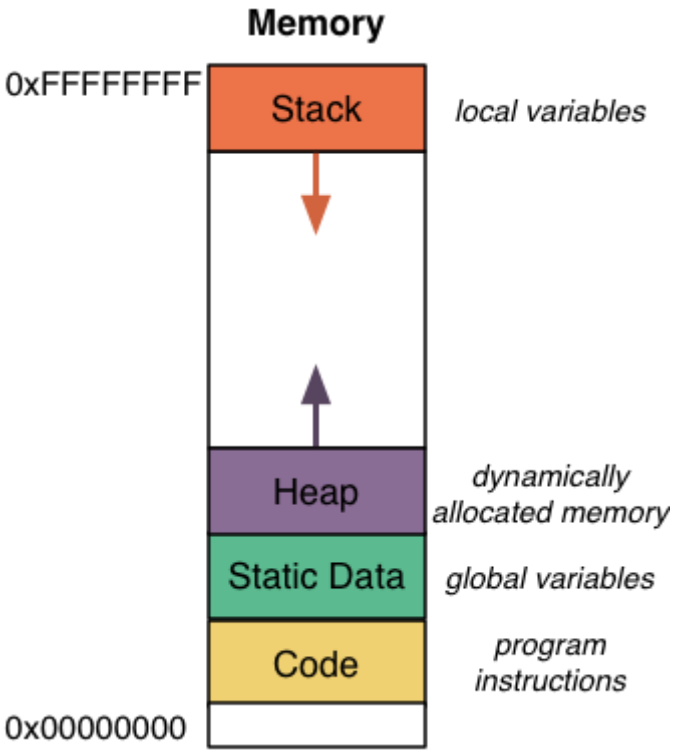
NULL

/run/media/lijunjie/资料/Learning Video/pointers-arrays-recursion/01_pointers/03_pointers-in-hardware/04_null_instructions.html

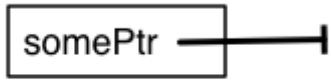
At the bottom of the figure below, there is a blank space below the code segment, which is an invalid area of the program's memory. You might wonder why the code does not start at address 0, rather than wasting this space. The reason is that programs use a pointer with the numeric value of 0—which has the special name, NULL to mean "does not point at anything." By not having any valid portion of the program placed at (or near) address 0, we can be sure that nothing will be placed at address 0. This means no properly initialized pointer that actually points to something would ever have the value NULL.



Knowing that a pointer does not point at an actual thing is useful for a variety of reasons. For one, we may sometimes have algorithms which need to answer "there is no answer". For example, if we think about our closest point example discussed earlier when the set of points is empty, we must return "there is no answer"—with pointers, we can return a pointer to a point, and return NULL in the "there is no answer" case. We may also have functions whose job it is to create something that return NULL if they fail to do so we will see functions that open files (so we can read data from the disk) which fail in this manner. Later in the course we will see how to dynamically allocate memory, which also return NULL if the memory cannot be allocated.We will also begin to learn about linked structures which store data with pointers from one item to the next, and use NULL to indicate the end of the sequence.

When we use NULL, we will represent it as an arrow with a flat head (indicating that it does not point at anything), as shown in the figure below. Whenever we have NULL, we can use the pointer itself (which just has a numeric value of 0), however, we cannot follow the arrow (as it does not point at anything). If we attempt to follow the arrow (*i.e.* , dereference the pointer), then our program will crash with a *segmentation fault* —an error indicating that we attempted to access memory in an invalid way (in fact, if our program tries to access any region of memory that is "blank" in the figure above, it will crash with a segmentation fault).

The NULL pointer has a special type that we have not seen yet— **void** * . A *v oid pointer* indicates a pointer to *any* type, and is compatible with any other pointer type—we can assign it to an **int** * , a **double*** , or any other pointer type we want. Likewise, if we have a variable of type **void** * , we can assign any other pointer type to it. Because we do not know what a **void** * actually points at, we can neither dereference it (the compiler has no idea what type of thing it should find at the end of the arrow), nor do pointer arithmetic on it.




```
somePtr = NULL;
```