C Final Test

Imperial College London

Department of Computing

Department Lists

Monday 10th June 2024
10:00 to 13:00
THREE HOURS
(including 10 minutes optional planning time)

- You can edit, compile and run your code using the terminal or an IDE like **VScode** or **CLion**. It is strongly recommended that you use **CBuild (CB)** to compile your code. No help will be available for IDEs, and no Makefiles are provided.
- There are **4 tasks, with 10, 40, 30 and 20 marks respectively, that sum to 100 marks**, and an **entirely optional BONUS CHALLENGE** (*available at the end of this document*). You should only attempt the Bonus challenge – or even read it – if you have a significant amount of spare time. Prizes may be available for the best two solutions to the Bonus challenge.
- **Important:** TEN MARKS will be deducted from solutions that do not compile. Comment out any code that does not compile.
- Your code needs to compile and work correctly in the test environment which will be the same as the lab machines.
- **Important:** You should only modify the C files as directed by the questions. All other files should be treated as read-only, and are going to be overwritten by our autotester. You can find all files in the top-level **~/dl** directory.

Problem Description

For this year's final test, you will write some parts of a program that builds and manipulates a list of departments, where a department comprises a name and a list of individual staff members.

There is one underlying generic linked list module that provides functionality both for lists of departments and for lists of people.

We will ask you to implement small key pieces of functionality in various modules, which can be combined to solve the overall problem.

Task 1 : Complete the Person module (10 marks)

- The first task is to complete a simple function in `person.c`: this is an ADT storing a person's title, first name, last name and age. Most of this module has already been written and tested.
- **Important** You should **only modify** `person.c`, in particular you must not change `person.h` or any other C source or header file.
- You will find a test program `testperson.c` that exercises the person module in several obvious ways, along with my C-Tools `summarisetests` Perl script that processes standard test format output, counting the tests, and telling you how many (and which tests) fail.
- Before you have modified `person.c` at all, we suggest that you compile things up using `cb`, and then run the following:

  ```
  ./summarisetests ./testperson
  ```

  This will produce the following output:

  ```
  10 tests: 8 pass, 2 fail
  failures:
    ./testperson uppercase(p2(dcw).firstname (expected='DUNCAN', got='Duncan'): FAIL
    ./testperson uppercase(p2(dcw)).lastname (expected='WHITE', got='White'): FAIL

  passes:
    ./testperson p(ldk).firstname == 'Lloyd'
    ./testperson p(ldk).lastname == 'Kamara'
    ...
  ```

  This shows that most of the code already exists and works correctly, but 2 tests fail - those that uppercase names.
- Start by familiarising yourself with the person ADT in `person.h`:

  ```
  #define NAMELEN  24
  #define TITLELEN 8

  // our "person" ADT is a pointer to a struct person.
  typedef struct person *person;
  ```

```
// a person structure: stores a title, name and age
struct person {
  char title[TITLELEN];
  char firstname[NAMELEN];
  char lastname[NAMELEN];
  int  age;
};
```

- Only one function remains to be completed by you:

```
// person_uppercase_names( p );
//   Uppercase both the person's firstname and lastname.
```

- Inside `person_uppercase_names(p)` you will need to uppercase both `p->firstname` and `p->lastname`. This either suggests a static "uppercase one string" helper function, called twice, or two near identical "uppercase this string inline" loops, one above the other.
- To uppercase a string, you should iterate over each character in a string using an appropriate idiomatic loop, and uppercase each character (modifying the string). Hint: `ctype.h`'s `toupper()` macro is the way to upper case one character: see `man toupper` for details.
- After implementing your uppercasing function, and recompiling:

```
./summarisetests ./testperson
```

  should show all tests passing, including the uppercasing ones that previously failed:

```
10 tests: 10 pass

passes:
  ./testperson p(ldk).firstname == 'Lloyd'
  ./testperson p(ldk).lastname == 'Kamara'
  ...
  ./testperson uppercase(p2(dcw).firstname (expected='DUNCAN', got='DUNCAN'): PASS
  ./testperson uppercase(p2(dcw)).lastname (expected='WHITE', got='WHITE'): PASS
  ...
```

- You are welcome to write additional separate test programs, or extend `testperson.c`, if you feel the need.
- If you can't get your `person` code to work, additional instructions can be made available telling you how to bypass your faulty `person.c` and link against a pre-compiled working version that we have provided. Raise your hand and ask.

Task 2 : Complete the Linked List module (40 marks - 3 subtasks 10/10/20 marks respectively)

- The second task is to complete 3 simple functions in the generic linked **list** module, which is very similar to the one presented in the lectures.
- Acquaint yourself with the core list types in `list.h` and `list.c`. `list.h` defines:

  ```
  #define LIST_ELEMENT void *

  // our ADT: a list - incomplete type
  typedef struct list *list;
  ```

  (and some pointer to function types).

  `list.c` defines:

  ```
  // the data: the actual internal singly linked list
  typedef struct listdata *listdata;
  struct listdata {
    LIST_ELEMENT  head;
    listdata      tail;
  };

  // our ADT comprises various element operators, the data and an iterator.
  struct list {
    list_cmpf    cmpf;  // the element comparator function
    list_sprintf spef;  // the element sprint function
    list_freef   fef;   // the element print function
    listdata     data;  // the data - the linked list itself
  };
  ```

- **Important** You should **only modify list.c** for any part of task 2. You must not alter `list.h` (or any other file). You also must not alter any of the types in `list.c`.
- You will find a test program `testlist.c` that exercises the list module in obvious ways. Initially most/all tests will fail, in fact the test program aborts tripping an assertion - feel free to show this by running

  ```
  ./testlist
  ```

- As you correctly implement each sub-task described below, recompile and then rerun `./testlist` – hopefully a few more tests will succeed before it crashes. Eventually, after you implement all 3 sub-tasks, the test program should no longer crash - and all tests should pass when your implementations are correct. When `./testlist` no longer crashes, you may wish to run it via summarisetests:

  ```
  ./summarisetests ./testlist
  ```

- Feel free to write additional test programs if you like.

Task 2a : The make_list constructor (10 marks)

- In `list.c`, write the following C function:

```
// list l = make_list( &compareelementf, &sprintelementf, &freelementf );
//   Create an empty list with the given element comparator,
//   sprint element and free element function pointers.
```

- You will of course need to use `malloc()` and check for failure.
- You should use `assert()` for all failure checks.
- Make sure that you properly initialize all fields. Do not rely on `calloc()` zero byte initializations.

Task 2b : The listdata_cons constructor (10 marks)

- In `list.c`, write the following C function:

```
// listdata l = listdata_cons( head, tail );
//   Add a list element head to the front of a listdata tail.
//   Return the new listdata.
```

- Again, use `malloc()`, check for failure using `assert()`, properly initialize all fields.

Task 2c : Complete the list_addsorted operator (20 marks)

- In `list.c`, complete the following partially written C function:

```
// list_addsorted( l, v );
//   Given a sorted list l (nil or not) and a list element v,
//   add v to the list's data at the right place so that the list
//   REMAINS SORTED, modifying l.
```

- The special cases have been written for you (if the list is empty, or if the new value is less than the first element in the list).
- The missing part - which you must complete - is the general case, shown here in four parts: The comments explain the logical specification of what you want to find:

```
// otherwise we find THE BEFORE INSERTION POINT bip:
// - a node whose head h is <= v (bip->head <= v) and
// - EITHER it is the very last node (bip->tail == NULL)
//   or the following node's head (bip->tail->head) > v)
listdata bip;
```

  Then there is a comment showing where you must add your code that sets `bip` to the correct value:

```
// Task 2c: WRITE YOUR CODE HERE
```

Then there's a series of assertions to check that `bip`'s value satisfies the specification:

```
assert( bip != NULL );
assert( l->cmpf( bip->head, v ) <= 0 );
assert( bip->tail == NULL || l->cmpf( bip->tail->head, v ) > 0 );
```

Finally we add the new node at the right place, right after `bip`:

```
// add the new node after the insertion point
new->tail = bip->tail;
bip->tail = new;
```

- **Important** You must only add code in the "write your own code here" section. Do not change the assertions - they are correct. If they trip: it's your code that is broken. Do not change any other code above or below.

Once you have got all this code working, `./testlist` should show all tests passing. As an additional test, `valgrind ./testlist` should show no memory leaks.
If you can't get your list code to work, additional instructions can be made available telling you how to bypass your faulty `list.c` and link against a pre-compiled working version that we have provided. Raise your hand and ask.
Once you have completed tasks 1 and 2, note that several other modules and test programs that we've written for you should now start working:

- A convenience module called `plist.[ch]` provides a list of people (persons).
- A separate test program `testplist.c` that exercises lists, people, and lists of people. `./testplist` should work now that you've finished `person.c` and `list.c`.
- An ADT module called `department.[ch]` that implements a department, storing a department name and a list of staff members (people).
- A separate test program `testdepartment.c` that exercises departments (and indirectly plists, lists and people). Now that you've finished `person.c` and `list.c`, `./testdepartment` should work too.
- A convenience module called `dlist.[ch]` provides a list of departments.
- A separate test program `testdlist.c` that exercises department lists, departments, lists of people etc. `./testdlist` should work now that you've finished `person.c` and `list.c`.

Task 3 : compute average age of members by department (30 marks)

- Our next task is to use our data structures (dlists, plists, people etc) to do some useful work on them. Specifically, let's try to compute the **average age of members** in each Department.
- This will require iterating over all members of all departments, counting the number of members of each department, and summing the ages of the members of each department. Then a second pass over the results calculates the average age for each department (by dividing the sum of the ages of all members of that department by the number of members of that department).
- In `ageinfo.h`, there's an `ageinfo` data structure:

```
typedef struct {
  // while processing departments and members
  int         currdept;      // current department (-1 before we start)

  // results:
  int         ndepts;        // number of depts
  deptnamestr *deptname;     // dynarray deptname[ndepts]
  int         *agesum;       // dynarray agesum[ndepts]
  int         *nmembers;     // dynarray nmembers[ndepts]
  double      *avgage;       // dynarray avgage[ndepts]
} ageinfo;
```

- In `ageinfo.c`, there's a function called `calc_ageinfo()` which fills in an `ageinfo` structure:

```
// ageinfo a;
// calc_ageinfo( dl, &a );
//   Calculate the sum of member ages and number of members
//   for each dept in dl, storing them in a, then compute the
//   per-dept average ages (also storing them in a).
//
void calc_ageinfo( list dl, ageinfo *a ) {
  // Initialise *a
  a->ndepts    = list_len( dl );
  a->currdept  = -1;
  a->deptname  = malloc( ndepts * sizeof(deptnamestr) );
  assert( a->deptname != NULL );
  a->agesum    = malloc( ndepts * sizeof(int) );
  assert( a->agesum != NULL );
  a->nmembers  = malloc( ndepts * sizeof(int) );
  assert( a->nmembers != NULL );

  // process all (dept,member) pairs, updating agesum[] and nmembers[]
  dlist_foreach_member( dl, &process_dm, a );

  assert( a->currdept == a->ndepts-1 );
```

```
    // now calculate the average ages:
    for( int i=0; i<a->ndepts; i++ ) {
      int nmbrs    = a->nmembers[i];
      assert( nmbrs > 0 );
      int sum      = a->agesum[i];
      double avg   = (double)sum / (double)nmbrs;
      a->avgage[i] = avg;
      printf( "dept %s: #mbrs %d, sum ages %d, avg age %g\n",
        a->deptname[i], nmbrs, sum, avg );
    }
  }
```

- As you see, after initializing the ageinfo structure, including allocating the dynamic array fields, it uses a `dlist_foreach_member()` iterator to walk across all the members of all the departments, one by one. That invokes a callback called `process_dm()` for each (department, member) pair, passing the callback the department name, the member and an `ageinfo *a` which can be modified.

- Your task is to implement `process_dm()`, which is specified as follows:

```
  // ageinfo a;
  // process_dm( deptname, member, &a );
  //   Process (deptname,member) pair, updating the ageinfo record:
  //   Sum up the ages of the current department, count the #members of
  //   the current dept. When the deptname changes, move onto the next dept.
  //
  static void process_dm( char *deptname, person member, void *state ) {
    ageinfo *a = state;
    // [some debugging code]
    // Task 3: WRITE YOUR CODE HERE
```

- `process_dm()` should work as shown by the following pseudo-code:

  1. On the very first call (when `a->currdept == -1`), it should set `a->currdept = 0`, set `a->deptname[0]` to `deptname`, set `a->agesum[0]` and `a->nmembers[0]` to zero.
  2. On all later calls, if `deptname` is not the same as `a->deptname[a->currdept]`, then it should increment `a->currdept`, set `a->deptname[a->currdept]` to `deptname`, set `a->agesum[a->currdept]` and `a->nmembers[a->currdept]` to zero.
  3. Unconditionally, it should then increment `a->nmembers[a->currdept]`, and add the current member's age to `a->agesum[a->currdept]`.

- **Important** You must only add code inside `process_dm()`. Do not change any other existing code above or below – although you may define additional static helper functions immediately above if you like.

- There is a new unit test program `./testageinfo` whose tests should work once you have correctly implemented `process_dm()`. You can also use `cb --test` to run all tests. All should pass if you've written everything correctly.

Task 4 : A small change to dlist.c (20 marks)

- After all this work in the foothills (on individual modules and their unit tests), we can finally reach the peak: There is one more program – the real main program called `readdata.c` which reads a prepared data file listing departments and their members called `DATA` (containing one member of a department per line in CSV format), builds a `dlist`, then displays the contents of the `dlist`, and finally calculates the average age of each department's members and displays them.
- When you run `readdata` as:

  ```
  ./readdata < DATA
  ```

  the final section of output is:

  ```
  List of departments and their members:

  [ dept CSG, members [ Mr Christian Bosio (age 50), Mr Geoff Bruce (age 60),
  ....  Mr Mark Wheelhouse (age 36), Mr Jaime Willis (age 32) ] ]

  Summary: 3 departments:

  Average ages (per dept)

  dept CSG: avg age 50.1667
  dept EdTech: avg age 30
  dept TF: avg age 33.25
  ```

- Looking at the list of departments and their members, we note that it shows them as a raw *list of lists*, very much as Haskell might display such a list (although an individual department has been formatted slightly).
- Your final task is to change that format so that instead the departments and their members are displayed as:

  ```
  dept CSG, member Mr Christian Bosio (age 50)
  dept CSG, member Mr Geoff Bruce (age 60)
  ...
  dept TF, member Mr Mark Wheelhouse (age 36)
  dept TF, member Mr Jaime Willis (age 32)
  ```

- The most elegant way to achieve this is to change the implementation of `dlist_print()` so that it invokes a suitable iterator to iterate over each **(department, member)** pair, providing a suitable callback and piece of state data to display the list in the correct format.
- You should understand enough about the available iterators, and how to write a customised callback – and how to use state data – to solve this problem now.
- As always, you should only modify `dlist_print()` in `dlist.c`. You must not modify other functions, although you will undoubtedly need to add a helper function – the iterator callback – immediately above `dlist_print()`.

The above 4 tasks total to 100 marks. You may stop here once you've completed them.

But, if you have completed the above tasks and have 30 minutes or more to spare, you may wish to have a look at the **Bonus Challenge** document.

Good luck!

Bonus Challenge: Change from a linked list to a dynamic array (0 Marks)

- **Only look at this if you have completed all the main tasks and have at least 30 minutes to spare**.
- **There are no extra marks available for this bonus challenge, although I intend to make a couple of prizes available for the best (neatest, most elegant) two solutions – if there are any.**
- Our goal is to change our **linked list of generic pointers** in `list.c` to a **dynamic array of generic pointers** while keeping the list interface in `list.h` (and all the code that uses it) entirely unchanged.
- Copy your entire solution into a new `bonus` subdirectory.
- Then go into that `bonus` subdirectory and work entirely in there.
- Start with a `cb --clean` to start clean.
- *..Plays Mission Impossible theme ..* Your mission, should you choose to accept it, is to alter `list.c` so that the `listdata` linked list data type is replaced entirely by a `LIST_ELEMENT *` dynamic array (with a capacity and a current length, as usual with dynamic arrays). The dynamic array will periodically need to be resized using `realloc()` when it becomes full.
- You will, in essence, need to remove the `listdata` structure, change the `struct list` data structure to replace the `listdata data` with the dynamic array fields, and change all references to `listdata` to array indexes. This will also involve modifying nearly every function in `list.c` in minor ways.
- If you get everything working, `cb --test` should work exactly as it did with the linked list implementation.

Congratulations if you get it to work!