CPS1011 – Programming Principles (in C)

Assignment for the Academic Year 2018/19

Xandru Mifsud

Typeset on December 9, 2018 using LaTeX and some patience.

1

tuples_t Library

A simple library to make the notion of tuples available in C99 is implemented, providing some similar functionality to that provided in Python. We begin with a walk through of some design considerations, followed by the functional specification of each of the implemented functions.

1 Design Considerations

1.1 Format Specification

The library is to handle a variety of data types, thus for this reason a *tagged union*[1] structure was chosen as the means of format specification for data input to a tuple. A simple example of such a structure is provided below,

```
typedef struct{
        enum{i, f, c} type;

        union{
                int i;
                float f;
                char c;
        } val;
} tagged_union;
```

Each data type is associated with a tag, allowing easy association between data and type during format specification. For example, an array of such tagged unions takes the form,

```
tagged_union example[2] = {{.type = i, .val.i = 10}, {.type = c,
↪  .val.c = 'T'}};
```

[1]https://en.wikipedia.org/wiki/Tagged_union

The decision was then taken to specify a `tuple_t` structure,

```
#define VAR_NAME_SIZE 64 // based on the ISO C Standard for naming variables

typedef struct{
        char id[VAR_NAME_SIZE];
        int next;
        tagged_union data;
} tuple_t;
```

where the tuple identifier size was chosen to be fixed at the standard variable identifier size used in ISO C99. The data part of a tuple must thus be specified using a tagged_union, as defined above. The integer `int next` specifies a shift in a `tuple_t` pointer such that it points to the first element of the next tuple in line. Logic for the population of this variable is decided by the library.

Thus upon tuple creation, a user must specify the following:

i. A pointer to a `tagged_union` type, be it an array or not

ii. Number of elements in the `tagged_union` passed (for reasons related to setting the value of `int next` for each tuple element)

iii. An `char` array identifier of size at most 64

It falls entirely in the responsibility of the user to ensure that the passed `tagged_union` has correct data and type associations. **Undefined behaviour may be caused by incorrect usage of `tagged_union`.**

1.2 Tuple and Memory Management

Tuples are all to be stored in a single dynamic array of type `tuple_t`, where each element represents an element to some tuple element. Elements belonging to the same tuple (i.e. of the same identifier) are stored sequentially in left-to-right order. Tuple and Memory Management is a key part to the following three desired features: Creation of a tuple, deletion of a tuple, and joining of two tuples.

We look into each of these by considering a series of examples, starting with the creation of a `tuple_t` array and its population with the first tuple, as illustrated in Figure 1.1.

```
tuples = (tuple_t*)malloc(0);
tagged_union example[2] = {{.type = i, .val.i = 10}, {.type = c, .val.c = 'T'}};

createTuple("1", &example, 2);
```

where `"1"` is simply the identifier of the tuple to be created and `2` is the size of the tuple. The above code does the following:

i. `malloc()` a dynamic array named `tuples` of type `tuple_t`, with initial size 0, in which all tuples and their elements are to be stored

ii. Specification of a `tagged_union` with tuple data by the user

iii. A call to a function `createTuple()` responsible for tuple and memory management upon creation of a new tuple

The still-to-be specified `createTuple()` function is to ensure that the identifier is unique, and if so,

i. Issue a `realloc` to expand the `tuples` array accordingly

ii. Populate the newly specified memory locations with the supplied data, in the order in which it appears from left-to-right

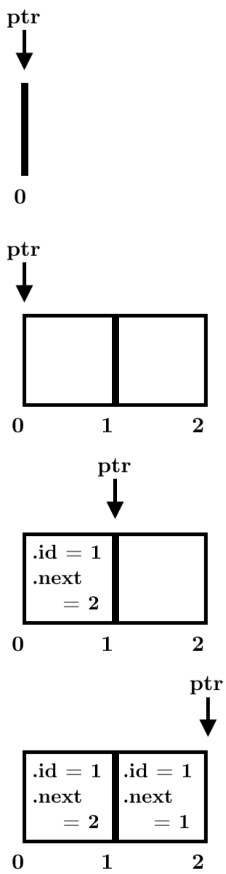

Figure 1.1: Diagrammatic representation of the procedural sequence for initial creation of first tuple.

We now proceed to consider tuple deletion, as illustrated in Figure 1.2. A function named `deleteTuple()` should check if a passed tuple identifier exists within `tuples`, and if so,

i. Point to the start of the tuple to be deleted, within `tuples`

ii. Use `memmove()` to shift down the memory contents *succeeding* the end of the tuple to be deleted to the location of the pointer (i.e. to the start of the tuple being deleted)

iii. Issue a `realloc` to reduce the size of `tuples` by the size of the deleted tuple

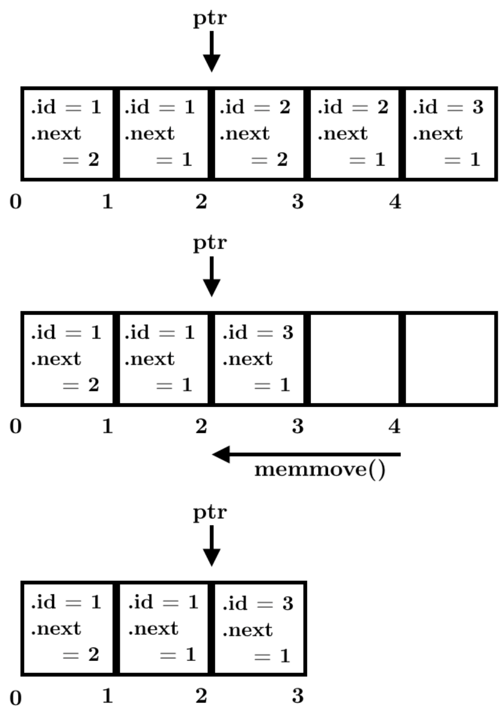

Figure 1.2: Diagrammatic representation of the procedural sequence for tuple deletion.

The last desired operation that requires significant Tuple and Memory Management is that of joining two tuples, in the order specified. This is achieved by a still-to-be-specified function `joinTuples()`, taking as input two `tuple_t` pointers and a new identifier, which if both pointers are valid and the identifier is unique,

i. Creates a `tagged_union` array of the aforementioned size, and populates it first with the data of the first tuple and then with that of the second, in left-to-right order

ii. Issues a call to `createTuple()` with the passed identifier and newly constructed `tagged_union` specification

1.3 Desired Functions and Modularity

Special considerations were taken to ensure that the design is as modular as possible. There is a reason why no further explanation and/or diagrammatic representation is given for the join operation - once the `tagged_union` is constructed, the rest of the function is equivalent to `createTuple()`, and hence why it is intended that a call to it is performed. This avoided significant code repetition, by designing `createTuple()` to be as structurally-agnostic as possible (i.e. of minimal dependencies).

Modularity was not restricted to this particular case only - convenience functions, such as those to check if a `malloc()` or `realloc()` operation was carried out successfully, were written to have minimal dependencies and thus could be used practically anywhere.

The code was designed so that any manipulation of supplied data and its type is carried out within `switch` statements, so that support for new data types can be simply added by designing appropriate `case` statements and extending the `tagged_union` structure to support the new type.

Besides the `createTuple()`, `deleteTuple()` and `joinTuples()` functions, a series of other features implemented through functions is desired,

i.

The functional decelerations for each of these, along with any constants used, and definition of the `tagged_union` and `tuple_t` structures, are found in the header file `tuple_t.h`. Further explanations on the implementation on each of these in `tuple_t.c` is provided in the next section.

2 Functional Listings

2.1 `tuple_t* getTupleByID()`

Fetches a pointer to the start of a tuple, by means of an supplied tuple identifier, if a match is found. Otherwise a `NULL` pointer is returned. Developers should explicitly handle `NULL` pointer return, to implement their own logic depending on their particular use case.

The returned pointer should not be stored indefinitely, especially if it is to be used after some creation, deletion, or join operation, as the `tuples` dynamic array may have been moved around for memory management purposes.

Parameters

i. `char search_id[VAR_NAME_SIZE]`: character array of acceptable length up to 64 bytes

Return

i. If a match is found, a `tuple_t*` pointer to the start of a tuple with matching identifier

ii. Otherwise, a `NULL` pointer

Firstly, a pointer to be returned is defined and initialised to `NULL`,

```
tuple_t *match = NULL;
```

The function then establishes the length of the supplied identifier, and stores it in a variable `int dimSO_in`.

```
while (search_id[k++] != '\0'){
        dimSO_in++;
}
```

The function then loops through `tuples`, with cases to skip from the start of one tuple to the next, since the elements in between need not be searched as they have the same identifier. Note that here, `tuples_size` is the current size of the `tuples` array. Skipping from one tuple to the next is carried out within line 5 in the listing below.

```
1 for (int j = 0; j < tuples_size; j++) {
2         // statements
3 
4         // if some condition satisfied
5         j += tuples[j].next - 1;
6         continue;
7 
8         // statements
9 }
```

Within the looping construct, the length of the identifier is found for the current tuple being pointed to, in a similar fashion to what was done for the search identifier. If the identifier lengths do not match, then definitely the identifiers are not equal and thus the pointer is incremented to the start of the next tuple.

If however the lengths match, a check if the individual pair-wise characters between the identifiers match. This is done in the following manner,

```
for (k = 0; k < dimS0_in; k++) {
        if (search_id[k] == tuples[j].id[k])
        match_count++;
}
```

where `int match_count` is used to store the number of matching characters. If the number of matching character-pairs is less then the length of the identifiers, then clearly they do not match in at least one position and hence the pointer is moved to the start of the next tuple.

Otherwise, the pointer `match` is set to the pointer of the first element of the current tuple, and a `break` is issued.

```
match = &tuples[j];
break;
```

If no match is found, `match` would still be `NULL` and thus that would be the return.

2.2 `void ptr_alloc_valid()`

Simple convenience function that checks if valid pointer is returned after a `malloc()` or `realloc()` operation; if `NULL`, `EXIT_FAILURE` is instructed so as to eliminate potential issues in memory management, prevent core dumps, etc...

Parameters

i. `void *ptr`: A pointer typecast to `(void*)`

A simple `NULL` check is carried out, and if true an error message is displayed and the program exits:

```
if (ptr == NULL) {
        puts("TUPLE_ALLOCATION_FAILURE: Memory allocation failed
        ↪  for array during either initial malloc or subsequent
        ↪  realloc.\n");
        exit(EXIT_FAILURE);
}
```

2.3 void createTuple()

Parameters

i. `char id[VAR_NAME_SIZE]`: character array of acceptable length up to 64 bytes

ii. `tagged_union in[]`: array of `tagged_union` in which the tuple element data is specified and formatted

iii. `int dimS0`: dimensionality about the 0^{th} axis of the input `tagged_union` array

Firstly, the function ensures that the specified identifier does not already exist, by means of the `getTupleByID()` function. If this is not so, an error is printed in the terminal:

```
if(getTupleByID(id) != NULL){
        printf("TUPLE_CREATE_ERROR: Tuple ID %s already
         registered in stack.\n", id);
}
```

If function execution proceeds, memory is allocated to the `tuples` array by increasing the size with `(tuples_size + dimS0) * sizeof(tuple_t)`. A call to `ptr_alloc_valid()` is carried out to ensure successful allocation.

```
tuples = realloc(tuples, (tuples_size + dimS0) *
 sizeof(tuple_t)); // adjust size to hold new tuple data
ptr_alloc_valid((void*) tuples);
```

Looping is then used to allocate data to the elements via the returned pointer. The loop specification is,

```
for(int j = tuples_size; j < (tuples_size + dimS0); j++){
        // statements to loop through
}
```

which specifies that the new tuple elements will be appended to the very end of `tuples`. The identifier variable `tuples[j].id` is populated using usual copying techniques for `char` arrays. The variable `tuples[j].next` is populated using the following logic,

```
tuples[j].next = dimS0 + tuples_size - j;
```

which specifies how much the pointer must be incremented to reach the end of the array (or rather, the start of the next tuple is further tuples are created later on).

The data formatting enforced by the `tagged_union` comes in handy when populating the data part of the tuple, which is done using a `switch` statement on the data type of the current element, `in[j - tuples_size].type`; an example `case` statement is provided,

```c
case i: tuples[j].data.type = i;
        tuples[j].data.val.i = in[j - tuples_size].val.i;
        break;
```

Note that, the `switch` statement is carried out on an `enum` type and thus correct population of the `tagged_union` supplied by the user is assumed, specifically between the data and it's associated type. Incorrect definitions may lead to undefined behaviour.

Lastly, the `tuples_size` is updated: `tuples_size += dimS0;`

2.4 `void deleteTuple()`

Parameters

i. `char id[VAR_NAME_SIZE]`: character array of acceptable length up to 64 bytes

Firstly, a function call to `getTupleByID()` is made to fetch the `tuple_t*` pointer to the tuple corresponding with the identifier. A check is also carried out in case `getTupleByID()` returns `NULL`.

```c
tuple_t* tuple_ptr = getTupleByID(id);

if(tuple_ptr == NULL){
        printf("TUPLE_DELETE_ERROR: Tuple ID %s not found and
         ↪  thus deletion cannot be performed.\n", id);
}
```

Since `getTupleByID()` fetches a pointer to the first element of a tuple, `tuple_ptr->next` corresponds to the size of the tuple. This is stored in `int size = tuple_ptr->next;` and then the size of the `tuples` array **after** the tuple to be deleted is found,

```c
long int copy_length = tuples_size - ((tuple_ptr + size) -
 ↪  tuples);
```

The `tuples_size` variable is decremented by `size`. After which, the memory contents of `tuples` **after** the tuple to be deleted are shifted using `memmove` to the location of the start of the tuple being deleted,

```c
tuples_size -= size;
memmove(tuple_ptr, tuple_ptr + size, copy_length *
 ↪  sizeof(tuple_t));
```

To aid with this, kindly refer back to Figure 1.2 and the Tuple and Memory Management discussion provided earlier on. An explanation on the use of `memmove` follows[2] since `tuple_ptr` points to the start of the tuple to be

[2] Kernighan, B. W. & Ritchie, D. M. (2012) *The C Programming Language*, 2nd Edition, Prentice-Hall, New Jersey, USA; pg. 250

deleted, then we must move `copy_length * sizeof(tuple_t)` bytes from `tuple_ptr + size` to `tuple_ptr`.

Lastly, a `realloc()` is carried out to free up memory and a call to `ptr_alloc_valid()` is made to ensure correct allocation.

```
tuples = realloc(tuples, tuples_size * sizeof(tuple_t));
ptr_alloc_valid((void*) tuples);
```

2.5 joinTuples()

While `createTuple()` and `deleteTuple()` are rather self-explanatory, `joinTuples()` merits explanation. The function takes an identifier (which must be unique), and two pointers to elements in `tuples`. These pointers however may not necessarily be at the start of a tuple. One can choose to fetch pointer to the start of a tuple by `getTupleByID()`, then shift it by some $0 < x <$ size of tuple, to join with *only a part of the tuple*. This is achieved through the use of the `.next` variable.

Developers should take care that the pointers passed are current, and point to the desired elements in `tuples`. With this in mind, it is recommended to fetch pointers at compile time using `getTupleByID()`.

Parameters

i. `char id[VAR_NAME_SIZE]`: character array of acceptable length up to 64 bytes

ii. `tuple_t* tuple_ptr_1`: pointer to an element in `tuples`, corresponding to the first tuple in the join (leftmost)

iii. `tuple_t* tuple_ptr_2`: pointer to an element in `tuples`, corresponding to the second tuple in the join (rightmost)

Firstly, a simple check is carried out to verify that the pointer are not `NULL`. If so, an error is printed and the join operation is not carried out,

```
if(tuple_ptr_1 != NULL && tuple_ptr_2 != NULL){
        // some nifty code goes here
}
else{
        printf("TUPLE_JOIN_ERROR: At least one of the tuple
         ↪ pointers specified is NULL.\n");
}
```

If the join operation is to be carried out, the size of the new resulting tuple is found and a `tagged_union` array is defined to hold the data specification for the new tuple,

```
int dimS0 = tuple_ptr_1->next + tuple_ptr_2->next;
tagged_union joinedData[dimS0];
```

The `tagged_union` array is then first populated with the data from the first tuple (or part of) and then with the data from the second tuple (or part of), in left-to-right order.

```
for (int j = 0; j < tuple_ptr_1->next; j++){
        joinedData[j] = tuple_ptr_1[j].data;
} // populate tagged_union with data of ptr_1 first

for (int j = tuple_ptr_1->next; j < dimS0; j++){
        joinedData[j] = tuple_ptr_2[j - tuple_ptr_1->next].data;
} // then populate tagged_union with data of ptr_1 first
```

Lastly, a call `createTuple(id, joinedData, dimS0);` is made, highlighting modularity in the code design.

2.6 `showTuple()`

The following is a simple function that prints out a formatted string with the contents of a tuple. If say the `tagged_union` array of a tuple is of the form,

```
{{.type = i, .val.i = 10}, {.type = c, .val.c = 'T'}, {.type =
 ↪ f, .val.f = 10.2}}
```

then the output string is of the form `(10, 'T', 10.2)`.

Developers should take care that the pointers passed are current, and point to the desired elements in `tuples`. With this in mind, it is recommended to fetch pointers at compile time using `getTupleByID()`. If one points to an element of a tuple that is not the first element, then the *succeeding* elements (inclusive) are shown.

Parameters

i. `tuple_t* tuple_ptr_1`: pointer to an element in `tuples`

Firstly, a check is carried out to ensure that the pointer is not `NULL`. If so, an error message is printed,

```
if (tuple_ptr != NULL){
        // hey, I really hope some code was actually written
         ↪ here
}
else{
        printf("TUPLE_SHOW_ERROR: Tuple pointer specified is
         ↪ NULL.\n");
}
```

The format is specified through a series of `switch` statements based on the data type of an element. We shall consider, for example, the `int` data type. If the first element happens to be an integer,

```
case i: printf("(%d,", tuple_ptr->data.val.i);
        break;
```

If some element between the first and last is an integer, where `int j` shifts the pointer to an element between the first and the last,

```
case i: printf(" %d,", (tuple_ptr + j)->data.val.i);
        break;
```

And if the last element happens to be an integer, where `int last` shifts the pointer to the last element,

```
case i: printf(" %d)", (tuple_ptr + last)->data.val.i);
        break;
```

2.7 `void saveAllTuples()`

The following is a simple function that creates a formatted text file holding information of all tuples currently in the heap, which can then be reloaded at a later time. The format specification is `<identifier> <next> <type> <val>\n`.

Parameters

i. `char path[]`: a character array specifying the file path to which the text file is to be created and saved, including the file name

A file pointer is first opened in write mode, at the specified `path[]`,

```
FILE *fp;
fp = fopen(path, "w");
```

Then, the function loops over each element in `tuples` in order, and writes down a line to the file,

```
for(int j = 0; j < tuples_size; j++){

        fprintf(fp, "%s %d ", tuples[j].id, tuples[j].next);

        // more code to follow
}
```

Depending on the type, a format string is chosen using a `switch` statement as done a number of times before now. Considering the integer type, a `case` statement looks something like this,

```
case i: fprintf(fp, "%c %d\n", 'i', tuples[j].data.val.i);
        break;
```

After looping through all of `tuples`, `fclose(fp);` is issued, to close the file and set an `EOF` marker.

2.8 `void loadAllTuples()`

This function provides the functionality to read tuple data from a file at the specified path, which must be formatted in the manner specified by `saveAllTuples()`. Otherwise, undefined behaviour is almost definitely guaranteed to occur. Any manual manipulation to said files may also result in undefined behaviour. Any data type errors will lead to a program exit, to restrict undefined behaviour as much as possible. During such occurrences, a graceful close is carried out, where all tuples currently in the heap are saved to the current directory.

Parameters

i. `char path[]`: a character array specifying the file path from which the file is to be loaded