Program Structure

Terry Marris March 2010

Student

Scenario

A student has a number, a name and a number of credits. No two students have the same number. A student gains credits for completing their course units.

A student file contains many student records.

StudentFile

Number	***Name***	***Credits***
001	*Terry Bull*	*3*
002	*Ann D'Pandy*	*6*
003	*May Day*	*3*
004	*Max Power*	*9*
005	*Pearl Button*	*6*
006	*Jo King*	*6*
007	*Priti Manek*	*9*

We provide a program that will:

- add new students to the StudentFile. A new student has zero credits. No two
- students have the same student number.
- update the number of credits for a student
- list all students

But first we program for just one student.

One Student

We start by creating just one student.

```
/* student.c - creates a single student. */
/* Terry Marris 27 March 2010 */

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

/* MODEL: algorithms and data structures. */

#define numberSize 5
#define nameSize 15
#define creditSize 3

typedef struct {
char number[numberSize];
char name[nameSize];
int credits;
} Student;
```

```
/* newStudent: creates a new student with the given number and name */
Student newStudent(char number[], char name[])
{
   Student student;
   strcpy(student.number, number);
   strcpy(student.name, name);
   student.credits = 0;
   return student;
}


/* addCredits: returns a student with the given credits added */
Student addCredits(Student student, int nCredits)
{
   student.credits = student.credits + nCredits;
   return student;
}


/* USER INTERFACE -VIEW: handles display operations for the model. */

/* printHeadings: displays headings for student attributes */
int printHeadings()
{
   printf("%-10s %-20s %s \n", "Number", "Name", "Credits");
   return 0;
}


/* printStudent: displays a students attributes */
int printStudent(Student student)
{
   printf("%-10s %-20s %d \n",
   student.number, student.name, student.credits);
   return 0;
}


/* USER INTERFACE - CONTROLLER: gets and processes user input;
includes prompts and menus. */

/* readString: reads a string from the keyboard, returns its length */
int readString(char string[], int maxLength)
{
   int c, i;
   i = 0;

   while (i < maxLength -1) {
      c = getchar();
      if (c == EOF || c == '\n')
         break;
      string[i] = c;
      i++;
   }
   string[i] = '\0';
   _flushall();
   return i;
}
```

```
/* readStudentNumber: reads student number from the keyboard */
int readStudentNumber(char number[])
{
   printf("Number? ");
   readString(number, numberSize);
   return 0;
}


/* readStudentName: reads student name from the keyboard */
int readStudentName(char name[])
{
   printf("Name? ");
   readString(name, nameSize);
   return 0;
}


/* readStudentCredits: reads student credits from the keyboard */
int readStudentCredits()
{
   char string[creditSize];

   printf("Credits? ");
   readString(string, creditSize);
   return atoi(string);
}


/* main: creates, amends and prints a student */
int main()
{
   Student student;
   char number[numberSize];
   char name[nameSize];
   int credits;

   readStudentNumber(number);
   readStudentName(name);
   student = newStudent(number, name);
   printHeadings();
   printStudent(student);

   credits = readStudentCredits();
   student = addCredits(student, credits);
   printStudent(student);
}
```

Model-View-Controller

We separate the user interface from the rest of the program. We split the program into three sections.

Model deals with objects in the problem domain. Here the problem domain is just one student. We create a student. We add credits to the student. The model does not know, and it does not care, where the input comes from or where the output goes to.

View deals with displaying the model. Here, we display the student's attributes - number, name, credits. (Attributes? What are the attributes of your boy or girl friend?)

Controller deals with input from the keyboard. It includes prompts and menus.

The main advantage of using the Model-View-Controller (MVC) pattern is that if you come to adapt the program to use Windows, the changes you need to make occur in a well-defined part of the program. The main disadvantage is that you have to think just a little bit more in the first place.

Model

A student has a number, a name, and a number of credits. So we write:

```
#define numberSize 5
#define nameSize 15
#define creditSize 3

typedef struct {
   char number[numberSize];
   char name[nameSize];
   int credits;
} Student;
```

The maximum size of *number* is 9999, a *name* has at most 14 characters, and we intend the maximum size for *credits* to be 99. We are not going to do sums with a student's *number*, so we define it as an array of *char* i.e. a string.

typedef introduces a new type. Here the type is a *struct* (for structure). A C structure is just like a record in Access. Here, we have named the struct *Student*.

To create a new student we supply a student number and the student's name. A new student has 0 credits.

```
/* newStudent: creates a new student with the given number and name
*/
Student newStudent(char number[], char name[])
{
   Student student;

   strcpy(student.number, number);
   strcpy(student.name, name);
   student.credits = 0;
   return student;
}
```

strcpy() copies a string from right to left, e.g. from *number* into *student.number*. *strcpy()* is defined in *stdlib.h*.

Like so many brilliant mathematicians and programmers, we think from right to left when it suits us.

```
student.credits = student.credits + nCredits;
```

says take *nCredits*, add it to *student.credits*, put the result in the *student.credits* on the left hand side of the equals sign. *nCredits* stands for number of credits.

```
/* addCredits: returns a student with the given credits added */
Student addCredits(Student student, int nCredits)
{
   student.credits = student.credits + nCredits;
   return student;
}
```

The function is given a *student* and a number of credits to deal with in the *Control* section shown below.

View

We display headings above the student's number, name and credits.

```
/* printHeadings: displays headings for student attributes */
int printHeadings()
{
   printf("%-10s %-20s %s \n", "Number", "Name", "Credits");
   return 0;
}
```

%-10s is a format specification. It says display a string left justified in a space 10 characters wide. Here, we have three format specifications. The order in which they are written matches the order in which the arguments are written. So, *%-10s* applies to *"Number"*, *%-20s* applies to *"Name"* and *%s* applies to *"Credits"*.

The *printStudent()* function works in a similar way. Here, the *%d* applies to *student.credits*, which is an integer (i.e. a whole number).

```
/* printStudent: displays a students attributes */
int printStudent(Student student)
{
   printf("%-10s %-20s %d \n",
   student.number, student.name, student.credits);
   return 0;
}
```

Notice *Student student* in the function header *int printStudent(Student student)*. *Student* with a capital s is the type we introduced above; *student* with a small s is the name we have given to a variable of type *Student*.

Controller

We need to input text and numbers at the keyboard. C provides some functions, each with limitations. *gets()* reads a string, but there is no control over its length and so it is possible to overflow a string variable. *fgets()* reads a string, controls its length but includes the newline character, which we usually need to remove. *scanf()* reads a string according to given format specifications but leaves unread characters in the keyboard buffer as well as the newline character. So we write our own function.

```
/* getString: reads a string from the keyboard, returns its length
*/
int getString(char string[], int maxLength)
{
   int c, i;
   i = 0;

   while (i < maxLength -1) {
      c = getchar();
      if (c == EOF || c == '\n')
         break;
      string[i] = c;
      i++;
   }
   string[i] = '\0';
   _flushall();
   return i;
}
```

We loop for as long as the given maximum length of the input is not reached. We remember to allow space for the end-of-string character, */0*. *getchar()* gets the next character from the input buffer (i.e. from the keyboard). If the character just entered is either the end-of-file character, or the newline character, we break out of the loop; if not we put the character into the string and increment the string array index, *i*. When finished looping we insert the endof-string character, and empty the keyboard buffer with *_flushall()*. *_flushall()* is not ANSI C, but seems to be provided by most modern C compilers running in a Windows environment. *EOF*, *getchar()* and *_flushall()* are all defined in *stdio.h*.

We input a student's number from the keyboard using our version of *readString()*.

```
/* readStudentNumber: reads student number from the keyboard */
int readStudentNumber(char number[])
{
   printf("Number? ");
   readString(number, numberSize);
   return 0;
}
```

To get the new student's number we print the prompt *Number?*, and read the user's input with *getString(number, numberSize)*. We remember that *numberSize* is defined as *5*. We allow four characters for the number, the fifth character is for the end-of-string marker. The number read is returned in the parameter *char number[ ]*.

readStudentName() works in a similar way.

```
/* readStudentName: reads student name from the keyboard */
int readStudentName(char name[])
{
   printf("Name? ");
   readString(name, nameSize);
   return 0;
}
```

readStudentCredits() returns an integer number input at the keyboard. The string entered is converted to an integer with *atoi()*. *atoi()* is defined in *stdlib.h*.

```
/* readStudentCredits: reads student credits from the keyboard */
int readStudentCredits()
{
   char string[creditSize];

   printf("Credits? ");
   readString(string, creditSize);
   return atoi(string);
}
```

We perform the testing in *main()*.

```
Student student;
char number[numberSize];
char name[nameSize];
int credits;
```

defines the variables for *student*, *number*, *name* and *credits*.

```
readStudentNumber(number);
readStudentName(name);
student = newStudent(number, name);
```

creates a new student.

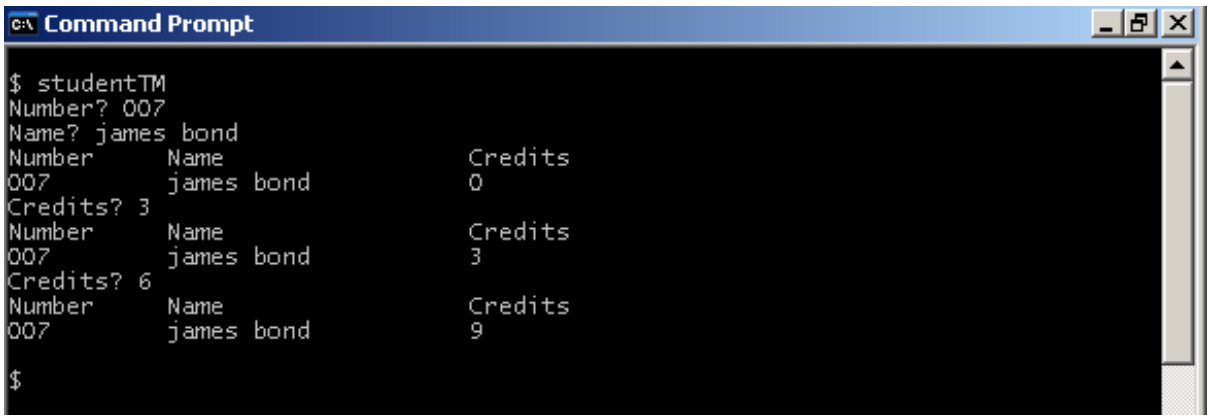
```
printHeadings();
printStudent(student);
```

prints headings and the student's attributes.

```
credits = readStudentCredits();
student = addCredits(student, credits);
```

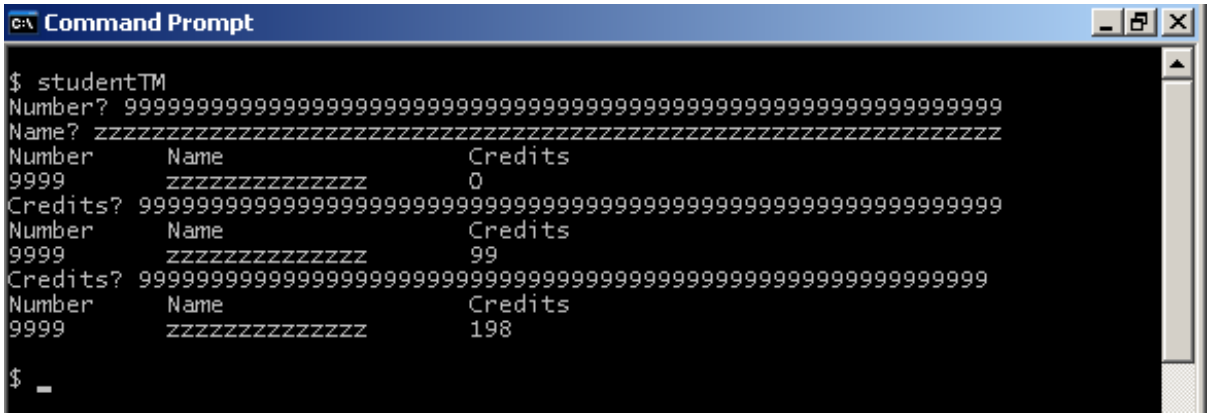
adds credits to the student. We do this test twice to show that the addition works as it should.

Test with normal data.



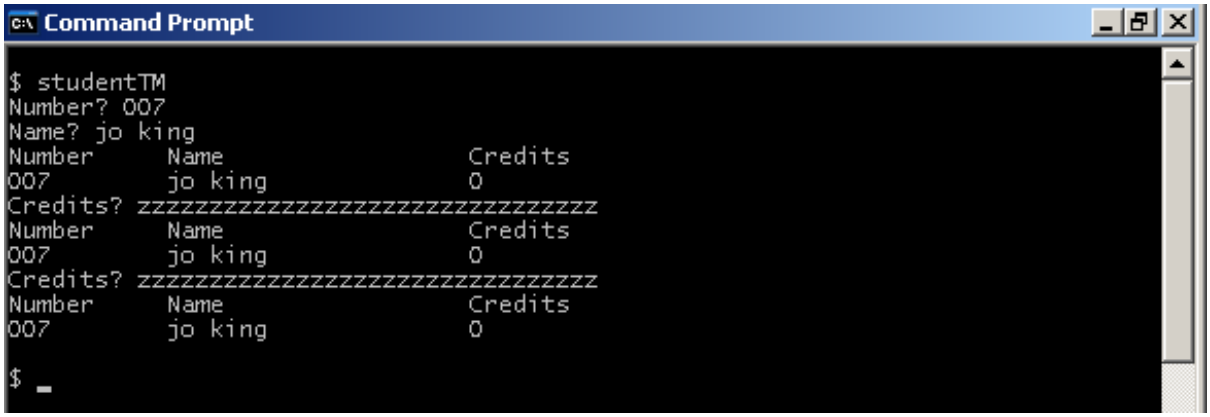
You could improve the display with some blank lines in appropriate places.

Test with very large data values.



We note that the total number of credits exceeds 99. We could possibly do some more programming to restrain the total number of credits to 99, but instead we acknowledge the problem and move on.

Test with invalid numeric input e.g. letters instead of digits for Credits



We note that invalid numeric input is translated as 0.

In the next section we extend the program so that it deals with a file of students.

Bibliography

KERNIGHAN B & RITCHIE D The C Programming Language Prentice Hall 1988
MARK WILLIAMS COMPANY ANSI C A Lexical Guide 1988
HOPKINS T & HORAN B Smalltalk: an Introduction to Application Development Using

VisualWorks Prentice Hall 1995
http://www.emcode.com/x/markup/tutorial/mvc.html accessed 26 Mar 2010

Student Records

Terry Marris March 2010

Scenario

A student has a number, a name and a number of credits. No two students have the same number. A student gains credits for completing their course units.

A student file contains many student records.

StudentFile

Number	***Name***	***Credits***
001	*Terry Bull*	*3*
002	*Ann D'Pandy*	*6*
003	*May Day*	*3*
004	*Max Power*	*9*
005	*Pearl Button*	*6*
006	*Jo King*	*6*
007	*Priti Manek*	*9*

We provide a program that will:

- add new students to the StudentsFile. A new student has zero credits. No two
- students have the same student number.
- update the number of credits for a student
- list all students in the StudentsFile

Many Students

Previously, in the previous section, we programmed for just one student. Now we see how to deal with a file of student records.

```
/* students.c - manages a file of student objects. */
/* Terry Marris 27 March 2010 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

/* MODEL: algorithms and data structures. */

#define numberSize 5
#define nameSize 15
#define creditSize 3
#define fileName "studentsFile.dat"

typedef enum { ok, duplicateStudent, studentNotFound } Report;

typedef struct {
   char number[numberSize];
   char name[nameSize];
   int credits;
} Student;
```

```
/* newStudent: creates a new student with the given number and name
*/
Student newStudent(char number[], char name[])
{
   Student student;

   strcpy(student.number, number);
   strcpy(student.name, name);
   student.credits = 0;
   return student;
}


/* addCredits: returns a student with the given credits added */
Student addCredits(Student student, int nCredits)
{
   student.credits = student.credits + nCredits;
   return student;
}


/* FILE OPERATIONS */

/* newFile: creates a new empty students file */
Report newFile()
{
   FILE *file = fopen(fileName, "wb");
   fclose(file);
   return ok;
}


/* isDuplicate: returns true (non-zero) if student number is in the
file */
int isDuplicate(char number[])
{
   Student student;

   FILE *file = fopen(fileName, "rb");
   fread(&student, sizeof(Student), 1, file);
   while (!feof(file)) {
      if (strcmp(number, student.number) == 0) {
         fclose(file);
         return 1;
      }
      fread(&student, sizeof(Student), 1, file);
   }
   fclose(file);
   return 0;
}
```

```
/* addToFile: appends the given student to file of students,
reports duplicateStudent if student.number already exists in the
file */
Report addToFile(Student student)
{
   if (isDuplicate(student.number))
      return duplicateStudent;

   FILE *file = fopen(fileName, "ab");
   fwrite(&student, sizeof(Student), 1, file);
   fclose(file);
   return ok;
}


/* fileSize: returns the number of records in the file */
int fileSize()
{
   int i = 0;
   Student student;

   FILE *file = fopen(fileName, "rb");
   fread(&student, sizeof(Student), 1, file);
   while (!feof(file)) {
      i++;
      fread(&student, sizeof(Student), 1, file);
   }
   fclose(file);
   return i;
}

int _iter; /* GLOBAL VARIABLE */

/* newIterator: initialises an iterator for the file */
int newIterator()
{
   _iter = 0;
   return _iter;
}


/* hasNext: returns true (non-zero) if there is a student in the
file in the current iteration that has not yet been visited */
int hasNext()
{
   return (_iter < fileSize());
}

/* nextStudent: returns the next student in the file */
Student nextStudent()
{
   Student student;

   assert(_iter >= 0 && _iter < fileSize());
   FILE *file = fopen(fileName, "rb+");
   fseek(file, _iter * sizeof(Student), SEEK_SET);
   fread(&student, sizeof(Student), 1, file);
   _iter++;
   fclose(file);
   return student;
}
```

```
/* updateFile: replaces the student returned by nextStudent with the
given student */
Report updateFile(Student student)
{
   if (!isDuplicate(student.number))
      return studentNotFound;
   _iter--;
   assert(_iter >= 0 && _iter < fileSize());
   FILE *file = fopen(fileName, "rb+");
   fseek(file, _iter * sizeof(Student), SEEK_SET);
   fwrite(&student, sizeof(Student), 1, file);
   _iter++;
   fclose(file);
   return ok;
}


/* USER INTERFACE -VIEW: handles display operations for the model
*/


/* printHeadings: displays headings for student attributes */
int printHeadings()
{
   printf("%-10s %-20s %s \n", "Number", "Name", "Credits");
   return 0;
}


/* printStudent: displays a students attributes */
int printStudent(Student aStudent)
{
   printf("%-10s %-20s %d \n",
          aStudent.number, aStudent.name, aStudent.credits);
   return 0;
}


/* USER INTERFACE -CONTROLLER: gets and processes user input,
includes prompts and menus. */

typedef enum { add, amend, view, quit, none } Choice;


/* readString: reads a string from the keyboard, returns its length
*/
int readString(char string[], int maxLength)
{
   int c, i;
   i = 0;

   while (i < maxLength -1) {
      c = getchar();
      if (c == EOF || c == '\n')
         break;
      string[i] = c;
      i++;
   }
```

```
   string[i] = '\0';
   _flushall();
   return i;
}


/* readStudentNumber: reads student number from the keyboard */
int readStudentNumber(char number[])
{
   printf("Number? ");
   readString(number, numberSize);
   return 0;
}


/* readStudentName: reads student name from keyboard */
int readStudentName(char name[])
{
   printf("Name? ");
   readString(name, nameSize);
   return 0;
}


/* readStudentCredits: reads student credits from keyboard */
int readStudentCredits()
{
   char string[creditSize];

   printf("Credits? ");
   readString(string, creditSize);
   return atoi(string);
}


/* printError: prints error message */
int printError(Report report)
{
   switch (report) {
   case ok:
      break;

   case duplicateStudent:
      printf("duplicate student number used\n");
      break;

   case studentNotFound:
      printf("student not found\n");
      break;

   default:
      printf("unexpected file handling error\n");
      exit(1);
      break;
   }
   return 0;
}
```

```
/* addNewStudent: gets new student from user, adds to file */
int addNewStudent()
{
   Student student;
   char number[numberSize];
   char name[nameSize];
   Report report;

   readStudentNumber(number);
   readStudentName(name);
   student = newStudent(number, name);
   report = addToFile(student);
   printError(report);
   return 0;
}


/* amendStudent: gets student from file, amends student record,
updates file */
int amendStudent()
{
   Student student;
   char number[numberSize];
   int credits;

   readStudentNumber(number);
   credits = readStudentCredits();
   newIterator();
   while (hasNext()) {
      student = nextStudent();
      if (strcmp(student.number, number) == 0) {
         student = addCredits(student, credits);
         updateFile(student);
         break;
      }
   }
   return 0;
}


/* viewAllStudents: displays all students */
int viewAllStudents()
{
   printHeadings();
   newIterator();
   while (hasNext())
      printStudent(nextStudent());
   return 0;
}


/* exitProgram: terminates program execution */
int exitProgram()
{
   exit(0);
   return 0;
}
```

```
/* menu: displays menu, returns user's choice */
Choice menu()
{
   char choice[3];

   printf("Student File Menu: A(dd, U(pdate, V(iew, Q(uit ? ");
   readString(choice, 3);
   choice[0] = toupper(choice[0]);
   switch(choice[0]) {
   case 'A':
      return add;
   case 'U':
      return amend;
   case 'V':
      return view;
   case 'Q':
      return quit;
   default:
      break;
   }
   return none;
}


/* processChoice: processes choice from menu */
int processChoice(Choice choice)
{
   switch (choice) {
   case add:
      addNewStudent();
      break;
   case amend:
      amendStudent();
      break;
   case view:
      viewAllStudents();
      break;
   case quit:
      exitProgram();
      break;
   case none:
      break;
   default:
      break;
   }
   return 0;
}


/* main: creates, updates and prints file of students */
int main()
{
   newFile();
   for ( ; ; )
      processChoice(menu());
   }
   return 0;
}
```

Model

We describe the elements that were not included in the previous section which dealt with just one student.

External FileName

We will have a file of students on memory stick or disk. We choose *studentsFile.dat* to be the name of this file as it is known to Windows.

```
#define fileName "studentsFile.dat"
```

Wherever *fileName* appears in the program, it will be replaced automatically with *"studentsFile.dat"* by the C compiler.

File Report

If we do a search for a student in the file, there are two possible outcomes: either we find what we are looking for (*ok*) or we do not (*studentNotFound*). So we define a new type named *Report* to help inform us whether our file handling was successful.

```
typedef enum {ok, duplicateStudent, studentNotFound } Report;
```

New File

To create a new, empty file is easy. We open the file in write binary mode (*"wb"*), then close it.

```
/* newFile: creates a new empty students file */
Report newFile()
{
   FILE *file = fopen(fileName, "wb");
   fclose(file);
   return ok;
}
```

FILE, *fopen()* and *fclose()* are all defined in *stdio.h*. *FILE* is the file type. **file* (say pointer to file) is the name we have chosen for the file. *fopen()* makes the connection between the file name used inside the program, and the file name used by Windows. *fclose()* breaks the connection and closes the file. *ok* is a value in the *Report* type we defined earlier. Any file in our current working directory (or folder) named *fileName* (= *studentsFile.dat* remember) is replaced with the new, empty one.

Check for Duplicate Student Number

We do not want two students with the same number. It is bad enough if two students have the same name. So, before adding a new student to the file we check to see if their student number has been used before.

```
/* isDuplicate: returns true (non-zero) if student number is in the
file */
int isDuplicate(char number[])
{
   Student student;
   FILE *file = fopen(fileName, "rb");

   fread(&student, sizeof(Student), 1, file);
   while (!feof(file)) {
      if (strcmp(number, student.number) == 0) {
         fclose(file);
         return 1;
      }
      fread(&student, sizeof(Student), 1, file);
   }
   fclose(file);
   return 0;
}
```

We provide the student number in the function header. The function does know where the number comes from, nor does it care how it got there. We define a *Student* variable to hold the students we retrieve from the file, one by one.

```
int isDuplicate(char number[])
{
   Student student;
```

We open the file for reading in binary mode and read the first student record from the file. If the read fails, because the end of file has been reached for example, then *feof()* is automatically set to true and the loop terminates before it has even started.

```
FILE *file = fopen(fileName, "rb");
fread(&student, sizeof(Student), 1, file);
while (!feof(file)) {
```

But if the first read is successful, *feof()* remains false and we enter the loop. Remember ! means not. So we loop for as long as the end of the file has not been reached.

Inside the loop we compare the number we supplied with the current student's number: if they are the same we have a duplicate. And so we close the file and return 1 for true. Remember in C true is a non-zero value, false is zero. return means go back straightaway to where you came from.

```
if (strcmp(number, student.number) == 0) {
   fclose(file);
   return 1;
}
```

If we do not exit the loop with the *return* statement, we carry on. We attempt to retrieve the next student record from the file, and then return to the top of the loop.

```
   fread(&student, sizeof(Student), 1, file);
}
```

When the loop terminates because *feof()* becomes true, we close the file and return false (0) since we have not found a duplicated student number in the file.

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

fread() is dedined in *stdio.h*.

Add Student to File

Now we can add a new student to the file.

```
/* addToFile: appends the given student to file of students,
reports duplicateStudent if student.number already exists in the
file */
Report addToFile(Student student)
{
   if (isDuplicate(student.number))
      return duplicateStudent;

   FILE *file = fopen(fileName, "ab");
   fwrite(&student, sizeof(Student), 1, file);
   fclose(file);
   return ok;
}
```

If the student's number supplied is also in the file, we return *duplicateStudent* error report.

```
if (isDuplicate(student.number))
return duplicateStudent;
```

If the student's number has not been used before, we carry on. We open the file in append binary mode (*"ab"*). This means that our student gets added onto the end of the file.

```
fwrite(&student, sizeof(Student), 1, file);
```

fwrite() adds a new record. *&student* (say address of student) is the student record we want to add. *sizeof(Student)* is the size of our student structure (record) in bytes. We want to write just one record, hence the 1. And file is the file we have just opened and want to write the new student record to. Finally, we close the file with *fclose()* and return *ok*.

fwrite() is defined in *stdio.h*.

File Size

We count how many records there are in the file .

```
/* fileSize: returns the number of records in the file */
int fileSize()
{
   int i = 0;
   Student student;

   FILE *file = fopen(fileName, "rb");
   fread(&student, sizeof(Student), 1, file);
```

```
   while (!feof(file)) {
      i++;
      fread(&student, sizeof(Student), 1, file);
   }
   fclose(file);
   return i;
}
```

We have a counter named *i*. *i* starts off at zero. We loop for as long as the end of the file has not been reached. Every time round the loop we add 1 to *i* and retrieve the next record. Finally, when the end of the file has been reached and the loop terminated, we close the file and return *i*, the record count.

The Iterator

An iterator visits each record in the file in turn. We declare the iterator variable, ***_iter***, as a global variable because several functions use and update it. We use an underscore in the variable name so we are not likely to use the same name by mistake for something else inside a function.

```
int _iter; /* GLOBAL VARIABLE */

/* newIterator: initialises an iterator for the file */
int newIterator()
{
   _iter = 0;
   return _iter;
}
```

newIterator() just sets the iterator to zero.

Has Next

hasNext() returns true if there is a record in the file that has not yet been visited in the current iteration.

```
/* hasNext: returns true (non-zero) if there is a student in the
file in the current iteration that has not yet been visited */
int hasNext()
{
   return (_iter < fileSize());
}
```

We just check that the value of ***_iter*** is less than the size of the file.

Next Student

We use ***_iter*** to refer to a record's position in the file. *nextStudent()* returns the next student record in the file.

```
/* nextStudent: returns the next student in the file */
Student nextStudent()
{
   Student student;
```

```
   assert(_iter >= 0 && _iter < fileSize());
   FILE *file = fopen(fileName, "rb+");
   fseek(file, _iter * sizeof(Student), SEEK_SET);
   fread(&student, sizeof(Student), 1, file);
   _iter++;
   fclose(file);
   return student;
}
```

We need to ensure that the value if *_iter* remains within bounds of between zero (including zero) and the file size. If would be an error to attempt to visit a record that does not exist in the file. We are paranoid about this so we write

```
assert(_iter >= 0 && _iter < fileSize());
```

Here, we assert, claim with convction, that the value of *_iter* remains within the bounds we set. If by some mischance it falls outside these bounds the *assert()* function halts program execution with a message to the user to contact the program writer. It is an error we do not expect to happen! *assert()* is defined in *assert.h*.

We open the file in read binary update mode (*"rb+"*). This allows us to both retrieve a record from the file and to write a record to the file, both at a specified position. The position in the file is set by

```
fseek(file, _iter * sizeof(Student), SEEK_SET);
```

fseek() is used to position the file pointer ready for the next read or write operation. The position is set by *_iter * sizeof(Student)*. *SEEK_SET* means from the beginning of the file.

Having retrieved a record we increment *_iter* ready for the next operation.

Update File

We update a file by replacing an existing record with an amended record.

```
/* updateFile: replaces the student returned by nextStudent with the
given student */
Report updateFile(Student student)
{
   if (!isDuplicate(student.number))
      return studentNotFound;
   _iter--;
   assert(_iter >= 0 && _iter < fileSize());
   FILE *file = fopen(fileName, "rb+");
   fseek(file, _iter * sizeof(Student), SEEK_SET);
   fwrite(&student, sizeof(Student), 1, file);
   _iter++;
   fclose(file);
   return ok;
}
```

A call to *updateFile()* must be immediately preceded by a call to *nextStudent()*. Failure to do so will result in the wrong record being updated.

First, we check to see if there is a student to update in the file; if we cannot find their number we know the student is not there.

Then, we backtrack *_iter* by reducing its value by 1. It now points to the record last visited by *nextStudent()*. The rest is straightforward: we assert that our iterator is within bounds, we open the file in read binary update mode, we position the file pointer with *fseek()*, we write the given record into that position (thereby overwriting what was there previously), we advance *_iter*, close the file and return *ok*

We have completed the discussion of our tools. Now we go on to use them.

View

The view remains unchanged from that described in the previous section featuring just a single, Student. So we have nothing more to say on that.

Controller

In controller we provide new students to be added, amended students to replace exisiting ones, and menus to be used. But first we look at the error report.

Error Reports

File handling errors arise from adding a new student with a student number already in use, and searching for a student that does not exist. The non-fatal errors reports are all dealt with in one place here.

```
/* printError: prints error message */
int printError(Report report)
{
   switch (report) {
   case ok:
      break;
   case duplicateStudent:
      printf("duplicate student number used\n");
      break;
   case studentNotFound:
      printf("student not found\n");
      break;
   default:
      printf("unexpected file handling error\n");
      exit(1);
      break;
   }
   return 0;
}
```

The only fatal error is the unexpected one. *exit(1)* immediately closes all open files and terminates program execution. Incidentally, the last *break* is not required. but we pit it in as a matter of neatness and consistency.

Add New Student

We get new student number and name from the user at the keyboard, create the new student, and add it to the file, all with the tools already created and described.

```
/* addNewStudent: gets new student from user, adds to file */
int addNewStudent()
{
   Student student;
   char number[numberSize];
   char name[nameSize];
   Report report;

   readStudentNumber(number);
   readStudentName(name);
   student = newStudent(number, name);
   report = addToFile(student);
   printError(report);
   return 0;
}
```

We report erros, such as duplicate student number used, by writing:

```
Report report;
...
report = addToFile(student);
printReport(report);
```

Amend Student

We get the student number and the number of credits to be added from the user at the keyboard.

We set up the iterator to look at each student in the file in turn.

We loop for as long as *hasNext()* tells us that there is yet another student in the file to look at.

We get the next student. We use *strcmp()* to see if that is the student we are looking for: if it is we amend the number of credits the student has. Then we update the file with the amended student.

```
/* amendStudent: gets student from file, amends student record,
updates file */
int amendStudent()
{
   Student student;
   char number[numberSize];
   int credits;

   readStudentNumber(number);
   credits = readStudentCredits();
   newIterator();
   while (hasNext()) {
      student = nextStudent();
      if (strcmp(student.number, number) == 0) {
         student = addCredits(student, credits);
         updateFile(student);
         break;
      }
   }
   return 0;
}
```

break teminates loop execution: there is no need to carry on looping if we have found and dealt with the student record we were looking for.

Notice that we made a call to *newIterator()* to initiallise *_iter*, and that *hasNext()* and *updateFile()* both use *_iter*.

View All Students

Now for the payback. All our hard work and head scratching is rewarded. *viewAllStudents()* is beautifully simple and elegant.

```
/* viewAllStudents: displays all students in the student file */
int viewAllStudents()
{
   printHeadings();
   newIterator();
   while (hasNext())
      printStudent(nextStudent());
   return 0;
}
```

We re-start the iterator. Loop for as long as there is another student record to visit. Print the next student.

Exit Program

There comes a time when we want to stop running the program. *exitProgram()* does just that with a call to *exit(0)*. *exit()* closes any open files and terminates program execution. It is defined in *stdio.h*.

```
/* exitProgram: terminates program execution */
int exitProgram()
{
   exit(0);
   return 0;
}
```

You might like to include

```
   printf("Press return to continue ... ");
   getchar();
```

if you need to hold the program run on screen before returning to your program development environment.

Menu

Now we come to the menu.

```
typedef enum { add, amend, view, quit, none } Choice;

/* menu: displays menu, returns user's choice */
Choice menu()
{
   char choice[3];
```

```
   printf("Student File Menu: A(dd, U(pdate, V(iew, Q(uit? ");
   readString(choice, 3);
   choice[0] = toupper(choice[0]);
   switch(choice[0]) {
   case 'A':
      return add;
   case 'U':
      return amend;
   case 'V':
      return view;
   case 'Q':
      return quit;
   default:
      break;
   }
   return none;
}
```

The menu is a simple one liner. The choices are *A* to add a new student, *U* to update the student file with an amended record, *V* to view all records in the file, and *Q* to quit. The first letter of the user's choice is converted to upper case with

```
choice[0] = toupper(choice[0]);
```

toupper() is defined in *string.h*.

The value returned by the menu is processed in *processChoice()*.

Process Choice

```
/* processChoice: processes choice from menu */
int processChoice(Choice choice)
{
   switch (choice) {
   case add:
      addNewStudent();
      break;

   case amend:
      amendStudent();
      break;

   case view:
      viewAllStudents();
      break;

   case quit:
      exitProgram();
      break;

   case none:
      break;

   default:
      break;
   }
   return 0;
}
```

The choices *add*, *amend*, *view* and *quit* are all returned by *menu()*. If the *choice* was *add* for example, then the function *addNewStudent()* defined above is called.

Main

The program is launched by a call to *processChoice(menu))* in an endless loop. The loop terminates when the user chooses *Quit* and *exitProgram()* is called.

```
/* main: creates, updates and prints file of students */
int main()
{
   newFile();
   for ( ; ; )
      processChoice(menu());
}
```

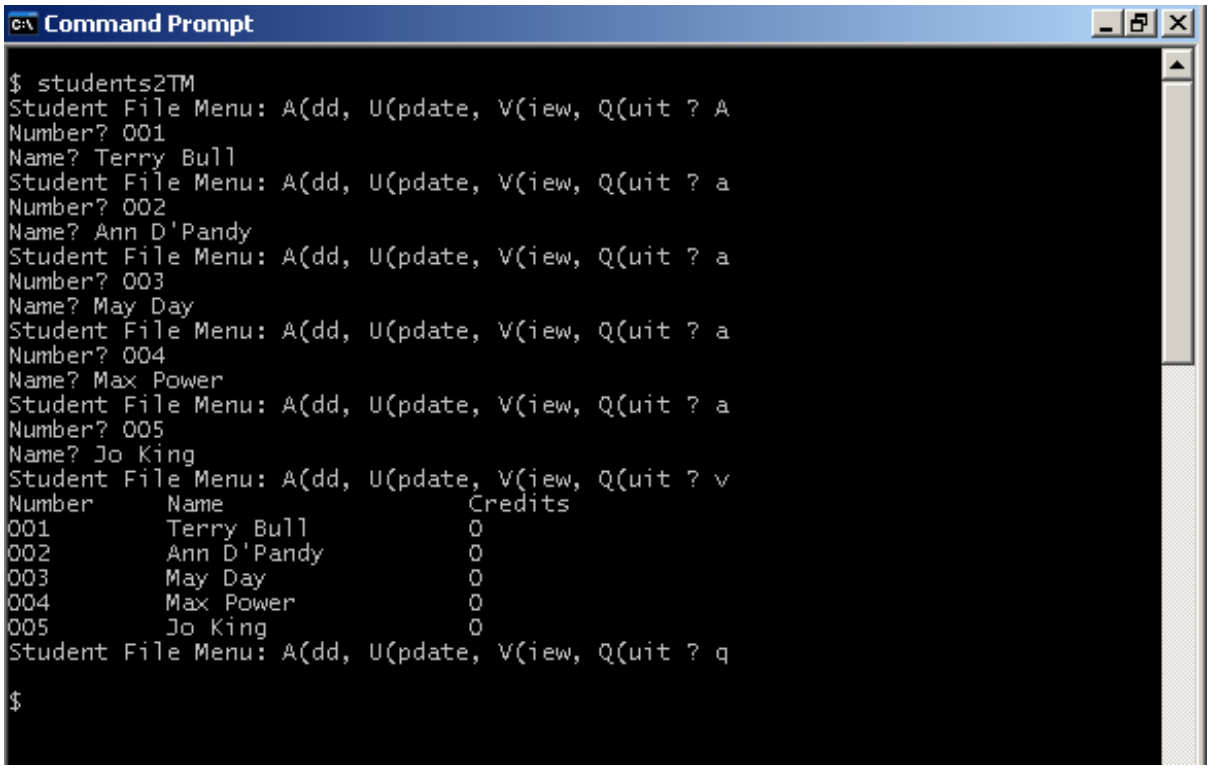
Another way of writing *processChoice(menu())* is:

```
Choice choice;
...
choice = menu();
processChoice(choice);
```

If you want to preserve the contents of your file from one day to the next then just disable the *newFile()* statement.

Testing

Normal data. We create five new students, add them to the file, view the contents of the file, then quit.



You could perhaps improve the layout by including blank lines in appropriate places.
We add 3, 5 and 7 credits to students 001, 003 and 005 respectively.

```
005         Jo King                 0
Student File Menu: A(dd, U(pdate, V(iew, Q(uit ? u
Number? 001
Credits? 3
Student File Menu: A(dd, U(pdate, V(iew, Q(uit ? u
Number? 003
Credits? 5
Student File Menu: A(dd, U(pdate, V(iew, Q(uit ? u
Number? 005
Credits? 7
Student File Menu: A(dd, U(pdate, V(iew, Q(uit ? v
Number      Name                Credits
001         Terry Bull          3
002         Ann D'Pandy         0
003         May Day             5
004         Max Power           0
005         Jo King             7
Student File Menu: A(dd, U(pdate, V(iew, Q(uit ? q

$
```

Error Testing: Duplicate Student. We add two students with the same student number.

```
Command Prompt

$ students2TM
Student File Menu: A(dd, U(pdate, V(iew, Q(uit ? a
Number? 001
Name? Terry Bull
Student File Menu: A(dd, U(pdate, V(iew, Q(uit ? a
Number? 001
Name? Jo King
duplicate student number used
Student File Menu: A(dd, U(pdate, V(iew, Q(uit ? q

$ _
```

Error Testing: Assertion Failure. When the assertion that _iter >= 0 and < fileSize() fails:

```
Command Prompt

$ students2TM
Student File Menu: A(dd, U(pdate, V(iew, Q(uit ? a
Number? 001
Name? Terry Bull
Student File Menu: A(dd, U(pdate, V(iew, Q(uit ? v
Number      Name                Credits
Assertion failed: _iter > 0 && _iter < fileSize(), file students2TM.c, line 131

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.

$
```

What happens if you try to amend a record that is not in the file? What happens when you make a choice that is not on the menu?

Conclusion

We have provided a program that:

- adds new students to the studentsFile. A new student has zero credits. No two
- students have the same student number.
- updates the number of credits for a student lists all students in the studenstFile

Each function in the program completes one small task.

We have used Model-View-Controller architecture, separating the user interface from the model.

The model maintains the data. View displays the data. Controller handles events such as input from the keyboard.

Events, such as key press and mouse click, are passed to a Controller. A Controller updates the Model or a View. A View gets data from the Model.

Exercises

1 Add an extra field in the Student struct named status with values of either current or deleted. Initially, a student is current.

2 Provide functions that will find a given student in the file and change their status from current to deleted, and conversely. The advantage of marking a student as deleted is that their student number cannot be re-used and the record remains as a historical document in the file.

3 Provide functions to print:
- **a** all current students, and
- **b** all deleted students.

Bibliography

KERNIGHAN B & RITCHIE D The C Programming Language Prentice Hall 1988
MARK WILLIAMS COMPANY ANSI C A Lexical Guide 1988
HOPKINS T & HORAN B Smalltalk: an Introduction to Application Development Using

VisualWorks Prentice Hall 1995
http://www.emcode.com/x/markup/tutorial/mvc.html accessed 26 Mar 2010