Lecture 10: Pointers

Pointers to pointers

Multidimensional arrays

We already discussed how it is possible to declare multidimensional arrays. For example, to declare a two-dimensional 2×3 element array of floats, you use

```
float a[2][3];
```

This statically declares a contiguous block of memory that is large enough to store 6 floats. Even though this is a multidimensional array, in memory it is still a single-dimensional array. The way C accesses element `a[i][j]` of this array is by accessing element `3*i+j` of the one-dimensional block of six floating point elements in memory. You can access the elements of this array with

```
((float *)a)[3*i + j];
```

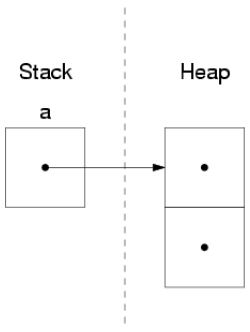
This will correctly access the elements of the array as a 1-dimensional array even though it was declared as a 2-d array. The reason this concept is important is because there are other ways to allocate a two-dimensional array, which is by using pointers to pointers. We can declare the two-dimensional array above as a pointer to a pointer with

```
float **a;
```

This simply declares a block in memory that is still 4 bytes that stores a pointer to a pointer of type `float *`. We can dynamically allocate an array that contains two pointers to floats with

```
a=(float **)malloc(2*sizeof(float *));
```

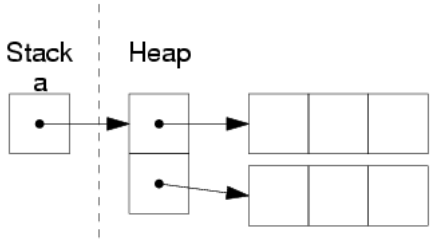
The pointer diagram for this declaration looks like



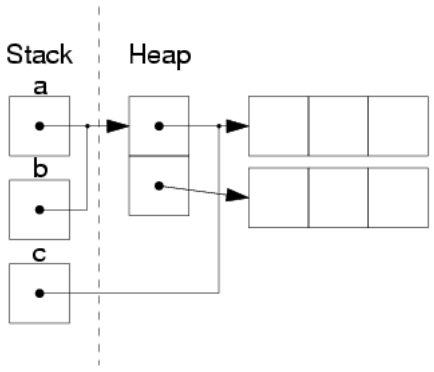
Now, to declare more space to store the actual values, we set each pointer to point to individual blocks of 3 floats in memory with

```
for(i=0;i<2;i++)
  a[i] = (float *)malloc(3*sizeof(float));
```

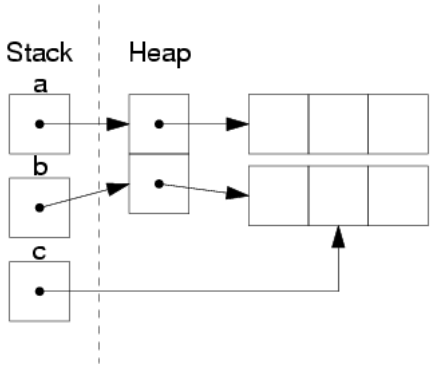
So that now the pointer diagram looks like:



The elements of this array can be accessed in the same way then can for arrays declared on the stack, such as with `a[1][1]`. You can determine exactly how the compiler determines this value using pointer arithmetic. The value of `a[i][j]` in pointer notation is given by `*(*(a+i)+j)`. If we decare `float **b = a`, and `float *c = *b`, then the pointer diagram looks like



Now we can perform pointer arithmetic to extract the value `a[1][1]` by setting `b=a+1` and `c=*b+1`, so that the diagram looks like



If we dereference the pointer `c`, then we get the value of `a[1][1]`, so the pointer arithmetic `x=*(*(a+1)+1)` is identical to `x=a[1][1]`.

Passing multi-dimensional arrays to functions

If we wanted to pass the above multi-dimensional arrays to functions, we need to be clear about the function declaration. To pass the statically allocated array to a function, we must specify the number of columns. For example, to declare a function that takes the array `float a[2][3]` as its argument, we would use

```
void Function(float a[][3])  { ... }
```

We could also have used `a[2][3]`, but C does not require the declaration of the first array index. The reason we need to do this is because the compiler needs to know the number of columns in order to jump from one row to the next in a statically declared array.

Alternatively, to declare a function to take a double-pointer `float **a`, you use

```
void Function(float **a) { ... }
```

Then, inside this function, you can refer to the elements of the multi-dimensional array using the standard `a[i][j]` notation. But this form of a function declaration will only work for dynamically declared arrays.

Definition of static multidimensional arrays

Since C stores static multidimensional arrays in one contiguous block of memory, you only need to declare the number of columns in these arrays when you initialize them, as in

```
float a[][3] = {0,1,2,3,4,5};
```

This will automatically create an array with 2 rows and three columns, even though the declaration consisted of a 1-d array of 6 elements. Of course, you can also explicitly declare the number of columns with

```
float a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Array of strings

An array can also consist of pointers to strings. You can initialize these arrays statically with

```
char *strings[] = {"UWC","Bellville","Cape Town"};
```

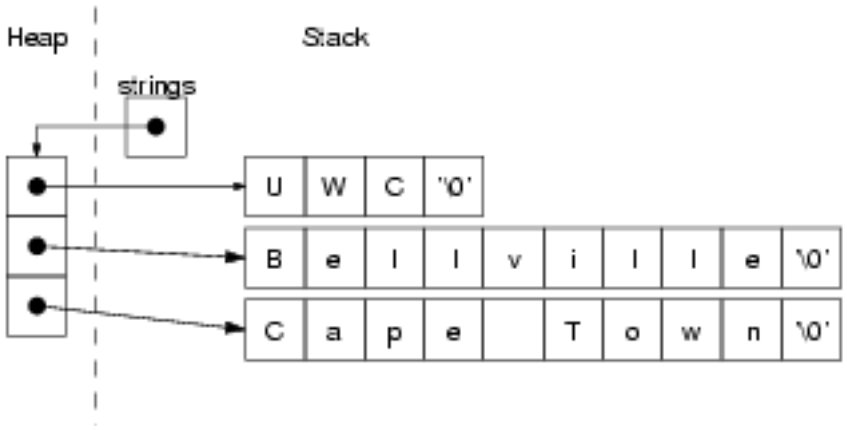
and these strings can be accessed with

```
printf("I work at %s in %s, right outside of %s\n",
        strings[0],strings[1],strings[2]);
```

You can also declare the array of strings dynamically, with

```
char **strings = (char **)malloc(3*sizeof(char *));
strings[0] = "UWC";
strings[1] = "Bellville";
strings[2] = "Cape Town";
```

But this defeats the purpose of the dynamic allocation, since the character constants allocate the strings on the stack, so that the pointer diagram looks like (taking careful note that the heap is unconventially on the left!):



A better, albeit more involved, way of dynamically storing strings in the heap is with

```
char **strings = (char **)malloc(3*sizeof(char *));
strings[0]=(char *)malloc((strlen("UWC")+1)*sizeof(char));
strcpy(strings[0],"UWC");
strings[1]=(char *)malloc((strlen("Bellville")+1)*sizeof(char));
strcpy(strings[1],"Bellville");
strings[2]=(char *)malloc((strlen("Cape Town")+1)*sizeof(char));
strcpy(strings[2],"Cape Town");
```

where we have been sure to allocate one more than the string length so that we could correctly store the null character `'\0'`. In the heap, the above code would look like:

Functions that take pointers to strings as their arguments do not need any special attention like statically declared multi-dimensional arrays. A string that is declared in either of the above two ways can be passed to a function which takes an array of strings that is defined as

```
void PrintStrings(char **strings) { ... }
```

This is because the compiler takes care of string declarations for us. If you declared a 2-d static character array with `strings[3][20]` and copied the strings into this array with

```
strcpy(strings[0],"UWC");
strcpy(strings[1],"Bellville");
strcpy(strings[2],"Cape Town");
```

a call to a function declared as `PrintStrings(strings);` would only work if it was declared as

```
void PrintStrings(char strings[][20]) { ... }
```

or

```
void PrintStrings(char strings[3][20]) { ... }
```

Pointers to structures

C provides the ability to declare pointers to structures. This is probably the most useful aspect of C, but it is also one of the more difficult aspects to understand. Consider the structure we defined in the last lecture as

```
typedef struct {
  string name;
  string IDNumber;
  int graduationYear;
  int testGrade;
} studentT;
```

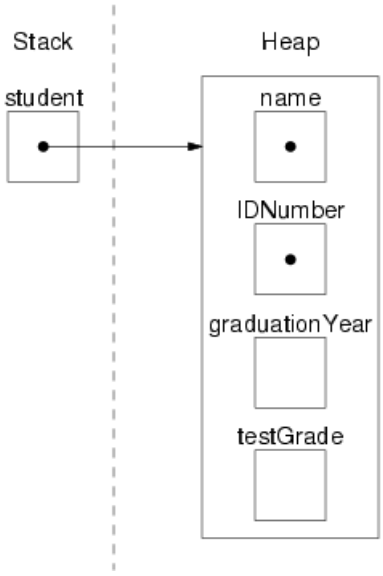
More often than not, we would like to create space for new structures in the heap rather than in the stack. For example, you may be reading in data about students from a file, and you would like the ability to create new `studentT` structs in the heap. To do so, you first declare a pointer to a `studentT` with

```
studentT *student;
```

and then you dynamically allocate space for a `studentT` type with

```
student = (studentT *)malloc(sizeof(studentT));
```

Using a pointer diagram, this would look like:



You can access the different fields of this dynamically allocated structure with, for example,

```
(*student).testGrade = 10;
```

The reason you need the parentheses is because the `.` operator takes precedence over the `*` operator here. Since you cannot access the field of a pointer to a struct, you need to dereference that pointer and then access that particular field. Since this is used so often and it is cumbersome to type, C provides a clever shorthand for this, which is given by

```
student->testGrade = 10;
```

You can think of this as saying "follow the pointer from the student type to its destination and retrieve (or set) that field."

Pointer arithmetic also applies to structures as well. For example, if you declare an array of structures and a pointer to a structure with

```
studentT students[10], *currentstudent;
```

then you can access the name of the fifth student with

```
currentstudent=students+4;
*currentstudent.name="You Doubleusee"
```

Self-referential structures

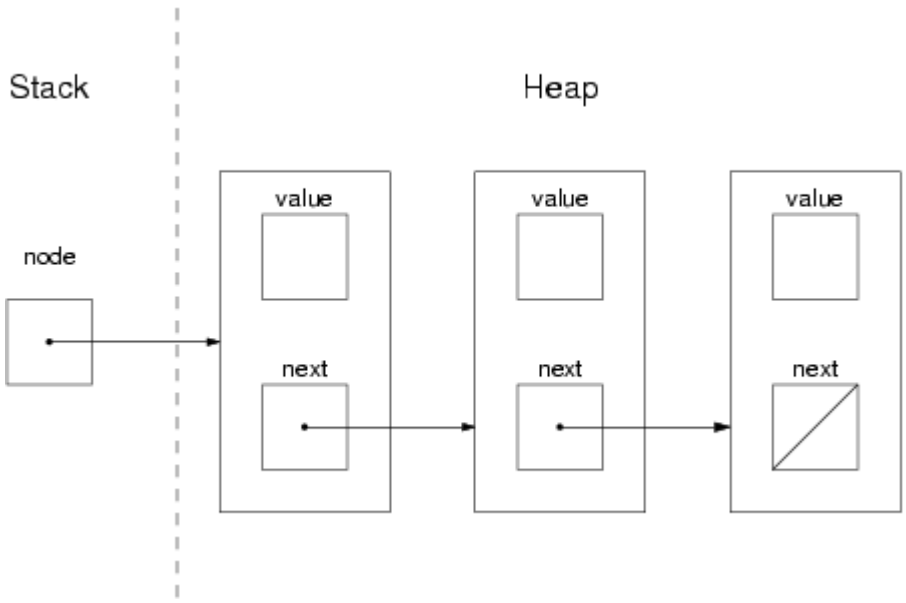
Even more confusing (and much more powerful) is the concept of a self-referential structure, or a structure that has a member that is of the same type as the structure in which it is defined. A common self-referential structure is a node in a linked-list, which is a member of a set of structures that are linked to one another with pointers. Each node of the linked-list is defined by the structure

```
typedef struct _nodeT {
  float value;
  struct _nodeT *next;
} nodeT;
```

Each node stores a `value` which is a floating point number, as well as a pointer to another node, which is given by the `next` field. For example, we can create three linked nodes in the heap with

```
nodeT *node = (nodeT *)malloc(sizeof(nodeT));
node->next = (nodeT *)malloc(sizeof(nodeT));
node->next->next = (nodeT *)malloc(sizeof(nodeT));
node->next->next->next = NULL;
```

the pointer diagram of this declaration would look like



The NULL pointer

In the above diagram, the `next` field of the last structure has a diagonal line through it. This represents the `NULL` pointer. When you initialize a pointer with `int *a`, the address that it initially stores depends on the compiler you are using. Essentially, its initial address is meaningless. You can set pointers to point to nothing, or `NULL`, with `a=NULL`. That way you can easily check if that pointer points to `NULL` with the statement

```
if(a==NULL) printf("a is NULL!\n");
```

or simply with

```
if(!a) printf("a is NULL!\n");
```

The `malloc` function returns `NULL` if there is no more memory, so whenever you allocate new memory, you should always check to make sure there is available memory with, for example,

```
a = (int *)malloc(10*sizeof(int));
if(!a) printf("Out of memory!\n");
```

If you're feeling adventurous and you would like your program to crash in a clean and easy fashion, you can always dereference the `NULL` pointer with

```
*((int *)NULL)); // DON'T DO THIS!
```

Memory leaks and orphaned memory

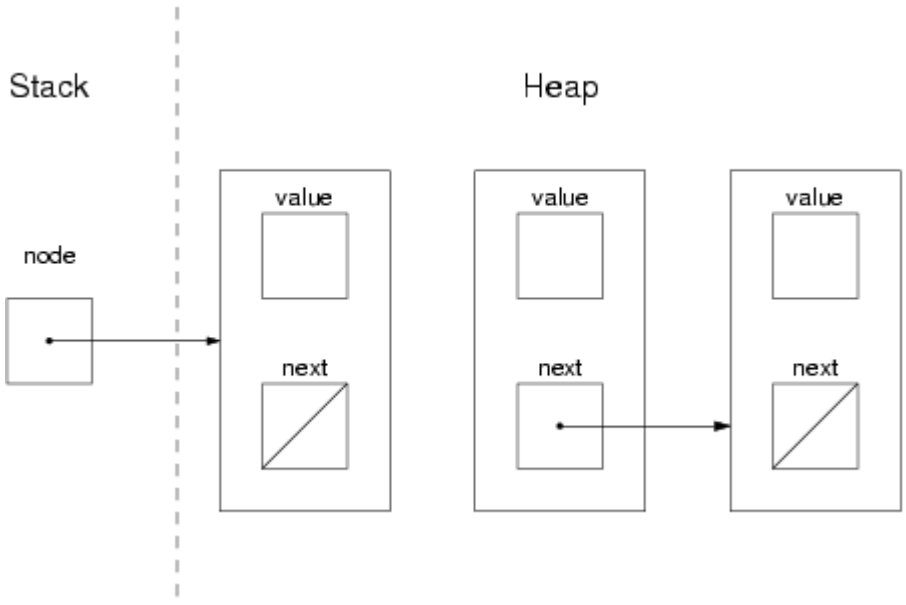
In the linked-list example you can see that the only way to access certain links in the list is from other pointers. For example, to access the value in the second link, you would use

```
x = node->next->value;
```

The dangerous thing about linked-lists is that it is possible to set the pointers so that you don't have access to certain links for the life of the program. For example, if you changed the `next` pointer of the first link with

```
node->next = NULL;
```

then the pointer diagram looks like



Since we have deleted the address of the second link in the pointer from the first link, there is no way to access the second or third links anymore! But at the same time, since we have not freed the memory associated with these links, then the space they are using cannot be

used to allocate more memory. This creates a memory leak and orphans these nodes. As a result, if you have a large program that continuously orphans memory like this, your program will eventually use up all available memory and will eventually crash. This is why it is so important to keep track of all of the memory you allocate and be sure you free it before you delete a pointer that accesses it. If we wanted to set the `next` pointer in the first link to `NULL` for some reason, we would first need to free up the space with the other links first! You need to free up the space associated with these one at a time, since they were all allocated separately. The correct way to do this would be with

```
free(node->next->next);
free(node->next);
node->next=NULL;
```

Note that if you attempted to free the links in any other order, you are not guaranteed to free all of the memory. For example, if you freed the links in this order

```
free(node->next);
free(node->next->next);
```

you would free up the space associated with the second link, but then the third link would be left unfreed.

Dangling pointers

Another problem that can arise with pointers is the concept of a dangling pointer. This results when you have a pointer pointing to a location in memory that is meaningless. For example, consider the function

```
void DanglingPointer(int *x) {
  int y=0;
  x = &y;
  return;
}
```

When the function is called, space is allocated for the variable `y` on the stack, and the pointer `x` contains the address of that space. But when the function returns control to the calling function, the space that was allocated for `y` is cleared and `x` points to a meaningless location in memory.