ; ссылаемся на переменную «b» как [b]  
  mov ecx, a        ; ссылаемся на переменную «a» как a  
  mov edx, b        ; ссылаемся на переменную «b» как b  
  mov [t1], eax     ; ссылаемся на переменную «t1» как [t1]  
  mov t2, ecx       ; ссылаемся на переменную «t2» как t2  
  ; имя_переменной в ASM – метка-указатель!  
  ; [имя_переменной] == имя_переменной  
  ;   так «правильнее»           так «проще»  
  
  mov [t3], ebx  
  mov t4, edx  
}
```

# Простейшая программа как ASM- вставка в *Visual C++*

```
system("cls");
```

```
cout << "[a] -> a = " << t1 << endl << endl;
```

```
cout << "a ->    a = " << t2 << endl << endl;
```

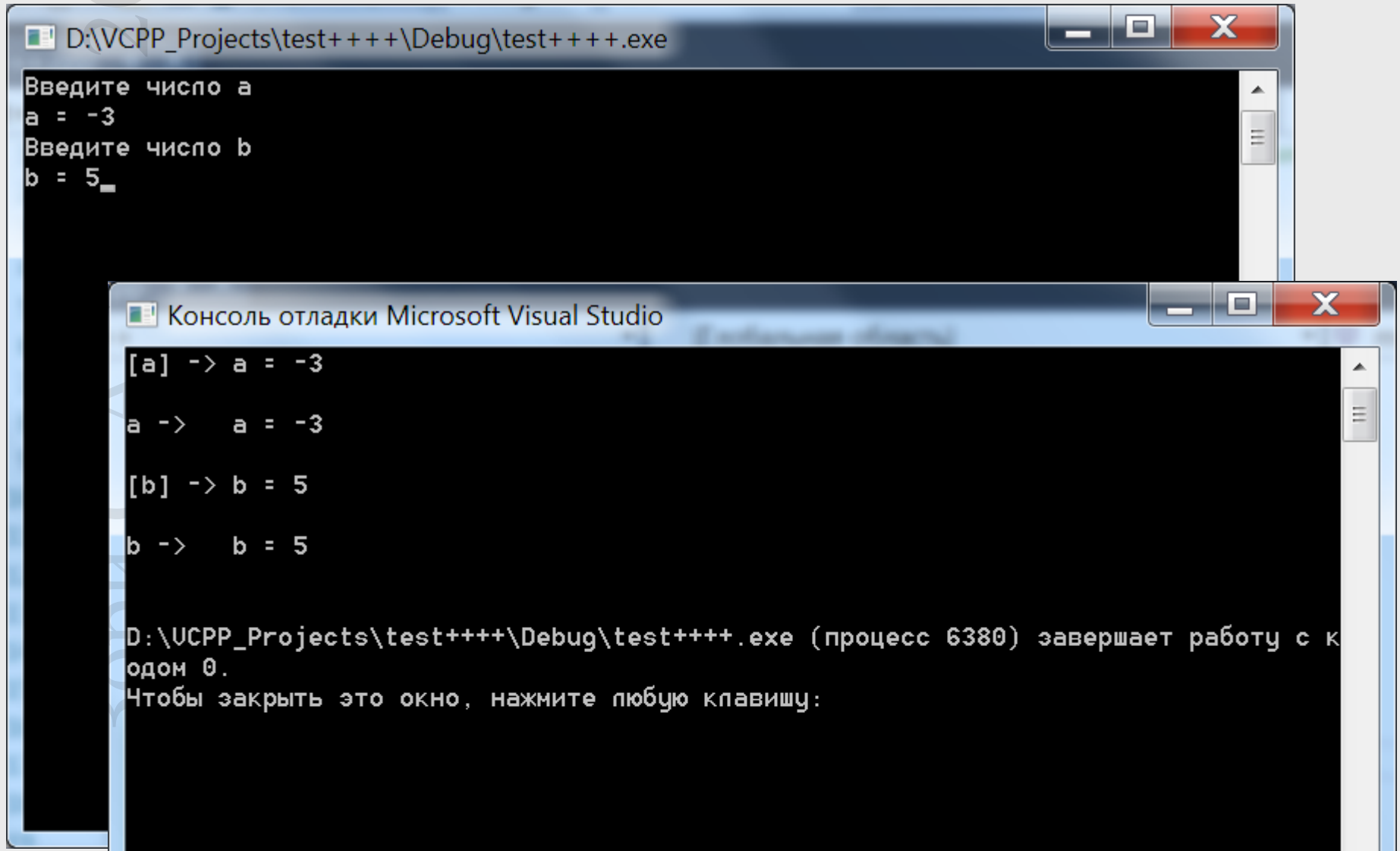
```
cout << "[b] -> b = " << t3 << endl << endl;
```

```
cout << "b ->    b = " << t4 << endl << endl;
```

```
return 0;
```

```
}
```

# Простейшая программа как ASM- вставка в *Visual C++*



The image shows two overlapping windows from a Windows operating system. The top window is titled "D:\VCPP\_Projects\test++++\Debug\test++++.exe" and contains a simple text-based interface in Russian. It prompts the user to "Введите число а" (Enter number a), shows "а = -3", prompts for "Введите число b" (Enter number b), and shows "b = 5\_". The bottom window is titled "Консоль отладки Microsoft Visual Studio" (Microsoft Visual Studio Debug Console). It displays the memory addresses and values for variables 'a' and 'b': "[a] -> а = -3", "а -> а = -3", "[b] -> b = 5", and "b -> b = 5". At the bottom of the console, it states: "D:\VCPP\_Projects\test++++\Debug\test++++.exe (процесс 6380) завершает работу с кодом 0." (D:\VCPP\_Projects\test++++\Debug\test++++.exe (process 6380) finishes work with code 0.) and "Чтобы закрыть это окно, нажмите любую клавишу:" (To close this window, press any key:).

```
D:\VCPP_Projects\test++++\Debug\test++++.exe
Введите число а
а = -3
Введите число b
b = 5_

Консоль отладки Microsoft Visual Studio
[a] -> а = -3
а -> а = -3
[b] -> b = 5
b -> b = 5

D:\VCPP_Projects\test++++\Debug\test++++.exe (процесс 6380) завершает работу с кодом 0.
Чтобы закрыть это окно, нажмите любую клавишу:
```