COMP10002 Workshop Week 9

1	Struct & Arrays of struct. Dicscuss Ex1,2,3 of lec07
2	malloc/free: Ex from lec07: Ex 4, 5, 6, 7 , and Ex 8.05, 8.07
3	Case Study[time permitted]: Popygons, including ex. 8.02, 8.03, malloc/realloc/free
Note	• grey exercises for self-learning, bring questions to W10 workshop • W10 workshop goes further than quiz3, preparing for ass2
From LMS	**Workshops:** • Exercises 8.02, 8.03, 8.05, and 8.07. • Exercises 1, 2, and 3 in lec07.pdf • Then, if you still have time, start looking at Exercises 4, 5, 6, and 7 in lec07.pdf **Quiz 3:** This will be held between 4:15pm and 5:00pm Melbourne time on Friday 16 October (end of Week 10), *covering material through to the end of the Week 8 lecture videos (that is, all of lec06.pdf and the first half of lec07.pdf through to lec07-d, including malloc() and realloc()* but not linked lists or binary search trees). A practice quiz will be available by Friday 9 October. **Assignment 2:** Due 11pm Friday 30 October

Toward using multi-component objects…

Supposing that I need to keeps MST scores of students of this class together with names and ID. Supposing that each name has maximum 30 characters.

```
#define MAX_S 25
#define MAX_NAME 30
…

char names[MAX_S][MAX_NAME+1];
int ids[MAX_S];
float scores[MAX_S];
int n= 0;     // current number of students
```

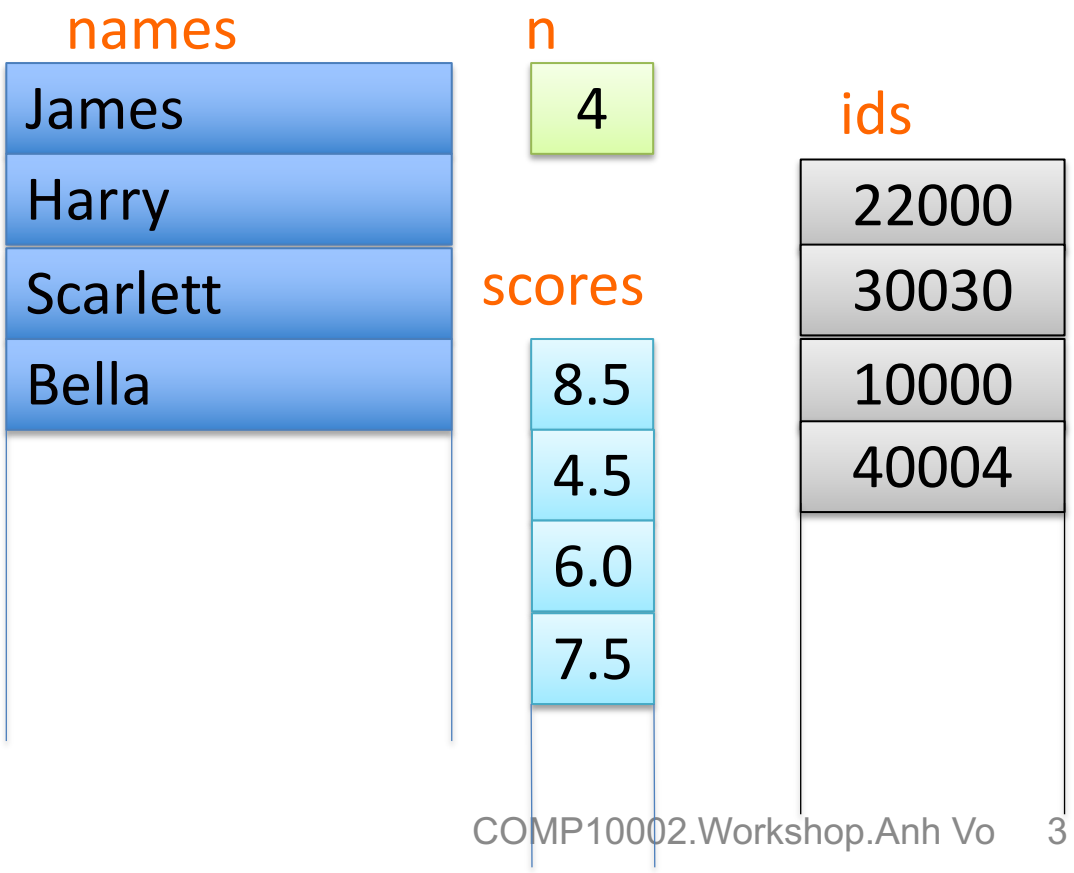
Is that a *correct* design?

Is that a *good* design?

Toward using objects…

```
while (n<4) {
    scanf("%s %d %f ", names[n], &ids[n], &scores[n]);
}
display(names[0], ids[0], scores[0]); //display the 1st stud
```

Hmmm…



Let's define a compound data type…

```
typedef struct {
   char name [MAX_NAME+1];
   int id;
   float score;
} student_t;
```

```
student_t bella= {"Bella", 40004, 7.5};
```

names ids scores


```
printf("name= %s,   id= %d,  score= %f\n",
      bella.name, bella.id,  bella.score );
```

Toward using objects…

```
typedef struct {
    ...
} student_t;
student_t studs[MAX_N]= {...};
for (n=0; n<4;n++) ...      // now n= 4
```

names ids scores n

James	22000	8.5
Harry	30030	4.5
Scarlett	10000	6.0
Bella	40004	7.5

4

Memo: Processing Structures

Initialising `struct` just like initialising arrays:

```
student_t s1= { "Bob" , 1234 , 97.75 };
```

Processing `struct` by doing so with each component (like arrays):

```
scanf("%s %d %f", s1.name, &s1.id, &s1.score);
printf("name= %s, id= %d, score= %.1f\n",
        s1.name, s1.id, s1.score);
```

But, unlike arrays, we can:

- make assignment: `s1= s2;`
- and hence, `struct` can be the output of a function:

```
student_t best_student(student_t s[], int n)
```

- but note: don't compare `struct`: `if (s1==s2)` makes nonsense

Memo: Pointers to Structures

```
typedef struct {
    char name[31];    // note that "name" is an array
    int id;
    float score;
} student_t;

student_t s1= {"Bob", 1234, 6.75}, s2;
student_t *ps= &s2;
```

The following 2 lines are equivalent:

```
scanf("%s %d %f",  s2.name,  &s2.id,  &s2.score);
scanf("%s %d %f", ps->name, &ps->id, &ps->score);
```

Note: `ps->name` *is just a shorthand* for `(*ps).name`

Arrays of structs

Arrays of structs are popular. Examples:

- a list of student records:

```
student_t class_list[MAX_S];
int n;
```

- a polygon:

```
typedef struct {
      double x, y;
} point_t;

typedef struct {
   int n;      // number of vertices
   point_t vertices[MAX_VERTICES];
} polygon_t;
```

Memo: Passing/returning struct with functions is BAD! Use pointers instead!

Compare:

```
double bad_perimeter(polygon_t p);
```
and
```
double perimeter(polygon_t *p);
```

In each case, how many bytes is passed by the following function call:

```
polygon triangle={3, {{0,0}, {1,0}, {0,1}} };
double length= bad_perimeter(p);
double length= perimeter(&p);
```

from lec07.pdf: Scenario for ex 1-3 [grok W08]

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.

lec07.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.

```
typedef char nstr[MAXL+1];
typedef struct {
  nstr given, mid, fam;
} name_t;
typedef struct {
  int dd,mm,yyyy;
} date_t;
typedef struct {
  name_t spouse;
  date_t start, end;
} mary_t;
typedef struct {
  int yyyy;
  int income, tax;
  date_t taxdate;
} work_t;
typedef struct {
  nstr title;
  name_t name;
  date_t dob;
  date_t dod;
  int is_dead;
  mary_t mary[10]; //change!
  int nm;
  work_t work[100]; //change!
  int nw;
} person_t;
```

lec07.E2: Write a function that calculates the average age of death for a country. Do not include people that are not yet dead.

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.

```
avg_longevity(  ) {

}
```

```c
typedef char nstr[MAXL+1];
typedef struct {
  nstr given, mid, fam;
} name_t;
typedef struct {
  int dd,mm,yyyy;
} date_t;
typedef struct {
  name_t spouse;
  date_t start, end;
} mary_t;
typedef struct {
  int yyyy;
  int income, tax;
  date_t taxdate;
} work_t;
typedef struct {
  nstr title;
  name_t name;
  date_t dob;
  date_t dod;
  int is_dead;
  mary_t mary[10]; //change!
  int nm;
  work_t work[100]; //change!
  int nw;
} person_t;
```

E3: Write a function that calculates, for a country, the total taxation revenue in a specified year.

```c
tax_revenue(   ) {

}
```

`malloc/realloc/free`: What (& Why & When?)

malloc= WHAT? Exercise: Write code fragment that

- use just a pointer `p` (ie. *not using another variable*), make `*p` be `10` and print out `*p`

```
int *p;
...
*p= 10;
printf("*p= %d\n", *p);
???
```

Memo: library functions malloc() and free()

```
void  *malloc(size_t n);
```

(dynamically) allocate a chunk of n bytes, and return the address of that chunk. It returns `NULL` if failed. Example:

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

--------- Need to check if `malloc` sucessful ----------

```
assert(a != NULL);   // OR
assert(a);
```

Note: memory created by malloc is not auto-freed!

```
void  free(void *p);
```

free the memory chunk that p points to. Example:

```
free(a);
a= NULL;
```

Exercise

Write a code fragment that build an array of 10 strings:

```
a
ab
abc
  …
abcdefghij
```

then print these strings. Each strings should not have any unused byte.

code:

Arrays. Scenario 1: known an upper bound of n

A task: input an array of int, output the array's elements in ascending order.

```
#define SIZE 1000

int main(int argc, char *argv[]) {
  int A[SIZE], n=0;

  for (n=0; n<MAX && scanf("%d", A+n)==1; n++);

  sort(A, n);
  print_array(A,n);
  return 0;
}
```

What if `SIZE` is unknown in advance? (ie. not available at compiler time?)

Automatic memory allocation (by compilers)

```
#define SIZE 1000

int main(int argc, char *argv[]) {
  int A[SIZE], n=0; // mem auto-allocated by compiler
  for (n=0; n<SIZE && scanf("%d", A+n)==1;
n++);
... // sorting & something else ...
  // A and n are  auto-released (auto-freed) by compiler
}
```

Memo 1: An array is controlled by three named objects:

- array name **A**
- current number of elements **n**
- capacity **SIZE**

Memo 2: memory for a variable is auto-allocated at the start of its scope, and auto-released at the end of its scope.

Scenario 2: If SIZE in unknown, and n is known at runtime:

```
int main(int argc, char *argv[]) {
   int *A, n=0; // n and pointer A auto-allocated by compiler

   scanf("%d", &n);

   A= <a memory chunk for n integers>
   // ie we demand (the system to allocate) a chunk of n integer cells
   // and we store the address of the chunk into A
   // → A would be equivalent to an array of n integers

   for (i=0; i<n && scanf("%d", A+i)==1; i++);
   …
   // Make sure that the memory chunk pointed to by A is freed
   // A and n are auto-freed by compiler
}
```

Scenario 2: SIZE in unknown, n is known at runtime:

```
int main(int argc, char *argv[]) {
   int *A, n=0; // n and pointer A auto-allocated by compiler
   scanf("%d", &n);

   A= malloc( n*sizeof(*A) );
   // that is, A=  a memory chunk for  n  integers
   //       that the system ***allocated*** for the function call malloc

   assert(A);  // terminates if  malloc failed

   for (i=0; i<n && scanf("%d", A+i)==1; i++);
   …
   free (A); // Free the memory chunk associated with  A
   // A and n are  auto-freed by compiler
}
```

Scenario 3: what if n is not available at the loop start?

```
#define INIT_SIZE 8
   int size=INIT_SIZE              // estimated capacity
   // declares an array as a triple  <A, n, size>
   int *A=NULL, n=0;
   A= malloc( size * sizeof(*A) );
   assert(A);

   for (i=0; scanf("%d", &x)==1; i++) {
      ???
      A[i]= x;
   }
   ...
```

n is not available in advance?

```
#define INIT_SIZE 8

   // declares an array as a triple <A, n, size>
   int *A=NULL, size=INIT_SIZE, n=0;
   A= malloc( size * sizeof(*A) );
   assert(A);

   for (i=0; scanf("%d", &x)==1; i++) {
     if (i==size) {
        // stuffs like free(A) then re-malloc A with a bigger memory
chunk
     }
     A[i]= x;
   }
   ...
```

n is not available in advance?

```
#define INIT_SIZE 8

  // declares an array as a triple <A, n, size>
  int *A=NULL, size=INIT_SIZE, n=0;
  A= malloc( size * sizeof(*A) );
  assert(A);

  for (i=0; scanf("%d", &x)==1; i++) {
    if (i==size) {
      size *= 2;   // makes size bigger
      // ?: free(A) then re-malloc A with a bigger memory chunk
      //    of course, with copying the old to the new chunk
    }
    A[i]= x;
  }
  ...
```

Scenario 3: If n is not available in advance:

```
#define INIT_SIZE 8

// declares an array as a triple <A, n, size>
int *A=NULL, size=INIT_SIZE, n=0;
A= malloc( size * sizeof(*A) );
assert(A);

for (i=0; scanf("%d", &x)==1; i++) {
  if (i==size) {
    size *= 2;  // makes size bigger
    A= realloc(A, size*sizeof(*A));
                // no need to copy or free the old chunk
    assert(A);
  }
  A[i]= x;
}
...
```

Memo: Tools for resizing (memory re-allocation)

```
void *realloc( void *old, size_t n);
```

resizes by performing 4 actions:

- allocate a new chunk of `n` bytes,
- copy the chunk pointed by `old` to that new chunk,
- free the memory chunk pointed to by `old`, and
- returns the address of that new chunk.

Note: similar to `malloc`, `calloc` might fail.

Example of use:

```
int *p;
p= malloc(10 * sizeof(*p));
assert(p);
...
p= realloc(p, 20 *sizeof(*p));
assert(p);
```

lec07.E4: 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;
}

… main(...) {
char *s= string_dupe("A hoax!");
```

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;
}
```

➔ Incorrect: the memory for t disappears at the end of string_dupe and cannot be used in the calling function like in:

```
char *s= string_dupe("A hoax!");
```

Is the solution correct? When should we free?

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) * sizeof(char) );
   strcpy(t,s);
   free t;
   return t;
}
```

Prev solution incorrect. When should we free?

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) * sizeof(char) );
   strcpy(t,s);
   return t;
}
... main ... {
  char *s= string_dupe("Tada!");
  ...
  free(s);
  return 0;
}
```

Case study: Working with 2D vectors Polygons

Objectives: Strengthen the understanding of:

- struct, arrays of struct
- malloc/realloc/free

Preparations:

Bring on your Editor and Terminal, and do the programming individually or together.

The Task (adapted from Exercises 8.2 and 8.3)

Define a structure `vector_t` *that could be used to store points in two dimensions* `x` *and* `y` *(such as on a map).*

Give suitable declarations for a type `poly_t` *that could be used to store a closed polygon, which is represented as a sequence of points in two dimensions. Note: there is no advance information about the number of vertices in polygons, and your code should use only as much memory as needed.*

Then:

a) *Write a function* `double distance(vector_t *p1, vector_t *p2)` *that returns the Euclidean distance between* `*p1` *and* `*p2`.

b) *Write a function that returns the length of the perimeter of a polygon.*

c) *Define a data type for a line segment (on a map) and write a function that computes the midpoint of a segment.*

Work on board in group OR Implement:

Exercise 4: Write a function `char *string_dupe(char *s)` that creates a copy of the string `s` and returns a pointer to it.

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 is available at github.com/anhvir/c102

end of today's workshop

Happy Break!

During the break, have some funs!
(aka. algorithms)

Note: Friday Week 10: quiz; Friday Week 11: no class