# EX9

**1. Introduction.** This is a literate program which solves the problem set in lab nine—Crazy Eights.

We will start with the, by now familiar, outline of the program. We don't need any functions other than main and there will be no global data either.

```
⟨ 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 the input file 5 ⟩
    ⟨ Read in the deck and initialize the data 7 ⟩
    ⟨ Do the main loop 8 ⟩
    ⟨ Finish and clean up 10 ⟩
  }
```

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 6.

This code is used in section 2.

**4.**

```
⟨ 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 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 read in the deck from the input file. There are a few ways we can define a card—struct, class, union—but in this case I will simply define two arrays, *suit* and *rank*. We will also define the *length* array which will be used to store the length of the longest allowable sequence of cards starting at the given card, and *maxIndex*, the index of the card with the longest allowable sub-sequence found so far. We also need to record the maximum length of the best solution we have found so far. I will also declare an array *last* which will track the first card of the sub-sequence corresponding to the entries in *length*.

**Note:** As a bit of a bonus I will also track and report the cards in the longest sequence. To do this we will use the array *next* to note the the index of the card which follows a given card in the sequence.

```
⟨ Variables of main 3 ⟩ +≡
  unsigned char suit[52], rank[52];
  int length[52], last[52];
  int maxIndex, maxLength;
  int next[52];
```

**7.** We will store the input in elements 0-51. We will also store the *length* and *last* card values to reflect a non-optimal solution in which we select a single card. Thus the initial legal sequence starting at card $i$ consists of just this card and has a *length* of 1.

**Note:** I will use a slightly different version of the algorithm from the one presented in the lecture notes. This version eliminates the need for the dummy starting card. Instead we will work backwards from the last card. Now we can set the maximum length for the last card to $i$ and record it as the card which starts the longest sub-sequence we have found so far.

**Note:** We will store $-1$ in the *next* array—this value will indicate the end of a sequence.

```
⟨ Read in the deck and initialize the data 7 ⟩ ≡
  for (int i = 0; i < 52; i++) {
    fin ≫ rank[i] ≫ suit[i];
    length[i] = 1;
    last[i] = i;
    next[i] = −1;
  }
  maxIndex = 51;
  maxLength = 1;
```

This code is used in section 2.

**8.** Now we can proceed with the main part of the algorithm. As already noted we will work backwards through the deck. for each card we will determine the length of the maximal sub-sequence that starts with this card by considering all subsequent cards and recording *length*[*i*] as *length*[*j*] + 1 where cards *i* and *j* are alike and card *j* has the greatest *length* of all following, similar cards. We start at the second last card as we already have the optimal (only possible) solution for the last card. Rather than use a dummy card we are assuming that each card forms a sequence of length 1 before we start. This is a valid assumption and represents the worst-case scenario for the problem—a single card is always a valid, although probably non-optimal, sequence.

```
⟨ Do the main loop 8 ⟩ ≡
  for (int i = 50; i ≥ 0; i−−) {
    for (int j = 1 + i; j < 52; j++) {
      if ⟨ Cards i and j are alike 9 ⟩ {
        if (length[i] ≤ length[j]) {
          length[i] = length[j] + 1;
          last[i] = last[j];
          next[i] = j;
        }
      }
    }
    if (length[i] > maxLength) {
      maxIndex = i;
      maxLength = length[i];
    }
  }
```

This code is used in section 2.

**9.** Two cards *i* and *j* are alike if:

- They have the same *suit*;
- They have the same *rank*;
- Either card has a *rank* of `'8'`.

```
⟨ Cards i and j are alike 9 ⟩ ≡
  (suit[i] ≡ suit[j] ∨ rank[i] ≡ rank[j] ∨ rank[i] ≡ '8' ∨ rank[j] ≡ '8')
```

This code is used in section 8.

**10.** Now that we have finished our search we can report the results and clean up. The maximal sub-sequence has a length of *maxLength* and starts at card *maxIndex*. It finishes at card *last*[*maxIndex*].

```
⟨ Finish and clean up 10 ⟩ ≡
  cout ≪ "The␣longest␣sequence␣has␣a␣length␣of␣" ≪ maxLength ≪ endl;
  cout ≪ "The␣longest␣sequence␣starts␣with␣" ≪ rank[maxIndex] ≪ suit[maxIndex] ≪
      "␣and␣ends␣with␣" ≪ rank[last[maxIndex]] ≪ suit[last[maxIndex]] ≪ endl;
  cout ≪ "The␣winning␣sequence␣of␣cards␣is:␣";
  for (int id = maxIndex; id ≠ −1; id = next[id]) cout ≪ rank[id] ≪ suit[id] ≪ "␣";
  cout ≪ endl;
  fin.close( );
```

This code is used in section 2.

**11. 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.