COMP10002 Workshop Week 9

Outlook:

1	Discuss: Ex 1, 2, 3 of lec07
2	Structures & arrays: Q&A
3	The legacy of malloc() and free()
4	Ex 4 from lec07.pdf, and more …
5	Lab: Implement Ex 4,5,6,7 of lec07.pdf (pp 19-20)

Scenario for ex 1-3

People have titles, a given name, a middle name, and a family name, all of up to 50 characters each. People also have dates of birth (dd/mm/yyyy), dates of marriage and divorce (as many as 10 of each), and dates of death (with a flag to indicate whether or not they are dead yet). Each date of marriage is accompanied by the name of a person. Assuming that people work for less than 100 years each, people also have, for each year they worked, a year (yyyy), a net income and a tax liability (both rounded to whole dollars), and a date when that tax liability was paid.

Countries are collections of people. Australia is expected to contain as many as 30,000,000 people; New Zealand as many as 6,000,000 people.

Ex 1,2,3 from lec07.pdf

Exercise 1
Give declarations that reflect the data scenario that is described.

Exercise 2
Write a function that calculates, for a specified country indicated by a pointer argument (argument 1) with a number of persons in it (argument 2), the average age of death. Do not include people that are not yet dead.

Exercise 3
Write a function that calculates, for the country indicated by a pointer argument (argument 1) with a number of persons in it (argument 2) the total taxation revenue in a specified year (argument 3).
Now that you see the processing mode implied by this exercise, do you want to go back now and revise your answer to Exercise 1? If you did, would you need to alter your function for Exercise 2 at all?

E1: Give declarations that reflect the data scenario that is described.

People have titles, a given name, a middle name, and a family name, all of up to 50 characters each. People also have dates of birth (dd/mm/yyyy), dates of marriage and divorce (as many as 10 of each), and dates of death (with a flag to indicate whether or not they are dead yet). Each date of marriage is accompanied by the name of a person. Assuming that people work for less than 100 years each, people also have, for each year they worked, a year (yyyy), a net income and a tax liability (both rounded to whole dollars), and a date when that tax liability was paid.

Countries are collections of people. Australia is expected to contain as many as 30,000,000 people; New Zealand as many as 6,000,000 people.

E2: Write a function that calculates, for a specified country indicated by a pointer argument (argument 1) with a number of persons in it (argument 2), the average age of death. Do not include people that are not yet dead.

E3: Write a function that calculates, for the country indicated by a pointer argument (argument 1) with a number of persons in it (argument 2) the total taxation revenue in a specified year (argument 3).

The legacy of malloc() and free()

For its running, each program or function claims some memory from the (operating) system. On exit, the program/function must return these memory to the system. Compare:

	A: Static Memory Allocation	B: Dynamic Memory Allocation

1
2
 	
5
	`void f_static(int a) {`
	`  char s[100];`
	`   …`
	`}`
		`void f_dynamic(int a) {`
		`  char *s= malloc(sizeof(char)*100);`
		`   …`
		`}`

	Each function has 2 local variables `a` and `s`, which has scope from line 2 to 5. When each function is invoked, memory is ***automatically allocated*** for the 2 variables, and is automatically returned (= being *freed*) to the system (at line 5). But:	
	- `s` is an array, so 100 bytes was allocated and freed automatically.	– `s` is a pointer, so 8 bytes was allocated and returned automatically. - the 100 bytes from `malloc` is ***allocated by the programmer***, so ***the programmer must explicitly return*** it to the system!

The legacy of malloc() and free()

`p= malloc(n)` requests a chunk of `n` bytes from the system, and makes `p` points to that chunk.

`free(p)` returns the chunk pointed by `p` to the system.

RULE: one execution of `malloc()` should be paired with one execution of `free()`

	Option A	Option B
1	`void f_static(int a) {`	`void f_dynamic(int a) {`
2	`  char s[100];`	`  char *s= malloc(sizeof(char)*100);`
	`   …`	`   …`
3		`  free(s);`
4		`  s= NULL;`
5	`}`	`}`

TRUE of FALSE for Option B:

1. At line 4, `s` is still accessible, so line 4 is ok.
2. At line 4, the 100 bytes requested by `malloc()` is still accessible

How to malloc()

Need to specify the size of chunk you want to malloc. With

```
    data_t *p
```

it can be done in 2 ways:

- using `data_t : p= malloc( sizeof(data_t) )`
- [GOOD] using `p` itself : `p = malloc( sizeof (*p) )`

Rule 1: number of malloc = number of stars

Examples

```
    int **a;
    a= malloc ( sizeof (*a) )
    *a= malloc ( sizeof (**a) )
```

OR, depending on situation:

```
    a= malloc ( m * sizeof( *a ))
    a[i] = malloc ( n * sizeof (**a) )
```

How about `int ***a` ?

Rule 2: take note on number of stars left and right of "=". Note that a pair `[ ]` is equivalent to one star.

How to malloc(): remember assert() after malloc()

`p= malloc( ... ) can fail`

[GOOD] always check after malloc:

```
    p = malloc( ... );
     assert(p);
```

Remember :

one `malloc`

- one `assert` (right after)

– one `free` (later)

Quiz 1

```
typedef char namestr[NAMESTRLEN+1];
typedef struct {
   namestr given, others, family;
} name_t;
name_t *me;
```

Which ones are valid?

A	`me= malloc( sizeof(name_t)      );`
B	`me= malloc( sizeof(name_t *)   );`
C	`me= malloc( sizeof(*name_t)     );`
D	`me= malloc( sizeof(me*)        );`
E	`me= malloc( sizeof(*me)        );`
F	`me= malloc( sizeof(me)         );`

Quiz 2

```
int *A;
int n = 10, i;
```

*Which ones are ok in making `A` an array of `n` integers? Which one is the best one **for you**?*

A	`A= malloc( n * 4 );`
B	`A= malloc( n * sizeof(*A)  );`
C	`A= malloc( n * sizeof(int)  );`

D	`for (i=0; i<n; i++ )`
	`  A[i]= malloc( sizeof(int) );`

E	`A= malloc( n );`

Quiz 3

```
#define MAXLEN 20
char **A;
int n = 10, i;
```

Which ones are ok in making A an array of 10 strings, each of which can have length of up to 19 ?

A	`A= malloc( n * MAXLEN * sizeof(char) );`
B	`A= malloc( n * sizeof(*A)  );`

C	`for (i=0; i<n; i++)`
	`   A[i]= malloc( MAXLEN * sizeof(*A)  );`

D	`A= malloc (n * sizeof (*A));`
	`for (i=0; i<n; i++)`
	`   A[i]= malloc( MAXLEN * sizeof(*A[i])  );`

Quiz 4: Is the solution correct?

Exercise 4

Write a function `char *string_dupe(char *s)` that creates a copy of the string `s` and returns a pointer to it.

```
char *string_dupe(char *s) {
   char t[MAXLEN];
   strcpy(t,s);
   return t;
}
```

Quiz 5: Is the solution correct?

Exercise 4

Write a function `char *string_dupe(char *s)` that creates a copy of the string `s` and returns a pointer to it.

```
char *string_dupe(char *s) {
   char *t= malloc( MAXLEN * sizeof(char) );
   strcpy(t,s);
   free t;
   return t;
}
```

Quiz 6: When should we free the malloc-ed elements?

Exercise 4

Write a function `char *string_dupe(char *s)` that creates a copy of the string `s` and returns a pointer to it.

```
char *string_dupe(char *s) {
   char *t= malloc( strlen(s) + 1 );
   assert(t);
   strcpy(t,s);
   return t;
}
```

Short exercise on argv[]

Exercise:

In `main(int argc, char *argv[])`, array `argv[]` actually contains the element `argv[argc]`, which has value `NULL`. Note that like the `‘\0’` member at the end of each string, this `NULL` element plays the role of a *sentinel*.

Write a code fragment that print out all strings in `argv[]`, ***without*** using `argc`.

Work on Group's Board OR Implement

Supposing that `char *string_dupe(char *s)` **exists!**

Exercise 5

Write a function `char **string_set_dupe(char **S)` that creates a copy of the set of string pointers `S`, assumed to have the structure of the set of strings in `argv` (including a sentinel pointer of `NULL`), and returns a pointer to the copy.

Exercise 6

Write a function `void string_set_free(char **S)` that returns all of the memory associated with the duplicated string set `S`.

Exercise 7

Test all three of your functions by writing scaffolding that duplicates the argument `argv`, then prints the duplicate out, then frees the space.

(What happens if you call `string_set_free(argv)`? Why?)

Note: skeleton for Ex4-7, and this slide set, are available at github.com/anhvir/c102