6. Structuring data

- C doesn't have classes but *does* allow you to group data items together
- Has a "data record" called a *struct*
- Think of it as an object without methods

structexample1.c

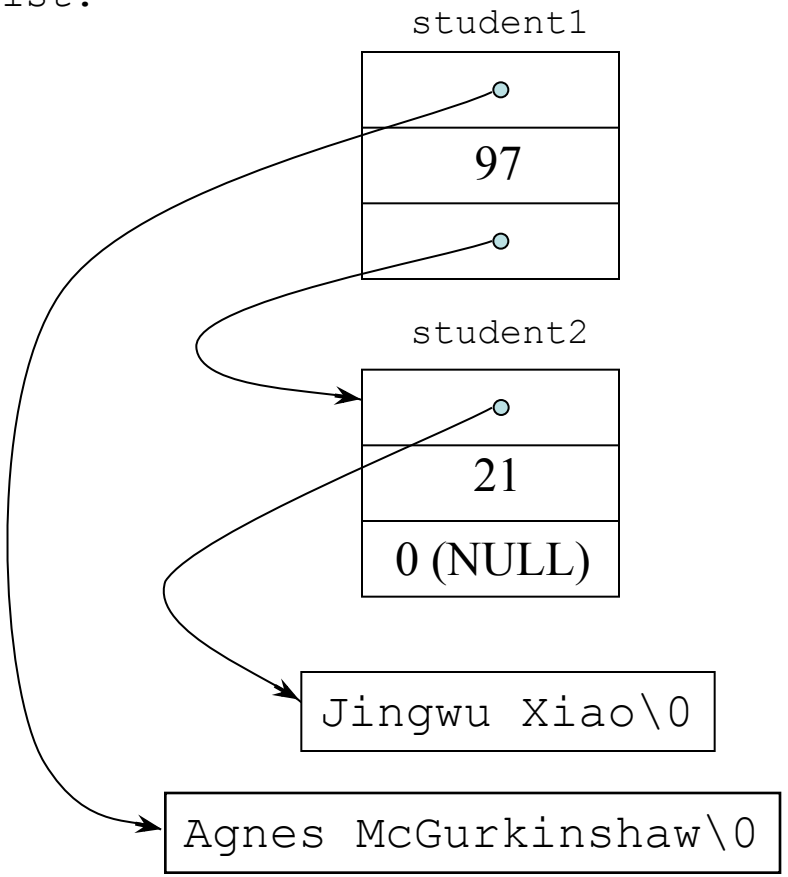
```
#include <stdio.h>
#include <stdlib.h>

/* Define a structure suitable for representing a student
 * in a linked 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
} student1;

// 'struct student_s' is now a type, and we can declare
// another global variable of that type.
struct student_s student2;
```

structexample1.c (cont'd)

```
// 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;
    };
}
```

structexample1.c *(cont'd)*

```
// main just defines the fields of the two students, links them
// together into a list, and prints the list.
int main(void)
{
    student1.name = "Agnes McGurkinshaw";
    student1.age = 97;
    student1.next = &student2;

    student2.name = "Jingwu Xiao";
    student2.age = 21;
    student2.next = NULL;

    printStudents(&student1);
}
```



Points to note

- Weird syntax:
  - **struct** <tag> { <field> ... } <identifier> ...

    A type — Variables of that type
- *struct* variables are not references
  - They're allocated memory when they're declared/defined
    - Either globally or within the stack frame
  - They're not in the heap
    - Unless you *malloc* space for them – see later
- My convention is to add the suffix "_s" to tag names to distinguish them from "normal" identifiers/types.
  - They can't conflict as they're in separate namespaces but *geany* doesn't seem to know this :-(

Points to note (cont'd)

- Dot notation selects fields of struct, e.g., `aStruct.field`
- `aStructPtr->field` is short for `(*aStructPtr).field`

structexample2.c

Uses *typedef*s and initialisers to make the code more readable. Variables now local to main.

```
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 student1 = {"...", 97, NULL};
     Student student2 = {"......", 21,
                                    NULL};
     Student* studentList = NULL;

     student1.next = &student2;
     studentList = &student1;
     printStudents(studentList);
     return EXIT_SUCCESS;
}
```

A larger example

We now extend the program to read students from a CSV file into a linked list.

File format: 1 student per line, name in column 1, age in column 2

Raises issues:

- Opening and reading a file (other than standard input)
- Processing the text lines to build a student struct
- Allocating space for an unknown number of such structs

structexample3.c

Reads an arbitrary no. of students from a file.
Allocates student structs from a *pool*

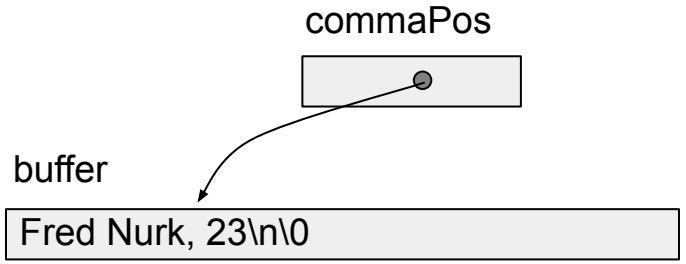
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_NUM_STUDENTS 10000
#define MAX_LINE_LENGTH 1000
#define MAX_NAME_SIZE 50        // The maximum allowable name length
// The declaration of the student record (or struct). Note that
// the struct contains the name as an array of 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 list
};

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

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

structexample3.c (cont'd)

```
// 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';  // Ensure termination
        student->age = age;
        student->next = NULL;
    }
    return student;
}
```

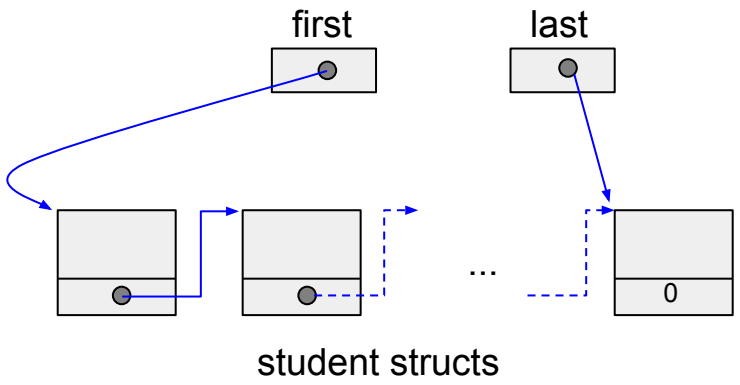
structexample3.c (cont'd)

```
// 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 a bad student record is read.
Student* readOneStudent(FILE* file)
{
    char buffer[MAX_LINE_LENGTH] = {0}; // Line buffer
    Student* student = NULL;           // Pointer to a student record

    // Read a line, extract name and age

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



structexample3.c (cont'd)

```
// Reads a list of students from a given file. List is terminated
// by a blank line or EOF or bad data.
// 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 first student in list
    Student* last = NULL;       // Pointer to last student in 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;
}
```



structexample3.c (cont'd)

```
// 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;
    };
}
```

structexample3.c (cont'd)

```
// 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");
    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!

    return EXIT_SUCCESS;
}
```

Another data structure: *enum*

Rather than this

```
#define SPADES 0
#define DIAMONDS 1
#define CLUBS 2
#define HEARTS 3
```

write this

```
typedef enum {SPADES, HEARTS, DIAMONDS, CLUBS} Suit;
```

Can then write:

```
Suit suit = HEARTS;
.
.
switch (suit) {
    case SPADES: ...
    case HEARTS: ...
    etc
}
```

The enumeration constants are actually just thinly disguised *int*s. Can still write:

```
suit = 3; /*  ☹  */
```

[But not in C++ ☺]