Lecture 9: Introduction to pointers

Pointers to variables

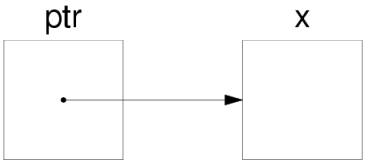
A pointer is nothing but a variable that contains an address of a location in memory. We can declare the variable `ptr` as a pointer to an integer as

```
int *ptr;
```

You can think of this as being a new type, which is `(int *)`, which means "pointer to an integer." When it is declared, a pointer does not have a value, which means that it does not yet store an address in memory, and hence it does not point to an address in memory. Let's say we want to have the pointer that we defined point to the address of another integer. This is done with

```
int x;
int *ptr;
ptr = &x;
```

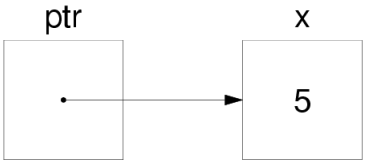
The unary operatory `&` returns the address of a particular variable. We can represent the above three lines of code graphically with



Since we did not assign the value of `x`, it does not contain a value. We can assign the value of `x` in one of two ways. The first way is the most obvious way, which is simply

```
x=5;
```

The pointer diagram now has a value at the location in which the variable is `x` is stored:



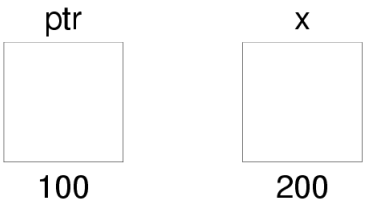
The other way we can assign the value of `x` is to use the pointer that points to its address. Since `ptr` points to its address, we can assign a value to the address that this pointer points to with

```
*ptr = 5;
```

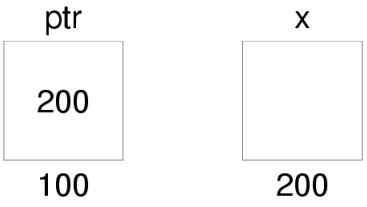
This is called dereferencing the pointer and it is identical to `x=5;`. The dereferencing operator says to set the contents of location pointed to by the pointer to the specified value.

Pointers themselves also take up memory, since they store a number that is an address to another location in memory. For the sake of argument, let's say that the pointer `ptr` is located at memory address `100` and that `x` is located at memory address `200`, as depicted the following diagram, which results when you declare

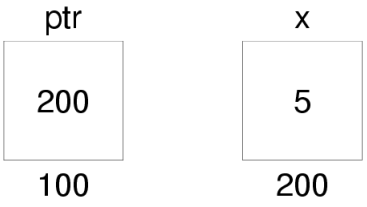
```
int x, *ptr;
```



When you set the pointer to point to the address of `x` with `ptr=&x`, you are assigning a value of `200` to the pointer, so that the pointer diagram now looks like



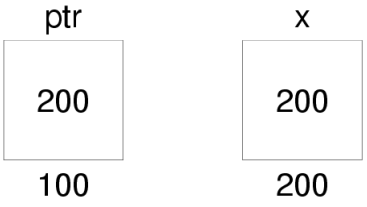
And when we set the value of `x` with either `x=5` or `*ptr=5`, the pointer diagram looks like



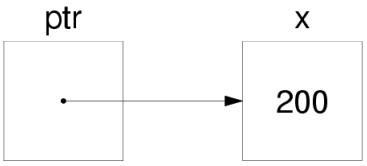
Since the pointer simply stores the address of another variable, there is nothing to stop us from setting the value of the integer `x` to the pointer `ptr` with

```
x=ptr;
```

In this case the pointer diagram would look like



But it is very rare that you would ever want to do this. Since pointers in general only store addresses, it is much easier to understand pointers if we represent them in memory as blocks with a dot at their center, with an arrow that points to the particular variable that exists at the address of the pointer, as we did before, as in

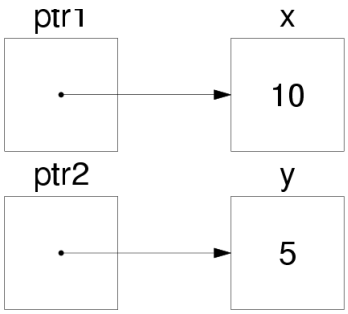


since `x` is not a pointer but it still contains the address of `ptr`, then its value is `200`.

Now consider the case when we declare more variables and pointers, such as

```
int x=10, y=5, *ptr1, *ptr2;
ptr1=&x;
ptr2=&y;
```

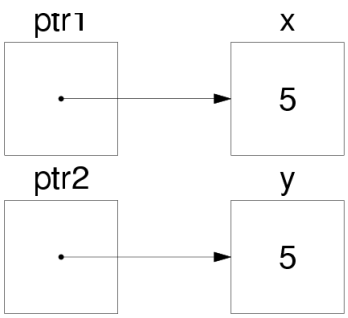
In this case the pointer diagram would look like



For this simple case, you can think of dereferencing a pointer as only changing the values at the addresses, while re-equating a pointer actually changes the pointers around. Consider the dereferencing operations

```
*ptr1=y;
*ptr2=x;
```

This first one changes the value of `x` to `y`, while the second one changes the value of `y` to the **new** value of `x`, which is the same as `y`, and hence does not change its value. The pointer diagram after this operation looks like



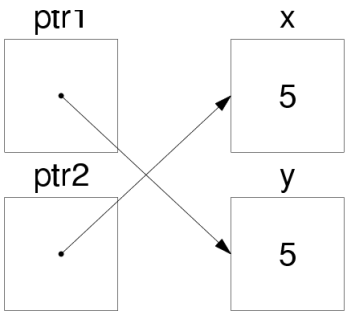
These operations are identical to

```
x=y;
y=x;
```

Now consider the case in which we swap the pointers with

```
ptr1=&y;
ptr2=&x;
```

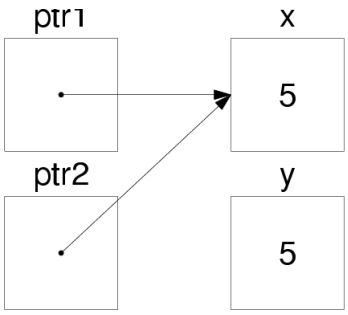
The pointer diagram now looks like



We can also set one pointer to the other,

```
ptr1=ptr2;
```

so now the pointers both point to the memory location of `x`, as in



Using pointers in functions

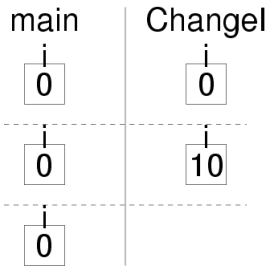
When you provide arguments to functions, the function stores the declared arguments as well as the variables within the function on the stack, and makes a **copy** of the function arguments to be used in the function. For example, consider the code

```
void ChangeI(int i);

main() {
  int i=0;
  ChangeI(i);
}
```

```
void ChangeI(int i) {
  i=10;
}
```

If we draw a pointer diagram for this function then we have



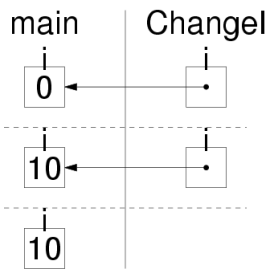
So we can see that since the function call merely makes a copy of its arguments, then the value of `i` is not changed in the calling `main` function. This is because the argument to the function is *passed by value*. We can get around this problem by passing the arguments to the function *by reference* using a pointer to `i`, as in

```
void ChangeI(int *i);

main() {
  int i=0;
  ChangeI(&i);
}

void ChangeI(int *i) {
  *i=10;
}
```

In this case we passed the address of the variable `i` to the function instead of its value. The call to the function still makes a copy of its arguments, but since that copy is the value of the address of `i`, then we can use that address to change the value in `main`. The pointer diagram for how this happens is given by



A classic example of why you need to pass variables to functions by reference rather than by value arises when you want to swap the values of two variables. Consider the following function that does not use pointers

```
void swap(int a, int b) {
  int tmp = b;
  b=a;
  a=tmp;
}
```

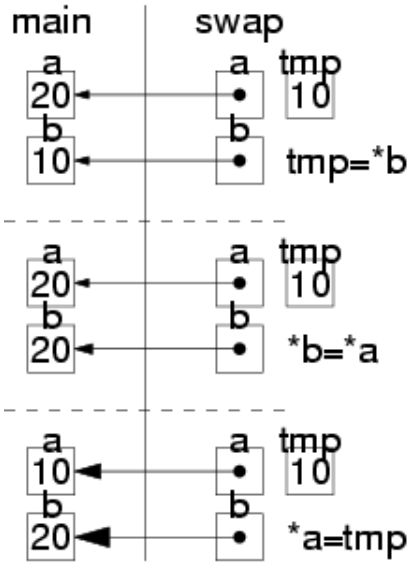
Clearly this does not work if we try and swap two values using `swap(a,b);` However, if we send the addresses of `x` and `y` with `swap(&a,&b);`, then it will work, as in

```
void swap(int *a, int *b) {
  int tmp = *b;
  *b=*a;
  *a=tmp;
}
```

You can see what is happening when you make a call to the `swap` function with

```
int a=20,b=10;
swap(&a,&b);
```

in the following pointer diagram:



Arrays and strings as pointers

On the stack

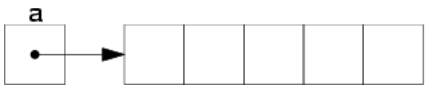
We have already seen how we can declare arrays and strings without pointers. For example, we can declare an array of 5 floats as

```
float a[5];
```

and we can declare a string with

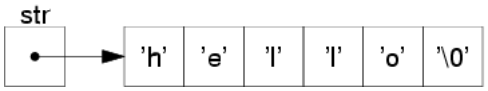
```
char str[] = "hello";
```

Both of these declarations are actually declaring pointers to a block of memory. When you declare an array of floats as above, you are declaring `a` to be a pointer that points to a block of memory on the local stack, and the pointer diagram looks like



You access different portions of that block of memory by dereferencing the pointer with, for example, `a[3]`.

The same goes for strings, except the compiler does a little extra on your behalf because it sees that `"hello"` is a string. So the pointer diagram for a string declaration looks like



but in addition to the five characters in `"hello"`, the additional null character `'\0'` is added on to ensure that this is a string.

Dynamically allocated in the heap

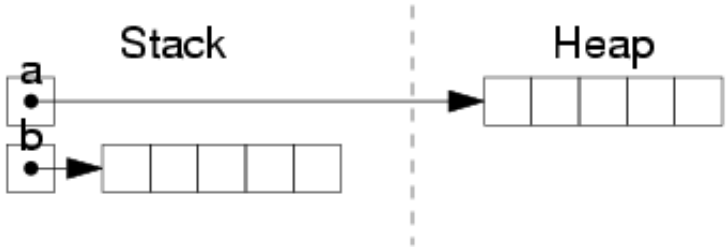
In the previous example, we declared arrays statically so that the memory that was allocated for them was allocated on the stack. This memory is allocated when the program begins and is allocated for the life of the program. It cannot be used for anything other than the array or string for which it was originally allocated. C provides us with the capability of allocating memory dynamically to store arrays and strings in the heap. This memory can be freed up for use by other arrays during the life of the program. To allocate space dynamically, we use the `malloc` function, which takes as its argument the amount of space we need to allocate. For example, if we want to allocate enough space in the heap for an array containing 5 floats, we would use

```
float *a = malloc(5*sizeof(float));
```

The `sizeof` command is conveniently used to tell us how many bytes we need for that particular type. In this case `sizeof(float)` returns `4` because a floating point number requires 4 bytes of memory. The pointer diagram for this allocation is similar to that for the static allocation in the previous section, except now the pointer `a` points to 5 4-byte blocks of memory in the heap, rather than the stack. For comparison, the declaration

```
float b[5];
```

is also shown in the pointer diagram.



The benefit of using arrays that are dynamically allocated in the heap is that we can free up the memory that was allocated for them with

```
free(a);
```

A note of **Warning:** you can only free space that is dynamically allocated in the heap. If you attempt to use `free` on a statically allocated variable, such as `free(b);`, your code will crash with a memory error that will read

```
Segmentation fault
```

Malloc and the void * pointer

If you use the `malloc` command as it is, it will be fine, but you most likely will get a compiler warning such as

```
warning: initialization makes pointer from integer without a cast
```

This is because the function prototype for the `malloc` function appears as

```
void *malloc(size_t size);
```

The `size_t` type-casts your integer input for you, but the return type of `malloc` is a `void *` pointer. C requires 4 bytes for every pointer you declare, regardless of its type, so it doesn't really matter what kind of pointer `malloc` returns. This is because no matter what kind of pointer you are using, it only needs 4 bytes to store a memory address. Therefore, `malloc` only needs to return the requisite 4-byte memory address. To fix the warning, you need to type-cast the return type of the function so that it is the same as the pointer you are equating it to. Since in this example we are creating a pointer to a floating point number, then we need to type-cast the return of the call to `malloc` as a `float *`, as in

```
float a = (float *)malloc(5*sizeof(float));
```

Pointer arithmetic

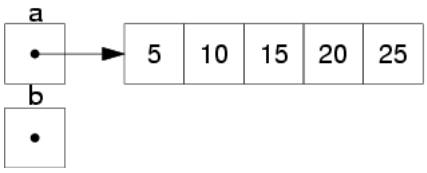
When you declare a pointer such as

```
int *x;
```

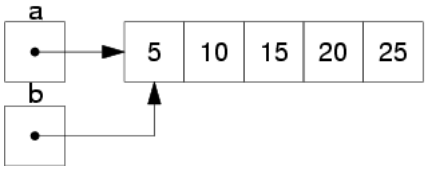
`x` is a pointer to a location in memory and, regardless of the type to which it points, we only need 4 bytes to store the pointer. But while every pointer only needs 4 bytes to store a memory address, the compiler also needs to know the type of variable that it points to in order to perform pointer arithmetic. As an example, consider the declaration

```
int x, *b, a[]={5,10,15,20,25};
```

In this declaration, `x` is an integer, and `a` and `b` are pointers to an integer (we could also have used `*a=(int *)malloc(5*sizeof(int))` and it would also have applied to this section). The difference between the two is that `a` points to the first element of a 5-integer block on the stack, while `b` doesn't point to anything yet. The pointer diagram for this declaration looks like



We can set `b` to point to the first element of the array with `b=a`, so now the pointer diagram looks like



To access the first element of the integer array `a`, we use

```
x=a[0];
```

This is identical to

```
x=*a;
```

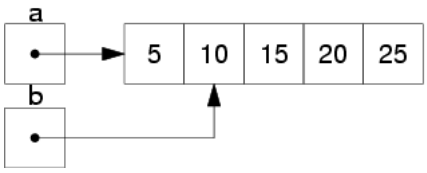
which dereferences the pointer `a` which simply points to the first element of the 5-integer block. To access the second element, we can use

```
x=a[1];
```

Using pointer arithmetic, we can let `b` point to the next element of the array by setting it with

```
b=a+1;
```

Now the pointer diagram looks like



This statement is why it is important for the compiler to know what types are stored at the addresses of pointers `a` and `b`. When it sees addition applied to pointers, such as the above, it knows that because the pointers store addresses of 4-byte integers, that incrementing a pointer by 1 means going to the next block of memory that is 4 bytes away. After doing this, we can dereference pointer `b` with

```
x=*b;
```

so that `x` is now `10`. This also works for any of the elements in the array, so that to access element `i`, we can use either

```
x=a[i];
```

or

```
x=*(a + i);
```

Integer arithmetic also allows us to increment or decrement pointers using the `++` or `--` operators. For example, to loop through the elements of `a`, we would use

```
b=a;
for(i=0;i<5;i++)
  printf(``%d ``,*b++);
```

The statement `*b++` says to first dereference the pointer `b` and get its value, and then to increment it, or to have it point to the next element in the array. Note that the operation `a++` **will not work**. This is because while the declaration `a[]` declares a pointer to an integer, it is a constant pointer and cannot be changed (The same goes for a declaration such as `a[5]`). If you attempt to do this then the compiler will give you an error like

```
wrong type argument to increment
```

Because it is very rare that you will actually initialize pointers like we initialized `a`, it is better to initialize non-constant pointers in the heap, and then setting the values afterwards, such as in

```
int i, *a = (int *)malloc(5*sizeof(int));
for(i=0;i<5;i++) *a++=5*(i+1);
```

The important thing to remember here is that the two statements

```
printf(``%d ``,*b++);
```

and

```
printf(``%d ``,(*b)++);
```

are **very** different. The first statement is identical to `*(b++)`, because the `++` operator takes precendence over the `*` operator. So in this case, we first dereference the pointer, and then add 1 to the pointer, so that it points to the next element. But in the second case, (*b)++, we only increment the value stored at the address of `b` and never increment the pointer. So the first statement will give us `5 10 15 20 25`, but the second statement will give us `5 6 7 8 9`.