Topic 7 – File I/O

Introduction

The Problem

- Virtually every program that we've written and looked at this semester has worked roughly the same way:

  - Data is read in from keyboard.
  - The program does something with the data.
  - The results are displayed.

- This approach allows us to usefully solve many problems but it has one big limitation.
  - Each time the program runs it "starts from scratch" and has no record of what happened when it ran before.

- For many simple problems this is not an issue.
- But often it is – particularly when there is a lot of data that needs to be processed.

- Consider a program that manages the ordering of stock for a large department store.
- If the program has no record of what happened the last time it was run then it will be virtually useless.
    - For example, every time the program is run someone will need to input into it all the details concerning the stock in the store.

- This might include information about each item, current stock available, price information and where the item can be re-ordered from.
- If the store has 1,000 stock items and information about each item takes 10 seconds to be re-entered then, every time the program stops running it will take nearly three hours of constant typing before it can be re-started.

- This by itself is clearly unworkable.

- But the problem becomes much worse when you consider that if the purpose of the program is to track stock then the program itself may be the only “reliable” and up-to-date source of information about the store's stock in the first place!

- If the program crashes or there is a power failure then the amount of data lost will be enormous.

- Clearly there is a need for programs to be able to maintain data, even when they are not executing.

Files and File Systems

- This is done by writing the data to some form of "secondary storage" such as a disk.

- The exact way that data is stored on the disk is determined by the operating system and is referred to as the "file system".

- However, virtually all file systems work in a similar way whereby data relating to a particular thing is stored together in something called a *file*.

- Files themselves are stored within logical locations on the disk called *directories*.
  - The analogy is often drawn between directories and physical folders which store papers so the term *folder* is often used.

- Directories themselves are usually arranged in a hierarchical tree structure, i.e., one within another.

- Files and directories are both given names, however, to uniquely refer to a specific file or directory it's location relative to the directories it is in must be given.
  - This is known as the *path* to that file or directory.

Records within Files

- Although the format of the data within files does differ (see below), the contents of a file can still be viewed in the same general way.

- Since files contain data about the same thing, there is always some general similarity between the different parts that make up a file.
- Because of this files can be thought of as being made up of a number of *records.*
- Each record is of the same general structure and records are usually of the same size (although not always).

- For example, a file might contain data about the stock stored in a warehouse.
- Each record in the file contains information about a particular stock item.
- So if there are 100 different types of stock items, the file will contain 100 records.

- The data which makes up each record can be broken down into fields, for example there might be a field for the name of the stock item and another for the number of that item currently in stock.
- This is very similar to a `struct` in C and, in fact, the records in a file can be thought of as an array of `struct`s stored on the disk.

- Now although files can be described as a collection of records, the structure of those records can differ enormously from one file to another.
- These differing structures are known as the file's *format*.

File Formats and Text Files

- The format of data within the file itself is completely the responsibility of the programs that will read from and write to the file.

- Most programs have their own format for encoding and storing data which often only they understand.

- However, there is a generic standard format whereby the data within the file is essentially not encoded at all.
- Instead the data is simply stored as lines of text characters, usually just in plain ASCII.

- These files are commonly referred to as *text files* and are the format we will be working with in this unit.

Working With Text Files

- The records in text files are made up of lines which represent a string of characters followed by a newline character.
  - In C this character is represented by `'\n'`.

- Note that the length of the records (the length of the line) is not fixed for text files – it just keeps going until a newline is found.
  - However, programs often artificially limit it to “word wrap” the text when displaying it to the screen.

- To gain access to a file (including non-text files) the program must make a request via the operating system to *open* the file.
  - Most programming languages provide built-in routines for doing this in a simple and convenient way.

- The attempt at opening a file will either succeed or fail.
    - Failure may occur if, for example, the file does not exist or the OS denies access to it for security reasons etc.

- When opening the file the program must also specify what it wants to do with the file once open.
- Files may be opened for reading, writing or appending.

- *Reading* simply means that data can be sequentially read out of the file but no changes can be made to the file.
- Files can also be opened for *writing* so that data can be sequentially added to the file.
    - Opening a non-existent file for writing causes an empty file to be created into which data can be written.
    - If a file that already exists is open for writing then the contents of that file will be erased and data can then be written into this now empty file.
- Opening a file for *appending* allows the program to write data to the file but this time the existing data in the file is not erased and the new data is added to the end of the file (appended).

- As indicated, when dealing with text files data can only be read from or written to that file in **sequential order**.
- This means making minor edits and changes to a file can sometimes be awkward.

WORKING WITH FILES IN C

Overview

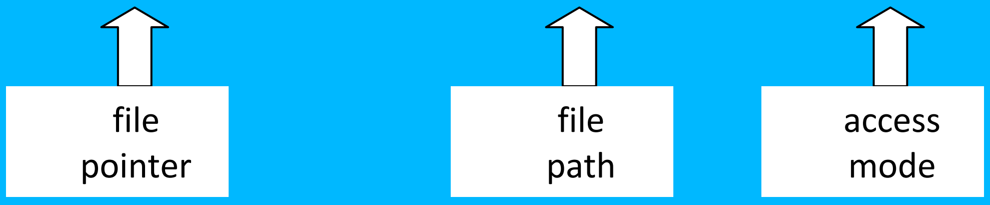
- In this unit we will be using the Standard C library routines for working with files.

- To start working with a file, it needs to be opened.
- This can be done using the `fopen()` function in the standard C library.
- To use this function, you need to provide the name (path) of the file to be opened, as well as the mode (read/write/append).
  - r = read
  - w = write
  - a = append

- The function will return what is known as a file pointer, which then allows you to access the file using other functions.

- For example:

```
srcFile = fopen ("myfile.txt", "r");
```



- Once the text file is open, you can use the `fgets()` we have already looked at to read strings from the file.

- When you're done with the file, it should be closed using `fclose()`

Here is a program that demonstrates how to read from a file:

```
#include <stdio.h>


int main(int argc, char *argv[])
{
   const int SIZE = 128;
   char line[SIZE];
   FILE *srcFile;

   if(argc == 1)
   {
        printf("No command line arguments
given!\n");
        return(0);
   }

   srcFile = fopen (argv[1], "r");

   while (fgets(line, SIZE, srcFile) != NULL)
   {
        printf("%s", line);
   }

   fclose(srcFile);

   return(0);
}
```

file pointer

Writing to Files

- To write to a file, you first need to open the file for writing (or appending).
- You can then use the `fprintf()` function to write to the file.
- This works exactly like `printf()`, except you tell it which file to write to instead of writing to a default “file” called the standard output (which usually refers to the screen).

- For example:

```
outputFile = fopen (“myfile.txt”, "w");

...

fprintf(outputFile, "%s", line);
```

Here is a program that demonstrates all of these concepts:

```
#include <stdio.h>

const int SIZE = 128;

int main(int argc, char *argv[])
{
  char line[SIZE];
  FILE *srcFile;
  FILE *destFile;

  if(argc == 1)
  {
      printf("No command line arguments 
given!");
      return(0);
  }

  srcFile = fopen (argv[1], "r");
  destFile = fopen (argv[2], "w");

  while (fgets(line, SIZE, srcFile) != NULL)
  {
      fprintf(destFile, "%s", line);
  }

  fclose(srcFile);
  fclose(destFile);

  return(0);
}
```

File Record Processing

Problems with Handling File Data

- One of the aspects of dealing with files is that it is quite possible for the amount of data in the file to be very large.
- This is particularly true when compared with the amount of data in terms of variables etc. that a program may declare and store in memory.

- For example, it is rare for any module in a program to declare more than say 10 different variables.
- If these occupy, on average, around four bytes then this is only 40 bytes of data!
- If the program has an array with 100 or so elements then this may grow to around 500 bytes.

- However, the sizes of files are generally measured in kilobytes (1024 bytes per kilobyte) and files of many megabytes (1024^2 bytes or 1024 kilobytes) are also very common.

- So there are a number of issues here with regards to storing data in memory that comes from a file:
    - Obviously it is not possible to declare a separate variable to hold each piece of data from a file so often an array is used.
    - The theoretical maximum file size that can be loaded into memory depends on how much memory is available – although this may be more than the amount of memory physically present (e.g., 256MB) due to so-called “virtual memory” systems.
    - However, some of this memory will already be used and for data from a file to be stored in memory by a program then the block of memory allocated to do this must be *contiguous* (all in a single block) and the amount of contiguous memory available may be relatively small.
    - Also operating systems may impose limitations on how much memory can be utilised by a single process (running program) and this is hard to know in advance.
    - There are other issues too since the amount of memory required to hold the file data may be larger that the actual size of the file if the data is compressed, e.g., image files.

Algorithmic Approaches

- All in all, dealing with the large amounts of data that can be held in a file can be quite tricky.

- There are two fundamental types of approach possible:

  - *Multiple Loaded Records*
  - *Single Record Batch Processing*

Multiple Loaded Records

- This involves loading multiple records from the file into memory.
  - Most of the time the entire file is loaded.

1. This will involve allocating sufficient memory to hold all of these records.
2. This will usually be done with an array.

- Because of the potential problems with allocating so much memory this is clearly not ideal.
- But for many problems it is the only option.
- For example, sorting the contents of a file generally requires that the entire file be held in memory.
- Manipulating compressed data such as that contained in image files also requires that data be completely loaded into memory.

- Another common example of a program that does this is a text editor like you have been using to write and edit your programs.

- The text editor works by opening a file for reading and then loading the entire contents into memory.
- The user then makes changes to the contents of this version of the file in memory – note that the original file on the disk is not touched at all during this process.
- Once the changes have been made they then "save" the file.
- This means the editor now re-opens the file on the disk, this time for "writing".
- The file is now truncated and all of its contents are deleted.
- The new version of the file which contains the changes the user made is now written back to the disk.

- Since the edits made by the user to the file can be to any part of the file and in any order and because the only ways of accessing the file are read, write and append, the only way of working with the file is to hold its entire contents in memory while editing it.

- It simply isn't possible to make edits directly to individual parts of the file so there is no alternative but to hold the entire file's data in memory all at once.

- Of course, some files are simply too big to be loaded into memory all at once.
  - For example, digital video is often many gigabytes in size.
- In this situation a specialised approach is required whereby the file is dealt with in separate portions or "chunks" at a time.

Single Record Batch Processing

- The alternative to loading multiple records is to only load as little of the file into memory at one time as possible.

- This typically involves reading only a single record in, processing it in memory and then reading in the next record.

- Because the records in the file are processed one after the other in order and typically no interaction with the user is required, this is often described as *batch processing*.

- When the next record is read then the previous record is discarded.
  - This means that the memory required to process the file is not cumulative.
  - Since the entire file is not being loaded into memory the memory required is of a fixed size, namely that for a single record.

- This sort of processing is clearly the more memory efficient of the two and also eliminates potential problems relating to limited available memory.
- So it is nearly always the best approach to use whenever this is possible.

SUMMARY

- Files are useful as there are many situations where a program needs to deal with a lot of data and this cannot be entered by the user from the keyboard each time the program is run.

- Files allow a program to store large amounts of data which remains on the disk after the program has finished executing.

- Files can be opened for reading, writing and appending only.
  - This means that apart from appending data to the end of the program, the only way of adding data to it is to rewrite the whole file.

- Algorithmically there are two basic ways of dealing with file data and these are:
  - to read the entire contents into memory all at once for processing.
  - to process only a single record at a time.