EX2

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

We will start with the outline of the program. You may note that this is not much different from the first example.

```
⟨ Headers 5 ⟩
using namespace std;
⟨ Prototypes for functions 12 ⟩
⟨ Global data 2 ⟩
⟨ The main program 3 ⟩
⟨ Implementation of functions 13 ⟩
```

2. To start with we need to declare the *heap*, which will be an array of **int**. To avoid extra parameters on the *makeHeap* and *siftUp* functions we will make the heap global. We also declare an integer value *size* which will contain the current nimber of values in the heap, initially it will be zero. To make the program a little more flexible we will use integer constants `HEAP_SIZE` for the size of the *heap* array.

```
⟨ Global data 2 ⟩ ≡
  const int HEAP_SIZE = 100;
  int heap[HEAP_SIZE];
  int size = 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 heap 7 ⟩
    ⟨ Convert the array into a heap 8 ⟩
    ⟨ Print the first five elements on the heap 9 ⟩
    ⟨ Finish and clean up 10 ⟩
  }
```

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;
```

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 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 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 can read the integer values from the input file straight into the *heap* array, changing size as we go. Note that the loop has an empty body as we do all the processing in the termination condition.

```
⟨ Read the file into the heap 7 ⟩ ≡
  while (fin ≫ heap[size++]) ;
```

This code is used in section 3.

8. On with the main program. At this point we have stored everything in the *heap* array, Now we must do the necesary work required to order the array into a heap. We do this via the function *makeHeap*.

⟨ Convert the array into a heap 8 ⟩ ≡
```
makeHeap( );
```
This code is used in section 3.

9. Now that the *heap* array actually contains a heap, it only remains to print out the first five values.

⟨ Print the first five elements on the heap 9 ⟩ ≡
```
for (int i = 0; i < 5; i++) cout ≪ heap[i] ≪ "␣";
cout ≪ endl;
```
This code is used in section 3.

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

⟨ Finish and clean up 10 ⟩ ≡
```
fin.close( );
```
This code is used in section 3.

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

12. Let's start with *makeHeap*—first the prototype.

```
⟨ Prototypes for functions 12 ⟩ ≡
  void makeHeap( );
```

See also section 14.

This code is used in section 1.

13. And the implementation. *makeHeap* calls siftDown on each non-leaf member of the *heap* array, working backwards towards *heap*[0], the top of the heap.

```
⟨ Implementation of functions 13 ⟩ ≡
  void makeHeap( )
  {
    int i;
    cerr ≪ "In␣makeHeap␣" ≪ size ≪ endl;
    for (i = size/2; i ≥ 0; i−−) {
      cerr ≪ "siftDown␣" ≪ i ≪ "␣" ≪ heap[i] ≪ endl;
      siftDown(i);
    }
    return;
  }
```

See also section 15.

This code is used in section 1.

14. All we need to do now is to code siftDown, a recursive function which puts element i into the correct location in the *heap* array.

```
⟨ Prototypes for functions 12 ⟩ +≡
  void siftDown(int);
```

15. *siftDown* works by comparing the value of the current element with those of its children, if any. If the larger child value is greater than the value of its parent we swap the values and call siftDown, once again, on the child. For arrays starting at zero the children of element i are stored in locations $2 * i + 1$ and $2 * i + 2$. If, when *siftDown* is called we are already at a leaf then we simply return.

```
⟨ Implementation of functions 13 ⟩ +≡
  void siftDown(int current)
  {
    int child = 2 * current + 1;
    if (child ≥ size) return;
    if (child + 1 < size ∧ heap[child] < heap[child + 1]) child++;
    if (heap[current] < heap[child]) {
      ⟨ Swap elements current and child 16 ⟩
      siftDown(child);
    }
    return;
  }
```

16. All that remains is to do the swap.

```
⟨ Swap elements current and child 16 ⟩ ≡
  int temp = heap[current];
  heap[current] = heap[child];
  heap[child] = temp;
```

This code is used in section 15.

17. 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.