CMSC 202 **Pointers** Spring 1996
Sections 101, 102 & 103 Handout #7

One of the best ways to design linked structures, or to understand a design of linked structures, is to use pictures. This handout gives a method for pictorially visualizing linked structures, shows examples of the use of the method, and suggests a number of exercises on the material presented.

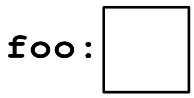
1 Drawing Linked Structures

The following conventions are useful for picturing linked structures:

- A variable is drawn as a labeled box. For example, if `foo` is declared to be an `int`, as in:

  ```
  int foo;
  ```

  then `foo` would be drawn as:

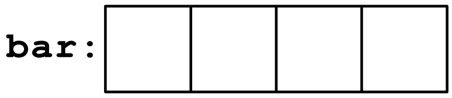
  

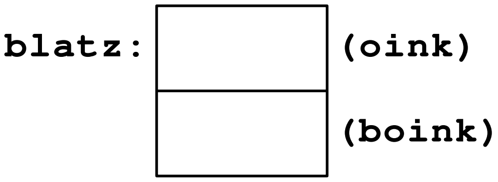
  The box represents the storage space that the compiler allocates for the variable. Named variables are *always* drawn with an associated box; there will never be a situation in which you write a variable name but do not draw its corresponding box. That's because whenever you declare a variable, the compiler allocates space for that variable.

- Complex variables are drawn as complex boxes. For example, the variables `bar` and `blatz` in the code:

  ```
  typedef int array_type[4];
  typedef struct {
    int oink;
    char boink;
  } struct_type;

  array_type bar;
  struct_type blatz;
  ```

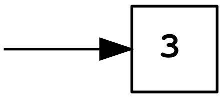
  would be drawn as:





Note the convention that array elements are spread out horizontally, while struct fields are spread out vertically.

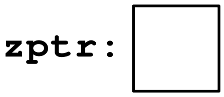
- A pointer *value* is drawn as an arrow. The arrow points to a picture of the object to which the pointer points. The pointer is said to *reference* this object. For example, a pointer to the integer `3` would be drawn as:



- A pointer *variable* is drawn as a labeled box, just like any other variable. For example, if `zptr` is defined as:

```
typedef int *int_ptr;

int_ptr zptr;
```

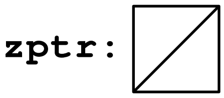
then `zptr` is drawn as:



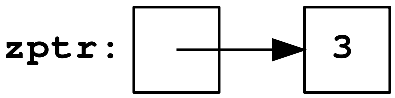
- `NULL`, which represents a pointer value that doesn't point to anything, is drawn as a slash through the box containing it. For example, if we initialize `zptr` to `NULL` with:

```
zptr = NULL;
```

then `zptr` will now look like:



- If a pointer variable actually points to an object, an appropriate pointer value is drawn in the box for that variable. However, only the *base* of the arrow representing the pointer value is placed in the box; the object to which the arrow points is drawn outside of the box. For example, suppose that `zptr` has been initialized to reference the number `3`. The picture of `zptr` will look like:



- A pointer variable receives a value through assignment. The space that the assigned pointer value references can either be newly-allocated space (as allocated by a function called `malloc`), or previously-allocated space. Consider the use of `malloc` first. `Malloc` allocates a new block of space, and returns a pointer to that space. The argument to `malloc` is the number of bytes of space needed. This value is almost always calculated using the `sizeof` function. `Sizeof` takes either the name of a type or the name of a variable as its argument, and returns the number of bytes occupied by an object of that type or by that variable. For example, the expression `sizeof(int)` returns `4` on many modern machines, while the expression `malloc(sizeof(int))` returns a pointer to `4` bytes of space on those same machines. Don't forget that to use `malloc` you must first `#include <stdlib.h>`.

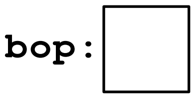
  We can graphically depict the result of `malloc` in the following way:

  1. Figure out what kind of space is being allocated.
  2. Draw a box to represent the allocated space.
  3. Draw an arrow pointing to the new box, and place its base in the box representing the variable being assigned to.

  As an example, consider the following definitions and declarations:

```
typedef struct {
  int buff;
  int puff;
} blotto_struct;
typedef blotto_struct *blotto_ptr;

blotto_ptr bop;
```

  We can draw `bop` like so:

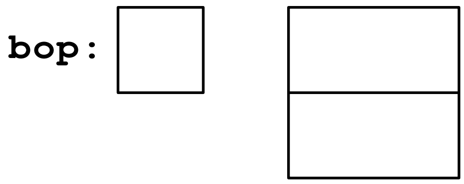


  Suppose we now do:

```
bop = malloc(sizeof(blotto_struct));
```

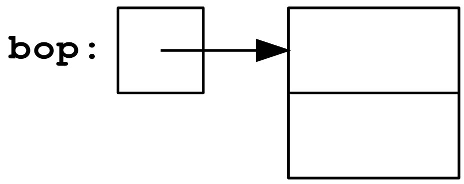
  To draw the result of this call to `malloc` and its corresponding assignment, we must perform the three steps outlined above. First, we need to determine what kind of space is being allocated. To do this, we examine the declaration of `bop`, and the pertinent `typedef`s. `Bop` is of type `blotto_ptr`, which means that it is a pointer to a `blotto_struct`. A `blotto_struct` is a struct of two fields. So, we will be allocating a struct of two fields.

  Step two of drawing the results of this call to `malloc` is to draw the appropriate box. We add a drawing of a struct of two fields (represented as a box with two sections) to our initial picture to get:



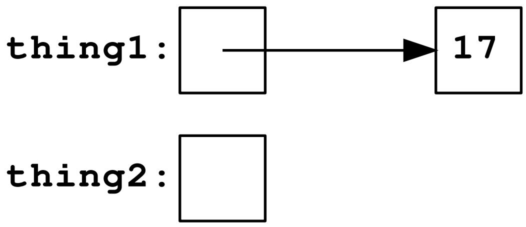
Notice that the new box has no name associated with it, so there is no way to access it directly.

The final step is to draw an arrow to the new box, and place its base in the box representing the variable on the left-hand side of the assignment statement. The variable being assigned to is `bop`, so this gives us:



The arrow in the above picture is not pointing to the first field of the struct. Rather, it is pointing to the struct as a whole.

- The other way to give a value to a pointer variable is to make it point to an existing box. Suppose that `thing1` and `thing2` are both pointers to `int`s, and that `thing1` has been initialized to reference the number 17. These variables are drawn as:



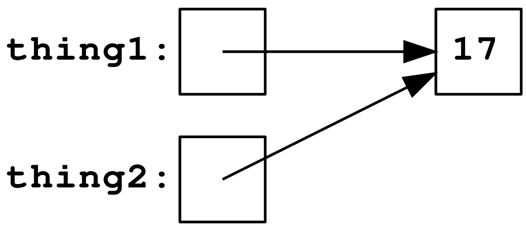
Suppose that we now execute:

```
thing2 = thing1;
```

To draw the results of a pointer assignment, we perform two steps:

1. Draw a new arrow that points to the box referenced by the pointer value on the right side of the assignment statement.
2. Place the base of the new arrow in the box representing the pointer variable on the left side of the assignment statement.

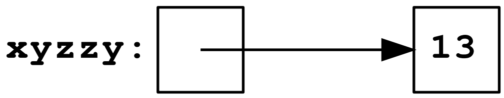
If we follow these two steps for the above assignment statement, we will get a new arrow that references the box containing 17, whose base is in the box representing `thing2`:



Another way to visualize pointer assignment is to imagine making a rubber clone of the arrow being assigned, then stretching the new arrow and placing its base in the box being assigned to, while holding its head fixed.

2 Using Drawings of Linked Structures

Now let's look at how components of a linked structure are selected or modified. The key to accessing a component of a linked structure is the use of the * operator. For example, suppose that `xyzzy` is a pointer variable that references the int `13`, and that we want to print out this number. The picture of `xyzzy` looks like:



We can print the `13` with the statement:

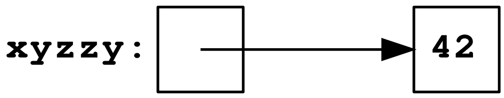
```
printf("%d", *xyzzy);
```

Here, the * means 'dereference the pointer value contained in `xyzzy`.' In terms of our pictorial representation, the * means 'follow the arrow.' If we break the expression `*xyzzy` up into its component parts, we get two pieces: `xyzzy` and `*`. The name `xyzzy` tells us to go to the variable labeled `xyzzy` in our picture. The * tells us to follow the arrow contained in the box for `xyzzy` to the box containing the `13`. Thus, `printf("%d", *xyzzy)` prints `13`.

Now suppose we want to change the value that `xyzzy` references. We can use the same expression on the left-hand side of an assignment statement:

```
*xyzzy = 42;
```

This statement says to start at the box representing the variable `xyzzy`, follow the arrow it contains, and place the value `42` in the resulting box. Pictorially, the result is:



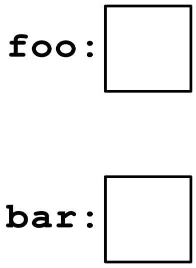
3 Example

Let's use our pictorial technique to determine what the following code does:

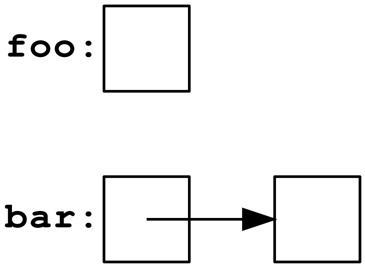
```
typedef int *int_ptr;
typedef int_ptr array_type[3];
typedef array_type *array_ptr;

{
  array_ptr foo;
  int_ptr bar;

  bar = malloc(sizeof(int));
  *bar = 42;
  foo = malloc(sizeof(array_type));
  (*foo)[0] = bar;
  (*foo)[1] = NULL;
  (*foo)[2] = malloc(sizeof(int));
  *(*foo)[2] = 17;
}
```

Before reading on, you might want to try your hand at drawing the picture that results from the execution of this code.

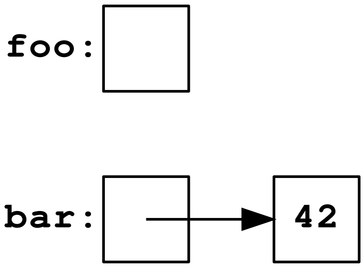
The variable declarations tell us that foo and bar are both pointer variables, so we draw them as simple boxes:



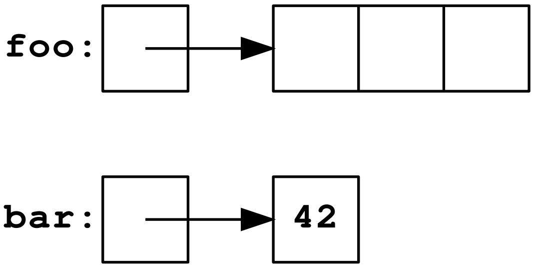
The first statement is `bar = malloc(sizeof(int))`. We determine that bar is a pointer to an int, so we draw a box for an int and make bar point to it:



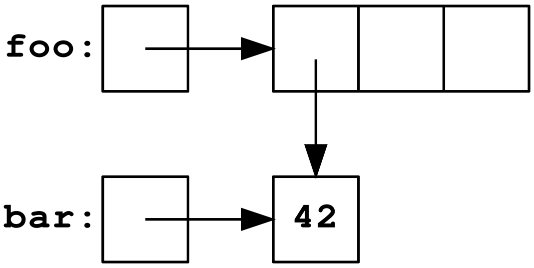
Statement two is `*bar = 42`. To draw the result of this statement, we start from the box for bar, follow the arrow, and place 42 in the box to which the arrow leads us:



Now the program executes `foo = malloc(sizeof(array_type))`. Since `foo` is a pointer to an array of three `int`s, we must allocate such an array and make `foo` point to it:

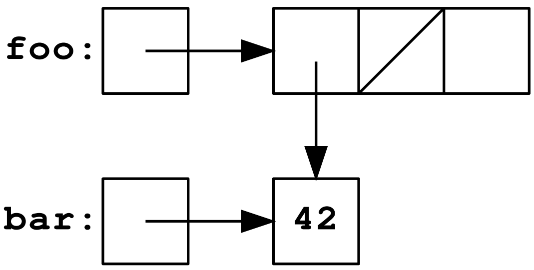


The next statement is `(*foo)[0] = bar`. Since `bar` contains a pointer value, we need to create a new arrow that references the same object, and place its base in `(*foo)[0]`:



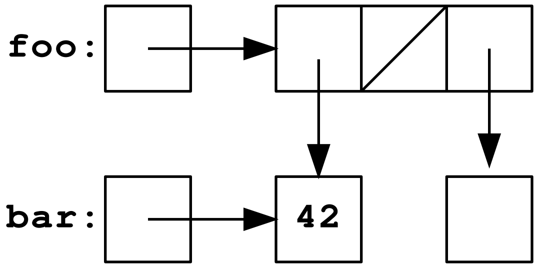
Notice that it would be incorrect to say `*foo = bar`. The value in `foo` does not point to the zeroth element of the array, but rather to the array as a whole. Thus, we need to include the `[0]` in the expression to select the particular element in which we're interested. It would also be incorrect to use `*foo[0] = bar`. Square brackets in C have higher precedence than the `*` operator, so this latter expression would be treated by the compiler as `*(foo[0]) = bar`.[1]

Next comes `(*foo)[1] = NULL`:

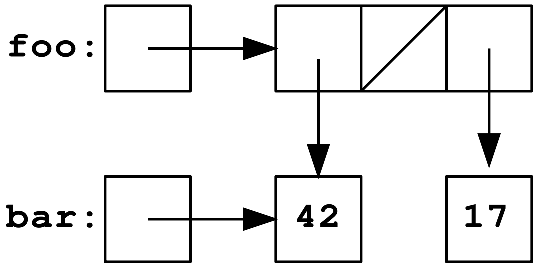


[1] In point of fact this expression would actually have the same effect as the correct expression `(*foo)[0] = bar` (for reasons that will be explained in Section 6). It is incorrect though, because it does not express the concept of first following the pointer, then subscripting the array.

Now the program creates a new pointer value, and place it in `(*foo)[2]` with the statement `(*foo)[2] = malloc(sizeof(int))`. This use of `malloc` gives us:

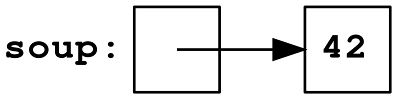


The final statement of the program is `*(*foo)[2] = 17`. To get to the box in which to draw the `17`, we have to start from the box labeled `foo`. We follow the arrow contained in `foo`'s box using the innermost `*`, select the number two element of the array we arrive at using `[2]`, and finally follow this arrow by once again using `*` (this time, the outermost one). This locates the box into which we will place the value `17`. Our final picture looks like:

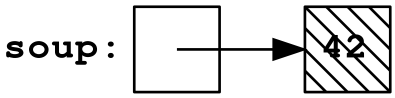


4 The `Free` Function

`Free` is a function provided by `stdlib.h` that allows us to recycle space previously allocated by `malloc`. We can gain insight into the `free` function by using pictures. Thus, if `soup` is a pointer variable that references the number 42:



then `free(soup)` will recycle the space *referenced* by `soup`:

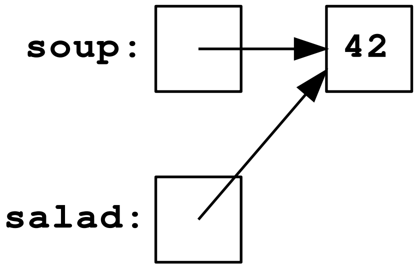


This space can later be reused by some other call to `malloc`. Thus, we can no longer be sure of the value contained in the box referenced by `soup`; `free` has effectively converted this space (which in our example contained the number 42) to garbage. We can be sure though that the box for `soup` will still exist; it is a named box, and as such cannot be created by `malloc` or destroyed by `free`.

Now suppose that we have executed the following code:

```
soup = malloc(sizeof(int));
*soup = 42;
salad = soup;
```

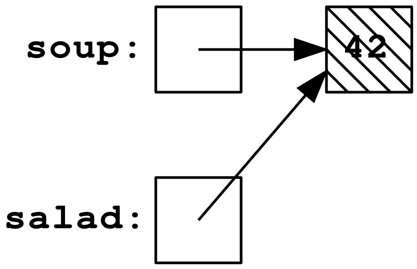
This gives us the following picture:



If we now execute:

```
free(soup);
```

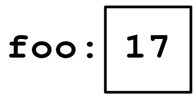
we will recycle the space referenced by `soup`. This gives us the picture:



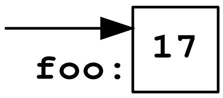
Notice that the space referenced by `salad` has also been recycled, even though the call to `free` never mentioned `salad`! The reason is that `free` affects the space referenced by its argument, not the argument itself. When you use `free`, it is your responsibility to ensure that you don't recycle space that is still in use.

5 The & Operator

In our pictorial representation, the `&` (or *address of*) operator allows us to create an arrow that points to an existing box. For example, if `foo` is an integer variable whose value is `17`:



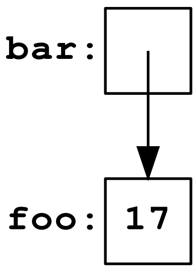
then &foo gives us a pointer to foo:



The code:

```
int foo;
int *bar;

foo = 17;
bar = &foo;
```

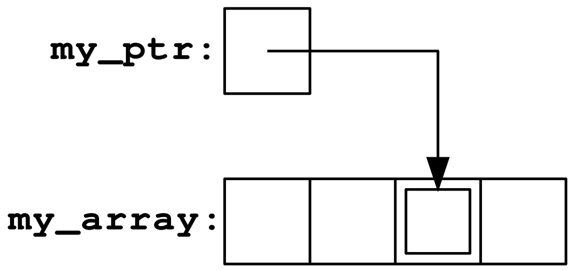
thus gives us the picture:



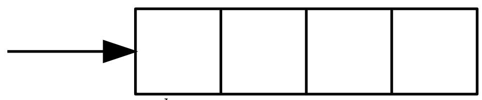
The & operator can be used on any box. For example, it is possible to obtain a pointer to a single element of an array:

```
int my_array[4];
int *my_ptr;

my_ptr = &(my_array[2]);
```

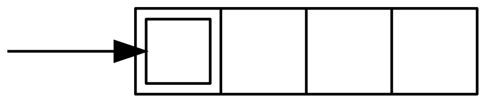
My_ptr now points to the [2] element of my_array:



To distinguish such a pointer from a pointer to an entire array, we highlight the referenced element with an extra box. Thus, this pointer value references an entire array:



while this one references only the zeroth element of the array:



The former is of type pointer to array, while the latter is of type pointer to array element.

6 Pointers and Arrays

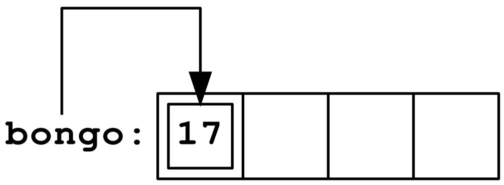
In C, there is a strong relationship between pointers and arrays. First, the name of an array is defined to be a pointer to that array's zeroth element. Thus, if `bongo` is an array:

```
int bongo[4];
```

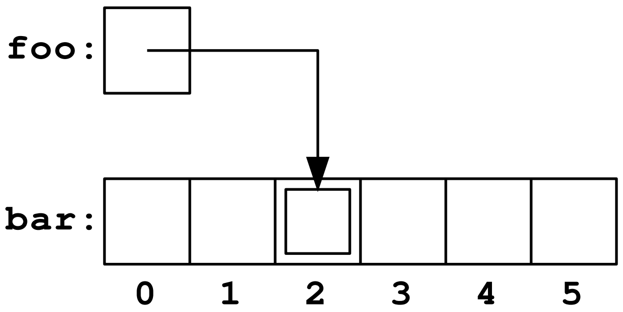
then the name `bongo` can be used as a reference to the zeroth element of the array:

```
*bongo = 17;
```

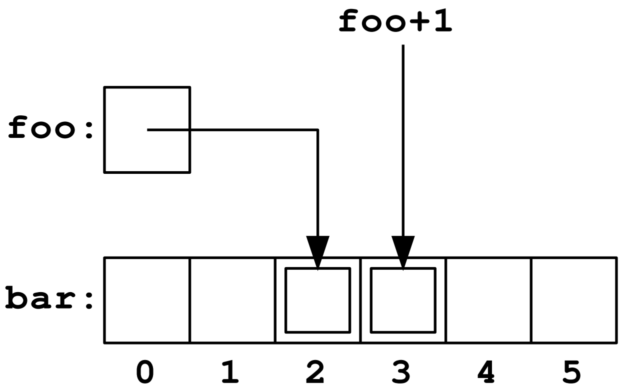
Pictorially, we can view this scenario as:



Second, if we have a pointer value that references a particular element of an array, then adding an integer n to that pointer will result in another pointer value that references the array element n elements away from the original element. For example, if `foo` references the `[2]` element of `bar`:



then `foo+1` will reference the `[3]` element of `bar`:



We can put these two facts together, and observe that the n^{th} element of an array `blatz` can be referenced with `blatz+`n. Thus, the expressions `blatz[`n`]` and `*(blatz+`n`)` are entirely equivalent, as are the expressions `&blatz[`n`]` and `(blatz+`n`)`.

7 Exercises

Here are several exercises that might help you to understand pointers and pictures of linked structures. Skill at performing these types of exercises is invaluable both for the remainder of this course, and for any future coding you might do involving pointers. In particular, similar problems will appear on homeworks and tests. The solutions to these exercises will appear in a separate handout.

1. Draw the results of the following code:

```
typedef char karpov[3];
typedef karpov *kasparov;

kasparov kamsky;

kamsky = malloc(sizeof(karpov));
(*kamsky)[0] = 'p';
(*kamsky)[1] = 'k';
(*kamsky)[2] = '4';
```

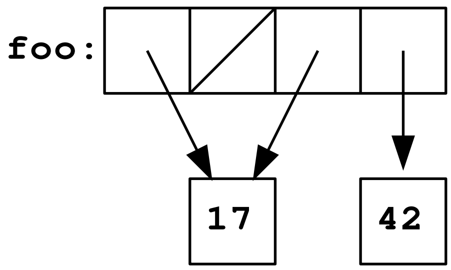
2. Draw the results of the following code:

```
typedef struct {
  int mince;
  char quince[3];
} soprano;
typedef soprano *alto;
typedef alto tenor[4];
typedef tenor *bass;

alto chapman;
bass redbone;

chapman = malloc(sizeof(soprano));
chapman->mince = 1;
chapman->quince[chapman->mince] = 'G';
redbone = malloc(sizeof(tenor));
(*redbone)[0] = malloc(sizeof(soprano));
(*redbone)[0]->quince[2] = 'Q';
(*redbone)[3] = chapman;
```

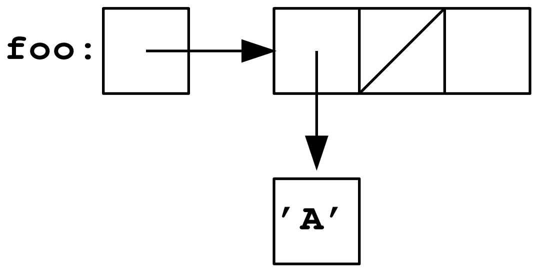
3. Draw the results of the following code:

```
typedef int *bart;
typedef bart *lisa;

lisa homer;
bart marge;

homer = malloc(sizeof(bart));
*homer = malloc(sizeof(int));
**homer = 42;
marge = *homer;
```

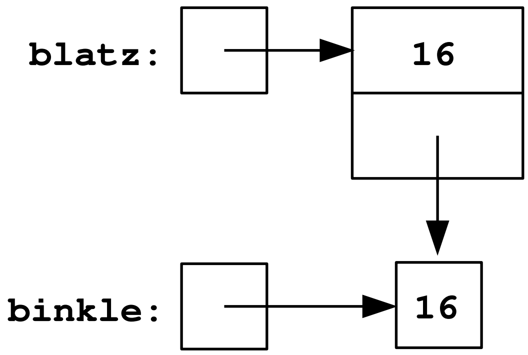
4. Write the typedefs, variable declarations, and code that will create the following picture:



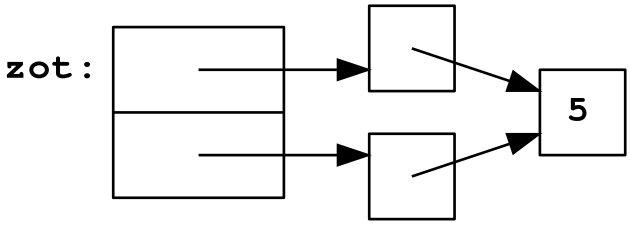
5. Write the typedefs, variable declarations, and code that will create the following picture:



6. Write the typedefs, variable declarations, and code that will create the following picture:



7. Write the typedefs, variable declarations, and code that will create the following picture:



8. Write the typedefs, variable declarations, and code that will create the following picture: