Started on	Saturday, 10 August 2019, 9:28 PM
State	Finished
Completed on	Friday, 16 August 2019, 2:54 PM
Time taken	5 days 17 hours
Marks	26.60/27.00
Grade	**9.85** out of 10.00 (**99**%)

Information

Structuring Data and Code

Goals

Python and Java provide classes and modules/packages to help you structure larger programs. C has neither of these, but it is still possible to build large systems.

This lab introduces C data structures ("*structs*"), which allow you to organise your data into sets of related items.
C *typedefs* are also introduced. The lab also shows how larger programs can be broken down into a number of files, each containing a set of related functions and their data.

The material in this lab is covered in King, Chapters 15 and 16.

Video playlists:

1. Playlist on structuring data ("structs").
2. Playlist on structuring code (separate compilation).

Information

Structuring Data

Firstly we focus on how to structure data, collecting multiple primitive variables into a composite variable called a *struct.* The following program is introduced in the video 6.1 Intro to structs.

structexample1.c

Look carefully through the following program `structexample1.c`, which introduces the use of C structures.

Hide `structexample1.c` code

```
/*
 * structexample1.c
 * A simple program to introduce C structs
 *
 * Richard Lobb
 * August 2014
 */

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

// Define a structure suitable for representing a student
// in a list of students. In this trivial example only
// the name and age is stored for each student.

// The following declaration declares both the structure type
// and a global instance 'student' of that type.
struct student_s {
    char* name;
    int age;
    struct student_s* next;   // Pointer to next student in a list
} student;

// 'struct student_s' is now a type, and we can declare
// another global variable of that type.
struct student_s anotherStudent;

// printOneStudent: prints a single student, passed by value
void printOneStudent(struct student_s student)
{
    printf("%s (%d)\n", student.name, student.age);
}

// printStudents: print all students in a list of students, passed
// by reference.
void printStudents(const struct student_s* student)
{
    while (student != NULL) {
        printOneStudent(*student);
        student = student->next;
    }
}

// main just defines fields of the two students, links them
// together into a list, and prints the list.
int main(void)
{
    student.name = "Agnes McGurkinshaw";
    student.age = 97;
    student.next = &anotherStudent;

    anotherStudent.name = "Jingwu Xiao";
    anotherStudent.age = 21;
    anotherStudent.next = NULL;

    printStudents(&student);
    return EXIT_SUCCESS;
}
```

Note the following:

1. A *struct* is a collection of variables grouped together, like an object with no methods. Access to the fields or members of a `struct` uses the same 'dot' notation as Python and Java.
2. The syntax for declaring a struct is a bit curious. In the declaration `struct blah {...} thing`, *blah* is the name of a structure, called a *structure tag* and *thing* is a variable of type `struct blah`. Note that *thing* is not of type `blah` but of type `struct blah`. Further variables of type `struct blah` can be declared, e.g. we could declare a new variable `struct blah other`.

3. The identifier used for the tag name is in a separate *namespace* from that used for variables and types, so we could actually have declared the struct as struct student { ... } student. However, that confuses geany's syntax colour algorithm and would likely confuse most of the class too. Hence in this course we're using the convention of adding the suffix "_s" to structure tag names to distinguish them from identifiers in the normal namespace.
4. The C syntax allows us to declare zero or more variables within the struct declaration and zero or more afterwards. For example, the following two code fragments are both equivalent (as far as the compiler is concerned) to the code in `structexample1.c`:

```
struct student_s {
    char* name;
    int age;
    struct student_s* next;
} student, anotherStudent;
```

and

```
struct student_s {
    char* name;
    int age;
    struct student_s* next;
};

struct student_s student, anotherStudent;
```

However, declaring multiple variables on a line is contrary to the style rules of ENCE260.

5. Fig. 1 illustrates how the `next` field is used to create a simple linked list. It also shows how the two `name` fields point to separate strings. When you read structure definitions containing pointers you need to be able to see figures like this in your head.

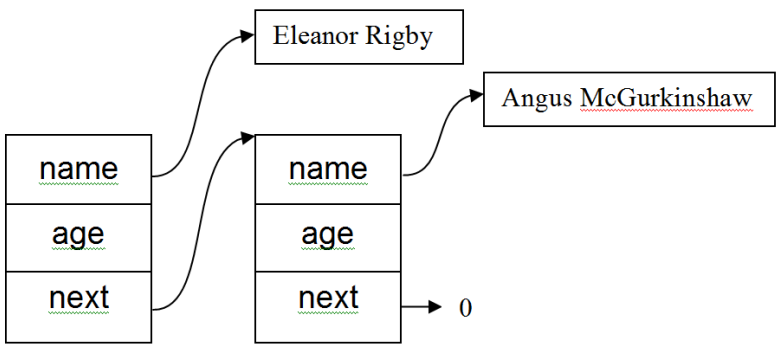


Figure 1: Two `Student` structs linked together. The `next` field of the second is NULL.

6. If we have a pointer to a struct, like the parameter `student` in the `printStudents` function, we can access the fields of the pointed-at struct by first dereferencing the pointer and then getting the fields with the dot notation. For example, if `p` is a pointer to a student structure, `*p` would be the student structure itself so that `(*p).name` would be the name of that student. However, C gives us a more convenient notation: `p->name`. Although the two notations are equivalent, please always use the latter.
7. The parameter to `printStudents` has been declared `const`. This prevents `printStudents` from modifying the student list it is given. When writing functions that take pointer parameters and do not need to modify the structure being pointed at you should always prefix the parameter declaration with the word `const`. This prevents you from inadvertently modifying the pointed-to structure and is also useful documentation for the user of your function, who can now be quite sure that the actual parameter will not be modified by calling your function.

Question **1**

Correct

Mark 1.00 out of 1.00

Which of the various C code lines below are syntactically legal following the three lines:

```
struct a {int b; } c;
struct a d;
struct a* e;
```

Note:

1. This example does not use the convention of suffixing the tag name with "_s". That would give too much away :-)
2. Simply typing the given code lines into a bit of code directly after the above declarations will probably not give you the right answers! A syntactically legal line is one that could be legal in some contexts; an illegal one could *never* be correct.

Select one or more:

- [x] `d.b = e->b;` ✔
- [ ] `d->b = 1;`
- [x] `c.b = 1;` ✔
- [x] `struct a f;` ✔
- [x] `d.b = c.b;` ✔
- [ ] `c->b = 1;`
- [ ] `a e;`
- [x] `e->b = 1;` ✔

Correct

Marks for this submission: 1.00/1.00.

Question **2**

Correct

Mark 3.00 out of 3.00

Modify *structexample1.c* so that the variables *student* and *anotherStudent* are not global but are local to *main*. [Do not change the names of the variables.] Additionally, modify the program so that the height of a student in metres is also recorded in the struct and printed by *printStudents*. The extra field, which must be called *height*, should be of type *double* and should be printed out to two decimal places in the form shown below. Agnes McGurkinshaw's height should be 1.64 m and Jingwu Xiao's height should be 1.83 m.

Note *typedefs*, which are introduced in the next section, are not allowed in this question - you must make only minimal changes to *structexample1.c*, above.

Submit your entire modified program.

Note: this question uses a specialised checker that makes the normal *Precheck* option unavailable. Therefore you have two free submissions before penalties start being applied.

For example:

Result
`Agnes McGurkinshaw (97), height 1.64 m` `Jingwu Xiao (21), height 1.83 m`

Answer: (penalty regime: 0, 0, 10, ... %)

```
/*
 * structexample1.c
 * A simple program to introduce C structs
 *
 * Richard Lobb
 * August 2014
 */

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

// Define a structure suitable for representing a student
// in a list of students. In this trivial example only
// the name and age is stored for each student.

// The following declaration declares both the structure type
// and a global instance 'student' of that type.
struct student_s {
    char* name;
    int age;
    double height;
    struct student_s* next;   // Pointer to next student in a list
} student;

// 'struct student_s' is now a type, and we can declare
// another global variable of that type.
struct student_s anotherStudent;

// printOneStudent: prints a single student, passed by value
void printOneStudent(struct student_s student)
{
    printf("%s (%d), height %.2lf m\n", student.name, student.age, student.height);
}

// printStudents: print all students in a list of students, passed
// by reference.
void printStudents(const struct student_s* student)
{
    while (student != NULL) {
        printOneStudent(*student);
        student = student->next;
    }
}

// main just defines fields of the two students, links them
// together into a list, and prints the list.
int main(void)
{
    struct student_s anotherStudent;
    struct student_s student;

    student.name = "Agnes McGurkinshaw";
    student.age = 97;
    student.height = 1.64;
    student.next = &anotherStudent;

    anotherStudent.name = "Jingwu Xiao";
    anotherStudent.age = 21;
    anotherStudent.height = 1.83;
```

```
61      anotherStudent.next = NULL;
62
63      printStudents(&student);
64      return EXIT_SUCCESS;
65  }
66
```

	Expected	Got	
✔	`Agnes McGurkinshaw (97), height 1.64 m` `Jingwu Xiao (21), height 1.83 m`	`Agnes McGurkinshaw (97), height 1.64 m` `Jingwu Xiao (21), height 1.83 m`	✔

Passed all tests! ✔

Correct

Marks for this submission: 3.00/3.00.

Information

2. `structexample2.c`

This modified version of the *structeaxample.c* program is explained in the 5 minute video 6.2 structexample2.c

Inspect the code of the program `structexample2.c` below and compare it with `structexample1.c` There are three changes:

1. The global variables have been made local to `main` as in the previous exercises.
2. A *typedef* has been used to give a name to the type `struct student_s` to enhance readability. `typedef`s are a way of declaring a type alias - the syntax is:
   `typedef oldType newType;`
   Note that in the case of structs, the *typedef* is legal even though the student struct itself hasn't yet been declared.
3. The student variables in the main function have had all their fields initialised within the declarations using a comma-separated list of field values within braces. This special syntax is usable only within initializers, not in general expressions.

Hide `structexample2.c` code

```
/*
 * structexample2.c
 * A development from structexample1 (q.v. for explanations),
 * this time using a typedef for the student structure.
 * The global variables have been moved into "main", too,
 * and have been given initialisers.
 * Otherwise it's the same as the previous example.
 *
 * Author: Richard Lobb
 * Date: August 2014
 */

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

typedef struct student_s Student;

struct student_s {
    char* name;
    int age;
    Student* next;  // Pointer to next student in a list
};


void printOneStudent(Student student)
{
    printf("%s (%d)\n", student.name, student.age);
}


void printStudents(const Student* student)
{
    while (student != NULL) {
        printOneStudent(*student);
        student = student->next;
    }
}


int main(void)
{
    // Declare and initialise the students and the list
    Student student = {"Agnes McGurkinshaw", 97, NULL};
    Student anotherStudent = {"Jingwu Xiao", 21, NULL};
    Student* studentList = NULL;

    // Set up the linked list structure
    student.next = &anotherStudent;
    studentList = &student;
    printStudents(studentList);
    return EXIT_SUCCESS;
}
```

Question **3**

Correct

Mark 3.00 out of 3.00

Write a global declaration of a new type `struct vector2d_s` that has two integer fields `x` and `y`. Such a type can be used to represent a location on a 2D integer grid. Also use *typedef* to define the type alias `Vector2d` for this new type. Follow the type declaration by two function implementations:

1. `Vector2d vector(int x, int y)`, which returns a Vector2d with fields `x` and `y`,
2. `Vector2d vectorSum(Vector2d v1, Vector2d v2)`, which returns the vector sum of v1 and v2.

The names *v1* and *v2* have been made legal in this question.

NB: The answer that you submit should contain *only* the declaration of the structure, the typedef'ed alias, and the two functions.

For example:

Test	Result

`Vector2d v1 = vector(100, -97);`
`Vector2d v2 = vector(11, 1);`
`Vector2d v3 = vectorSum(v1, v2);`
`printf("(%d, %d) + (%d, %d) = (%d, %d)\n",`
`  v1.x, v1.y, v2.x, v2.y, v3.x, v3.y);`
	`(100, -97) + (11, 1) = (111, -96)`

Answer: (penalty regime: 0, 10, ... %)

```
typedef struct vector2d_s Vector2d;


struct vector2d_s {
    int x;
    int y;
};


Vector2d vector(int x, int y)
{
    Vector2d store;
    store.x = x;
    store.y = y;
    return store;
}




Vector2d vectorSum(Vector2d v1, Vector2d v2)
{
    Vector2d summation;
    summation.x = v1.x + v2.x;
    summation.y = v1.y + v2.y;
    return summation;

}
```

	Test	Expected	Got	

✔	`Vector2d v1 = vector(100, -97);`
	`Vector2d v2 = vector(11, 1);`
	`Vector2d v3 = vectorSum(v1, v2);`
	`printf("(%d, %d) + (%d, %d) = (%d, %d)\n",`
	`  v1.x, v1.y, v2.x, v2.y, v3.x, v3.y);`
		`(100, -97) + (11, 1) = (111, -96)`	`(100, -97) + (11, 1) = (111, -96)`	✔

✔	`Vector2d v1 = vector(1, 2);`
	`Vector2d v2 = vector(10, 11);`
	`Vector2d v3 = vectorSum(v1, v2);`
	`printf("(%d, %d) + (%d, %d) = (%d, %d)\n",`
	`  v1.x, v1.y, v2.x, v2.y, v3.x, v3.y);`
		`(1, 2) + (10, 11) = (11, 13)`	`(1, 2) + (10, 11) = (11, 13)`	✔

Passed all tests! ✔

Correct

Marks for this submission: 3.00/3.00.

Information

3. `structexample3.c`

`structexample3.c` below extends the program by reading in an arbitrarily large number of students from a .csv file, constructing a linked list of them. This is a big change from the much simpler earlier version. It is explained in the video 6.3 structexample3.c

One student is read from each input line. Student structs are taken from an array, or "pool" of structs. A complication here is that we must have space somewhere to store each student's name. In the previous programs the names were literals (string constants) created by the compiler. It was sufficient just to store a pointer to each literal within the student struct. Now, however, we have to make space for the names (strings) that we read in. We do that by changing the declaration of the *student* struct so that it contains space for the student's name within it.

Hide `structexample3.c` code

```
/* structexample3.c
 * A development from structexample2, this time reading
 * in an arbitrary number of students from a file, allocating
 * student structures from an array of such structures
 * (often referred to as a "pool").

 * Students could have just been read into the array directly, but
 * then one would lose the benefits of having a linked list
 * structure that allows, for example, cheap insertion and
 * deletion of students. [Note that we don't actually use those wonderful
 * features in this program, but we might do in more advanced
 * applications.]

 * The input file is assumed to be a simple .csv (comma-separated-values)
 * file, with the student in the first column and the age in the second.
 *
 * This is not a very pretty program (string processing in C is never very
 * pretty), but it is instructive :-)
 *
 * Richard Lobb, June 2013.
 */

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

#define MAX_LINE_LENGTH 80      // The longest line this program will accept
#define MAX_NUM_STUDENTS 500    // The maximum number of students this program can handle
#define MAX_NAME_SIZE 50        // The maximum allowable name length

// The declaration of the student record (or struct). Note that
// the struct contains the name as an array of characters, rather than
// containing just a pointer to the name as before.

typedef struct student_s Student;

struct student_s {
    char name[MAX_NAME_SIZE];
    int age;
    Student* next;              // Pointer to next student in a list
};

// Create a pool of student records to be allocated on demand

Student studentPool[MAX_NUM_STUDENTS];  // The student pool
int firstFree = 0;

// Return a pointer to a new student record from the pool, after
// filling in the provided name and age fields. Returns NULL if
// the student pool is exhausted.
Student* newStudent(const char* name, int age)
{
    Student* student = NULL;
    if (firstFree < MAX_NUM_STUDENTS) {
        student = &studentPool[firstFree];
        firstFree += 1;
        strncpy(student->name, name, MAX_NAME_SIZE);
        student->name[MAX_NAME_SIZE - 1] = '\0';  // Make sure it's terminated
        student->age = age;
        student->next = NULL;
    }
    return student;
}

// Read a single student from a csv input file with student name in first column,
// and student age in second.
// Returns: A pointer to a Student record, or NULL if EOF or an invalid
// student record is read. Blank lines, or lines in which the name is
// longer than the provided name buffer, or there is no comma in the line
// are considered invalid.
Student* readOneStudent(FILE* file)
{
    char buffer[MAX_LINE_LENGTH] = {0};  // Buffer into which we read a line from stdin
    Student* student = NULL;       // Pointer to a student record from the pool

    // Read a line, extract name and age

    char* inputLine = fgets(buffer, MAX_LINE_LENGTH, file);
```

```
    if (inputLine != NULL) {        // Proceed only if we read something
        char* commaPos = strchr(buffer, ',');
        if (commaPos != NULL) {
            int age = atoi(commaPos + 1);
            *commaPos = '\0';  // null-terminate the name
            student = newStudent(buffer, age);
        }
    }
    return student;
}

// Reads a list of students from a given file. Input stops when
// a blank line is read, or an EOF occurs, or an illegal input
// line is encountered.
// Returns a pointer to the first student in the list or NULL if no
// valid student records could be read.
Student* readStudents(FILE *file)
{
    Student* first = NULL;     // Pointer to the first student in the list
    Student* last = NULL;      // Pointer to the last student in the list
    Student* student = readOneStudent(file);
    while (student != NULL) {
        if (first == NULL) {
            first = last = student;   // Empty list case
        } else {
            last->next = student;
            last = student;
        }
        student = readOneStudent(file);
    }
    return first;
}

// printOneStudent: prints a single student, passed by value
void printOneStudent(Student student)
{
    printf("%s (%d)\n", student.name, student.age);
}


// printStudents: print all students in a list of students, passed
// by reference
void printStudents(const Student* student)
{
    while (student != NULL) {
        printOneStudent(*student);
        student = student->next;
    }
}


// Main program. Read a linked list of students from a csv file, then display
// the contents of that list.
int main(void)
{
    FILE* inputFile = fopen("studlist.txt", "r");
    if (inputFile == NULL) {
        fprintf(stderr, "File not found\n");
    } else {
        Student* studentList = readStudents(inputFile);
        printStudents(studentList);

        // The program could now do various things that make use of
        // the linked list, like deleting students and adding new ones,
        // but the program is already quite long enough!
    }
}
```

Using a fixed-size array for the pool of student structs and a fixed-size array for each student's name is clearly problematic. How long should each of these be? There is obviously no good answer to that question. In the next lab we'll see how we solve the problem by dynamically allocating memory as we need it, but for now we'll use the fixed-size array approach.

Read carefully through the code and try to understand it. You should draw yourself a picture or two in order to visualise the data structures - you can't program in C if you can't "see" the data structures in your head and you'll never learn to see them if you don't practice by drawing them on paper.

The program opens the file `studlist.txt` using the function `fopen` which corresponds almost exactly to Python's `open` function. The value returned by `fopen` is a pointer to a hidden structure of type `FILE`; you don't need to know anything more about that type than you did when using files in Python. The file is then read one line at a time by calls to the *fgets* function. Type *man fgets* for details. The FILE* object records the current read position in the file and each call to *fgets* reads the next line, starting from where the last call left off.

The `fprintf` function is used to print to a specific file, rather than to `stdout`; the first argument to `fprintf` is the `FILE*` variable returned by `fopen`. A call to `fprintf(stdout, ...)` is exactly equivalent to a call to `printf(...)`. The program also introduces a new `FILE*` variable, `stderr`, which denotes the so-called *standard error* stream. This is used for output of error messages rather than "standard" output.

Now build and run the program making sure that the required input file `studlist.txt` (click the link to download it) is in the same directory as `structexample3.c`. Was the output what you expected?

Note: try to answer the following questions by inspecting the code rather than by mindlessly running it to see what happens. You're certainly welcome to check your predictions in practice but remember that the actual answers to the quiz questions are valueless - it's the understanding that's required to reach those answers that matters.

Question **4**

Correct

Mark 1.00 out of 1.00

Roughly how many bytes of memory does the "studentPool" array require, assuming a 64-bit machine (i.e., a machine in which pointers require 8 bytes).

Select one:

- 31 kB ✔
- 50 kB
- 27 MB
- 10 kB
- 500 kB

Although we can never be sure precisely how much space a compiler will allocate for variables, we would expect that each student record would require 50 bytes for the name, 4 bytes for the age and 8 for the 'next' pointer. That's 62 bytes per student, and there are 500 students. So 31 kB is roughly the amount of space required.

Correct

Marks for this submission: 1.00/1.00.

Question **5**

Correct

Mark 1.00 out of 1.00

What would *structexample3.c* do if you tried to use it to read more than 500 students?

Select one:

- The program would ignore all students after the 500th in the file. ✔
- The program handles an arbitrarily large number of students, so would work correctly.
- The program would crash with a segmentation fault.
- The program would abort with an error message.

Correct

Marks for this submission: 1.00/1.00.

Question **6**

Correct

Mark 1.00 out of 1.00

In the input file, the last line defines a student Lord Albert Augustus Edward Joseph Eldersfield III. But when you run *structexample3*, he appears to be Lord Albert Augustus Edward Joseph Eldersfield II. What happened to that third 'I'?

Select one:

- It wasn't returned by *fgets* because it wouldn't fit in the given line buffer.
- It wasn't copied from the line buffer by *strncpy* in the *newStudent* function.
- It got clobbered by the line `student->name[MAX_NAME_SIZE-1] = '\0';` ✔ Without this line, you would get the extra I at the end! However, the line of code is necessary to guarantee that the name buffer is correctly terminated because *strncpy* doesn't necessarily terminate it correctly if the maximum count is reached (as in this case).
- Because the maximum name length is 50, including the terminating null, the call to *strcspn* resulted in the *nameLength* variable being set to a value of 49.

Correct

Marks for this submission: 1.00/1.00.

Question **7**

Correct

Mark 1.00 out of 1.00

Suppose one of the lines, somewhere in the middle of the input file has a space instead of a comma separating the name and the age. Which of the following statements are correct? [You must select *all* correct statements to get full marks.]

Note that the purpose of the question is get you to learn something by thinking about the code. Please don't answer it just by moronically running the program with modified data. Learning avoidance isn't usually a wise strategy at University.

Select one or more:

- The computed age is zero.
- The offending line is ignored. ✔
- The computed name is missing its last character.
- All input lines after the offending line are ignored. ✔
- The age is still correctly computed, but the name is wrong.
- The name is still correctly computed, but the age is wrong.

Correct

Marks for this submission: 1.00/1.00.

Question **8**

Correct

Mark 3.60 out of 4.00

Modify the *structexample3* program so that

1. It reads input from *stdin* rather than a specified file. This is achieved just by changing the first line of *main* to

```
FILE* inputFile = stdin;
```

   This change makes it easier for us to show you what data we are using to test your program.

2. Each student read is inserted at the front of the list of students, not at the end.

The list of students will now be displayed in reverse order.

For example:

Input	Result
`Fred Nurk, 21` `Arwen Evensong, 927`	`Arwen Evensong (927)` `Fred Nurk (21)`

Answer: (penalty regime: 0, 10, ... %)

```
/* structexample3.c
 * A development from structexample2, this time reading
 * in an arbitrary number of students from a file, allocating
 * student structures from an array of such structures
 * (often referred to as a "pool").

 * Students could have just been read into the array directly, but
 * then one would lose the benefits of having a linked list
 * structure that allows, for example, cheap insertion and
 * deletion of students. [Note that we don't actually use those wonderful
 * features in this program, but we might do in more advanced
 * applications.]

 * The input file is assumed to be a simple .csv (comma-separated-values)
 * file, with the student in the first column and the age in the second.
 *
 * This is not a very pretty program (string processing in C is never very
 * pretty), but it is instructive :-)
 *
 * Richard Lobb, June 2013.
 */

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

#define MAX_LINE_LENGTH 80      // The longest line this program will accept
#define MAX_NUM_STUDENTS 500    // The maximum number of students this program can handle
#define MAX_NAME_SIZE 50        // The maximum allowable name length

// The declaration of the student record (or struct). Note that
// the struct contains the name as an array of characters, rather than
// containing just a pointer to the name as before.

typedef struct student_s Student;

struct student_s {
    char name[MAX_NAME_SIZE];
    int age;
    Student* next;              // Pointer to next student in a list
};

// Create a pool of student records to be allocated on demand

Student studentPool[MAX_NUM_STUDENTS];  // The student pool
int firstFree = 0;

// Return a pointer to a new student record from the pool, after
// filling in the provided name and age fields. Returns NULL if
// the student pool is exhausted.
Student* newStudent(const char* name, int age)
{
    Student* student = NULL;
    if (firstFree < MAX_NUM_STUDENTS) {
        student = &studentPool[firstFree];
        firstFree += 1;
        strncpy(student->name, name, MAX_NAME_SIZE);
        student->name[MAX_NAME_SIZE - 1] = '\0';  // Make sure it's terminated
        student->age = age;
        student->next = NULL;
```

```
    }
    return student;
}

// Read a single student from a csv input file with student name in first column,
// and student age in second.
// Returns: A pointer to a Student record, or NULL if EOF or an invalid
// student record is read. Blank lines, or lines in which the name is
// longer than the provided name buffer, or there is no comma in the line
// are considered invalid.
Student* readOneStudent(FILE* file)
{
    char buffer[MAX_LINE_LENGTH];  // Buffer into which we read a line from stdin
    Student* student = NULL;       // Pointer to a student record from the pool

    // Read a line, extract name and age

    char* inputLine = fgets(buffer, MAX_LINE_LENGTH, file);
    if (inputLine != NULL) {        // Proceed only if we read something
        char* commaPos = strchr(buffer, ',');
        if (commaPos != NULL) {
            int age = atoi(commaPos + 1);
            *commaPos = '\0';  // null-terminate the name
            student = newStudent(buffer, age);
        }
    }
    return student;
}

// Reads a list of students from a given file. Input stops when
// a blank line is read, or an EOF occurs, or an illegal input
// line is encountered.
// Returns a pointer to the first student in the list or NULL if no
// valid student records could be read.
Student* readStudents(FILE *file)
{
    Student* first = NULL;     // Pointer to the first student in the list
    Student* last = NULL;      // Pointer to the last student in the list
    Student* student = readOneStudent(file);
    while (student != NULL) {
        if (first == NULL) {
            first = last = student;   // Empty list case
        } else {
            student->next = first;
            first = student;
        }
        student = readOneStudent(file);
    }
    return first;
}

// printOneStudent: prints a single student, passed by value
void printOneStudent(Student student)
{
    printf("%s (%d)\n", student.name, student.age);
}


// printStudents: print all students in a list of students, passed
// by reference
void printStudents(const Student* student)
{
    while (student != NULL) {
        printOneStudent(*student);
        student = student->next;
    }
}


// Main program. Read a linked list of students from a csv file, then display
// the contents of that list.
int main(void)
{
    FILE* inputFile = stdin;
    if (inputFile == NULL) {
        fprintf(stderr, "File not found\n");
    } else {
        Student* studentList = readStudents(inputFile);
        printStudents(studentList);

        // The program could now do various things that make use of
        // the linked list, like deleting students and adding new ones,
        // but the program is already quite long enough!
    }
}
```

```
147
```

	Input	Expected	Got	
✔	Fred Nurk, 21 Arwen Evensong, 927	Arwen Evensong (927) Fred Nurk (21)	Arwen Evensong (927) Fred Nurk (21)	✔
✔	Albert Humperdink II, 1	Albert Humperdink II (1)	Albert Humperdink II (1)	✔
✔	A, 1 B, 2 C, 3 D, 4	D (4) C (3) B (2) A (1)	D (4) C (3) B (2) A (1)	✔

Passed all tests! ✔

Correct

Marks for this submission: 4.00/4.00. Accounting for previous tries, this gives **3.60/4.00**.

Question **9**

Correct

Mark 4.00 out of 4.00

Using the answer to the previous question as a starting point, further modify the *structexample3* program so that it takes input in a different format: each line consists of an age, a comma, a space and a name, e.g.

```
23, Angus McGurkinshaw
```

Given erroneous input the program should behave as before (see question 7).

For example:

Input	Result
21, Fred Nurk 927, Arwen Evensong	Arwen Evensong (927) Fred Nurk (21)

Answer: (penalty regime: 0, 10, ... %)

```
/* structexample3.c
 * A development from structexample2, this time reading
 * in an arbitrary number of students from a file, allocating
 * student structures from an array of such structures
 * (often referred to as a "pool").

 * Students could have just been read into the array directly, but
 * then one would lose the benefits of having a linked list
 * structure that allows, for example, cheap insertion and
 * deletion of students. [Note that we don't actually use those wonderful
 * features in this program, but we might do in more advanced
 * applications.]

 * The input file is assumed to be a simple .csv (comma-separated-values)
 * file, with the student in the first column and the age in the second.
 *
 * This is not a very pretty program (string processing in C is never very
 * pretty), but it is instructive :-)
 *
 * Richard Lobb, June 2013.
 */

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

#define MAX_LINE_LENGTH 80      // The longest line this program will accept
#define MAX_NUM_STUDENTS 500    // The maximum number of students this program can handle
#define MAX_NAME_SIZE 50        // The maximum allowable name length

// The declaration of the student record (or struct). Note that
// the struct contains the name as an array of characters, rather than
// containing just a pointer to the name as before.

typedef struct student_s Student;

struct student_s {
    char name[MAX_NAME_SIZE];
    int age;
    Student* next;              // Pointer to next student in a list
};

// Create a pool of student records to be allocated on demand

Student studentPool[MAX_NUM_STUDENTS];  // The student pool
int firstFree = 0;

// Return a pointer to a new student record from the pool, after
// filling in the provided name and age fields. Returns NULL if
// the student pool is exhausted.
Student* newStudent(const char* name, int age)
{
    Student* student = NULL;
    if (firstFree < MAX_NUM_STUDENTS) {
        student = &studentPool[firstFree];
        firstFree += 1;
        strncpy(student->name, name, MAX_NAME_SIZE);
        student->name[MAX_NAME_SIZE - 1] = '\0';  // Make sure it's terminated
        student->age = age;
        student->next = NULL;
    }
    return student;
}

```

```
66  // Read a single student from a csv input file with student name in first column,
67  // and student age in second.
68  // Returns: A pointer to a Student record, or NULL if EOF or an invalid
69  // student record is read. Blank lines, or lines in which the name is
70  // longer than the provided name buffer, or there is no comma in the line
71  // are considered invalid.
72  Student* readOneStudent(FILE* file)
73  {
74      char buffer[MAX_LINE_LENGTH];  // Buffer into which we read a line from stdin
75      Student* student = NULL;       // Pointer to a student record from the pool
76
77      // Read a line, extract name and age
78
79      char* inputLine = fgets(buffer, MAX_LINE_LENGTH, file);
80      //printf("buffer is %s hoho\n", buffer);
81      if (inputLine != NULL) {        // Proceed only if we read something
82          char* commaPos = strchr(buffer, ',');
83          char* endOfLine = strchr(buffer, '\n');
84          *endOfLine = '\0';
85          if (commaPos != NULL) {
86              int age = atoi(buffer);
87              //printf("age is %d", age);
88              *commaPos = '\0';  // null-terminate the name
89              //printf("commaPos is %s", (commaPos + 2));
90              student = newStudent((commaPos + 2), age);
91          }
92      }
93      return student;
94  }
95
96  // Reads a list of students from a given file. Input stops when
97  // a blank line is read, or an EOF occurs, or an illegal input
98  // line is encountered.
99  // Returns a pointer to the first student in the list or NULL if no
100 // valid student records could be read.
101 Student* readStudents(FILE *file)
102 {
103     Student* first = NULL;     // Pointer to the first student in the list
104     Student* last = NULL;      // Pointer to the last student in the list
105     Student* student = readOneStudent(file);
106     while (student != NULL) {
107         if (first == NULL) {
108             first = last = student;   // Empty list case
109         } else {
110             student->next = first;
111             first = student;
112         }
113         student = readOneStudent(file);
114     }
115     return first;
116 }
117
118 // printOneStudent: prints a single student, passed by value
119 void printOneStudent(Student student)
120 {
121     printf("%s (%d)\n", student.name, student.age);
122 }
123
124
125 // printStudents: print all students in a list of students, passed
126 // by reference
127 void printStudents(const Student* student)
128 {
129     while (student != NULL) {
130         printOneStudent(*student);
131         student = student->next;
132     }
133 }
134
135
136 // Main program. Read a linked list of students from a csv file, then display
137 // the contents of that list.
138 int main(void)
139 {
140     FILE* inputFile = stdin;
141     if (inputFile == NULL) {
142         fprintf(stderr, "File not found\n");
143     } else {
144         Student* studentList = readStudents(inputFile);
145         printStudents(studentList);
146
147         // The program could now do various things that make use of
148         // the linked list, like deleting students and adding new ones,
149         // but the program is already quite long enough!
150     }
```

```
151 }
152
```

	Input	Expected	Got	
✔	21, Fred Nurk 927, Arwen Evensong	Arwen Evensong (927) Fred Nurk (21)	Arwen Evensong (927) Fred Nurk (21)	✔
✔	1, Albert Humperdink II	Albert Humperdink II (1)	Albert Humperdink II (1)	✔
✔	1, A 2, B 3, C 4, D	D (4) C (3) B (2) A (1)	D (4) C (3) B (2) A (1)	✔

Passed all tests! ✔

Correct

Marks for this submission: 4.00/4.00.

Information

Debugging in C

As you know by now, C is a much less forgiving programming language than Python or Java. When you have a bug in your code, the result is quite often just *Segmentation fault, core dumped*, or, worse, quietly wrong answers.

Usually the best way to debug C programs is to add lots of extra *printf* statements to allow you to trace the execution of your program. To do this you have to ask yourself questions like "What might be going wrong? How can I check if that's the case? What variables should I print?" Such questions focus your mind on your code and likely issues with it, which is usually a better first step than mindlessly firing up a debugger and single-stepping through the code like an automaton.

However, there certainly *are* cases where debuggers are useful, and you should at least be aware of their existence so you can decide on the best approach.

There are (at least) four C debugging tools installed on the lab machines:

1. *gdb*, the *GNU debugger*. This is the classic command-line debugger loved or hated by many generations of programmers. To run it, type *gdb ./executablefilename*. You debug by typing commands to it, such as *run*, *print*, *step*, *next*, *break* etc. If you want to run the program with standard input redirected to a file, type the run command in the form *run < inputfile.txt*.
2. *ddd*, the Data Display Debugger. This is a GUI wrapper around *gdb*. You can still type *gdb* commands directly but you can also use GUI control panels, double click variables to see their values, etc. It even displays pointers between structs as arrows, which makes it almost unique in the world of debuggers. The downside is its almost unbelievably clunky look and feel, but if you can overcome your horror at that (and various other quirks) it's actually a rather nice tool.
3. *kdbg*, the KDE debugger. This is another GUI wrapper around *gdb*, this time using the KDE (K Desktop Environment) toolkit. It looks much more professional than *ddd* but doesn't do graphical display of linked structures. It's worth a look, although for installing on a home Linux you'll need to download the entire KDE environment, which is a lot of code.
4. *valgrind*. This tool is invaluable for debugging programs that use dynamic memory (next week's topic). It tracks all your accesses into the heap and detects buffer overruns and underruns (within the heap) and memory leaks. The quiz server often runs your code under the protection of valgrind to help detect errors that might otherwise go undetected.

All of these tools can be used to help debug a program with a command of the form *toolname ./progname*, e.g. *gdb ./structexample3*, *ddd ./structexample3*, *valgrind ./structexample3.* If used to run a program that is seg-faulting, all of them will tell you the line in your code at which you crashed (provided you compiled with the *-g* flag).

You're encouraged to try out each of these tools with *structexample3*. In particular try using *ddd* to get a display of the linked list of students directly after the call to *readStudents*.

Optional Exercise

The following program, `cards.c`, was originally written to demonstrate the use of enums. These are being downplayed in the course this year, though they're still in the lecture notes. Although the details of the program are not the focus of this exercise, you are encouraged to look at the code to see how enums can be used. But what's important right now is that the program has a deliberate bug.

Build and run `cards`. On a lab machine you will probably (but not necessarily) get a crash as a result of a segmentation fault. Run the program under the control of each of the above tools to identify the line at which the code crashes. If you're feeling keen, you might want to see if you can identify the actual bug.

Hide `cards.c` code

```
/* Example of the use of C enumerations (enums)
 * Creates a deck of cards, shuffles it, then prints out the
 * shuffled deck.
 *
 * Richard Lobb, August 2014.
 */

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

/* Declare the prototype for drand48, which is a non-ANSI library
 * function.
 */
double drand48(void);

#define NUM_SUITS 4
#define NUM_IN_SUIT 13
#define NUM_IN_DECK 52
#define NUM_SWAPS 10000

typedef enum {SPADES, HEARTS, DIAMONDS, CLUBS} Suit;
typedef enum {ACE=1, JACK=11, QUEEN, KING} Rank;

char* suits[] = {"spades", "hearts", "diamonds", "clubs"};
char* ranks[]  = {"UNDEFINED", "ace", "2", "3", "4", "5", "6", "7", "8",
                  "9", "10", "jack", "queen", "king"
                 };

typedef struct {
    Rank rank;
    Suit suit;
} Card;


/* Shuffle a deck of cards by making a large number of swaps
 * of two randomly selected cards in the deck */
void shuffle(Card* deck)
{
    int swap = 0;
    int i = 0, j = 0;
    Card temp = {ACE, SPADES};
    for (swap = 0; swap < NUM_SWAPS; swap += 1) {
        i = drand48() * NUM_IN_DECK;   // Random in [0,51]
        j = drand48() * NUM_IN_DECK;
        temp = deck[i];
        deck[i] = deck[j];
        deck[j] = temp;
    }
}

// main program: create a deck, shuffle it and display the result
int main()
{
    Card deck[NUM_IN_DECK];
    int cardNum = 0;

    for (Suit suit = SPADES; suit <= CLUBS; suit += 1) {
        for (Rank rank = ACE; rank < KING; rank += 1) {
            deck[cardNum].suit = suit;
            deck[cardNum].rank = rank;
            cardNum += 1;
        }
    }

    shuffle(deck);
    printf("Shuffled deck:\n");
    for (int iCard = 0; iCard < NUM_IN_DECK; iCard += 1) {
        printf("%s of %s\n", ranks[deck[iCard].rank],
                             suits[deck[iCard].suit]);
    }
    return EXIT_SUCCESS;
}
```

Information

Multi-file programs

All the C programs we've seen so far have resided in a single file. However, this approach will obviously be unmanageable with larger programs, and we need to find a way to structure our source code into separate files, which we can develop and debug separately.

Firstly, watch the two videos 7.1 Introducing modularisation and 7.2 Modularising structexample3.c.

Then download, unzip and inspect the contents of the archive processstudents.zip. You'll see it contains the code of `structexample3.c`, broken into three separate compilation units, which we'll refer to as modules: the *student* module, the *studentlist* module and the main program (which is in the file `processstudents.c`). Inspect the code carefully, starting with `student.c` and `student.h`, noting the following:

1. The "header" file, `student.h`, contains the declaration of the *Student* type plus the declarations of the functions that deal with individual student records. The definitions of the functions are in `student.c`. [A function *declaration* consists of just the function prototype or "signature" without the actual body, where the *definition* includes the body of the function.] In a loose sense you can think of `student.h` and `student.c` together as defining a class *Student*. The header file defines the *interface* to the class while the C file is the *implementation* of it.
2. Similarly you can think of `studentlist.h` and `studentlist.c` together as defining a class *StudentList,* with "methods" for reading and printing a list of students.
3. Although header files can contain any code, the usual convention is to put only declarations in them, keeping all definitions of variables and functions in .c files.
4. There is nothing magical about the use of the extension ".h". We could have called the header file `student.header`, `student.decls` or `student.tiddlyPom` but by convention we always use the extension "`.h`".
5. If one module uses functions from another module, it should #include the header file from the used module. This ensures that the compiler knows the type signatures of all the called functions. However, that creates a potential problem. If the main program #includes both `studentlist.h` and `student.h` and `studentlist.h` itself #includes `student.h`, won't we finish up with two copies of `student.h` in the one compilation unit for the main program? The answer unfortunately is "yes". So... to prevent this - it would almost inevitably cause errors - we need to make use of a new preprocessor directive: `#ifndef somesymbol` ... `#endif`. This directive should be read as "if somesymbol is not defined ...", and results in the body of the the document being included only if the symbol isn't currently defined. Since the first thing the included block does is define the specified symbol, further #includes of that file will not actually contain any text. This use of #ifndef is called an *include guard*. For more details, see `http://en.wikipedia.org/wiki/Include_guard`.

To see how a new version of the program could be built by separately compiling each module then linking the object files together, open a terminal window, change into the new directory (command *cd your/directory/path*) , and type the following sequence of commands, observing what files are present at each stage of the process. Don't bother typing the comments, which begin with '#' - they are just for explanation.

```
ls   # Directory listing at the start of the process
gcc -g -std=c99 -Wall -c student.c   # Compile the student module
ls   # What new relocatable object file was produced?
gcc -g -std=c99 -Wall -c studentlist.c  # Compile studentlist module
ls   # What new file this time?
gcc -g -std=c99 -Wall -c processstudents.c  # Compile main program
ls   # Surely you can guess this time!
gcc -o processstudents student.o studentlist.o processstudents.o # Link it all
ls   # Now you have a main program, right?
rm *.o  # "Clean up" by removing all the .o files
./processstudents  # Run the program
```

Each compilation step separately compiles a single `.c` file, using the -c option so that a relocatable object file is produced but no attempt is made to link the result into an executable.

Note that the line with the comment "Link it all" isn't actually a compilation step, even though it invokes the gcc compiler. Because there are no source files in the input to the compiler - only object files - the compiler just passes the input files on to the linker, `ld`, to link with the C library files. Direct invocation of the linker would be possible for this step, but finding the correct set of library files is problematic, so it's easiest to let the compiler do it.

Question **10**

Correct

Mark 1.00 out of 1.00

Select menu items so that the following paragraph makes sense. Be warned that you must choose all the correct options to get marks in this question.

In the command `gcc -g -std=c99 -Wall -c student.c`

gcc ✔ is the name of the C compiler, which is compiling the program student.c ✔ (which includes the header file student.h ✔ via the C preprocessor, cpp ✔). The -c ✔ option requests that the program be compiled but not linked. The output from the compilation is a relocatable object file ✔ , which is named student.o ✔ .

Correct

Marks for this submission: 1.00/1.00.

Question **11**

Correct

Mark 1.00 out of 1.00

Which of the following statements best describes the purpose of the command line below?

```
gcc -o processstudents student.o studentlist.o processstudents.o
```

Select one:

- It links the three .o files to produce an executable file called 'processstudents'. ✔
- It compiles the three .o files and processstudents.c to produce an executable output file.
- It compiles the three .o files, the two #included .h files and processstudents.c to produce an executable output file.
- It compiles the three .o files to produce an executable file called 'processstudents'.
- It compiles the three .o files and the two #included .h files to produce an executable file called 'processstudents'.

Correct

Marks for this submission: 1.00/1.00.

Information

Why bother with separate compilation?

You might be wondering why we're bothering compiling each file separately when we can compile and link a set of .c source files in a single command (see below). The key idea is that when that file is changed **only that module** would need to be compiled and then the whole program can be relinked. Large systems may contain hundreds or even thousands of source files, and the saving in time from compiling only the changed modules can be enormous. However, this benefit can only be realised if the interface to the changed module, that is, it's *.h* header, remains unchanged. If a *.h* header is changed, all modules that `#include` that header will also need to be recompiled.

Managing of the dependencies between various source files so as to ensure that only the minimum number of files are recompiled is usually done by means of a *makefile* - see later.

In this section of the course, the programs will be small enough that it's not worth trying to avoid compiling unchanged code; we can just compile and link everything after each source code change. For our current *processstudents* program, compiling and linking of all files can be done with the single command

```
gcc -std=c99 -Wall -o processstudents student.c studentlist.c processstudents.c
```

That command says to compile (and link, since the *-c* option is omitted), the three `.c` files, putting the output into an executable file called `processstudents`. You might want to set that command as the *Build* command in *Geany*, but please realise that you'll need to change it for each new program you build from then on. So please make sure you know what you're doing!

In the next section of the course, where you will be dealing with many more source files, separate compilation will become important for efficiency reasons.

Question **12**

Correct

Mark 1.00 out of 1.00

If you had to change the *processstudents* program to keep the student list sorted, you would need to modify only one of the source files, namely studentlist.c ✔ . The studentlist ✔ module would need to be recompiled. Also, you would need to rebuild the executable program, for example by repeating the command that was given earlier with the comment Link it all ✔ .

Correct

Marks for this submission: 1.00/1.00.

Question **13**

Correct

Mark 1.00 out of 1.00

If you changed just `studentlist.h` and recompiled using the command `gcc -c *.c`, all source files would be recompiled. However that recompiles some files that do not require recompilation. What are the files that DO require recompilation when a change to `studentlist.h` is made?

Select one:

- None of them.
- student.c
- All of them.
- processstudents.c
- studentlist.c and processstudents.c ✔
- student.c and processstudents.c
- student.c and studentlist.c
- studentlist.c

Correct

Marks for this submission: 1.00/1.00.

Question **14**

Correct

Mark 1.00 out of 1.00

Modifying the *studentlist* module

The main program, *processstudents.c,* simply reads and prints a list of students. Suppose that instead of printing the entire list of students, we instead wanted to print only the information for a particular student whose name is entered from standard input. With our current very simple *student* struct, the only extra information displayed will be the age.

The modified processstudents.c might then be as follows:

```
#include <stdio.h>
#include <string.h>
#include "student.h"
#include "studentlist.h"

int main(void)
{
    char name[MAX_NAME_SIZE] = {0};
    const Student* student = NULL;
    FILE* inputFile = fopen("studlist.txt", "r");
    if (inputFile == NULL) {
        fprintf(stderr, "File not found\n");
    } else {
        Student* studentList = readStudents(inputFile);
        puts("Enter the name of a student");
        fgets(name, MAX_NAME_SIZE, stdin); // Reads a line from stdin. Trust me
        if (name[strlen(name) - 1] == '\n') {
            name[strlen(name) - 1] = '\0';
        }
        student = findStudent(studentList, name);
        if (student == NULL) {
            printf("'%s' not found\n", name);
        } else {
            printOneStudent(*student);
        }
    }
}
```

Here, the *findStudent* function is a function that should be defined as part of the *studentlist* module. This means that the declaration should be in *studentlist.h* and the definition in *studentlist.c*. The function signature should be

```
const Student* findStudent(const Student* studentList, const char* name);
```

In this question your job is to implement the *findStudent* function by modifying *studentlist.h* and *studentlist.c* in the appropriate way so that the modified *processstudents* program behaves as shown in the example table below.

How to submit your answer

This question uses a different approach from any CodeRunner questions you have met so far. For this question the code for the modified *processStudents.c* is preloaded into the answer box. **You must not modify it. Any changes you make will be ignored.** Instead, you should attach your modified *studentlist.h* and *studentlist.c* using the *Attachments* box underneath the answerbox. The unmodified files *student.h* and *student.c* are automatically included when testing, as is the data file *studlist.txt*.

Notes

1. Remember that when testing C CodeRunner questions in geany you will usually use the keyboard for standard input. Everything you type on the keyboard is echoed to standard output. Thus for example when you run the above program with an appropriate *studentlist* module, you might see on the screen output like:

   ```
   Enter the name of a student
   Richard Lobb
   Richard Lobb (101)
   ```

   where the characters in bold font are what you typed on the keyboard.

   However, if you instead take the standard input from a file, as CodeRunner does when testing your code, the input is not echoed to standard output, so instead you see just the two lines of output shown in the example table below.

2. If you select Lord Albert Augustus Edward Joseph Eldersfield III, he should be displayed as Lord Albert Augustus Edward Joseph Eldersfield II (7). This is because his full name (ending with III rather than II) is 50 characters long but the student struct has a name buffer of exactly 50 characters. The last 'I' character is overwritten by the null terminating character.

3. The file *studlist.txt* used to test your program is an extended version of the one supplied earlier. This one has some extra students included with names that are very similar to students in the earlier version. If your code is correct, you shouldn't ever see the extra students. If you do see the extra ones, your check for name equality must be flawed. How are you testing for equality of two strings?

For example:

Input	Result
Richard Lobb	Enter the name of a student Richard Lobb (101)
Tweedledum	Enter the name of a student 'Tweedledum' not found

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```
#include <stdio.h>
#include <string.h>
#include "student.h"
#include "studentlist.h"

int main(void)
{
    char name[MAX_NAME_SIZE] = {0};
    const Student* student = NULL;
    FILE* inputFile = fopen("studlist.txt", "r");
    if (inputFile == NULL) {
        fprintf(stderr, "File not found\n");
    } else {
        Student* studentList = readStudents(inputFile);
        puts("Enter the name of a student");
        fgets(name, MAX_NAME_SIZE, stdin); // Reads a line from stdin. Trust me
        if (name[strlen(name) - 1] == '\n') {
            name[strlen(name) - 1] = '\0';
        }
        student = findStudent(studentList, name);
        if (student == NULL) {
            printf("'%s' not found\n", name);
        } else {
            printOneStudent(*student);
        }
    }
}
```

studentlist.c

studentlist.h

	Input	Expected	Got	
✔	Richard Lobb	Enter the name of a student Richard Lobb (101)	Enter the name of a student Richard Lobb (101)	✔
✔	Tweedledum	Enter the name of a student 'Tweedledum' not found	Enter the name of a student 'Tweedledum' not found	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **15**

Correct

Mark 1.00 out of 1.00

The answer box has been preloaded with a program that reads a line of text from standard input, then reads an integer, and outputs an encoded version of the input line using a mysterious function called *confab*. You are not expected to know (or care) about the exact operation of *confab*. Instead you are required to extract the *confab* function into a separate module, which will consist of two files *confab.h* and *confab.c*. The header file must contain only the **declaration** of *confab*; its **definition** will be in confab.c. [If you don't yet know the difference between a declaration and a definition, now would be a good time to find out.]

To submit your answer you should remove the *confab* function from the main program and attach your files *confab.h* and *confab.c* using the *Attachments* widget below the answer box. You will also need to add the following line to the main program:

```
#include "confab.h"
```

Note: the program uses the function *ceil* from the math library. As with the *sqrt* function in lab 1, you will need to link the program with the *-lm* flag. This must go at the end of the linker command, after all the object files.

For example:

Input	Result
`Don't wait until the last day before starting.` `3`	`D'wtnlhltabo ai.ota t ea yersrnn iuit sd fettg`
`ALL HAIL CAESAR` `2`	`ALHI ASRL ALCEA`

Answer: (penalty regime: 0, 10, ... %)

Reset answer

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "confab.h"
#define MAX_LINE_SIZE 1000

int main(void)
{
    char line[MAX_LINE_SIZE] = {0};
    char codedLine[MAX_LINE_SIZE] = {0};
    int rows = 0;

    fgets(line, MAX_LINE_SIZE, stdin);  // Read a line from stdin
    line[strlen(line) - 1] = '\0';      // Replace the newline terminator with a null byte
    scanf("%d", &rows);                 // Read a mysterious number
    confab(line, rows, codedLine);      // Encode the input line
    printf("%s\n", codedLine);          // Print the encoded string
}

```

confab.c

confab.h

	Input	Expected	Got	
✔	`Don't wait until the last day before starting.` `3`	`D'wtnlhltabo ai.ota t ea yersrnn iuit sd fettg`	`D'wtnlhltabo ai.ota t ea yersrnn iuit sd fettg`	✔

	Input	Expected	Got	
✔	ALL HAIL CAESAR 2	ALHI ASRL ALCEA	ALHI ASRL ALCEA	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

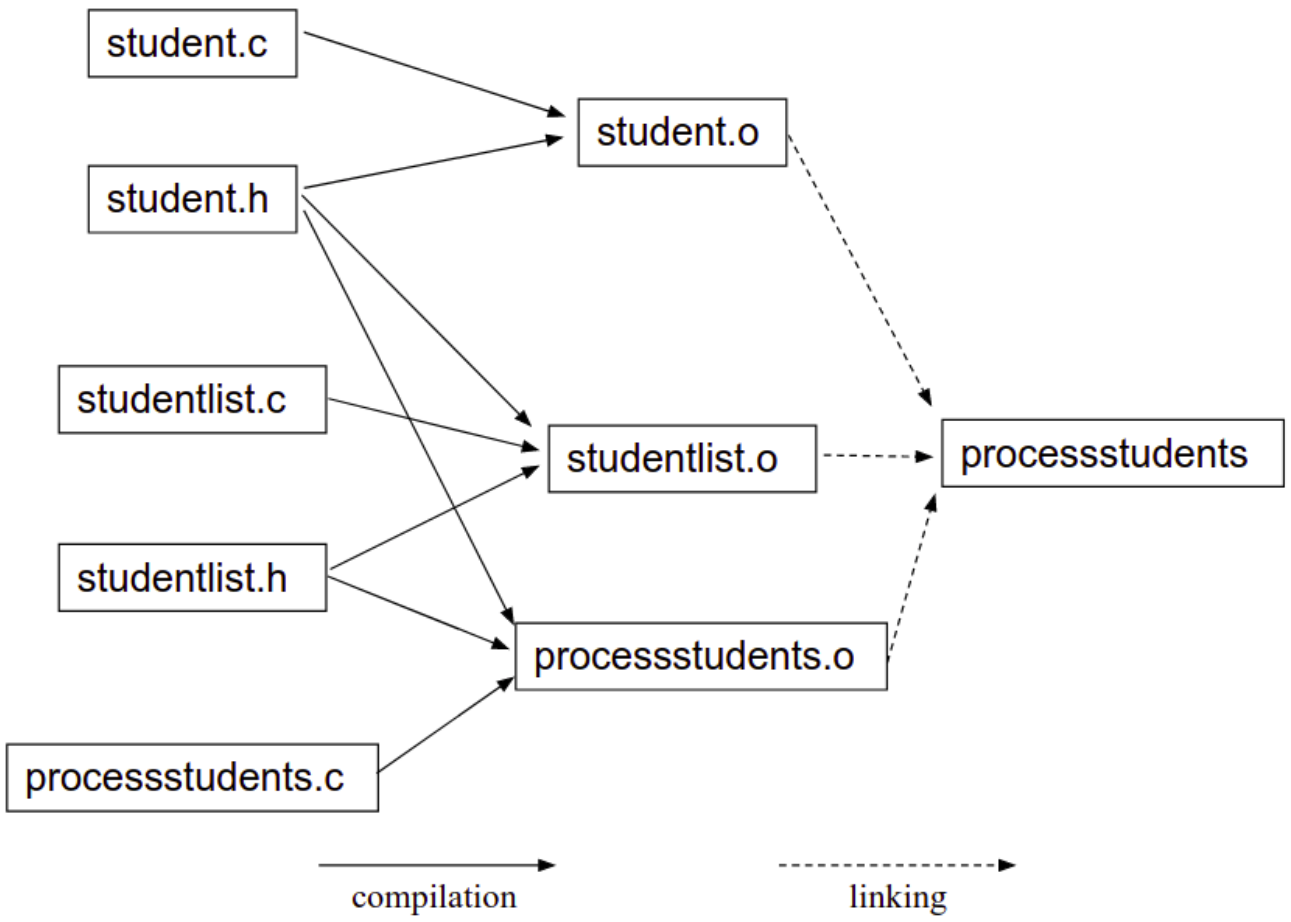
Introducing *Makefiles*

Makefiles are introduced in the video 7.3 Makefiles. For full documentation of *make*, see the GNU documentation here.

The ability to compile files separately, as shown above, can give large benefits in compile and build times. Also, by breaking the system into separate modules, each with its own interface (.h) file, large systems become much more maintainable. However, it also becomes very difficult to keep track of the dependencies; knowing which files need to be rebuilt after any modifications becomes very difficult very quickly.

Building of large systems is generally performed by programs like `make` or `ant`. These programs take as input a textual description of the dependency graph for all the modules - in the case of `make` it's called a *makefile* - and by inspecting the *last-modified* timestamps on all the files they can perform a minimal set of compilations.

Inspect the supplied makefile in the `ProcessStudents` directory and make sure you understand it; it's the same as the version discussed in lectures. The dependence graph, reproduced from the lecture notes, is:



Delete any .o files and the executable `processstudents` program, then type the command *make* in the terminal window; obviously (?) you must be in the `ProcessStudents` directory for this to work. Observe the commands that are executed.

Now edit just `student.c` in some trivial way, e.g. by adding a space to the end of a line. Then type the *make* command again. Which commands are executed now?

Question **16**

Correct

Mark 1.00 out of 1.00

Select all the correct statements regarding the *make* program from the following list. Where appropriate, assume it is being used in the context of the *processstudents* program.

Select one or more:

- The *Makefile* defines a dependency graph and gives the rules for building the nodes in it. ✔
- If you run the command *make* with no parameters and without a *Makefile* present in the current directory, *make* compiles all the .c files in the directory.
- If you run a command like *make thing*, and a *Makefile* is present, *make* tries to build the node *thing* in the dependency graph. ✔
- If you run a command like *make thing*, and no *Makefile* is present, *make* looks for a source file from which it might be able to construct an executable *thing*, such as *thing.c, thing.cpp* etc and then tries to build an executable program from that source file using default build rules. ✔ That's correct. *make* has built-in default rules for various languages, including C, C++, Pascal, Fortan and assembler.
- The rule lines that specify how to build nodes in a *Makefile* must begin with a TAB ('\t') character. ✔ That's correct. And very annoying it is, too, when you're using editors that replace tabs with spaces! Fortunately *geany* at least is smart enough to recognise that Makefiles are a special case and does not replace tabs with spaces.

Your answer is correct.

Correct

Marks for this submission: 1.00/1.00.

Question **17**

Correct

Mark 1.00 out of 1.00

The confab program again

This question has the same spec as Q15 with one important change - you must now attach a *Makefile* for your confab-encoding project in addition to *confab.h* and *confab.c*. The contents of the answer box will need to be modified exactly as before and your confab.h and confab.c attachments should be the same, too. The addition of the *Makefile* is the only change.

The contents of the modified answer box will be saved as *coder.c* and your *Makefile* must have *coder* as its primary target, so that the command

```
make coder
```

rebuilds the coder program, recompiling only when necessary, according to whatever files have been updated. A range of different scenarios will be run when testing, checking that the minimum of work is done in each case.

There must also be a target *clean* that removes all .o files from the working directory.

Important Note on TAB characters

The (ancient) *Makefile* specification requires that the command lines associated with a particular target are indented with a single TAB character ('\t'). However, most GUI editors including *geany* are (or should be) set to replace TAB characters with a suitable number of space characters, which immediately breaks Makefiles.

There are at least three solutions:

1. Use a command line editor like *nano* or *vim* to create and edit your *Makefile*.
2. Insert the following line at the start of your *Makefile*

   ```
   .RECIPEPREFIX = +
   ```

   and start all your command lines with a '+' instead of a TAB. Note the full stop at the start of the word .RECIPEPREFIX
3. Temporarily re-configure *geany* to allow tabs (not recommended).

For example:

Input	Result
`Don't wait until the last day before starting.` `3`	`D'wtnlhltabo ai.ota t ea yersrnn iuit sd fettg`
`ALL HAIL CAESAR` `2`	`ALHI ASRL ALCEA`

Answer: (penalty regime: 0, 10, ... %)

Reset answer

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "confab.h"
#define MAX_LINE_SIZE 1000

int main(void)
{
    char line[MAX_LINE_SIZE] = {0};
    char codedLine[MAX_LINE_SIZE] = {0};
    int rows = 0;

    fgets(line, MAX_LINE_SIZE, stdin);  // Read a line from stdin
    line[strlen(line) - 1] = '\0';      // Replace the newline terminator with a null byte
    scanf("%d", &rows);                 // Read a mysterious number
    confab(line, rows, codedLine);      // Encode the input line
    printf("%s\n", codedLine);          // Print the encoded string
}

```

confab.c

confab.h

Makefile

	Input	Expected	Got	
✔	`Don't wait until the last day before starting.` `3`	`D'wtnlhltabo ai.ota t ea yersrnn iuit sd fettg`	`D'wtnlhltabo ai.ota t ea yersrnn iuit sd fettg`	✔
✔	`ALL HAIL CAESAR` `2`	`ALHI ASRL ALCEA`	`ALHI ASRL ALCEA`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

◄ Quiz 4: Pointers and Strings

Jump to...

Quiz6: Dynamic Memory ►