Object file sections:

Symbol table provides a mapping of symbols to addresses just like the one in your assembler. But it only contains things that:

1. Might be needed by another file (globals and functions you define)
2. Unresolved variables (globals and functions defined by other files, but you have referenced them)
3. Static variables (note: local static variables are marked as unlinkable)

Relocation table holds addresses of _instructions_ that need to be fixed up with updated addresses. It contains fields that tell you:

- What address the instruction that needs to be fixed is at
- What type of instruction you have (lw, sw, jal, etc)
- What symbol you are using there.

The types of instructions you might see relocated are:

- Calls to functions
- Uses of things in the data section (globals and statics)

This would be anything that use a symbol in the object file's symbol table (globals or static vars)
Note: Declarations statements in C need not be relocated, because they are only used by the compiler and don't translate to instructions in the object file.

Where data goes

- All local variables, except local static variables, go into stack.
- Dynamically allocated memory goes into heap.
- Global, static variables and string constants go into data segment.
- In ARM, the first 4 parameters are passed via registers (r0-r3). The rest go into stack.

Caller/callee assumptions

- For simplicity, in 370, we assume that inter-procedural analysis is not possible. The only exception is when we explicitly state in a question that a particular function (e.g., main()) has no callers (in which case you would not generate any save and restores for callee save registers).

- Assume uninitialized variables are not live across a function call

1) Do static local variables go in the symbol table?

Specifically, in the following code:

```
1: void foo() {
2:   static int static_local_var = 5;
3:  static_local_var++;
4: }
```

Does static_local_var go in the symbol table? Does the reference on line 3 go in the relocation table.

The answer to both is **YES**. static_local_var behaves like a global variable - it lasts across calls to foo(). Hence, it needs storage allocated in the data segment. Since the layout of the data segment is not known until link time (because the linker may be combining data segments from multiple files), there must be an entry in the symbol table and the reference on line 3 must go in the relocation table. Note that there will be two things special about this entry: (1) the symbol name will be static_local_var.some_random_stuff_added_by_the_compiler. The suffix (usually the line number for gcc) is needed because multiple functions may all declare static variables with the same name. Second, the symbol table entry will be marked "local" and not "global" because the variable is only visible in this file (and, in fact, only in the function foo()).

You can confirm this by compiling the code above and dumping the symbol table with objdump:

```
gcc -c -o test.o test.c
objdump -t test.o

test.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS* 0000000000000000 test.c
0000000000000000 l    d  .text 0000000000000000 .text
0000000000000000 l    d  .data 0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l     O .data 0000000000000004 static_local_var.1593
0000000000000000 l    d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame 0000000000000000 .eh_frame
0000000000000000 l    d  .comment 0000000000000000 .comment
0000000000000000 g     F .text 0000000000000006 foo
```

The fifth line of this file is the symbol table entry for "static_local_var.1593" indicating it is in the data segment.

2) Do string literals go in .text or .data?

They go into a segment called .rodata (read-only data) that behaves much like the data segment. In EECS 370, we don't make a distinction between these two segments for the sake of simplicity.

You can see that with the following:

```
void foo() {
  char * string_literal = "literal";
}
```

```
 objdump -s test.o

test.o:     file format elf64-x86-64

Contents of section .text:
 0000 554889e5 48c745f8 00000000 c9c3       UH..H.E.......
Contents of section .rodata:
 0000 6c697465 72616c00                     literal.
Contents of section .comment:
 0000 00474343 3a202847 4e552920 342e342e  .GCC: (GNU) 4.4.
 0010 37203230 31323033 31332028 52656420  7 20120313 (Red
 0020 48617420 342e342e 372d3429 00        Hat 4.4.7-4).
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 0e000000 00410e10 8602430d  .........A....C.
 0030 06490c07 08000000                    .I......
```

Note the "contents of section .rodata" in the middle, which contains the string "literal".