EX8

1. Introduction. This is a literate program which solves the problem set in lab eight—Dynamic stack management.

We will start with the, by now familiar, outline of the program. The stack, along with its size and the number of valid entries will be made global.

```
⟨ Headers 4 ⟩
using namespace std;
int *stack;
int stackSize = 0;
int stackPointer = 0;
⟨ Prototypes for functions 10 ⟩
⟨ The main program 2 ⟩
⟨ Implementation of functions 11 ⟩
```

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

```
⟨ The main program 2 ⟩ ≡
  int main( )
  {
    ⟨ Variables of main 3 ⟩
    ⟨ Open the input file 5 ⟩
    ⟨ Set up the initial stack 6 ⟩
    ⟨ Do the main loop 7 ⟩
    ⟨ Finish and clean up 9 ⟩
  }
```

This code is used in section 1.

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

```
⟨ Variables of main 3 ⟩ ≡
  char filename[20];
  ifstream fin;
```

See also section 8.

This code is used in section 2.

4.

```
⟨ Headers 4 ⟩ ≡
#include <iostream>
#include <fstream>
```

See also section 12.

This code is used in section 1.

5. Right—now we can get the file opened, ready for input. 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 the input file 5 ⟩ ≡
  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 2.

6. Before we start looping over the input we need to create an initial stack. The size of the stack will be read from *fin*.

```
⟨ Set up the initial stack 6 ⟩ ≡
  fin ≫ stackSize;
  stack = new int[stackSize];
```

This code is used in section 2.

7. We can now proceed with the main loop. Each time round we read a *command* which will be either "push" followed by an integer *value* or "pop".
Note: The input command is identified by the simple expedient of looking at the second character of the command—if this is a `'u'` we assume that *command* contains the word "push", otherwise we assume it contains "pop". This hack saves doing a complete string comparison but runs the slight risk that it may interpret erroneous input as a call to *pop*.

```
⟨ Do the main loop 7 ⟩ ≡
  while (fin ≫ command) {
    if (command[1] ≡ 'u') {
      fin ≫ value;
      push(value);
    }
    else {
      value = pop( );
    }
  }
  cout ≪ "Stack␣contains␣" ≪ stackPointer ≪ "␣entries." ≪ endl;
```

This code is used in section 2.

8. We need to declare *command* and *value*.

```
⟨ Variables of main 3 ⟩ +≡
  char command[20];
  int value;
```

9. We should also close our input file.

```
⟨ Finish and clean up 9 ⟩ ≡
  fin.close( );
  return 0;
```

This code is used in section 2.

10. Functions. There are only two functions required for this exercise, *push* and *pop*. We will declare the prototypes first:

```
⟨ Prototypes for functions 10 ⟩ ≡
  void push(int);
  int pop( );
```

This code is used in section 1.

11. Now comes the implementation: The *push* function adds a *value* to the top of the stack and increments *stackPointer*. If the stack is now full we double its size before we return.
Note: Rather than copy the entries in *stack* one at a time we can use *memcpy* to shift the values in one go. This hack will result in a faster copy. The last parameter to *memcpy* is a byte count which is why the multipler of 4 is used.

```
⟨ Implementation of functions 11 ⟩ ≡
  void push(int value)
  {
    stack[stackPointer++] = value;
    if (stackPointer ≡ stackSize) {
      stackSize *= 2;
      int *temp = new int[stackSize];
      memcpy(temp, stack, 4 * stackPointer);
      delete[] stack;
      stack = temp;
      cout ≪ "Stack␣doubled␣from␣" ≪ stackPointer ≪ "␣to␣" ≪ stackSize ≪ endl;
    }
    return;
  }
```

See also section 13.
This code is used in section 1.

12. We need to add a header for *memcpy*( ).

```
⟨ Headers 4 ⟩ +≡
#include <cstring>
```

13. The *pop* function decrements *stackPointer* and returns the value popped from the top of the *stack*. If the *stack* is empty when *pop* is called we simply return 0.

```
⟨ Implementation of functions 11 ⟩ +≡
  int pop( )
  {
    if (stackPointer ≡ 0) return 0;
    return stack[--stackPointer];
  }
```

14. 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.