CMSC 202 | **Questions for Test 1** | Spring 1996
Sections 101, 102 & 103 | Handout #10

Exam 1 will be a full–period, closed–book exam on 29 February. The exam will be taken entirely from the following questions (although some of the questions may be changed in minor ways). This handout is designed to provide you with a study guide for the exam, and to eliminate the possibility of 'trick questions.'

# 1 General Questions

1.1. Given the following code in the indicated files:

**foo.c:**

```
main()
{
  int x = 42;
  ProcA(x);
  ProcB();
}
```

**m1.h:**

```
extern void ProcA(int);
```

**m2.h:**

```
extern void ProcB(void);
```

**m1.c:**

```
void
ProcA(int y)
{
  assert(y > 0);
  printf("%d\n", y);
}
```

**m2.c:**

```
void
ProcB(void);
{
  int y;
  scanf("%d",&y);
}
```

(a) Write any additional compiler directives such as `#include`, `#ifndef`, *etc.* needed in each file.

(b) Write a simple Makefile to compile the above code into an executable named `foo`. The Makefile must compile each `.c` file separately into a `.o` file, before building the executable `foo`.

1.2. Answer the following True/False

(a) It is legal for a recursive function to have more than one base case.
(b) It is legal for a recursive function to have more than one recursive case.
(c) It is never correct to make a recursive call with exactly the same arguments as the original call.
(d) A recursive function does not need a base case if it makes only one recursive call.
(e) A recursive function does not need a base case if every one of its recursive calls reduces the value of the arguments.
(f) In C, it is legal to add an integer to a pointer.
(g) In C, the expression `foo->fieldx` is equivalent to the expression `*foo.fieldx`.
(h) In C, a reference to an array is always converted to a pointer to the array's zero$^{th}$ element.
(i) If `A` is an array in C, then the expression `A[1]` is equivalent to the expression `*A + 1`
(j) In C, if a struct `S` contains two fields, `f1` and `f2`, then `S.f2` is equivalent to `S.f1 + 1`
(k) In C, if you dereference a `NULL` pointer, you will get the value 0.
(l) A pointer value is a pointer variable to which a value has not yet been assigned.
(m) In C, `malloc` returns a pointer value.
(n) In C, if two pointer values of the same type point to the same item, then they are the same pointer value.

1.3. Briefly (*i.e.*, in 1–3 sentences) define each of the following terms:

- Base case
- Pointer value
- Pointer variable
- Data abstraction
- Procedural abstraction
- `Assert` statement.

## 2 Recursion Problems

2.1. All exercises in the recursion handout (Handout 4).

2.2. Write a recursive function called `power` that takes a `float` $x$ and a non–negative `int` $n$ as arguments, and returns $x^n$, expressed as a `float`. Hint: $x^n$ can be defined by the following two equations:

$$\begin{aligned} x^0 &= 1.0 \\ x^n &= x \times x^{n-1} \text{ for } n \geq 1 \end{aligned}$$

2.3. Write a recursive function named `change_char` that takes as arguments an array of characters A, and two characters, old and new. Each occurrence of old in A is to be replaced by new. A is null–terminated (*i.e.*, the last character in A is the '`\0`' character.) All array manipulations are to be done by pointer arithmetic (*i.e.*, do not use subscript notation).

2.4. Write a recursive C function called `count_digit` that counts the number of occurrences of a given digit in the decimal representation of a number. `count_digit` takes two arguments: a digit (*i.e.* an `int` between zero and nine inclusive) occurrences of which are to be counted, and a number (an `int`) which might contain occurrences of that digit. For example, the call `count_digit(8, 18488)` should return three, because there are three eights in the decimal representation of 18488. Recall that if `num` is an integer, then `num/10` is the integer part of `num` divided by ten, while `num%10` is the remainder of `num` divided by ten. All looping in your function must be accomplished with recursion—you may not use any of C's iteration constructs. You may use a helper function if you like, but none is necessary.

2.5. Write a recursive C function called `double_char` that copies one string of characters into another one, replacing every instance of a given character with two copies of that character as it proceeds. `Double_char` takes three arguments: a string to copy from (a null–terminated array of characters); a character to double; and a string to copy to (another array of characters). For example, if the source string is `"xyzzy"` and the character to double is `'y'`, the target string (*i.e.*, the third argument) should be set to the string `"xyyzzyy"`. Don't forget to terminate the target string with the ASCII null character (*i.e.*, with `'\0'`). All looping must be accomplished with recursion—you may not use any of C's iteration constructs. You may use a helper function if you like, but none is necessary.

2.6. Write a recursive C function called `remove_char` that copies one string of characters into another one, removing every instance of a given character as it proceeds. `Remove_char` takes three arguments: a string to copy from (a null–terminated array of characters); a character to remove; and a string to copy to (another array of characters). For example, if the source string is `"xyzzy"` and the character to remove is `'y'`, the target string (*i.e.*, the third argument) should be set to the string `"xzz"`. Don't forget to terminate the target string with the ASCII null character (*i.e.*, with `'\0'`). All looping must be accomplished with recursion—you may not use any of C's iteration constructs. You may use a helper function if you like, but none is necessary.

2.7. Assume you have the following function:

```
int SplitNum(int num, int *left, int *right)
```

The function takes an integer `num`, and two integer pointers `left` and `right`. It returns the number of digits in `num`. Upon return, `left` points to an integer composed of the left half of the digits in `num` and `right` points to an integer composed of the right half. For example,

```
{
  int ndigits, left, right;
  ndigits = SplitNum(123, &left, &right);
}
```

results in `ndigits` equal to 3, `left` equal to 12, and `right` equal to 3.

Write a recursive function `int CountDigits(int n, int d)` that returns the number of occurrences of the digit `d` in the integer `n`. `CountDigits` must use `SplitNum` to effect its recursion.

2.8. Write a recursive function called `count_occurrences` that counts the number of occurrences of a given integer in an integer array. Your function should take three arguments: an array of `int`s, the number of elements in the array, and the integer, occurrences of which in the array are to be counted.

# 3 Pointer Problems

3.1. All exercises in the pointers handout (Handout 7).

3.2. All homework 3 problems (Handout 8).

3.3. Draw the picture for the following code segment

```
typedef struct gak *gak_ptr;
typedef struct gak {
  gak_ptr field1;
  gak_ptr field2;
} gak_type;

gak_type dalloway;
gak_type lighthouse;

lighthouse.field1 = &dalloway;
lighthouse.field1->field2 = malloc(sizeof(gak_type));
dalloway.field2->field2 = lighthouse.field1;
lighthouse.field2 = dalloway.field2;
lighthouse.field2->field2->field2 = lighthouse.field2->field2;
dalloway.field1 = lighthouse.field2;
dalloway.field2->field1->field1 = lighthouse.field1->field1;
```

3.4. Draw the picture for the following code segment

```
typedef int *intptr;
typedef intptr array[3];
typedef struct {
  char nose;
  array elbow;
} frenulum;
typedef frenulum *anvil;

anvil blatz;

blatz = malloc(sizeof(frenulum));
(*blatz).nose = 'A';
blatz->elbow[0] = malloc(sizeof(int));
blatz->elbow[1] = malloc(sizeof(int));
blatz->elbow[2] = blatz->elbow[0];
*(blatz->elbow[1]) = 17;
blatz->elbow[0] = blatz->elbow[1];
*(blatz->elbow[2]) = 42;
```

3.5. Draw the picture for the following code segment

```
typedef struct click_tag *click_ptr;
typedef struct clack_tag *clack_ptr;
typedef struct click_tag {
  clack_ptr boston;
  click_ptr accent;
} click_struct;
typedef struct clack_tag {
  int car;
  click_ptr guys;
} clack_struct;

click_struct puzzler;

puzzler.boston = malloc(sizeof(clack_struct));
puzzler.accent = malloc(sizeof(click_struct));
puzzler.accent->accent = &puzzler;
puzzler.accent->accent->boston->guys = puzzler.accent;
puzzler.boston->guys->accent = puzzler.accent;
puzzler.accent->accent->boston = puzzler.boston;
puzzler.boston->guys->boston->car = 42;
```

3.6. Draw the picture for the following code segment

```
typedef int *intptr;
typedef intptr *lamar;
typedef intptr bob[3];
typedef bob *pat;

pat buchanan;
bob dole;
lamar alexander;
buchanan = &dole;
alexander = dole + 1;
dole[1] = malloc(sizeof(int));
(*buchanan)[2] = *alexander;
*dole = NULL;
**(alexander + 1) = 17;
```

3.7. Draw the picture for the following code segment

```
typedef struct charlie {
  struct charlie *chocolate;
  int factory;
} wonka;
typedef wonka *candy;
typedef candy bar[3];
typedef candy *willy;

willy augustus;
bar gloop;

gloop[0] = malloc(sizeof(wonka));
gloop[2] = gloop[0];
*(gloop + 1) = malloc(sizeof(wonka));
gloop[1]->chocolate = gloop[0];
augustus = gloop + 2;
(*(gloop + 2))->chocolate = gloop[1];
(**augustus).factory = 17;
(*augustus)->chocolate->factory = 42;
```

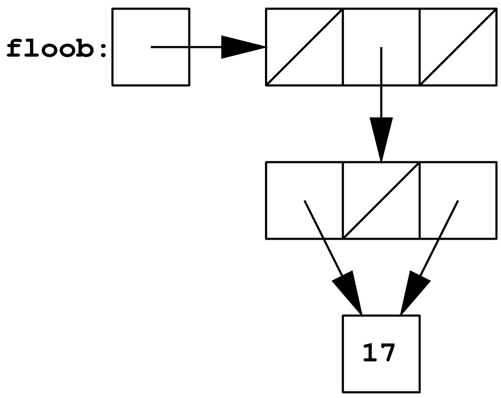
3.8. Draw the picture for the following code segment

```
typedef int *femur;
typedef femur tibia[3];

tibia radius;
int ulna;

radius[1] = malloc(sizeof(int));
radius[0] = &ulna;
radius[2] = radius[0];
*radius[0] = 88;
ulna = 12;
*radius[2] = 17;
radius[2] = radius[1];
*radius[1] = 42;
```

3.9. For each of the following pictures, show a sequence of `typedef`s, variable declarations, and statements that will build the structure depicted. Remember that an arrow pointing to a box represents a pointer to the entire box, while an arrow to a box–within–a–box represents a pointer to just that one array element.



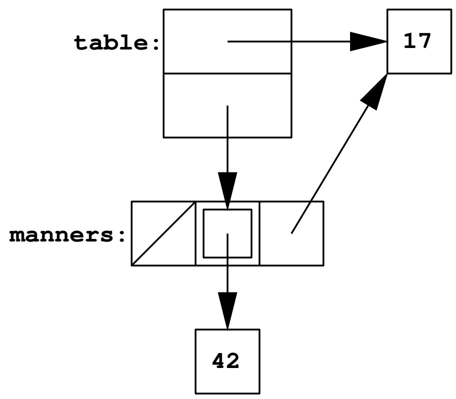
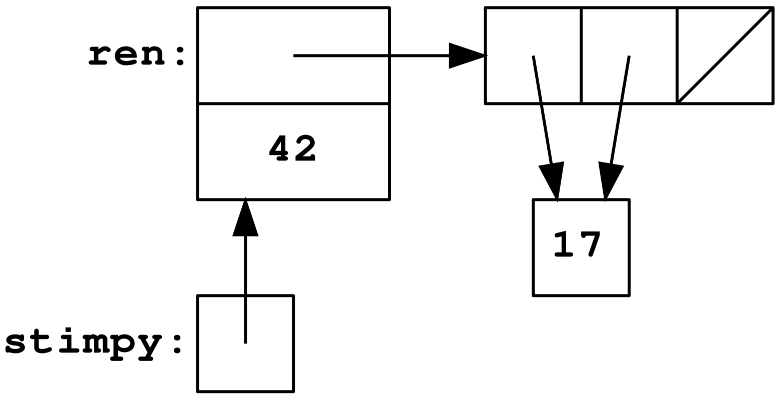
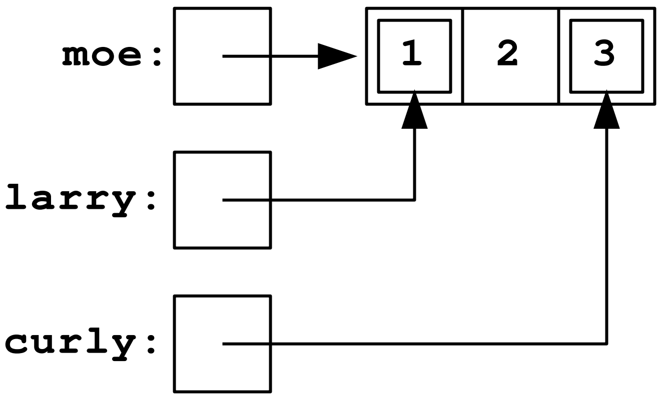
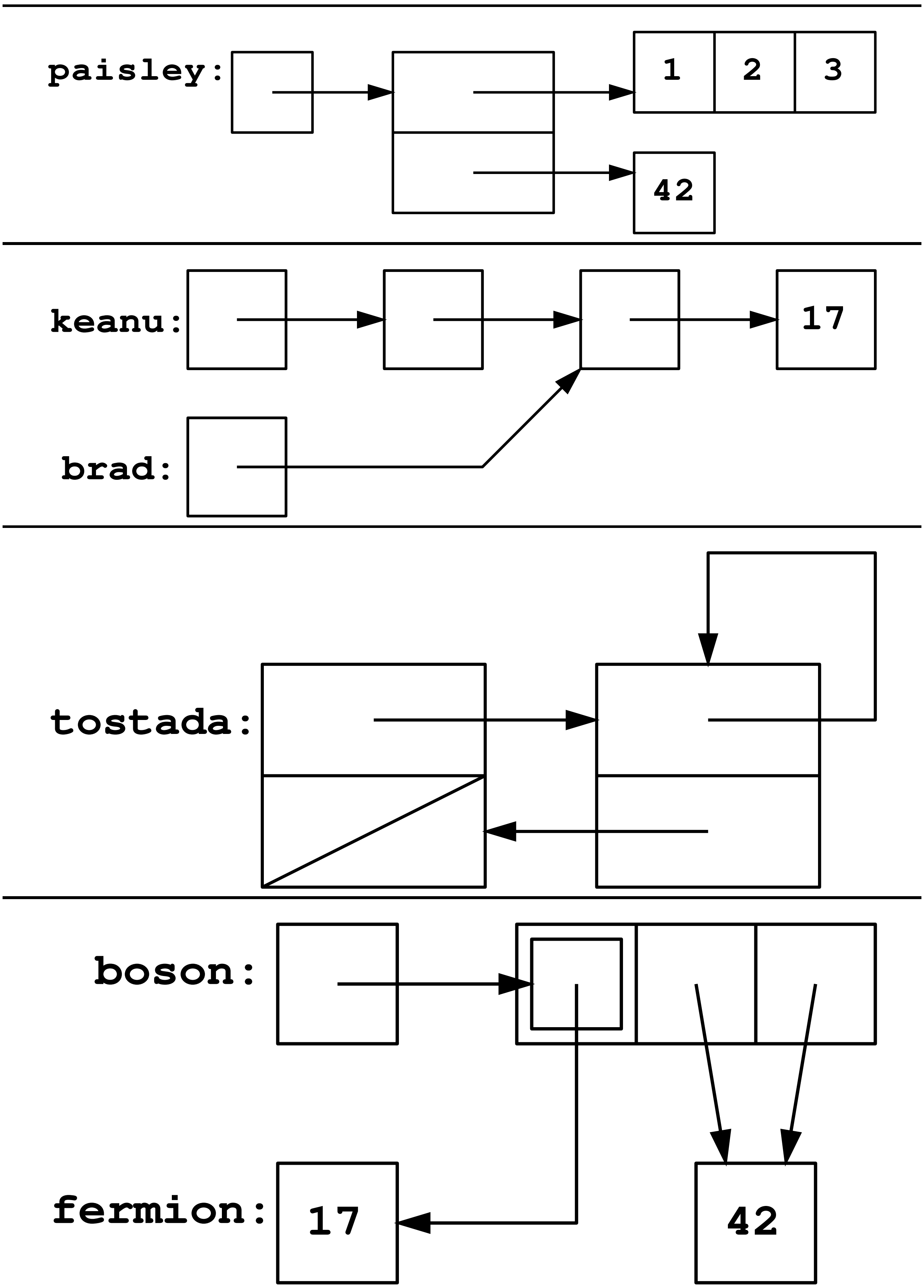


table:
17
manners:
42


ren:
42
17
stimpy:


moe:
1
2
3
larry:
curly:


paisley:
1
2
3
42
keanu:
17
brad:
tostada:
boson:
fermion:
17
42