READBRIEFCASE

1. Overview. In June, 2012. I was contacted by a doctor's office that wanted to migrate away from the *Laserfiche* product, but couldn't find an easy way to carry their massive collection of TIF images into their new system. Originally, they asked me to try to pull the information out of the *Laserfiche* database, but that wasn't feasible because they didn't have passwords for it. There were 500GB of image files belonging to tens of thousands of patients, so populating the new system manually was not an option!

After playing with the product for awhile, I realized that it allowed exporting batches of files in a so-called "briefcase." I immediately recalled a Microsoft "briefcase" file in old versions of Office, but these files did not appear to match that format. Still, using a hex-editor, it wasn't too hard to see unencrypted TIF files inside, and also character data identifying patients and doctors. Nothing seemed to be compressed or encrypted. So I told the company I could take a shot at reverse-engineering the export file-format and creating an organized group of files out on a disk.

Over the next couple of days, I worked out the structure of the files. It was pretty typical, with tagged section names, most of which were followed by either a length in bytes or a count of sub-records. The hardest part was working out the mapping between files and owners. Thankfully the section that ultimately provided that information was called "relations," making it an obvious choice for close scrutiny.

2. Here is an overview of the entire conversion program:

```
#include <map>
#include <string>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
#include <sys/stat.h>
  using std::map;
  using std::string;
  using std::istream;
  using std::ostream;
  using std::ifstream;
  using std::ofstream;
  ⟨ Support Functions 7 ⟩
  ⟨ Helper Classes 10 ⟩
  ⟨ Global Data 4 ⟩
  int main(int argc, char **argv)
  {
    ⟨ Open the briefcase 3 ⟩;
    ⟨ Process the briefcase 12 ⟩;
    ⟨ Write out the index 34 ⟩;
    ⟨ Cleanup 5 ⟩;
    return 0;
  }
```

3. People will call the program with an argument for the file to read.

```
⟨ Open the briefcase 3 ⟩ ≡
  if (argc < 2) {
    std::cerr << "Usage:␣␣readbriefcase␣<fn>" << std::endl;
    return -1;
  }
  ifstream in(argv[1], std::ifstream::in | std::ifstream::binary);
  if (¬in) {
    std::cerr << "Cannot␣open␣input␣file␣<" << argv[1] << ">" << std::endl;
  }
  input_filename ← argv[1];
  std::cout << "Reading␣input␣file:␣" << input_filename << std::endl;
```

This code is used in section 2.

4.

```
⟨ Global Data 4 ⟩ ≡
  string input_filename;
```

See also sections 16, 20, and 26.

This code is used in section 2.

5. Of course I will close the file when I'm done with it. Might as well go ahead and do that.

```
⟨ Cleanup 5 ⟩ ≡
  in.close( );
```

This code is used in section 2.

6. Some Utilities. Before I get too far into parsing the briefcase, I'll take some time to provide a few utility functions.

7. I'm going to want to report on the current file position from several places. To help with that, I'll override **operator**≪ so I can pass the file to **ostream** and get the reporting I want. For good measure, I report both the decimal and hex positions. That was a big help during the reverse-engineering process, because I was looking for pointers to the important file locations in hex dumps of the breifcases..

```
⟨ Support Functions 7 ⟩ ≡
  ostream &operator≪(ostream &os, ifstream &in)
  {
    std::streampos loc ← in.tellg( );
    os ≪ "at␣file␣location␣" ≪ std::dec ≪ loc ≪ "␣(0x" ≪ std::hex ≪ loc ≪ ')' ≪ std::dec;
    return os;
  }
```

See also sections 8 and 11.

This code is used in section 2.

8. Since the structure of the briefcase is based on tagged sections, another operation I'll be doing a lot is looking for a tag. Usually, it will be where I expect it to be, but since my knowledge of the file format is imperfect, I'll allow for some wiggle-room and search ahead up to *max_search* bytes sometimes. Other times, when I'm positive where the tag should be, I call the function with *max_search* set to 0.

The search loop was tricky because there are multiple ways to fail, and also a way to backtrack. The invariant I chose to maintain was:

$$(\forall x \mid 0 \le x < \text{idx} \;:\; \text{in}(x) = \text{tag}(x))$$

You can see that when the characters don't match, I maintain the invariant by backing up and re-starting the search. When the loop finishes, I've either established that idx = tag.size, or I've detected a failure condition. The two failures I watch out for are EOF and travelling beyond *max_search* without a match.

```
⟨ Support Functions 7 ⟩ +≡
  int find_tag(ifstream &in, const string &tag, int max_search ← 1024)
  {
    std::cout ≪ "Looking␣for␣" ≪ tag ≪ "␣within␣" ≪ max_search ≪ '␣' ≪ in ≪ std::endl;
    int idx ← 0;      /* string index */
    int distance ← 0;      /* travel distance */
    while (idx < tag.size( )) {
      if (in.eof( ) ∨ distance > max_search) goto failure;
      if (in.get( ) ≡ tag[idx]) {
        ++idx;
      }
      else {
        in.seekg(−idx, ifstream::cur);
        ++distance;
        idx ← 0;
      }
    }
    std::cout ≪ "...found␣it␣in␣" ≪ distance ≪ "␣steps␣" ≪ in ≪ std::endl;
    return distance;
  failure:
    ⟨ Complain if the tag wasn't found 9 ⟩;
  }
```

9. When the match fails, issue a complaint and return an error code.

```
⟨ Complain if the tag wasn't found 9 ⟩ ≡
  std::cout ≪ "XXX did not find tag " ≪ tag ≪ '!' ≪ std::endl;
  return −1;
```

This code is used in section 8.

10. Just as it was convenient to present the file position in both decimal and hex, it will be helpful to do the same with several integers along the way. An easy way to do this is to wrap it in a helper class, which I call *dechex* here.

```
⟨ Helper Classes 10 ⟩ ≡
  class dechex {
  private:
    unsigned int num;
  public:
    dechex(unsigned int n) : num(n) { }
    friend ostream &operator≪(ostream &, const dechex &);
  };
  ostream &operator≪(ostream &os, const dechex &x)
  {
    os ≪ std::dec ≪ x.num ≪ " (0x" ≪ std::hex ≪ x.num ≪ ')';
  }
```

See also sections 23, 24, and 35.

This code is used in section 2.

11. Finally, here are some helper functions to read a words and doublewords from the stream. We'll go ahead and define them here. N.B.: This code assumes ints are at least 32 bits wide, and that the machine is little-endian (briefcase files are little-endian). On a big-endian machine, you'll need to swap the bytes in *ans*.

```
⟨ Support Functions 7 ⟩ +≡
  unsigned int read_next_dword(istream &in)
  {
    unsigned int ans ← 0;
    in.read(reinterpret_cast⟨char *⟩(&ans), 4);
    return ans;
  }
  unsigned int read_next_word(istream &in)
  {
    unsigned int ans ← 0;
    in.read(reinterpret_cast⟨char *⟩(&ans), 2);
    return ans;
  }
```

12. Parsing the File. At a high level, a breifcase file consists of a header, followed by a list of "objects" (names of people to whom documents will be assigned), followed by a list of "relations" (a mapping from people to their documents), followed by the documents. I'll map out the code here, and then fill in these sections one-by-one.

```
⟨ Process the briefcase 12 ⟩ ≡
  ⟨ Parse the Header 13 ⟩;
  ⟨ Parse the Objects 14 ⟩;
  ⟨ Parse the Relations 18 ⟩;
  ⟨ Parse the Documents 21 ⟩;
```

This code is used in section 2.

13. The file should start with `LFWIN`, then the next word should be `#0004`, I think. Next there is a bunch of text saying "`UNENCRYPTED`" over and over. Then, I get a doubleword telling me when this briefcase is over. The file may have multiple briefcases in it, but I think we only want the type 4 ones, based on the data I'm looking at..

```
⟨ Parse the Header 13 ⟩ ≡
  if (find_tag(in, string("LFWIN"), 0) ≠ 0) return −1;
  unsigned int lfwin_type ← read_next_word(in);
  std::cout ≪ "Briefcase␣type␣" ≪ lfwin_type ≪ std::endl;
  in.seekg(48, ifstream::cur);      /* skip UNENCRYPTEDUNEN... */
  unsigned int lfwin_end ← read_next_dword(in);
```

This code is used in section 12.

14. Now that I'm sure I'm in a briefcase file, I need to find the "`OBJECTS`" directory. In every file I've seen so far, it's been at location `#580`. But, I might as well search for it. The tag is followed by a dword count of the objects I can read.

```
⟨ Parse the Objects 14 ⟩ ≡
  int objects_distance ← find_tag(in, string("OBJECTS"), 2000);
  if (objects_distance < 0) return −1;
  unsigned int number_objects ← read_next_dword(in);
  std::cout ≪ "There␣are␣" ≪ dechex(number_objects) ≪ "␣objects␣to␣read." ≪ std::endl;
```

See also section 15.

This code is used in section 12.

15. Now I know how many objects there are, and I need to read them in, one by one. Each is 45 bytes long, and is made of a dword *id* followed by a name of length 41. I read these in and store them in a map, with the name adjusted so that it can serve as a file name.

```
⟨ Parse the Objects 14 ⟩ +≡
  char object_name[41];
  for (unsigned int i ← 0; i < number_objects; ++i) {
    unsigned int id ← read_next_dword(in);
    in.read(object_name, 41);
    ⟨ Sanitize object name for filename use 17 ⟩;
    briefcase_objects[id] ← object_name;
    std::cout ≪ "READ:␣" ≪ dechex(id) ≪ ":␣" ≪ object_name ≪ std::endl;
  }
```

16. I need to define the *breifcase_objects* map from the previous section.

```
⟨ Global Data 4 ⟩ +≡
  typedef map⟨unsigned int, string⟩ object_dictionary;
  object_dictionary briefcase_objects;
```

17. To turn object names into file names, I replace all "special" characters with underscores.

```
⟨ Sanitize object name for filename use 17 ⟩ ≡
  for (int pos ← 0; pos < 41; ++pos) {
    if (object_name[pos] ≡ '␣' ∨ object_name[pos] ≡ ',' ∨
          object_name[pos] ≡ '&' ∨ object_name[pos] ≡ '/') {
      object_name[pos] ← '_';
    }
  }
```

This code is used in section 15.

18. At this point in the file (just past the object list), I should be at the "`RELATIONS`" table. I give up to 4 bytes of leeway to locate it. It starts with the tag name, followed by a count of the relations, and the root object of the briefcase:

```
⟨ Parse the Relations 18 ⟩ ≡
  int relations_distance ← find_tag(in, string("RELATIONS"), 4);
  if (relations_distance < 0) return −1;
  unsigned int number_relations ← read_next_dword(in);
  std::cout ≪ "There␣are␣" ≪ dechex(number_relations) ≪ "␣relations␣to␣read." ≪ std::endl;
  unsigned int briefcase_root ← read_next_dword(in);
  std::cout ≪ "Briefcase␣root␣is␣" ≪ dechex(briefcase_root) ≪ ":␣" ≪
      briefcase_objects[briefcase_root] ≪ std::endl;
```

See also section 19.

This code is used in section 12.

19. Each relation is just two dwords, which I read into a map.

```
⟨ Parse the Relations 18 ⟩ +≡
  for (unsigned int i ← 0; i < number_relations; ++i) {
    unsigned int parent_id ← read_next_dword(in);
    unsigned int document_id ← read_next_dword(in);
    briefcase_relations[document_id] ← parent_id;
    std::cout ≪ "READ:␣" ≪ dechex(document_id) ≪ '␣' ≪ briefcase_objects[document_id] ≪ "␣->␣" ≪
        dechex(parent_id) ≪ '␣' ≪ briefcase_objects[parent_id] ≪ std::endl;
  }
```

20. I need to define the relations map:

```
⟨ Global Data 4 ⟩ +≡
  typedef map⟨unsigned int, unsigned int⟩ object_relations;
  object_relations briefcase_relations;
```

21. The Documents. Now I get to the juiciest part of the affair: reading each document. Each document starts with a the letters "DOC" followed by the id of the document. It can be mapped to its parent via the relations table I previously stored off. Then comes a word with the number of pages in the document. Then comes an unknown word, probably indicating the type of document. It seems to usually be #0002. Next is a word giving the length of the document metadata. When it contains the patient metadata, the length will be #61. Further notes: if the number of pages is #FFFF, then it won't have a "PAGE" record, but rather a "*doc*" record (I think!).

Note that there doesn't seem to be a count of documents anywhere, so I just parse as many as I can find until I reach the end of the briefcase.

```
⟨ Parse the Documents 21 ⟩ ≡
  ⟨ Data needed for parsing DOC records 28 ⟩
  while (true) {
    int doc_distance ← find_tag(in, string("DOC"), 4);
    if (doc_distance < 0) {
      find_tag(in, string("ONE"), 4);
      if (in.tellg( ) ≥ lfwin_end) {
        std::cout ≪ "Reached␣the␣end␣of␣the␣briefcase␣normally." ≪ std::endl;
          /* Possible enhancement: check for extra II* records... */
      }
      break;
    }
    ⟨ Process a DOC record 22 ⟩;
  }
```

This code is used in section 12.

22. When a document is found, I just read the potions of the record (described above), one bit at a time.

```
⟨ Process a DOC record 22 ⟩ ≡
  unsigned int doc_id ← read_next_dword(in);
  const string &doc_name ← briefcase_objects[doc_id];
  unsigned int doc_owner_id ← briefcase_relations[doc_id];
  const string &doc_owner_name ← briefcase_objects[doc_owner_id];
  ⟨ Check and report on which document we found 39 ⟩;
  unsigned int doc_pages ← read_next_word(in);
  unsigned int doc_type ← read_next_word(in);
  std::cout ≪ "The␣document␣has␣" ≪ dechex(doc_pages) ≪ "␣pages,␣and␣is␣type␣" ≪
      dechex(doc_type) ≪ std::endl;
  patient_data & pdat ← briefcase_patients[doc_owner_id];
  if (doc_type ≠ 0) {
    unsigned int doc_metadata_length ← read_next_word(in);
    if (doc_metadata_length ≡ #61) {
      ⟨ Read and update patient metadata 27 ⟩;
    }
    else {
      std::cout ≪ "XXX␣This␣DOC␣does␣not␣have␣length␣0x61,␣instead␣" ≪
          dechex(doc_metadata_length) ≪ ",␣so␣skipping." ≪ std::endl;
      in.seekg(doc_metadata_length, ifstream::cur);
    }
  }
  else {
    std::cout ≪ "Doc␣type␣0␣means␣no␣metadata␣to␣read." ≪ std::endl;
  }
```

See also section 29.

This code is used in section 21.

23. In the above fragment, I introduced a new map for objects of type "*patient_data*." This is what we'll use to remember stuff like their SSN and doctor, etc. I'll define a class to store this now:

```
⟨ Helper Classes 10 ⟩ +≡
  class patient_data {
  private:
    string last_name;
    string first_name;
    string middle_name;
    string patient_number;
    string social_security_num;
    string doctor;
    string directory_name;
  public:
    patient_data( ) { }
    void fill(const char *const ln,
        const char *const fn,
        const char *const mn,
        const char *const pnum,
        const char *const ssn,
        const char *const doc,
        const string &parent_obj);
    const string &get_directory( ) const
    {
      return directory_name;
    }
    friend ostream &operator≪(ostream &, const patient_data &);
  };
```

24. Every time I look up a patient, I send it everything I found in the "DOC" record. The first time, the **patient_data** object will be empty, so it takes a copy of the data. Every time after that, the object has an opportunity to check that the data for the patient hasn't changed. If it has changed, I don't try to reconcile the differences, as this is an off-line conversion app. I do at least try to snag a SSN if it was missing in the first record I saw. This stuff was obviously filled out manually at some point, and people make typos and leave out data at times.

```
⟨ Helper Classes 10 ⟩ +≡
  void patient_data::fill(const char *const ln,
          const char *const fn,
          const char *const mn,
          const char *const pnum,
          const char *const ssn,
          const char *const doc,
          const string &parent_obj)
  {
    if (last_name.length( ) ≡ 0 ∧ patient_number.length( ) ≡ 0) {
      std::cout ≪ "Filling␣in␣patient␣info␣for␣the␣first␣time." ≪ std::endl;
      last_name ← ln;
      first_name ← fn;
      middle_name ← mn;
      patient_number ← pnum;
      social_security_num ← ssn;
      doctor ← doc;
      ⟨ Assign a directory to this patient 25 ⟩;
    }
    else {
      ⟨ Try to recover a missing SSN 38 ⟩;
    }
  }
```

25. There are tens of thousands of patients in these briefcase files. So, I don't want to throw all the data in a single directory! I build up a directory tree based on the patients' last names. At the leaves, they each get their own directory, filled with their documents. Let's say the patient is:

```
"Dave␣L␣Thomas␣504-33-3293␣393823␣[10]␣LEVERTON"
```

Then, the directory should be:

```
"T/THO/Thomas_Dave_L_504333293_LEVE_393823"
```

⟨ Assign a directory to this patient 25 ⟩ ≡

```
std::ostringstream d;
if (last_name.length() > 0) {
  d ≪ last_name[0];
}
else {
  d ≪ '_';
}
d ≪ '/' ≪ std::setfill('_') ≪ std::setw(3) ≪ std::left ≪ last_name.substr(0, 3) ≪ '/';
if (last_name.length() > 0) {
  d ≪ last_name ≪ '_' ≪ first_name ≪ '_' ≪ middle_name ≪ '_';
  d ≪ social_security_num ≪ '_';
  if (doctor.length() ≥ 5) {
    d ≪ std::setw(4) ≪ doctor.substr(5, 4) ≪ '_';
  }
  d ≪ patient_number;
}
else {
  for (std::size_t i ← 0; i < parent_obj.length(); ++i) {
    char c ← parent_obj[i];
    if ((c ≡ '␣') ∨ (c ≡ '[') ∨ (c ≡ ']')) c ← '_';
    d ≪ c;
  }
}
directory_name ← d.str();
```

This code is used in section 24.

26. I still need to define a global mapping from owner *id*s to **patient_data** records:

⟨ Global Data 4 ⟩ +≡

```
typedef map⟨unsigned int, patient_data⟩ patient_info;
patient_info briefcase_patients;
```

27. The format of the document metadata in the #61-length case is as follows:
– 21-char last name,
– 21-char first name
– 2-char middle name
– 7-char patient number
– unknown dword
– 21-char SSN
– 21-char doctor name.

```
⟨ Read and update patient metadata 27 ⟩ ≡
  in.read(scratch_area, #61);
  pdat.fill(scratch_area,      /* last (21) */
   scratch_area + 21,      /* first (21) */
   scratch_area + 42,      /* middle (2) */
   scratch_area + 44,      /* patient number (7 + 4) */
   scratch_area + 55,      /* SSN (21) */
   scratch_area + 76,      /* doctor (21) */
   doc_owner_name);
  ⟨ Create directories to hold the files 37 ⟩;
```

This code is used in section 22.

28.

```
⟨ Data needed for parsing DOC records 28 ⟩ ≡
  char scratch_area[#61];      /* 0x61 = 97 */
```

This code is used in section 21.

29. So I've read in the doc metadata, if there was any. Next I should have *doc_pages*-worth of page records.

```
⟨ Process a DOC record 22 ⟩ +≡
  for (int page_no ← 0; page_no < doc_pages; ++page_no) {
    ⟨ Process a PAGE record 30 ⟩;
  }
```

30. A PAGE record starts with the letters "PAGE" For now, I'm assuming that the DOC record's page count wasn't #FFFF. After the tag name, it has a word containing the page number. Next is a dword with unknown meaning, followed by a dword for the length of the page.

```
⟨ Process a PAGE record 30 ⟩ ≡
  int page_distance ← find_tag(in, string("PAGE"), 4);
  if (page_distance < 0) return −1;
  unsigned int reported_page_number ← read_next_word(in);
  ⟨ Report on whether the reported page number was expected or not 31 ⟩;
  unsigned int unknown_page_dword ← read_next_dword(in);
  unsigned int page_length ← read_next_dword(in);
  std::cout ≪ "Page␣is␣length␣" ≪ dechex(page_length) ≪ std::endl;
  ⟨ Output the TIF file 32 ⟩;
  ⟨ Skip past any addenda 33 ⟩;
```

This code is used in section 29.

31. I keep the user in the loop about which page I'm on, and if the numbers in the file match my expectations.

```
⟨ Report on whether the reported page number was expected or not 31 ⟩ ≡
  if (reported_page_number ≡ page_no) {
    std::cout ≪ "This␣is␣page␣" ≪ page_no + 1 ≪ "␣of␣" ≪ doc_pages ≪ std::endl;
  }
  else {
    std::cout ≪ "XXX␣Expecting␣page␣number␣" ≪ page_no ≪ "␣and␣got␣" ≪
        dechex(reported_page_number) ≪ std::endl;
  }
```

This code is used in section 30.

32. The TIF file should immediately follow the `PAGE` record. So, if it looks like a TIF file is next, output it. The two options I look for are "`II*`" and "`MM*`". The first couple of lines below just make sure the check fails when the page length is less than three characters. After that, I just build up a file name and write out the data verbatim.

```
⟨ Output the TIF file 32 ⟩ ≡
  if (page_length ≥ 3) {
    in.read(scratch_area, 3);
  }
  else {
    scratch_area[0] ← 'X';
  }
  if ((scratch_area[0] ≡ 'I' ∨ scratch_area[0] ≡ 'M') ∧
        (scratch_area[0] ≡ scratch_area[1]) ∧
        (scratch_area[2] ≡ '*')) {      /* we have a TIF file */
    std::ostringstream fn_maker;
    fn_maker ≪ pdat.get_directory();
    fn_maker ≪ '/' ≪ doc_name ≪ "_pg" ≪ std::setw(4) ≪ std::setfill('0') ≪ std::right ≪
        (reported_page_number + 1) ≪ ".tif";
    string filename ← fn_maker.str();
    std::cout ≪ "Creating␣file␣<" ≪ filename ≪ '>' ≪ std::endl;
    ofstream out_tif(filename.c_str(), std::ofstream::out | std::ofstream::binary);
    if (¬out_tif) {
      std::cout ≪ "XXX␣problem␣writing␣output␣file␣" ≪ filename ≪ std::endl;
    }
    out_tif.write(scratch_area, 3);
    for (unsigned int counter ← 3; counter < page_length; ++counter) {
      out_tif.put(in.get());
    }
    out_tif.close();
  }
  else {
    std::cout ≪ "Not␣a␣TIF␣file,␣skipping." ≪ std::endl;
    if (page_length ≥ 3) in.seekg(page_length − 3, std::ifstream::cur);
  }
```

This code is used in section 30.

33. After a PAGE TIF file, there will sometimes be addenda. It seems that it will either take the form: "`0000 0000 0000 0000`," meaning no addenda, or "`xxxx xxxx` (addendum length x) `yyyy yyyy` (addendum length y) `0000 0000`." Either way, I just want to skip it.

```
⟨ Skip past any addenda 33 ⟩ ≡
  unsigned int addendum ← read_next_dword(in);
  unsigned int number_of_addenda ← 0;
  while (addendum > 0) {
    ++number_of_addenda;
    std::cout ≪ "Addendum␣of␣length␣" ≪ dechex(addendum) ≪ '␣' ≪ in ≪ std::endl;
    in.seekg(addendum, ifstream::cur);
    if (¬in ∨ in.eof()) {
      std::cout ≪ "XXX␣Problem␣moving␣past␣addenda..." ≪ std::endl;
      return −1;
    }
    std::cout ≪ "After␣addendum␣" ≪ in ≪ std::endl;
    addendum ← read_next_dword(in);
  }
  if (number_of_addenda ≡ 0) {
    addendum ← read_next_dword(in);
    while (addendum ≠ 0) {
      std::cout ≪ "XXX␣There␣was␣no␣official␣addendum,␣yet␣the␣next␣dword␣wasn't␣0␣but␣" ≪
          dechex(addendum) ≪ '␣' ≪ in ≪ std::endl;
      std::cout ≪ "XXX␣skipping␣that␣many␣bytes␣and␣hoping␣for␣the␣best...␣" ≪ std::endl;
      in.seekg(addendum, ifstream::cur);
      if (¬in ∨ in.eof()) {
        std::cout ≪ "XXX␣Problem␣moving␣past␣addenda..." ≪ std::endl;
        return −1;
      }
      addendum ← read_next_dword(in);
    }
  }
```

This code is used in section 30.

34. The Patient Index. When I'm done printing out TIF files, I will also have all the patient data I'm going to have. I cycle through it to make an index by social security number. It may help to search for a hard-to-find patient this way.

```
⟨ Write out the index 34 ⟩ ≡
  ofstream index("index.txt", ofstream::app);
  for (patient_info::iterator it ← briefcase_patients.begin(); it ≠ briefcase_patients.end(); ++it) {
    index ≪ it→second;
  }
  index.close();
```

This code is used in section 2.

35. Below I define the format of the **patient_data** when written to the index. It's just the SSN followed by the directory where the patient files live. I don't need to also print their name, since that is embedded in the directory name.

```
⟨ Helper Classes 10 ⟩ +≡
  ostream &operator≪(ostream &os, const patient_data &pd)
  {
    if (pd.social_security_num.length() ≡ 9) {
      os ≪ pd.social_security_num[0];
      os ≪ pd.social_security_num[1];
      os ≪ pd.social_security_num[2];
      os ≪ '-';
      os ≪ pd.social_security_num[3];
      os ≪ pd.social_security_num[4];
      os ≪ '-';
      os ≪ pd.social_security_num[5];
      os ≪ pd.social_security_num[6];
      os ≪ pd.social_security_num[7];
      os ≪ pd.social_security_num[8];
    }
    else if (pd.social_security_num.length() ≠ 0) {
      os ≪ pd.social_security_num;
    }
    os ≪ "␣␣" ≪ pd.directory_name ≪ std::endl;
    return os;
  }
```

36. Odds and ends. There are a few things left to fill out, to make the program complete:

37. I need to make directories so the files have somewhere to go.

```
⟨ Create directories to hold the files 37 ⟩ ≡
  const string &fulldir ← pdat.get_directory( );
  std::cout ≪ "Ensuring␣directory␣<" ≪ fulldir ≪ ">␣is␣available." ≪ std::endl;
  size_t curloc ← fulldir.find_first_of('/');
  while (curloc ≠ string::npos) {
    string partial ← fulldir.substr(0, curloc);
    std::cout ≪ "creating␣partial␣" ≪ partial ≪ std::endl;
    mkdir(partial.c_str( ), °755 );
    curloc ← fulldir.find_first_of('/', curloc + 1);
  }
  mkdir(fulldir.c_str( ), °755 );
```

This code is used in section 27.

38. When I get a second or third or fourth document for the same patient, I have a chance to grab their SSN if it was missing from the initial record. This will help to build a better index. I could also take the time to report on other discrepancies (such as if they spelled the name differently this time), but there's no way to automate those corrections, so I don't do that.

```
⟨ Try to recover a missing SSN 38 ⟩ ≡
  if ((social_security_num.length( ) ≡ 0) ∧ (*ssn ≠ '\0')) {
    social_security_num ← ssn;
  }
```

This code is used in section 24.

39. When I first encounter a new document record, I'll do some basic reporting for debug purposes.

```
⟨ Check and report on which document we found 39 ⟩ ≡
  std::cout ≪ "\n\n********␣new␣document\n";
  std::cout ≪ "Found␣document␣" ≪ dechex(doc_id) ≪ '␣' ≪ doc_name ≪ "␣belonging␣to␣" ≪
      doc_owner_name ≪ std::endl;
  if (doc_name.length( ) ≡ 0) {
    std::cout ≪ "XXX␣BAD␣DOCUMENT␣ID!" ≪ std::endl;
  }
  if (doc_owner_name.length( ) ≡ 0) {
    std::cout ≪ "XXX␣BAD␣DOCUMENT␣OWNER!" ≪ std::endl;
  }
```

This code is used in section 22.

40. Index.