C-Programming Cont'd part 3

Overview

- structures in C provide another tool for grouping elements together
  - a powerful concept that you will use in many C programs that you develop

- suppose you want to store a date inside a program
  - we could create variables for month, day, and year to store the date

```
int  month = 9, day = 25, year = 2015;
```

- suppose your program also needs to store the date of purchase of a particular item
  - you must keep track of three separate variables for each date that you use in the program
  - these variables are logically related and should be grouped together

- it would be much better if you could somehow group these sets of three variables together
  - this is precisely what the structure in C allows you to do

Creating a structure

- a structure declaration describes how a structure is put together
  - what elements are inside the structure

- the struct keyword enables you to define a collection of variables of various types called a structure that you can treat as a single unit

```
struct  date
{
   int  month;
   int  day;
   int  year;
};
```

- the above statement defines what a date structure looks like to the C compiler
  - there is no memory allocation for this declaration

- the variable names within the date structure, month, day, and year, are called members or fields
  - members of the structure appear between the braces that follow the struct tag name date

Using a structure

- the definition of date defines a new type in the language
  - variables can now be declared to be of type struct date

```
struct date  today;
```

- you can now declare more variables of type struct date

```
struct date  purchaseDate;
```

- the above statement declares a variable to be of type struct date
  - memory is now allocated for the variables above
  - memory is allocated for three integer values for each variable

- be certain you understand the difference between defining a structure and declaring variables of the particular structure type

Accessing members in a struct

- now that you know how to define a structure and declare structure variables, you need to be able to refer to the members of a structure

- a structure variable name is not a pointer
  - you need a special syntax to access the members

- you refer to a member of a structure by writing the variable name followed by a period, followed by the member variable name
  - the period between the structure variable name and the member name is called the member selection operator
  - there are no spaces permitted between the variable name, the period, and the member name

- to set the value of the day in the variable today to 25, you write

```
today.day = 25;
today.year = 2015;
```

- to test the value of month to see if it is equal to 12

```
if  ( today.month == 12 )
   nextMonth = 1;
```

Example

```
struct  date
{
       int  month;
       int  day;
       int  year;
};

struct date  today;

today.month = 9;
today.day = 25;
today.year = 2015;

printf ("Today's date is %i/%i/%.2i.\n", today.month, today.day, today.year % 100);
```

Defining the structure and variable at the same time

- you do have some flexibility in defining a structure
  - it is valid to declare a variable to be of a particular structure type at the same time that the structure is defined
  - include the variable name (or names) before the terminating semicolon of the structure definition
  - you can also assign initial values to the variables in the normal fashion

```
struct  date
{
      int  month;
      int  day;
      int  year;
} today;
```

- in the above, an instance of the structure, called today, is declared at the same time that the structure is defined
  - today is a variable of type date

Un-named Structures

- you also do not have to give a structure a tag name
  - If all of the variables of a particular structure type are defined when the structure is defined, the structure name can be omitted

```
struct
{                          // Structure declaration and...
 int  day;
 int  year;
 int  month;
} today;                   // ...structure variable declaration combined
```

- a disadvantage of the above is that you can no longer define further instances of the structure in another statement
  - all the variables of this structure type that you want in your program must be defined in the one statement

---

Initializing Structures

- initializing structures is similar to initializing arrays
  - the elements are listed inside a pair of braces, with each element separated by a comma
  - the initial values listed inside the curly braces must be constant expressions

```
struct date  today = { 7, 2, 2015 };
```

- just like an array initialization, fewer values might be listed than are contained in the structure

```
struct date  date1 = { 12, 10 };
```

- sets date1.month to 12 and date1.day to 10 but gives no initial value to date.year

---

Initializing structures

- you can also specify the member names in the initialization list
  - enables you to initialize the members in any order, or to only initialize specified members

.member = value

struct date date1 = { .month = 12, .day = 10 };

- set just the year member of the date structure variable today to 2015

struct date today = { .year = 2015 };

Assignment with compound literals

- you can assign one or more values to a structure in a single statement using what is known as compound literals

today = (struct date) { 9, 25, 2015 };

- this statement can appear anywhere in the program
  - it is not a declaration statement
  - the type cast operator is used to tell the compiler the type of the expression
  - the list of values follows the cast and are to be assigned to the members of the structure, in order
  - listed in the same way as if you were initializing a structure variable

- you can also specify values using the .member notation

today = (struct date) { .month = 9, .day = 25, .year = 2015 };

- the advantage of using this approach is that the arguments can appear in any order

Arrays of structures

- you have seen how useful a structure is in enabling you to logically group related elements together
  - for example, it is only necessary to keep track of one variable, instead of three, for each date that is used by the program
  - to handle 10 different dates in a program, you only have to keep track of 10 different variables, instead of 30

- a better method for handling the 10 different dates involves the combination of two powerful features of the C programming language
  - structures and arrays.
  - it is perfectly valid to define an array of structures
  - the concept of an array of structures is a very powerful and important one in C

- declaring an array of structures is like declaring any other kind of array

struct date myDates[10];

- defines an array called myDates, which consists of 10 elements
  - each element inside the array is defined to be of type struct date

Initializing an array of structures

• initialization of arrays containing structures is similar to initialization of multidimensional arrays

```
struct date  myDates[5] = {  {12, 10, 1975},  {12, 30, 1980},  {11, 15, 2005} };
```

• sets the first three dates in the array myDate to 12/10/1975, 12/30/1980, and 11/15/2005

• the inner pairs of braces are optional

```
struct date  myDates[5] = {  12, 10, 1975,  12, 30, 1980,  11, 15, 2005 };
```

• initializes just the third element of the array to the specified value

```
struct date myDates[5] = { [2] = {12, 10, 1975} };
```

• sets just the month and day of the second element of the myDates array to 12 and 30

Initializing an array of structures

• initialization of arrays containing structures is similar to initialization of multidimensional arrays

```
struct date  myDates[5] = {  {12, 10, 1975},  {12, 30, 1980},  {11, 15, 2005} };
```

• sets the first three dates in the array myDate to 12/10/1975, 12/30/1980, and 11/15/2005

• the inner pairs of braces are optional

```
struct date  myDates[5] = {  12, 10, 1975,  12, 30, 1980,  11, 15, 2005 };
```

• initializes just the third element of the array to the specified value

```
struct date myDates[5] = { [2] = {12, 10, 1975} };
```

• sets just the month and day of the second element of the myDates array to 12 and 30

```
struct date myDates[5] = { [1].month = 12, [1].day = 30 };
```

Structures containing arrays

- it is also possible to define structures that contain arrays as members
  - most common use is to set up an array of characters inside a structure

- suppose you want to define a structure called month that contains as its members the number of days in the month as well as a three-character abbreviation for the month name

```
struct  month
{
   int    numberOfDays;
   char   name[3];
};
```

- this sets up a month structure that contains an integer member called numberOfDays and a character member called name
  - member name is actually an array of three characters

Structures containing arrays

- you can now define a variable to be of type struct month and set the proper fields inside aMonth for January

```
struct month  aMonth;
aMonth.numberOfDays = 31;
aMonth.name[0] = 'J';
aMonth.name[1] = 'a';
aMonth.name[2] = 'n';
```

- you can also initialize this variable to the same values

```
struct month  aMonth = { 31, { 'J', 'a', 'n' } };
```

- you can set up 12-month structures inside an array to represent each month of the year

```
struct month  months[12];
```

Nested Structures

- you want to have a convenient way to associate both the date and the time together
  - define a new structure, called, for example, dateAndTime, which contains as its members two elements
  - date and time

```
struct dateAndTime
{
   struct date   sdate;
   struct time   stime;
};
```

- the first member of this structure is of type struct date and is called sdate
- the second member of the dateAndTime structure is of type struct time and is called stime

- variables can now be defined to be of type struct dateAndTime

```
struct dateAndTime  event;
```

Accessing members in a nested structure

- to reference the date structure of the variable event, the syntax is the same as referencing any member

```
event.sdate
```

- to reference a particular member inside one of these structures, a period followed by the member name is tacked on the end
  - the below statement sets the month of the date structure contained within event to October, and adds one to the seconds contained within the time structure

```
event.sdate.month = 10;
++event.stime.seconds;
```

Accessing members in a nested structure

- the event variable can be initialized just like normal
  - sets the date in the variable event to February 1, 2015, and sets the time to 3:30:00.

```
struct dateAndTime  event = { { 2, 1, 2015 }, { 3, 30, 0 } };
```

- you can use members' names in the initialization

```
struct dateAndTime event =
        { { .month = 2, .day = 1, .year = 2015 },
          { .hour = 3, .minutes = 30, .seconds = 0 }
        };
```

An array of nested structures

- it is also possible to set up an array of dateAndTime structures

```
struct dateAndTime  events[100];
```

- the array events is declared to contain 100 elements of type struct dateAndTime
  - the fourth dateAndTime contained within the array is referenced in the usual way as events[3]

- to set the first time in the array to noon

```
events[0].stime.hour    = 12;
events[0].stime.minutes = 0;
events[0].stime.seconds = 0;
```

Declaring a structure within a structure

- you can define the Date structure within the time structure definition

```
struct Time
{
  struct Date
  {
    int day;
    int month;
    int year;
  } dob;

int hour;
int minutes;
int seconds;
};
```

- the declaration is enclosed within the scope of the Time structure definition
  - it does not exist outside it
  - it becomes impossible to declare a Date variable external to the Time structure

Structures and Pointers

- C allows for pointers to structures
- pointers to structures are easier to manipulate than structures themselves
- in some older implementations, a structure cannot be passed as an argument to a function, but a pointer to a structure can.
- even if you can pass a structure as an argument, passing a pointer is more efficient
- many data representations use structures containing pointers to other structures

Declaring a struct as a pointer

- you can define a variable to be a pointer to a struct

```
struct date  *datePtr;
```

- the variable datePtr can be assigned just like other pointers
  - you can set it to point to todaysDate with the assignment statement

```
datePtr = &todaysDate;
```

- you can then indirectly access any of the members of the date structure pointed to by datePtr

```
(*datePtr).day = 21;
```

- the above has the effect of setting the day of the date structure pointed to by datePtr to 21
  - parentheses are required because the structure member operator . has higher precedence than the indirection operator *

Using structs as pointers

- to test the value of month stored in the date structure pointed to by datePtr

```
if  ( (*datePtr).month == 12  )
        ...
```

- pointers to structures are so often used in C that a special operator exists
  - the structure pointer operator ->, which is the dash followed by the greater than sign, permits

```
(*x).y
```

to be more clearly expressed as

```
x->y
```

- the previous if statement can be conveniently written as

```
if  ( datePtr->month == 12 )
```

Example

```
struct date
{
        int  month;
        int  day;
        int  year;
};

struct date  today, *datePtr;

 datePtr = &today;

datePtr->month = 9;
datePtr->day = 25;
datePtr->year = 2015;

printf ("Today's date is %i/%i/%.2i.\n",  datePtr->month, datePtr->day, datePtr->year % 100);
```

Structures containing pointers

- a pointer also can be a member of a structure

```
struct  intPtrs
{
    int  *p1;
    int  *p2;
};
```

- a structure called intPtrs is defined to contain two integer pointers
  - the first one called p1
  - the second one p2

- you can define a variable of type struct intPtrs

```
struct intPtrs  pointers;
```

- the variable pointers can now be used just like other structs
  - pointers itself is not a pointer, but a structure variable that has two pointers as its members

Example

```
struct  intPtrs
{
    int  *p1;
    int  *p2;
};

struct intPtrs  pointers;
int  i1 = 100, i2;

pointers.p1 = &i1;
pointers.p2 = &i2;
*pointers.p2 = -97;

printf ("i1 = %i, *pointers.p1 = %i\n", i1, *pointers.p1);
printf ("i2 = %i, *pointers.p2 = %i\n", i2, *pointers.p2);
```

Character arrays or character pointers??

```
struct names {
    char first[20];
    char last[20];
};
```

OR

```
struct pnames {
    char * first;
    char * last;
};
```

- you can do both, however, you need to understand what is happening here

Character arrays or character pointers??

```
struct names veep = {"Talia", "Summers"};
struct pnames treas = {"Brad", "Fallingjaw"};
printf("%s and %s\n", veep.first, treas.first);
```

- the struct names variable veep
  - strings are stored inside the structure
  - structure has allocated a total of 40 bytes to hold the two names

- the struct pnames variable treas
  - strings are stored wherever the compiler stores string constants
  - the structure holds the two addresses, which takes a total of 16 bytes on our system
  - the struct pnames structure allocates no space to store strings
  - it can be used only with strings that have had space allocated for them elsewhere
    - such as string constants or strings in arrays

- the pointers in a pnames structure should be used only to manage strings that were created and allocated elsewhere in the program

Character arrays or character pointers??

- one instance in which it does make sense to use a pointer in a structure to handle a string is if you are dynamically allocating that memory
  - use a pointer to store the address
  - has the advantage that you can ask malloc() to allocate just the amount of space that is needed for a string

Example

```
struct namect {
    char * fname;  // using pointers instead of arrays
    char * lname;
    int letters;
};
```

- understand that the two strings are not stored in the structure
  - stored in the chunk of memory managed by malloc()
  - the addresses of the two strings are stored in the structure
  - addresses are what string-handling functions typically work with

Example

```
void getinfo (struct namect * pst)
{
    char temp[SLEN];
    printf("Please enter your first name.\n");
    s_gets(temp, SLEN);

    // allocate memory to hold name
    pst->fname = (char *) malloc(strlen(temp) + 1);

     // copy name to allocated memory
    strcpy(pst->fname, temp);
    printf("Please enter your last name.\n");
    s_gets(temp, SLEN);
    pst->lname = (char *) malloc(strlen(temp) + 1);
    strcpy(pst->lname, temp);
}
```

Structures as arguments to functions

• after declaring a structure named Family, how do we pass this structure as an argument to a function?

```
struct Family {
  char name[20];
  int age;
  char father[20];
  char mother[20];
};

bool siblings(struct Family member1, struct Family member2) {
  if(strcmp(member1.mother, member2.mother) == 0)
    return true;
  else
    return false;
}
```

Pointers to Structures as function arguments

• you should use a pointer to a structure as an argument
  • it can take quite a bit of time to copy large structures as arguments, as well as requiring whatever amount of memory to store the copy of the structure.
  • pointers to structures avoid the memory consumption and the copying time (only a copy of the pointer argument is made)

```
bool siblings(struct Family *pmember1, struct Family *pmember2)
{
  if(strcmp(pmember1->mother, pmember2->mother) == 0)
    return true;
  else
    return false;
}
```

• you can also use the const modifier to not allow any modification of the members of the struct (what the struct is pointing to)

```
bool siblings(Family const *pmember1, Family const *pmember2)
{
  if(strcmp(pmember1->mother, pmember2->mother) == 0)
    return true;
  else
    return false;
}
```

Pointers to Structures as function arguments

- you can also use the const modifier to not allow any modification of the pointers address
  - any attempt to change those structures will cause an error message during compilation

```
bool siblings(Family *const pmember1, Family *const pmember2)
{
  if(strcmp(pmember1->mother, pmember2->mother) == 0)
    return true;
  else
    return false;
}
```

- the indirection operator in each parameter definition is now in front of the const keyword
  - not in front of the parameter name
  - you cannot modify the addresses stored in the pointers
  - its the pointers that are protected here, not the structures to which they point

Example

```
struct funds {
    char   bank[FUNDLEN];
    double bankfund;
    char   save[FUNDLEN];
    double savefund;
};

double sum(struct funds moolah)
{
    return(moolah.bankfund + moolah.savefund);
}
```

```
int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

    printf("Stan has a total of $%.2f.\n",
sum(stan));

    return 0;
}
```

Reminder

- I mentioned earlier that you should always use pointers when passing structures to a function
  - it works on older as well as newer C implementations and that it is quick (you just pass a single address)

- however, you have less protection for your data
  - some operations in the called function could inadvertently affect data in the original structure
  - use const qualifier solves that problem

- advantages of passing structures as arguments
  - the function works with copies of the original data, which is safer than working with the original data
  - the programming style tends to be clearer

- main disadvantages to passing structures as arguments
  - older implementations might not handle the code
  - wastes time and space
  - especially wasteful to pass large structures to a function that uses only one or two members of the structure

- programmers use structure pointers as function arguments for reasons of efficiency and use const when necessary

- passing structures by value is most often done for structures that are small

Requirements

- write a C program that creates a structure pointer and passes it to a function
  - create a structure named item with the following members
    - itemName – pointer
    - quantity – int
    - price – float
    - amount – float (stores quantity * price)

  - create a function named readItem that takes a structure pointer of type item as a parameter
    - this function should read in (from the user) a product name, price, and quantity
    - the contents read in should be stored in the passed in structure to the function

  - create a function named print item that takes as a parameter a structure pointer of type item
    - function prints the contents of the parameter

  - the main function should declare an item and a pointer to the item
    - you will need to allocate memory for the itemName pointer
    - the item pointer should be passed into both the read and print item functions

---

Overview

- up until this point, all data that our program accesses is via memory
  - scope and variety of applications you can create is limited

- all serious business applications require more data than would fit into main memory
  - also depend on the ability to process data that is persistent and stored on an external device such as a disk drive

- C provides many functions in the header file stdio.h for writing to and reading from external devices
  - the external device you would use for storing and retrieving data is typically a disk drive
  - however, the library will work with virtually any external storage device

- with all the examples up to now, any data that the user enters is lost once the program ends
  - if the user wants to run the program with the same data, he or she must enter it again each time
  - very inconvenient and limits programming
  - referred to as volatile memory

Files

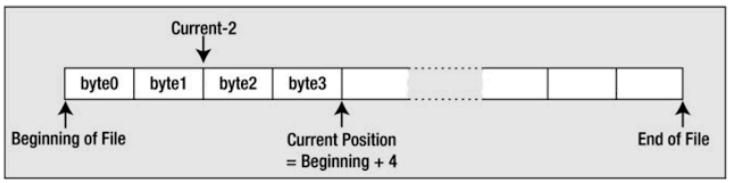
- programs need to store data on permanent storage
  - non-volatile
  - continues to be maintained after your computer is turned off

- a file can store non-volatile data and is usually stored on a disk or a solid-state device
  - a named section of storage
  - stdio.h is a file containing useful information

- C views a file as a continuous sequence of bytes
  - each byte can be read individually
  - corresponds to the file structure in the Unix environment

---

Files



Taken from Beginning C, Horton

- a file has a beginning and an end and a current position (defined as so many bytes from the beginning)
- the current position is where any file action (read/write) will take place
  - you can move the current position to any point in the file (even the end)

Text and binary files

- there are two ways of writing data to a stream that represents a file
  - text
  - binary
- text data is written as a sequence of characters organized as lines (each line ends with a newline)
- binary data is written as a series of bytes exactly as they appear in memory
  - image data, music encoding – not readable
- you can write any data you like to a file
  - once a file has been written, it just consists of a series of bytes
- you have to understand the format of the file in order to read it
  - a sequence of 12 bytes in a binary file could be 12 characters, 12 8-bit signed integers, 12 8-bit unsigned integers, etc.
  - in binary mode, each and every byte of the file is accessible

Streams

- C programs automatically open three files on your behalf
  - standard input - the normal input device for your system, usually your keyboard
  - standard output - usually your display screen
  - standard error - usually your display screen
- standard input is the file that is read by getchar() and scanf()
- standard output is used by putchar(), puts(), and printf()
  - redirection causes other files to be recognized as the standard input or standard output.
- the purpose of the standard error output file is to provide a logically distinct place to send error messages
- a stream is an abstract representation of any external source or destination for data
  - the keyboard, the command line on your display, and files on a disk are all examples of things you can work with as streams
  - the C library provides functions for reading and writing to or from data streams
    - you use the same input/output functions for reading and writing any external device that is mapped to a stream

Accessing Files

- files on disk have a name and the rules for naming files are determined by your operating system
  - You may have to adjust the names depending on what OS your program is running

- a program references a file through a file pointer (or stream pointer, since it works on more than a file)
  - you associate a file pointer with a file programmatically when the program is run
  - pointers can be reused to point to different files on different occasions

- a file pointer points to a struct of type FILE that represents a stream
  - contains information about the file
    - whether you want to read or write or update the file
    - the address of the buffer in memory to be used for data
    - a pointer to the current position in the file for the next operation
  - the above is all set via input/output file operations

- if you want to use several files simultaneously in a program, you need a separate file pointer for each file
  - there is a limit to the number of files you can have open at one time
    - defined as FOPEN_MAX in stdio.h

Opening a File

- you associate a specific external file name with an internal file pointer variable through a process referred to as opening a file
  - via the fopen() function
    - returns the file pointer for a specific external file

- the fopen() function is defined in stdio.h

```
FILE *fopen(const char * restrict name, const char * restrict mode);
```

- The first argument to the function is a pointer to a string that is the name of the external file you want to process
  - you can specify the name explicitly or use a char pointer that contains the address of the character string that defines the file name
  - you can obtain the file name through the command line, as input from the user, or defined as a constant in your program

- the second argument to the fopen() function is a character string that represents the file mode
  - specifies what you want to do with the file
  - a file mode specification is a character string between double quotes

- assuming the call to fopen() is successful, the function returns a pointer of type FILE* that you can use to reference the file in further input/output operations using other functions in the library

- if the file cannot be opened for some reason, fopen() returns NULL

File Modes (only apply to text files)

Mode	Description
"w"	Open a text file for *w*rite operations. If the file exists, its current contents are discarded.
"a"	Open a text file for *a*ppend operations. All writes are to the end of the file.
"r"	Open a text file for *r*ead operations.
"w+"	Open a text file for update (reading and writing), first truncating the file to zero length if it exists or creating the file it it does not exist
"a+"	Open a text file for update (reading and writing) appending to the end of the existing file, or creating the file if it does not yet exist
"r+"	Open a text file for update (for both reading and writing)

Write Mode

• If you want to write to an existing text file with the name myfile.txt

```
FILE *pfile = NULL;
char *filename = "myfile.text";
pfile = fopen(filename, "w");  // Open myfile.txt to write it
If(pfile != NULL)
  printf("Failed to open %s.\n", filename);
```

• opens the file and associates the file with the name myfile.txt with your file pointer pfile
  • the mode as "w" means you can only write to the file
  • you cannot read it

• If a file with the name myfile.txt does not exist, the call to fopen() will create a new file with this name

• if you only provide the file name without any path specification, the file is assumed to be in the current directory
  • you can also specify a string that is the full path and name for the file

• on opening a file for writing, the file length is truncated to zero and the position will be at the beginning of any existing data for the first operation
  • any data that was previously written to the file will be lost and overwritten by any write operations

Append Mode

•If you want to add to an existing text file rather than overwrite it
  •specify mode "a"
  •the append mode of operation.

•this positions the file at the end of any previously written data
  •If the file does not exist, a new file will be created

```
pFile = fopen("myfile.txt", "a");      // Open myfile.txt to add to it
```

•do not forget that you should test the return value for null each time

•when you open a file in append mode
  •all write operations will be at the end of the data in the file on each write operation
  •all write operations append data to the file and you cannot update the existing contents in this mode

Read Mode

•if you want to read a file
  •open it with mode argument as "r"
  •you can not write to this file

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

•this positions the file to the beginning of the data

•if you are going to read the file
  •it must already exist

•If you try to open a file for reading that does not exist, fopen() will return a file pointer of NULL

•you always want to check the value returned from fopen

Renaming a file

- renaming a file is very easy
  - use the rename() function

```
int rename(const char *oldname, const char *newname);
```

- the integer that is returned will be 0 if the name change was successful and nonzero otherwise

- the file must not be open when you call rename(), otherwise the operation will fail

```
if(rename( "C:\\temp\\myfile.txt", "C:\\temp\\myfile_copy.txt"))
  printf("Failed to rename file.");
else
  printf("File renamed successfully.");
```

- this will change the name of myfile.txt in the temp directory on drive C to myfile_copy.txt

- if the file path is incorrect or the file does not exist, the renaming operation will fail

Closing a file

- when you have finished with a file, you need to tell the operating system so that it can free up the file
  - you can do this by calling the fclose() function

- fclose() accepts a file pointer as an argument
  - returns EOF (int) if an error occurs
    - EOF is a special character called the end-of-file character
    - defined in stdio.h as a negative integer that is usually equivalent to the value –1
  - 0 if successful

```
fclose(pfile);                  // Close the file associated with pfile
pfile = NULL;
```

- the result of calling fclose() is that the connection between the pointer, pfile, and the physical file is broken
  - pfile can no longer be used to access the file

- if the file was being written, the current contents of the output buffer are written to the file to ensure that data is not lost

- It is good programming practice to close a file as soon as you have finished with it
  - protects against output data loss

- you must also close a file before attempting to rename it or remove it.

Deleting a file

- you can delete a file by invoking the remove() function
  - declared in stdio.h

```
remove("myfile.txt");
```

- will delete the file that has the name myfile.txt from the current directory

- the file cannot be open when you try to delete it

- you should always double check with operations that delete files
  - you could wreck your system if you do not

Reading characters from a text file

- the fgetc() function reads a character from a text file that has been opened for reading

- takes a file pointer as its only argument and returns the character read as type int

```
int mchar = fgetc(pfile);              // Reads a character into mchar with pfile a File pointer
```

- the mchar is type int because EOF will be returned if the end of the file has been reached

- the function getc(), which is equivalent to fgetc(), is also available
  - requires an argument of type FILE* and returns the character read as type int
  - virtually identical to fgetc()
  - only difference between them is that getc() may be implemented as a macro, whereas fgetc() is a function

- you can read the contents of a file again when necessary
  - the rewind() function positions the file that is specified by the file pointer argument at the beginning

```
rewind(pfile);
```

Example

```
#include <stdio.h>

int main () {
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");

   if(fp == NULL) {
      perror("Error in opening file");
      return(-1);
   }
```

```
// read a single char
while((c = fgetc(fp)) != EOF)
    printf("%c", c);


fclose(fp);
fp = NULL;
return(0);

}  // return from main
```

FileRead

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    //printf("hello world\n");
    FILE *fp;
    int c;

    fp=fopen("newfile.txt","r");
    //r -read
    //w - only write if file exists else create a new file then write
    //a - append to existing text
    if(fp!=NULL){
       while((c=fgetc(fp))!=EOF){
          printf("%c",c);
       }
    }
    else{
       perror("Error in opening file");
       return (-1);
    }
    return 0;
}
```

Reading a string from a text file

- you can use the fgets() function to read from any file or stream

char *fgets(char *str, int nchars, FILE *stream)

- the function reads a string into the memory area pointed to by str, from the file specified by stream
  - characters are read until either a '\n' is read or nchars-1 characters have been read from the stream, whichever occurs first
  - if a newline character is read, it's retained in the string
    - a '\0' character will be appended to the end of the string
  - if there is no error, fgets() returns the pointer, str
  - if there is an error, NULL is returned
  - reading EOF causes NULL to be returned

Example

```
#include <stdio.h>

int main () {
   FILE *fp;
   char str[60];

   /* opening file for reading */
   fp = fopen("file.txt" , "r");

   if(fp == NULL) {
      perror("Error opening file");
      return(-1);
   }
   if( fgets (str, 60, fp)!=NULL ) {
       /* writing content to stdout */
       printf("%s", str);
   }

   fclose(fp);
   fp = NULL;
   return(0);

} // end main
```

Reading formatted input from a file

- you can get formatted input from a file by using the standard fscanf() function

int fscanf(FILE *stream, const char *format, ...)

- the first argument to this function is the pointer to a FILE object that identifies the stream
- the second argument to this function is the format
  - a C string that contains one or more of the following items
    - whitespace character
    - non-whitespace character
    - format specifiers
    - usage is similar to scanf, but, from a file

Overview

```
#include <stdio.h>
#include <stdlib.h>

int main () {
  char str1[10], str2[10], str3[10];
  int year;
  FILE * fp;

  fp = fopen ("file.txt", "w+");
  if (fp != NULL)
    fputs("Hello how are you", fp);

  rewind(fp);

  fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);

  printf("Read String1 |%s|\n", str1 );
  printf("Read String2 |%s|\n", str2 );
  printf("Read String3 |%s|\n", str3 );
  printf("Read Integer |%d|\n", year );

  fclose(fp);

  return(0);
} // end of main
```

Writing characters to a text file

- the simplest write operation is provided by the function fputc()
  - writes a single character to a text file

```
int fputc(int ch, FILE *pfile);
```

- the function writes the character specified by the first argument to the file identified by the second argument (file pointer)
  - returns the character that was written if successful
  - Return EOF if failure

- In practice, characters are not usually written to a physical file one by one
  - extremely inefficient

- the putc() function is equivalent to fputc()
  - requires the same arguments and the return type is the same
  - difference between them is that putc() may be implemented in the standard library as a macro, whereas fputc() is a function

Example

```
#include <stdio.h>

int main () {
  FILE *fp;
  int ch;

  fp = fopen("file.txt", "w+");

  for( ch = 33 ; ch <= 100; ch++ ) {
    fputc(ch, fp);
  }

  fclose(fp);
  return(0);
}
```

Writing a string to a text file

- you can use the fputs() function to write to any file or stream

int fputs(const char * str, FILE * pfile);

- the first argument is a pointer to the character string that is to be written to the file
- the second argument is the file pointer
- this function will write characters from a string until it reaches a '\0' character
  - does not write the null terminator character to the file
    - can complicate reading back variable-length strings from a file that have been written by fputs()
    - expecting to write a line of text that has a newline character at the end

Example

```
#include <stdio.h>

int main () {
  FILE *filePointer;

  filePointer = fopen("file.txt", "w+");

  fputs("This is Jason Fedin Course.", filePointer);
  fputs("I am happy to be here", filePointer);

  fclose(filePointer);

  return(0);
}
```

Writing formatted output to a file

- the standard function for formatted output to a stream is fprintf()

int fprintf(FILE *stream, const char *format, ...)

- the first argument to this function is the pointer to a FILE object that identifies the stream
- the second argument to this function is the format
  - a C string that contains one or more of the following items
    - whitespace character
    - non-whitespace character
    - format specifiers
    - usage is similar to printf, but, to a file
- if successful, the total number of characters written is returned otherwise, a negative number is returned

Example

```
#include <stdio.h>
#include <stdlib.h>

int main () {
  FILE * fp;

  fp = fopen ("file.txt", "w+");
  fprintf(fp, "%s %s %s %s %d", “Hello", “my", “number“, “is”, 555);

  fclose(fp);
  return(0);
}
```

File Positioning

- for many applications, you need to be able to access data in a file other than sequential order
- there are various functions that you can use to access data in random sequence
- there are two aspects to file positioning
  - finding out where you are in a file
  - moving to a given point in a file
- you can access a file at a random position regardless of whether you opened the file

Finding out where you are

- you have two functions to tell you where you are in a file
  - ftell()
  - fgetpos()

```
long ftell(FILE *pfile);
```

- this function accepts a file pointer as an argument and returns a long integer value that specifies the current position in the file

```
long fpos = ftell(pfile);
```

- the fpos variable now holds the current position in the file and you can use this to return to this position at any subsequent time
  - value is the offset in bytes from the beginning of the file

ftell() example

```
FILE *fp;
int len;

fp = fopen("file.txt", "r");
if( fp == NULL ) {
   perror ("Error opening file");
   return(-1);
}
fseek(fp, 0, SEEK_END);

len = ftell(fp);
fclose(fp);

printf("Total size of file.txt = %d bytes\n", len);
```

fgetpos()

- the second function providing information on the current file position is a little more complicated

```
int fgetpos(FILE * pfile, fpos_t * position);
```

- the first parameter is a file pointer
- the second parameter is a pointer to a type that is defined in stdio.h
  - fpos_t - a type that is able to record every position within a file

- the fgetpos() function is designed to be used with the positioning function fsetpos()

- the fgetpos() function stores the current position and file state information for the file in position and returns 0 if the operation is successful
  - returns a nonzero integer value for failure

```
fpos_t here;
fgetpos(pfile, &here);
```

- the above records the current file position in the variable here
- you must declare a variable of type fpos_t
  - cannot declare a pointer of type fpos_t* because there will not be any memory allocated to store the position data

fgetpos() example

```
FILE *fp;
fpos_t position;

fp = fopen("file.txt","w+");
fgetpos(fp, &position);
fputs("Hello, World!", fp);

fclose(fp);
```

Setting a position in a file

- as a complement to ftell(), you have the fseek() function

int fseek(FILE *pfile, long offset, int origin);

- the first parameter is a pointer to the file you are repositioning
- the second and third parameters define where you want to go in the file
  - second parameter is an offset from a reference point specified by the third parameter
  - reference point can be one of three values that are specified by the predefined names
    - SEEK_SET - defines the beginning of the file
    - SEEK_CUR - defines the current position in the file
    - SEEK_END - defines the end of the file

- for a text mode file, the second argument must be a value returned by ftell()

- the third argument for text mode files must be SEEK_SET.
  - for text files, all operations with fseek() are performed with reference to the beginning of the file
  - for binary files, the offset argument is simply a relative byte count
    - can therefore supply positive or negative values for the offset when the reference point is specified as SEEK_CUR

fseek() example

```
#include <stdio.h>

int main () {
   FILE *fp;

   fp = fopen("file.txt","w+");
   fputs("This is Jason", fp);

   fseek( fp, 7, SEEK_SET );
   fputs(" Hello how are you", fp);
   fclose(fp);

   return(0);
}
```

fsetpos()

- you have the fsetpos() function to go with fgetpos()

int fsetpos(FILE *pfile, const fpos_t *position);

- the first parameter is a pointer to the open file
- the second is a pointer of the fpos_t type
  - the position that is stored at the address was obtained by calling fgetpos()

fsetpos(pfile, &here);

- the variable here was previously set by a call to fgetpos()

- the fsetpos() returns a nonzero value on error or 0 when it succeeds

- this function is designed to work with a value that is returned by fgetpos()
  - you can only use it to get to a place in a file that you have been before
  - fseek() allows you to go to any position just by specifying the appropriate offset

fgetpos() example

```
#include <stdio.h>

int main () {
   FILE *fp;
   fpos_t position;

   fp = fopen("file.txt","w+");
   fgetpos(fp, &position);
   fputs("Hello, World!", fp);

   fsetpos(fp, &position);
   fputs("This is going to override previous content", fp);
   fclose(fp);

   return(0);
}
```

31 people have written a note here.

Requirements

- write a program to find the total number of lines in a text file
- create a file that contains some lines of text
- open your test file
- use the fgetc function to parse characters in a file until you get to the EOF
  - If EOF increment counter
- display as output the total number of lines in the file

stddef.h

<stddef.h> - contains some standard definitions

Define	Meaning
`NULL`	A null pointer constant
`offsetof (structure, member)`	The offset in bytes of the member *`member`* from the start of the structure *`structure`*; the type of the result is `size_t`
`ptrdiff_t`	The type of integer produced by subtracting two pointers
`size_t`	The type of integer produced by the `sizeof` operator
`wchar_t`	The type of the integer required to hold a wide character (see Appendix A, "C Language Summary")

Taken from Programming in C, Kochan

limits.h

<limits.h> - contains various implementation-defined limits for character and integer data types

Define	Meaning
CHAR_BIT	Number of bits in a char (8)
CHAR_MAX	Maximum value for object of type char (127 if sign extension is done on chars, 255 otherwise)
CHAR_MIN	Minimum value for object of type char (–127 if sign extension is done on chars, 0 otherwise)
SCHAR_MAX	Maximum value for object of type signed char (127)
SCHAR_MIN	Minimum value for object of type signed char (–127)
UCHAR_MAX	Maximum value for object of type unsigned char (255)
SHRT_MAX	Maximum value for object of type short int (32767)
SHRT_MIN	Minimum value for object of type short int (–32767)
USHRT_MAX	Maximum value for object of type unsigned short int (65535)
INT_MAX	Maximum value for object of type int (32767)
INT_MIN	Minimum value for object of type int (–32767)
UINT_MAX	Maximum value for object of type unsigned int (65535)
LONG_MAX	Maximum value for object of type long int (2147483647)
LONG_MIN	Minimum value for object of type long int (–2147483647)
ULONG_MAX	Maximum value for object of type unsigned long int (4294967295)
LLONG_MAX	Maximum value for object of type long long int (9223372036854775807)
LLONG_MIN	Minimum value for object of type long long int (–9223372036854775807)
ULLONG_MAX	Maximum value for object of type unsigned long long int (18446744073709551615)

Taken from Programming in C, Kochan

stdbool.h

<stdbool.h> - file contains definitions for working with Boolean variables (type _Bool)

Define	Meaning
bool	Substitute name for the basic _Bool data type
true	Defined as 1
false	Defined as 0

Taken from Programming in C, Kochan

String functions

- to use any of these functions, you need to include the header file <string.h>

- char *strcat (s1, s2)
  - concatenates the character string s2 to the end of s1, placing a null character at the end of the final string. The function returns s1

- char *strchr (s, c)
  - searches the string s for the first occurrence of the character c. If it is found, a pointer to the character is returned; otherwise, a null pointer is returned

- int strcmp (s1, s2)
  - compares strings *s1* and *s2* and returns a value less than zero if *s1* is less than *s2*, equal to zero if *s1* is equal to *s2*, and greater than zero if *s1* is greater than *s2*

- char *strcpy (s1, s2)
  - copies the string s2 to s1, returning s1

- size_t strlen (s)
  - returns the number of characters in s, excluding the null character

String functions

- char *strncat (s1, s2, n)
  - copies s2 to the end of s1 until either the null character is reached or n characters have been copied, whichever occurs first. Returns s1

- int strncmp (s1, s2, n)
  - performs the same function as strcmp, except that at most n characters from the strings are compared

- char *strncpy (s1, s2, n)
  - copies s2 to s1 until either the null character is reached or n characters have been copied, whichever occurs first. Returns s1

- char *strrchr (s, c)
  - searches the string s for the last occurrence of the character c. If found, a pointer to the character in s is returned; otherwise, the null pointer is returned

- char *strstr (s1, s2)
  - searches the string s1 for the first occurrence of the string s2. If found, a pointer to the start of where s2 is located inside s1 is returned; otherwise, if s2 is not located inside s1, the null pointer is returned

- char *strtok (s1, s2)
  - breaks the string s1 into tokens based on delimiter characters in s2

character functions

- to use these character functions, you must include the file <ctype.h>

- isalnum, isalpha, isblank, iscntrl, isdigit, isgraph, islower, isspace, ispunct, isupper, isxdigit

- int tolower(*c*)
  - returns the lowercase equivalent of *c*. If *c* is not an uppercase letter, *c* itself is returned

- int toupper(*c*)
  - returns the uppercase equivalent of *c*. If *c* is not a lowercase letter, *c* itself is returned.

I/O functions

- to use the most common I/O functions from the C library you should include the header file <stdio.h>

- included in this file are declarations for the I/O functions and definitions for the names EOF, NULL, stdin, stdout, stderr (all constant values), and FILE

- int fclose (filePtr)
  - closes the file identified by filePtr and returns zero if the close is successful, or returns EOF if an error occurs

- int feof (filePtr)
  - returns nonzero if the identified file has reached the end of the file and returns zero otherwise

- int fflush (filePtr)
  - flushes (writes) any data from internal buffers to the indicated file, returning zero on success and the value EOF if an error occurs

I/O functions

- int fgetc (filePtr)
  - returns the next character from the file identified by filePtr, or the value EOF if an end-of-file condition occurs. (Remember that this function returns an int)

- int fgetpos (filePtr, fpos)
  - gets the current file position for the file associated with filePtr, storing it into the fpos_t (defined in <stdio.h>) variable pointed to by fpos. fgetpos returns zero on success, and returns nonzero on failure

- char *fgets (buffer, i, filePtr)
  - reads characters from the indicated file, until either i − 1 characters are read or a newline character is read, whichever occurs first

- FILE *fopen (fileName, accessMode)
  - opens the specified file with the indicated access mode

- int fprintf (filePtr, format, arg1, arg2, ..., argn)
  - writes the specified arguments to the file identified by filePtr, according to the format specified by the character string format

I/O functions

- int fputc (c, filePtr)
  - writes the value of c to the file identified by filePtr, returning c if the write is successful, and the value EOF otherwise

- int fputs (buffer, filePtr)
  - writes the characters in the array pointed to by buffer to the indicated file until the terminating null character in buffer is reached

- int fscanf (filePtr, format, arg1, arg2, ..., argn)
  - reads data items from the file identified by filePtr, according to the format specified by the character string format

- int fseek (filePtr, offset, mode)
  - positions the indicated file to a point that is offset (a long int) bytes from the beginning of the file, from the current position in the file, or from the end of the file, depending upon the value of mode (an integer)

- int fsetpos (filePtr, fpos)
  - sets the current file position for the file associated with filePtr to the value pointed to by fpos, which is of type fpos_t (defined in <stdio.h>). Returns zero on success, and nonzero on failure

I/O functions

- long ftell (filePtr)
  - returns the relative offset in bytes of the current position in the file identified by filePtr, or −1L on error

- int printf (format, arg1, arg2, ..., argn)
  - writes the specified arguments to stdout, according to the format specified by the character string. Returns the number of characters written.

- int remove (fileName)
  - removes the specified file. A nonzero value is returned on failure

- int rename (fileName1, fileName2)
  - renames the file fileName1 to fileName2, returning a nonzero result on failure.

- int scanf (format, arg1, arg2, ..., argn)
  - reads items from stdin according to the format specified by the string format

Conversion functions

- to use these functions that convert character strings to numbers you must include the header file <stdlib.h>

- double atof (s)
  - converts the string pointed to by s into a floating-point number, returning the result

- int atoi (s)
  - converts the string pointed to by s into an int, returning the result

- int atol (s)
  - converts the string pointed to by s into a long int, returning the result

- int atoll (s)
  - converts the string pointed to by s into a long long int, returning the result

---

Dynamic Memory functions

- to use these function that allocate and free memory dynamically you must include the stdlib.h header file

- void *calloc (n, size)
  - allocates contiguous space for n items of data, where each item is size bytes in length. The allocated space is initially set to all zeroes. On success, a pointer to the allocated space is returned; on failure, the null pointer is returned

- void free (pointer)
  - returns a block of memory pointed to by pointer that was previously allocated by a calloc(), malloc(), or realloc() call

- void *malloc (size)
  - allocates contiguous space of size bytes, returning a pointer to the beginning of the allocated block if successful, and the null pointer otherwise

- void *realloc (pointer, size)
  - changes the size of a previously allocated block to size bytes, returning a pointer to the new block (which might have moved), or a null pointer if an error occurs

---

Math functions

- to use common math functions you must include the math.h header file and link to the math library

- double acosh (x)
  - returns the hyperbolic arccosine of x, $x \geq 1$

- double asin (x)
  - returns the arcsine of x as an angle expressed in radians in the range $[-\pi/2, \pi/2]$. x is in the range $[-1, 1]$

- double atan (x)
  - returns the arctangent of x as an angle expressed in radians in the range $[-\pi/2, \pi/2]$

- double ceil (x)
  - returns the smallest integer value greater than or equal to x. Note that the value is returned as a double

---

Math functions

- double cos (r)
  - returns the cosine of r

- double floor (x)
  - returns the largest integer value less than or equal to x. Note that the value is returned as a double

- double log (x)
  - returns the natural logarithm of x, x ≥ 0

- double nan (s)
  - returns a NaN, if possible, according to the content specified by the string pointed to by s

- double pow (x, y)
  - returns xy. If x is less than zero, y must be an integer. If x is equal to zero, y must be greater than zero

- double remainder (x, y)
  - returns the remainder of x divided by y

Math functions

- double round (x)
  - returns the value of x rounded to the nearest integer in floating-point format. Halfway values are always rounded away from zero (so 0.5 always rounds to 1.0)

- double sin (r)
  - returns the sine of r

- double sqrt (x)
  - returns the square root of x, x ≥ 0

- double tan (r)
  - returns the tangent of r

- And so many more …….
  - complex arithmetic

Utility functions

- to use these routines, include the header file <stdlib.h>

- int abs (n)
  - returns the absolute value of its int argument n

- void exit (n)
  - terminates program execution, closing any open files and returning the exit status specified by its int argument n
  - EXIT_SUCCESS and EXIT_FAILURE, defined in <stdlib.h>
  - other related routines in the library that you might want to refer to are abort and atexit

- char *getenv (s)
  - returns a pointer to the value of the environment variable pointed to by s, or a null pointer if the variabl doesn't exist
  - used to get environment variables

Utility functions

- void qsort (arr, n, size, comp_fn)
  - sorts the data array pointed to by the void pointer arr
  - there are n elements in the array, each size bytes in length. n and size are of type size_t
  - the fourth argument is of type "pointer to function that returns int and that takes two void pointers as arguments."
  - qsort calls this function whenever it needs to compare two elements in the array, passing it pointers to the elements to compare

- int rand (void)
  - returns a random number in the range [0, RAND_MAX], where RAND_MAX is defined in <stdlib.h> and has a minimum value of 32767

- void srand (seed)
  - seeds the random number generator to the unsigned int value seed

- int system (s)
  - gives the command contained in the character array pointed to by s to the system for execution, returning a system-defined value
  - system ("mkdir /usr/tmp/data");

Assert Library

- the assert library, supported by the assert.h header file, is a small one designed to help with debugging programs

- it consists of a macro named assert()
  - it takes as its argument an integer expression
  - If the expression evaluates as false (nonzero), the assert() macro writes an error message to the standard error stream (stderr) and calls the abort() function, which terminates the program

```
z = x * x - y * y;  /* should be + */
assert(z >= 0);  // asserts that z is greater than or equal to 0
```

Other useful header files

- time.h
  - defines macros and functions supporting operations with dates and times

- errno.h
  - defines macros for the reporting of errors

- locale.h
  - defines functions and macros to assist with formatting data such as monetary units for different countries

- signal.h
  - defines facilities for dealing with conditions that arise during program execution, including error conditions

- stdarg.h
  - defines facilities that enable a variable number of arguments to be passed to a function