1. Introduction. This is a file de-duplication program for ECE 551 assignment 097. The specification in `README` is cited below.

> A de-duplication program searches through a set of directories and removes files that are exactly identical to other files. However, to reduce the risk of catastrophe as you test this program, you are not actually going to remove the files, but instead, print a shell script, which would remove the files if you ran it.
> - It must take as command line arguments 1 or more directory names, find all regular files (recursively) contained within those directories, and find duplicates to remove.
> - As your program's output is a shell script, it should first print `#!/bin/bash`
> - Whenever your program finds a duplicated file, it must print lines of a shell script to standard output. First, it must print a comment indicating what file it is removing and what file it is a duplicate of, then it should print an `rm` command, which would remove the file. The program should print this output using the full pathnames of the files. For example:
>
> ```
> #Removing /home/ece551/test/file1 (duplicate of /home/ece551/test/dir1/file2).
>
> rm /home/ece551/test/file1
> ```
>
> Your script should have exactly the same format.
> - Of the identical files, you may remove any of the copies, as long as you leave one of them.
> - Your program must work correctly in the presence of multiple copies of the same file (so if `file1`, `file2`, `file3`, and `file4` are the same, your program must delete three of them and leave one; however, it is considered correct regardless of which one you leave).
> - Your program should ignore symlinks, as they do not actually represent another copy of the data.
> - Your program may **not** perform an N^2 comparison of all of the files (checking every file's contents against every other file's contents). Instead, you must use a hashtable, in which you hash the contents of the files, and only compare files that have the same hash. You can expect that we may test on a large number of large-ish files and will require your program to complete in a reasonable time.
> - You may assume that the contents of the directories/files will not change while your program runs (unless you change them).
> - Provide a `Makefile` that compiles your code into a program called dedup

2. This program is written in C++. The overall structure is very simple. Interestingly there're no external functions defined.

⟨ headers 3 ⟩
⟨ class definitions 5 ⟩
⟨ main program 40 ⟩

3. Some standard headers are included here, as they are the most commonly used.

⟨ headers 3 ⟩ ≡
```
#include <iostream>
#include <string>
```
See also sections 4, 20, and 31.

This code is used in section 2.

4. Directories. C++ has file streams, but it does not have the concept of directories (or not until C++17). POSIX has **DIR** ∗, which looks like a friend of **FILE** ∗.

```
⟨ headers 3 ⟩ +≡
#include <dirent.h>      /∗ DIR ∗ and related functions are declared here ∗/
```

5. But I would like to handle directories in an object-oriented manner. Thus I define a class to represent a directory.

```
⟨ class definitions 5 ⟩ ≡
  class Directory {
  private: ⟨ private fields in Directory 13 ⟩
  protected: ⟨ protected fields in Directory 6 ⟩
  public: ⟨ public fields in Directory 7 ⟩
  };
```

See also sections 16, 25, and 38.

This code is used in section 2.

6. Of course, a **Directory** holds a **DIR** ∗ under the hood.

```
⟨ protected fields in Directory 6 ⟩ ≡
  DIR ∗dir;
```

See also section 14.

This code is used in section 5.

7. The default constructor sets *dir* to NULL, i.e., no directory has been opened.

```
⟨ public fields in Directory 7 ⟩ ≡
  Directory( )
  : dir(Λ) { }
```

See also sections 8, 9, 10, 11, 12, and 15.

This code is used in section 5.

8. Use *open* to open a directory.

```
⟨ public fields in Directory 7 ⟩ +≡
  void open(const char ∗dirname)
  {
    if (is_open( )) {
      close( );
    }
    dir = opendir(dirname);
  }
```

9. Accordingly, use *close* to close a directory.

```
⟨ public fields in Directory 7 ⟩ +≡
  void close( )
  {
    closedir(dir);
    dir = Λ;
  }
```

10. To test if a directory is open, use *is_open*. It simply tests if *dir* is NULL, because *opendir* returns NULL on failure, and I manually set *dir* to NULL when a directory is closed.

```
⟨ public fields in Directory 7 ⟩ +≡
  bool is_open( ) const
  {
    return dir ≠ Λ;
  }
```

11. It's handy to have a constructor that automatically calls *open*.

```
⟨ public fields in Directory 7 ⟩ +≡
  Directory(const char *dirname)
  : dir(Λ) {
    open(dirname);
  }
```

12. The advantage of using an directory object is that no resource will be leaked even in strange situations.

```
⟨ public fields in Directory 7 ⟩ +≡
  ~Directory( )
  {
    if (is_open( )) {
      close( );
    }
  }
```

13. According to the rule of three, I also need a copy constructor and an assignment operator. But I cannot think of any proper way to copy a **DIR** *. After looking into the implementation of **std**::**ios_base** (which is in a similar situation as it does not make much sense to copy a stream), I learned a trick to disallow copy-construction and assignment.

```
⟨ private fields in Directory 13 ⟩ ≡
  Directory(const Directory &);
  Directory &operator=(const Directory &);
```

This code is used in section 5.

14. POSIX offers *readdir_r* to read one entry in the directory. It writes the entry into a **struct dirent**.

```
⟨ protected fields in Directory 6 ⟩ +≡
  struct dirent entry;
```

15. Wrap the call to *readdir_r* in a method.

I can keep on implementing other methods, such as rewind, seek and tell, but I don't think I'll need them in this program. So stop here.

⟨ public fields in **Directory** 7 ⟩ +≡

```
char *read( )
{
  struct dirent *res;
  if (dir ≡ Λ) {
    return Λ;
  }
  readdir_r(dir, &entry, &res);
  if (res ≡ Λ) {
    return Λ;
  }
  return entry.d_name;
}
```

16. File tree walker. There is a library function named `ftw` that (as its name suggests) does exactly the job. With a global hashtable to detect duplication, this assignment is done. This sounds too easy, and makes my just-written **Directory** class useless. Also, global variables are bad.

So, write a directory walker on my own.

```
⟨ class definitions 5 ⟩ +≡
  class FileTreeWalker {
  protected: ⟨ protected fields in FileTreeWalker 18 ⟩
  public: ⟨ public fields in FileTreeWalker 19 ⟩
  };
```

17. The main thing the file tree walker would do is to recursively access all files. I will need to maintain the "current path" during the process.

On Linux, there is a constant `PATH_MAX`—no path will have more bytes than that. I can store an **char**[`PATH_MAX`] in my class, but why arbitrary limits? C++ has **std**::**string** so there's really no space limitations.

18.
```
⟨ protected fields in FileTreeWalker 18 ⟩ ≡
  std::string path;
```
See also sections 21 and 24.

This code is used in section 16.

19. *walk* will start the tree walking starting from *root*.

```
⟨ public fields in FileTreeWalker 19 ⟩ ≡
  int walk(const char *root)
  {
    path.assign(root);
    return walk_internal( );
  }
```
This code is used in section 16.

20. Different actions are taken depending on the file type. I'll need to call *lstat* to get the file type.

```
⟨ headers 3 ⟩ +≡
#include <sys/stat.h>      /* for struct stat and related functions */
```

21. The actual walking process happens in *walk_internal*. The reason to use another layer of function call is that I maintain the current path in the field named *path*, instead of creating temporary strings and pass them to *walk*.

```
⟨ protected fields in FileTreeWalker 18 ⟩ +≡
  int walk_internal( )
  {
    struct stat st;
    int rc = 0;
    lstat(path.c_str( ), &st);
    if (S_ISREG(st.st_mode)) {
      rc = work(path);
    }
    else if (S_ISDIR(st.st_mode)) {
      ⟨ walk on children in the directory 22 ⟩
    }     /* don't handle other fancy file types */
    return rc;
  }
```

22. For directories, I need to go through all its children.

```
⟨ walk on children in the directory 22 ⟩ ≡
  Directory d(path.c_str());
  char *p;
  size_t oldlen = path.size();
  if (path[oldlen − 1] ≠ '/') {
    path.push_back('/');      /* path separator */
    oldlen++;
  }
  while ((p = d.read()) ≠ Λ) {
    ⟨ continue if p is "." or ".." 23 ⟩
    path.append(p);      /* child name */
    rc = walk_internal();
    if (rc ≠ 0) {
      break;
    }
    path.resize(oldlen);
  }
```

This code is used in section 21.

23. *readdir_r* returns `"."` and `".."` as children of the directory, but don't recursively walk on them, for an obvious reason.

```
⟨ continue if p is "." or ".." 23 ⟩ ≡
  if (p[0] ≡ '.' ∧ (p[1] ≡ '\0' ∨ (p[1] ≡ '.' ∧ p[2] ≡ '\0'))) {
    continue;
  }
```

This code is used in section 22.

24. Another method called *work* actually work on the files. For debugging purpose, I chose to print out the file path.

```
⟨ protected fields in FileTreeWalker 18 ⟩ +≡
  virtual int work(const std::string &path)
  {
    std::cerr ≪ path ≪ std::endl;
    return 0;
  }
```

25. Set of unique files. This is a specialized version of a hash table. Only file path and hash value is stored. When comparing for equalty, the content of the file will be re-read on demand.

```
⟨ class definitions 5 ⟩ +≡
  class FileSet {
  private: ⟨ private fields in FileSet 26 ⟩
  public: ⟨ public fields in FileSet 30 ⟩
  };
```

26. To store files in the hash table, I need a hash function and an equal function.

```
⟨ private fields in FileSet 26 ⟩ ≡
  static size_t hash(std::istream &);
  static bool equal(std::istream &, std::istream &);
```

See also section 29.

This code is used in section 25.

27.

```
size_t FileSet::hash(std::istream &f)
{
  size_t h = 6549;
  while (¬f.eof()) {
    h = h * 1558 + ((unsigned char) f.get() ⊕ 233);
  }
  return h;
}
```

28.

```
bool FileSet::equal(std::istream &f1, std::istream &f2)
{
  while (¬f1.eof() ∧ ¬f2.eof()) {
    if (f1.get() ≠ f2.get()) {
      return false;
    }
  }
  return f1.eof() ∧ f2.eof();
}
```

29. The hash table is open addressing. I have an array of hash slots. Each slot contains the file path and the hash value of the file content.

```
⟨ private fields in FileSet 26 ⟩ +≡
  struct Slot {
    std::string path;
    size_t hash;
  };
  Slot *hvec;      /* array of slots */
  size_t hsiz;     /* size of the above array */
  size_t hcnt;     /* entries in the hash table */
```

30. The array of slots is dynamically allocated, because I do not know in advance how many files I will be processing.

```
#define HASH_INISIZ 64
⟨ public fields in FileSet 30 ⟩ ≡
  FileSet( )
  : hvec(new Slot[HASH_INISIZ]( )), hsiz(HASH_INISIZ), hcnt(0) { }
  ~FileSet( )
  {
    delete[ ] hvec;
  }
```

See also section 32.

This code is used in section 25.

31. ⟨ headers 3 ⟩ +≡

```
#include <fstream>
```

32. In fact, **FileSet** has only one operation, *add*. It tries to add a file to the set, but also has an interesting property: if the file is not a duplicate, *path* will be returned; otherwise it returns the stored path of an existing file, of which the to-be-added file is a duplicate.

```
⟨ public fields in FileSet 30 ⟩ +≡
  const std::string &add(const std::string &path)
  {
    Slot *p, *pos, *end;
    std::ifstream f(path.c_str( ));
    size_t h = hash(f);
    p = pos = &hvec[h % hsiz];
    end = &hvec[hsiz];
    while (¬p→path.empty( )) {
      ⟨ if found a duplicate, return the stored path 33 ⟩
      p++;      /* linear probing */
      if (p ≡ end) {
        p = hvec;      /* wrap around */
      }
    }
    ⟨ add path to hash table 35 ⟩
    return path;
  }
```

33. ⟨ if found a duplicate, return the stored path 33 ⟩ ≡

```
  if (p→hash ≡ h) {      /* very likely two files are identical */
    ⟨ return path if the two files have the same path 34 ⟩
    std::ifstream fs(p→path.c_str( ));
    f.clear( );
    f.seekg(0);
    if (equal(fs, f)) {      /* yes, they are indeed identical */
      return p→path;
    }
  }
```

This code is used in section 32.

34. I don't know why I would see the same path twice. Maybe the user gave some arguments that are not disjoint?

```
⟨ return path if the two files have the same path 34 ⟩ ≡
  if (p→path ≡ path) {
    return path;
  }
```

This code is used in section 33.

35.

```
⟨ add path to hash table 35 ⟩ ≡
  p→path = path;
  p→hash = h;
  hcnt++;
  if (hcnt > hsiz/2) {
    ⟨ rehash 36 ⟩
  }
```

This code is used in section 32.

36. When the load factor is greater than 0.5, a rehashing takes place and the size of the array is doubled. There's no need to call *hash* during rehashing since the hash values are stored.

```
⟨ rehash 36 ⟩ ≡
  Slot *nvec = new Slot[hsiz * 2]( );
  size_t nsiz = hsiz * 2;
  for (size_t i = 0; i < hcnt; i++) {
    Slot *p = &hvec[i];
    Slot *q = &nvec[p→hash % nsiz];
    Slot *end = &nvec[nsiz];
    while (¬q→path.empty( )) {
      q++;
      if (q ≡ end) {
        q = nvec;
      }
    }
    *q = *p;
  }
  delete[ ] hvec;
  hvec = nvec;
  hsiz = nsiz;
```

This code is used in section 35.

37. The main program. It's easy to see how things will work now: Override **FileTreeWalker** :: *work*, and in each invocation of *work*, call **FileSet** :: *add* to add that file into the set (or be told it is a duplicate).

38. ⟨ class definitions 5 ⟩ +≡

```
class Dedup : FileTreeWalker {
protected:
  FileSet fset;
  virtual int work(const std::string &);
public:
  void run(const char *path)
  {
    walk(path);
  }
};
```

39.

```
int Dedup::work(const std::string &path)
{
  const std::string &stored_path = fset.add(path);
  if (&stored_path ≠ &path) {
      /* comparing addresses is OK, due to the implementation of FileSet::add */
    std::cout ≪ "#Removing␣" ≪ path ≪ "␣(duplicate␣of␣" ≪ stored_path ≪ ").\n\nrm␣" ≪
        path ≪ "\n\n";
  }
  return 0;
}
```

40. ⟨ main program 40 ⟩ ≡

```
int main(int argc, char *argv[])
{
  if (argc < 2) {
    std::cerr ≪ "I␣need␣at␣least␣one␣argument,␣sorry.\n";
    return 1;
  }
  std::cout ≪ "#!/bin/bash\n";
  Dedup dd;
  for (int i = 1; i < argc; i++) {
    dd.run(argv[i]);
  }
  return 0;
}
```

This code is used in section 2.

41. Index.

add: 32, 37, 39.
append: 22.
argc: 40.
argv: 40.
assign: 19.
c_str: 21, 22, 32, 33.
cerr: 24, 40.
clear: 33.
close: 8, 9, 12.
closedir: 9.
cout: 39, 40.
d: 22.
d_name: 15.
dd: 40.
Dedup: 38, 39, 40.
DIR: 4, 6, 13.
dir: 6, 7, 8, 9, 10, 11, 15.
Directory: 5, 6, 7, 11, 12, 13, 16, 22.
dirent: 14, 15.
dirname: 8, 11.
empty: 32, 36.
end: 32, 36.
endl: 24.
entry: 14, 15.
eof: 27, 28.
equal: 26, 28, 33.
f: 27, 32.
false: 28.
FileSet: 25, 27, 28, 30, 32, 37, 38, 39.
FileTreeWalker: 16, 37, 38.
fs: 33.
fset: 38, 39.
f1: 28.
f2: 28.
get: 27, 28.
h: 27, 32.
hash: 26, 27, 29, 32, 33, 35, 36.
`HASH_INISIZ`: 30.
hcnt: 29, 30, 35, 36.
hsiz: 29, 30, 32, 35, 36.
hvec: 29, 30, 32, 36.
i: 36, 40.
ifstream: 32, 33.
ios_base: 13.
is_open: 8, 10, 12.
istream: 26, 27, 28.
lstat: 20, 21.
main: 40.
nsiz: 36.
nvec: 36.
oldlen: 22.
open: 8, 11.
opendir: 8, 10.
p: 22, 32, 36.
path: 18, 19, 21, 22, 24, 29, 32, 33, 34, 35, 36, 38, 39.
`PATH_MAX`: 17.
pos: 32.
push_back: 22.
q: 36.
rc: 21, 22.
read: 15, 22.
readdir_r: 14, 15, 23.
res: 15.
resize: 22.
root: 19.
run: 38, 40.
`S_ISDIR`: 21.
`S_ISREG`: 21.
seekg: 33.
size: 22.
Slot: 29, 30, 32, 36.
st: 21.
st_mode: 21.
stat: 20, 21.
std: 13, 17, 18, 24, 26, 27, 28, 29, 32, 33, 38, 39, 40.
stored_path: 39.
string: 17, 18, 24, 29, 32, 38, 39.
walk: 19, 21, 38.
walk_internal: 19, 21, 22.
work: 21, 24, 37, 38, 39.

⟨ add *path* to hash table 35 ⟩ Used in section 32.
⟨ class definitions 5, 16, 25, 38 ⟩ Used in section 2.
⟨ headers 3, 4, 20, 31 ⟩ Used in section 2.
⟨ if found a duplicate, return the stored path 33 ⟩ Used in section 32.
⟨ main program 40 ⟩ Used in section 2.
⟨ private fields in **Directory** 13 ⟩ Used in section 5.
⟨ private fields in **FileSet** 26, 29 ⟩ Used in section 25.
⟨ protected fields in **Directory** 6, 14 ⟩ Used in section 5.
⟨ protected fields in **FileTreeWalker** 18, 21, 24 ⟩ Used in section 16.
⟨ public fields in **Directory** 7, 8, 9, 10, 11, 12, 15 ⟩ Used in section 5.
⟨ public fields in **FileSet** 30, 32 ⟩ Used in section 25.
⟨ public fields in **FileTreeWalker** 19 ⟩ Used in section 16.
⟨ rehash 36 ⟩ Used in section 35.
⟨ walk on children in the directory 22 ⟩ Used in section 21.
⟨ **continue** if p is `"."` or `".."` 23 ⟩ Used in section 22.
⟨ **return** *path* if the two files have the same path 34 ⟩ Used in section 33.

DEDUP

	Section	Page
Introduction	1	1
Directories	4	2
File tree walker	16	5
Set of unique files	25	7
The main program	37	10
Index	41	11