EX3

1. Introduction. This is a literate program which solves the problem set in lab three—virtual initialization of arrays.

We will start with the, by now familiar, outline of the program. This time we will not use global data and we have no functions other than *main*.

⟨ Headers 4 ⟩

using namespace std;

⟨ The main program 2 ⟩

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 and validate the input file 5 ⟩
    ⟨ Store values into the array 7 ⟩
    ⟨ Check values in the array 12 ⟩
    ⟨ Finish and clean up 14 ⟩
  }
```

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 sections 6, 8, 10, and 13.

This code is used in section 2.

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

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

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 and validate 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 reading the file we need to declare the three arrays we are going to use to store, and track the validity of, the input data. These consist of *data* which will contain the values we read in and two arrays, *forward* and *backward*, that will keep track of which entries in *data* contain valid values. We will use the constant `ARRAY_SIZE` to determine the size of our arrays.

```
⟨ Variables of main 3 ⟩ +≡
  const int ARRAY_SIZE = 100;
  int data[ARRAY_SIZE];
  unsigned int forward[ARRAY_SIZE];
  unsigned int backward[ARRAY_SIZE];
```

7. Store values. We are now ready to do the main input loop. we can read the integer values from the input file and store them in the *data* array. The pairs of *what* and *where* values are term=nated by a line containing $-1-1$.

```
⟨ Store values into the array 7 ⟩ ≡
  while (true) {
    fin ≫ what ≫ where;
    if (what ≡ −1 ∧ where ≡ −1) break;
    ⟨ Put element what into data[where] 9 ⟩
  }
```

This code is used in section 2.

8. Better declare *what* and *where*.

```
⟨ Variables of main 3 ⟩ +≡
  int what, where;
```

9. The process of storing a value into our *data* array is a little complex. there are two possible cases:

1. this is the first time we have stored a value in this location—in this case we need to validate the entry before we store it;
2. we already have something stored here—in this case we simply store the changed value.

We will sanity check *where* to ensure that it is a valid index and throw an error message if it is not.

```
⟨ Put element what into data[where] 9 ⟩ ≡
  if (where < ARRAY_SIZE) {
    if (backward[where] ≥ numValid ∨ forward[backward[where]] ≠ where) {
      ⟨ Validate location where 11 ⟩
    }
    data[where] = what;
  }
  else cerr ≪ "Index␣value␣" ≪ where ≪ "is␣outside␣the␣range␣of␣the␣data␣array." ≪ endl;
```

This code is used in section 7.

10. Ooh - I just used a new variable, *numValid*, the number of valid entries in the *data* array. I had better declare and intialize it.

```
⟨ Variables of main 3 ⟩ +≡
  int numValid = 0;
```

11. The validation process works as follows:

1. we record *numValid* in *backward*[*where*];
2. we record *where* in *forward*[*numValid*];
3. we increment *numValid*.

```
⟨ Validate location where 11 ⟩ ≡
  backward[where] = numValid;
  forward[numValid] = where;
  numValid++;
```

This code is used in section 9.

12. Check Values. Now for the second part of the program. We have read in all the *what*, *where* pairs and stored the values in our *data* array. The second part of the program involves reading a sequence of *probe* values and reporting the status of *data*[*probe*]. This location will either contain a valid value or will be unititialized. This time the sanity check is on *probe*.

```
⟨ Check values in the array 12 ⟩ ≡
  while (true) {
    fin ≫ probe;
    if (probe ≡ −1) break;
    if (probe < ARRAY_SIZE) {
      if (backward[probe] ≥ numValid ∨ forward[backward[probe]] ≠ probe)
        cout ≪ "Location␣" ≪ probe ≪ "␣is␣unititialized." ≪ endl;
      else cout ≪ "Location␣" ≪ probe ≪ "␣contains␣the␣value␣" ≪ data[probe] ≪ "." ≪ endl;
    }
    else cerr ≪ "Index␣value␣" ≪ probe ≪ "is␣outside␣the␣range␣of␣the␣data␣array." ≪ endl;
  }
```

This code is used in section 2.

13. We should also declare *probe*.

```
⟨ Variables of main 3 ⟩ +≡
  int probe;
```

14. We should also close our input file.

```
⟨ Finish and clean up 14 ⟩ ≡
  fin.close( );
  return 0;
```

This code is used in section 2.

15. 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.