1. Introduction. This is a literate program which solves the problem set in lab one—implementing a stack.

We will start with the outline of the program.

```
⟨ Headers 5 ⟩
using namespace std;
⟨ Prototypes for functions 13 ⟩
⟨ Global data 2 ⟩
⟨ The main program 3 ⟩
⟨ Implementation of functions 14 ⟩
```

2. To start with we need to declare the *stack*, which will be an array of character arrays, one per word. To avoid extra parameters on the *push*, *pop* and *isEmpty* functions we will make the stack global. We also declare an integer value *top* which will point at the current top of the stack, initially it will be zero. To make the program a little more flexible we will use integer constants `STACK_SIZE` for the size of the *stack* array and `WORD_SIZE` for the maximum word length.

```
⟨ Global data 2 ⟩ ≡
  const int STACK_SIZE = 100;
  const int WORD_SIZE = 20;
  char stack[STACK_SIZE][WORD_SIZE];
  int top = 0;
```

This code is used in section 1.

3. Main. Ok, let's start writing *main*. The skeleton of the *main* program is as follows.

```
⟨ The main program 3 ⟩ ≡
  int main( )
  {
    ⟨ Variables of main 4 ⟩
    ⟨ Open and validate the input file 6 ⟩
    ⟨ Read the file into the stack 8 ⟩
    ⟨ Pop the words from the stack 10 ⟩
    ⟨ Finish up 11 ⟩
  }
```

This code is used in section 1.

4. The first thing we need to do is declare the variables we need to input words. Let's start with the character array *filename* and the input stream *fin*.

```
⟨ Variables of main 4 ⟩ ≡
  char filename[20];
  ifstream fin;
```

See also section 7.

This code is used in section 3.

5. Hang on—we need a couple of header files *iostream* for stream-based input and *fstream* for managing files.

```
⟨ Headers 5 ⟩ ≡
#include <iostream>
#include <fstream>
```

This code is used in section 1.

6. Right—now we can get the file sorted. We will prompt for the input file name using *cerr* so that we can redirect the output without getting the prompt in the output file and so that we can see the prompt even when we redirect standard output. We will then read in the file name and open an input stream. We should test for errors too, I guess.

```
⟨ Open and validate the input file 6 ⟩ ≡
  cerr ≪ "Please␣enter␣the␣name␣of␣the␣input␣file:␣";
  cin ≫ filename;
  fin.open(filename);
  if (¬fin) {
    cerr ≪ "Error␣opening␣file␣" ≪ filename ≪ ".␣Program␣will␣exit." ≪ endl;
    return 0;
  }
```

This code is used in section 3.

7. We are now ready to do the main input loop. we need a character array *word* to read the input file into.

```
⟨ Variables of main 4 ⟩ +≡
  char word[WORD_SIZE];
```

8. and now we are ready to do the code. We will code this as a loop which attempts to read in a *word* and, if successful, *push* it onto the *stack*.

⟨ Read the file into the stack 8 ⟩ ≡
```
  while (fin ≫ word) {
    push(word);
  }
```
This code is used in section 3.

9. I could define the function *push* now or save it for later. Let's do it later.

10. On with the main program. At this point we have stored everything on the *stack* now we simply reverse the process, calling *pop* until the *stack* is empty.

⟨ Pop the words from the stack 10 ⟩ ≡
```
  while (¬isEmpty()) cout ≪ pop() ≪ endl;
```
This code is used in section 3.

11. To finish up we should close the input stream.

⟨ Finish up 11 ⟩ ≡
```
  fin.close();
```
This code is used in section 3.

12. Functions. We will declare our functions, including the prototypes, here.

13. Let's start with *push*—first the prototype.

⟨ Prototypes for functions 13 ⟩ ≡
```
  void push(char word[ ]);
```
See also sections 15 and 17.

This code is used in section 1.

14. And the implementation. We need to ensure that we do not overflow the stack so we wil test that first.

⟨ Implementation of functions 14 ⟩ ≡
```
  void push(char word[ ])
  {
    if (top ≡ STACK_SIZE) return;
    int i = 0;
    do {
      stack[top][i] = word[i];
      i++;
    } while (word[i]);
    top++;
    return;
  }
```
See also sections 16 and 18.

This code is used in section 1.

15. Next comes pop.

⟨ Prototypes for functions 13 ⟩ +≡
```
  char *pop( );
```

16. This function does two things: it returns the *top* element of the *stack*; it removes the *top* element from the *stack*. We can do this in one instruction if we are clever.

⟨ Implementation of functions 14 ⟩ +≡
```
  char *pop( )
  {
    return stack[−−top];
  }
```

17. Last but not least is the function to test whether the stack has anything on it.

⟨ Prototypes for functions 13 ⟩ +≡
```
  bool isEmpty( );
```

18. The implementation of this is simple enough—we just need to know if the value of *top* is 0.

⟨ Implementation of functions 14 ⟩ +≡
```
  bool isEmpty( )
  {
    return top ≡ 0;
  }
```

19. Index. This index is automatically created. It lists all the variables used in the program and the section(s) in which they are used. Underlined entries indicate where a variable is defined. The remaining sections of this document are also created automatically.

EX1