C Final Test

Imperial College London

Department of Computing

Analysing C Include Structure

Monday 12th June 2023
10h00 to 13h00
THREE HOURS
(including 10 minutes planning time)

- You can edit, compile and run your code using the terminal or an IDE like **CLion**. You may use **CBuild (CB)** to compile your code. No help will be available for IDEs, and no Makefiles are provided.
- There are **4 tasks, with 20, 20, 40 and 20 marks respectively, that sum to 100 marks**, and an **entirely optional BONUS CHALLENGE** (separate section at end). 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 ∼/**acis** directory.

Problem Description

For this year's final test, you will write some parts of a **mfbuild**-like tool to analyse C source file #include structure and generate Makefiles.

We will focus on providing small reusable pieces of functionality, which can be combined to analyse the include structure of C programs.

Task 1 : For every line in a file (20 marks)

Our first task is some simple File I/O: In `everyline.c`, write the following callback-based C function:

```
// int n = foreveryline( filename, eachlinef );
//   open the given filename, read every line from that file,
//   remove the trailing newline (if present) and invoke the given
//   callback eachlinef with the filename, the line number and the line.
//   Return the number of lines read, -1 if the file can't be opened.
```

You will of course need to:

- Open the given `filename` for reading,
- Check that it succeeded (returning -1 if not),
- Read a line at a time, keeping track of the line number as you go,
- Remove a trailing newline from the line you've read (if there is one),
- Invoke the `eachlinef` callback (a pointer to a function), passing it the filename, the line number and (of course) the line.

Plus at the end:

- Close the file, and
- Return the total number of lines read.

Important You should **only modify everyline.c** for this task. You must not alter `everyline.h` (or any other file), although you should definitely acquaint yourself with the types that `everyline.h` contains, in particular a `line` **type** with a suitable max line length constant, and the **every line callback function type** `everylinecb`.

You will find a test program `testeveryline.c` that exercises this framework in an obvious way (recall that `__FILE__` is the C pre-processor's way of delivering the current C source filename as a string).

Feel free to write additional test programs if you like.

Task 2 : Whitespace-separated word matching (20 marks)

Next, let's do some simple pointer-based string manipulation: In `match.c`, write the following function:

```
// char *wds[] = { "one", "two", NULL };
// char *result = matchwords( target, wds );
//      Given a target string, and a NULL terminated array of words (each
//      word is a char *), attempt to match every word in wds against the
//      target, starting the matching process at the start of the target.
//      More specifically, you are to:
//      - match (and skip) any amount of whitespace (including none) at
//        the start of the target,
//      - then match wds[0] against the next part of the target exactly as is,
//      - then match (and skip) any amount of whitespace (including none),
//      - then match wds[1] etc.
//      If the target doesn't match all words, return NULL
//      If the target matches all words (not necessarily matching the
//      whole string), then match and skip any final amount of whitespace
//      (including none) after the last word is matched, and return a pointer
//      to the first unmatched char beyond that.
```

For clarity, *whitespace* means a space or a tab character.

So for example, the target string

```
"    one    two   hello"
```

would successfully match the above list of words, and the result would be a pointer to the `'h'` of hello.

The target string

```
"onetwohello"
```

would also match, the result again being a pointer to the `'h'`.

Hint: `ctype.h`'s `isspace()` macro will be useful, as will `strncmp()` and `strlen()` and other basic string utilities.

You will find a unit test program `testmatch.c` that tests the word matcher in an obvious way, producing multiple lines of OKs or FAILs.

You should **only modify** `match.c` for this task; you must not alter `match.h` or any other C source file.

Task 3 : Binary Search Tree of (string key, generic pointer value) pairs (40 marks - 3 subtasks 10/10/20 marks respectively)

Our next task is to build an ADT (data structure) to store and manipulate information that we will need later.

Specifically, let's build a **Binary Search Tree** that stores **(key,value)** pairs: The **keys are arbitrary strings**, and the **values are generic** (`void *`) **pointers**.

You will find a partially implemented Binary Search Tree module `bst.[ch]`.

Familiarise yourself with its data structures - especially `bintree`, where each node stores a **(k,v)** pair, a **left** and **right** subtree:

```
typedef struct bintree *bintree;
struct bintree {
  char *key;           // the key: a string
  void *value;         // the value: a generic pointer
  bintree left, right; // the left and right subtrees
};
```

Plus the `bst` ADT storing a `bintree`, a **print (k,v)** function pointer and a **free value** function pointer (which may be NULL):

```
struct bst {
  bst_printkv_func pf;  // the (k,v) print function
  bst_freev_func ff;    // the value free function
  bintree t;            // the binary tree of (k,v) pairs itself
};
typedef struct bst *bst;      // our ADT..
```

As in all binary trees, the idea is that the keys are distinct, and always sorted – all **(k,v)** pairs whose keys are **alphabetically smaller** than the key in the current node will be found in the **left** subtree, and similarly all **(k,v)** pairs whose keys are **alphabetically larger** than the current node are in the **right** subtree.

You should **only modify** `bst.c`; you must not alter `bst.h` or any other C source file.

There is, as usual, a unit test program called `testbst.c`, which will help you to test your BST as you gradually complete it. You may write additional test programs if you like.

There are three routines for you to complete, all in **bst.c**:

Task 3a : The BST constructor (10 marks)

```
// bst d = make_empty_bst( pf, ff );
//   Create a new empty BST, with an element print function <pf>,
//   an element free function <ff>, and an empty bintree.
//   Abort if any memory allocation fails.
//
```

Task 3b : The bintree constructor (10 marks)

```
// bintree t = makenode( key, value );
//   Make a new bintree node containing (<key>,<value>),
//   duplicating <key> via strdup().
//   Abort if any memory allocation - even the strdup() - fails.
//
```

Task 3c : The BST "add (key,value) pair" operator (20 marks)

```
// add_bst( b, key, value );
//   Add (<key>,<value>) to <b>.  If the given key is already present,
//   free the old value associated with it (i.e. in the bintree node with
//   that <key>), and then store the new value in that bintree node,
//   otherwise add a completely new (<key>,<value>) node.
//   The key (but NOT the value) should be duplicated.
//   Abort if any memory allocation fails.
//
```

Feel free to write static helper functions if you like, and of course later tasks may call the functions you wrote in earlier tasks. For example, you probably should call `makenode()` from `add_bst()`.

Task 4 : use BSTs to build our data structures (20 marks)

Note that there is also – provided for you – a module called `set.[ch]` that constructs a simple set of strings using a dynamic array, and the `testset.c` unit test program.

Now that we have built our BSTs (and been given sets), we will use them.

Start by examining a program called `findallincludes.c`, which is a thin wrapper around the function `analyse()` in an accompanying module called `analyse.[ch]`.

This uses your solutions to task 1 and 2 to read all `*.[ch]` files in the current directory, and then to find all lines in all those files that match the following text:

```
  #include "
```

and also find which files include a line matching the text:

```
  int main(
```

Before changing anything, compile and run `findallincludes`, and spend a minute or two understanding what `findallincludes.c` (and `analyse.c`) is detecting and reporting.

You should see that it identifies **.c** and **.h** files that exist, identifies which **.c** files define `main()`, and identifies which **.h** files are directly included by each **.c** file.

Examining the code further, in `analyse.h` you will see the following type declarations:

```
struct analysis {
  set existset;   // set of .c and .h filenames that exist
  set mainset;    // set of .c filenames containing main()
  bst c2inc;      // .c filename -> set of included .h files
};
typedef struct analysis *analysis;
```

and you will see that the `analyse()` function in `analyse.c` returns an `analysis` (pointer).

Your task is to modify `analyse.c` (the `analyse()` function and three recording functions) to:

1. Build `a->existset`: the set of all **.c** and **.h** filenames that exist.
2. Build `a->mainset`: the set of all **.c** filenames which define `main()`.
3. Build `a->c2inc`: the mapping from filenames to the set of files that are **included directly** in that file. **Note that each value of the BST should be a set of strings**.

Note that `analyse()` already calls a helper function to `malloc()` an `analysis` block. You will need to somehow make that `analysis` pointer available to the recording functions so that they may modify it – adding code in those functions to build the analysis **existset, mainset** and **c2inc** BST – and finally return the completed `analysis` from `analyse()`.

You should **only modify** `analyse.c` – you must not alter `analyse.h, findallincludes.c` or any other C source file – there is already code in `main()` in `findallincludes.c` to display the analysis data structures (if they are not NULL), and free them.

Hints:

1. You'll need to create the analysis sets (`a->existset` and `a->mainset`) and the BST `a->c2inc`.
2. To create the `c2inc` BST, you'll need helper print and free functions. They are already present in `analyse.c`, so you can create `c2inc` as follows:

   ```
   a->c2inc = make_empty_bst( &print_wrapper, &free_wrapper );
   ```

3. **IMPORTANT**: note that `c2inc's` values must be sets, i.e. it's a **BST of (string, set of string) pairs**. The way to enforce this is as follows:

   When storing a new fact of the form

   "src file **S** *directly includes header file* **H***"*

   into `c2inc`, you'll need to:

   - Check whether **S** is already in `c2inc`,
   - If it is, retrieve the associated value from `c2inc` which has **S** as the corresponding key. This value is the set to use below.
   - If not, create a new empty set via `set newset = make_set()`, and add the **(k,v)** pair `(S, newset)` to `c2inc`; in this case `newset` is the set to use below.
   - Now, we have a set, which is either newly created (and empty), or long existing (and non empty):
   - Add **H** to that set.

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 might think that it would be very satisfying, having done all this work, to be able to compile some C code using your work. Read the following **Bonus Challenge** section about this.

Good luck!

Bonus Challenge: Write a Makefile (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.

You might think that it would be very satisfying, having done all this work, to be able to compile some C code using your work – by building a `Makefile` and then running m̂ake.

The data structures we have are enough to do almost all of this.

There is a `BONUS` subdirectory, containing some additional source code. Go into that directory, and read the instructions in the `README` file there. You will find that a partial Makefile generator has been written in `writemakefile.c`.

It generates (on stdout) an incomplete `Makefile`. It figures out what to build (the basenames of all the source files containing `main()`), generates standard `all` and `clean` make targets, generates one target per object file in `c2inc` (because if `X.c` directly includes `X.h` and `Y.h` then `X.o` depends on `X.h` and `Y.h`), and then generates one link target per main program.

Note that it DOES NOT SUPPORT NESTED INCLUDES: Forget nested includes for now.

Unfortunately, the link targets that it generates are not complete. It only records that `X` depends on `X.o` for each main program `X.c`. It doesn't work out all the `.o` files of all the modules that `X` **directly OR INDIRECTLY** uses.

..Plays Mission Impossible theme ..

Your mission, should you choose to accept it, is to complete the function `allobjects( M, A )` in `writemakefile.c`.

`M` is the source filename of a single main program, and `A` is an initialized analysis pointer. `allobjects()` builds and returns a space-separated `malloc()`ed string listing all the `.o` files that `M` will need to be linked with – including `M.o` itself. (Actually `M` includes the `.c` extension, so we really mean the basename of `M` with `.o` appended).

This involves computing the **TRANSITIVE CLOSURE** of `c2inc{M}`. There are many algorithms for computing transitive closures. You may pick any of them that you know, and adapt it if you like.

But here's an outline of a simple queue-based algorithm that can be used to implement `allobjects`:

- Create a `todo queue` with only `M` on it. **Note that** the `todo queue` should contain strings of the form `X.c`.
- Create an empty `done set` of strings.
- While the `todo queue` is not empty:
  - Dequeue one C source filename `T` from the `todo queue` (this will be of the form `X.c`).
  - Add `T` to the `done set`.

  - Look up `c2inc{T}`: a set of included files, each of the form `Y.h`.
  - Foreach included file `Y.h` in that set:
    * Map the `Y.h` filename to the corresponding `Y.c` source filename.
    * If the file `Y.c` exists (use the `existset` to check), and `Y.c` is not in the `done set`, then enqueue `Y.c` onto the `todo queue`.

- When the `todo queue` is finally empty: set the `result` string to empty, then iterate over the members of the `done set`, mapping each `X.c` file to the corresponding `X.o` object file, and appending `X.o` to the `result`, adding a space between each object filename.
- Then when you've built the `result` string, `strdup(result)` and return the duplicated result.

To help you implement this algorithm, you will find a simple queue-of-strings module called `queue.[ch]` in the BONUS directory. This has the useful property that duplicate elements are silently ignored.

To store the `result` string, you could either implement the simplest possible "growable" dynamic string (with a `capacity`, a `length` and a `char *` pointer, as discussed in the lectures) - or simply make `result` a huge char array (10000 chars long, say) - and use `assert()` to prevent buffer overrun.

If you get the algorithm correct, the resultant string of object files will be added to each main program's link target, and the Makefile generated (on stdout) will be complete.

Capture it via:

```
./writemakefile > Makefile
```

and CAREFULLY EXAMINE it. You may compare it with the output of

```
cb --makefile
```

When the generated output looks correct, try:

```
cb clean
make -n
```

and if those commands look sane, try running:

```
make
```

If you got `allobjects()` correct, the `Makefile` will be correct, with each executable correctly depending on all the object files that need to be linked together, and thus make will read your freshly generated `Makefile` and recompile all the code you've been working on.

Congratulations if you get it to work!