This means:
1. Allocate an array of 17 bytes in the .rodata section of memory.
2. Fill those bytes with the 8-bit ASCII character codes from the given string, with the null(0) byte appended.
3. Make the symbol .LC0 symbolize the address of the first byte, which contains the code for W.
This means

1. Assemble the following instructions into the "text" section of memory.

2. Make the symbol main be global. That means roughly other compilation units will "know" about it via the linker. (By default, \texttt{.LC\_\_} was local.)

3. Record the type of the object whose address is symbolized by main

4. \texttt{main:} is a label.

Symbol main will symbolize the address of the first instruction given below.
main:
  leal 4(%esp), %ecx

Mnemonic for
load effective address long

ATT/GNU assembly language syntax:
1. Source operand(s) are on the LEFT ← 4(%esp)
2. Destination operand is last, on the RIGHT → %ecx
3. CPU registers are named by %...
4. A memory operand is denoted by ( ) (The "e")
5. %esp is the stack pointer register, means 32 bits

i386 Instruction set architecture:
One addressing mode is Register+displacement
The (effective) address of operand 4(%esp) is 4 + the value in the %esp register.
The CPU calculates 4 + that value.
main:

```
leal 4(%esp), %ecx
```

Effective address of operand 4(%esp) is 4 + value in %esp = 4 + %esp

Destination operand is %ecx.

That's an i386 general purpose 32bit register.

This `leal` calculates 4 + %esp and then copies that value into %ecx

`lea` type instructions are special, they look like memory accesses but do NOT access memory,
main:
    leal  4(%esp), %ecx
    andl $-16, %esp

`andl`: bitwise AND long 32 bits.
Source operand 1: Immediate (denoted by $)
$-16 is 0xFFFFFFF0
2nd Source & Destination operand is %esp
0xFFFFFFF0 & %esp → %esp makes the low order 4 bits of %esp become 0s, leaving the other 32 - 4 = 28 bits unchanged.
%esp is a memory address - it's ≥ 0.
Effect of `andl $-16, %esp`
is: Round %esp down to a multiple of 16 = 2^4
main:
  leal 4(%esp), %ecx //Save 4+old esp in ecx
  andl $-16, %esp    //Decr esp, align stack
  pushl -4(%ecx)   //Save ret. addr. in our aligned stack
  pushl %ebp
  movl %esp, %ebp
main:
leal 4(%esp), %ecx  //Save 4+old esp in ecx
andl $-16, %esp     //Decr esp, align stack
pushl -4(%ecx)     //Save ret. addr. in our aligned stack
pushl %ebp
movl %esp, %ebp

Higher Value Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>β-4 esp</td>
<td>old ebp</td>
</tr>
<tr>
<td>α+4 esp</td>
<td>%ebp</td>
</tr>
<tr>
<td>α+4 ecx</td>
<td></td>
</tr>
</tbody>
</table>

Higher Value Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>β-8 esp</td>
<td>old ebp</td>
</tr>
<tr>
<td>β-4 esp</td>
<td>old ebp</td>
</tr>
<tr>
<td>α+4 esp</td>
<td></td>
</tr>
</tbody>
</table>

pushl %ebp
pushes the current value of the Base Pointer (was set by the caller) onto our stack,
main:
leal 4(%esp), %ecx  //Save 4+old esp in ecx
andl $-16, %esp  //Decr esp, align stack
pushl -4(%ecx)  //Save ret. addr. in our aligned stack
pushl %ebp  //Save caller's ebp in our stack
movl %esp, %ebp

As usual, move instructions actually copy.

Higher Value Addresses
\[\alpha+4\]
\[\alpha\]
\[\beta\]
\[\beta-4\]
\[\beta-8\]

Higher Value Addresses
\[\alpha+4\]
\[\alpha\]
\[\beta\]
\[\beta-4\]
\[\beta-8\]
main:
  leal 4(%esp), %ecx //Save 4+old esp in ecx
  andl $-16, %esp    //Decr esp, align stack
  pushl -4(%ecx)    //Save ret. addr. in our aligned stack
  pushl %ebp        //Save caller's ebp in our stack
  movl %esp, %ebp   //Set ebp to the base of OUR frame.
  subl $20, %esp
main:
leal 4(%esp), %ecx //Save 4+old esp in ecx
andl $-16, %esp    //Decr esp, align stack
pushl -4(%ecx)    //Save ret. addr. in our aligned stack
pushl %ebp       //Save caller's ebp in our stack
movl %esp, %ebp //Set ebp to the base of OUR frame.
subl $20, %esp    //Allocate 20 bytes for our locals.

movl %ecx, -12(%ebp)  
movl -12(%ebp), %eax  
cmpl $2, (%eax)  
je .L7  

.L7: Higher Value Addresses

MOVE 1
MOVE 2
Jump to here if ==.

.L7: Higher Value Addresses

\[\alpha + 4\] 
\[\beta - 12\] 
\[\beta - 8\] 
\[\beta - 8\] 
\[\alpha + 4\] 
\[\beta - 12\] 
\[\beta - 8\] 
\[\beta - 8\]
main:
leal 4(%esp), %ecx //Save 4+old esp in ecx
andl $-16, %esp    //Decr esp, align stack
pushl -4(%ecx)    //Save ret. addr. in our aligned stack
pushl %ebp        //Save caller's ebp in our stack
movl %esp, %ebp   //Set ebp to the base of OUR frame.
subl $20, %esp    //Allocate 20 bytes for our locals.

movl %ecx, -12(%ebp) //Copy address of argc to a local.
movl -12(%ebp), %eax //Copy it also to %eax
cmpl $2, (%eax)    //Compare the value of argc with 2
je .L7              //Jump to address .L7 if argc==2
...

.L7: Higher Value Addresses

A reasonable question:
Why didn't the compiler just compile
    cmpl $2, (%ecx) //??
Possible answers:
1. Compilers with optimization turned off are very stupid.
2. The address of argc is needed to be in the local and in %ecx for other reasons, probably stupid ones.
write( 1, "W...", 17); is compiled to:

```
je       .L7
movl $17, 8(%esp)
movl $.LC0, 4(%esp)
movl $1, (%esp)
call write
movl $1, -8(%ebp)
jmp      .L9
```

The 3 movl's push the 3 parameters, 17, the addr. of the error message string, and 1 (file descr.). Then, the call instruction pushes the return address.