Notes on CP Tutorial Sheet for week 7 (2012/13):

"Structured Data"

This first question is just checking, but it's got some mean bits!

```
mark x = 2.6;
 mark is int, so this casts the float to an int by dropping the
 fractional part.
foobar y;
 declares y as a struct
y.a = x;
 sets y.a to 2
y.b = &y.a;
 sets y.b to the address of y.a - emphasize here that &y.a means  &(y.a).
x++;
 just increments x (but not of course y.a)
int z = *y.a;
 is a compile-time error, since y.a (emphasize that *y.a means *(y.a))
 is not a pointer.
 It's possible to force y.a to be treated as a pointer by doing a
 cast: int z = *((int *)y.a) , but on DICE this will still produce a
 warning, because on DICE pointers are 8 bytes not 4 bytes.
 (This was originally a bug, but we adopted it as a feature. So also
 ask them about the originally intended  int z = *y.b;)
```

"Structs, strings and arrays"

The second question gives an opportunity for reinforcing strings and arrays. One answer is:

```
typedef int id_t;
typedef enum { AUTUMN, SPRING } semester_t;
#define MAX_NAME_LENGTH 64
typedef struct {
  id_t id;
  char[MAX_NAME_LENGTH] forename;
  char[MAX_NAME_LENGTH] surname;
  semester_t entered;
} student_t;
```

Points to think about:
Do we need to "abstract" id_t away from int?
Do we need the enum for semester_t, when actually we talk about Semester 1 and Semester 2?
If we avoid the MAX_NAME_LENGTH issue by making the name fields be just char *, what risks and problems do we have? (Need to follow pointers when reading/writing the record, need to allocate memory separately for the names, etc. etc.)

"Programming"

The programming exercise is about string manipulation. The basic algorithm that should be produced (which may be tricky for students who are not native English-speakers) is:
(*) if the word ends in s, z, sh or ch, then add es.
(*) Otherwise, if the word ends in a consonant, or y preceded
   by a vowel, add s.
(*) Otherwise, if the word ends in y, change the y to ies.

(*) Otherwise add s.
Then, of course, there are the exceptions, which just have to be looked up.

The question about index is nasty: the plural of index depends on context. indexes is always acceptable (at least to assertively Anglo-Saxon English speakers), but in mathematical (and other technical) uses, indices is strongly preferred, whereas for indexes of books, indices would be very unusual. So be warned that the plural function needs a full understanding of English .....

A significant string-related issue is how much space? The function is passed a char *plural, so it can modify the string pointed to by plural, but not change plural itself. So the function had better come with a contract saying that the caller must allocate enough space in plural to store any possible plural - at least three bytes (children!) more than the length of singular. Should we pass the available length of plural explicitly? (Yes!)
You could consider the alternative of having the function return a char * that it allocates itself; or even of passing a char ** for the same purpose!