Memory in a computer is restricted to two types, permanent and non-permanent memory. Hard drives are an example of permanent memory. To write data to the hard drive we use the file IO functions that we learned about in the last lecture. RAM is an example of non-permanent memory. When a program executes all, of the memory that it uses (aside from fileIO functions) is RAM.

RAM is laid out as a series of addresses (represented as hexadecimal numbers). Each address is one byte. Each byte is one bit. Each bit is a 1 or a 0.

A character is one byte (technically 7 bits, with a leading 0). For example:

'c' has ascii code 99 and is represented in binary as 01100011

Since a character is 7 bits with a leading 0 ascii codes go from 0 to 127 (00000000 is binary for 0 and 01111111 is binary for 127). If we display a byte that stores a character as a number we will see the ascii code of the character. If we try to display a byte whose leading bit is a 1 instead of a 0 then the number that is printed will be negative. (This is called 2's compliment notation). For example:

10000000 will be displayed as -128

Not what you expected right? This is because 2's compliment notation starts at -128 and counts up. So:

10000001 will be displayed as -127

This means that with 2s compliment we can display numbers from -128 through 127. (To get the negative of a number we flip the bits and add 1)

The different primitives in C take up a different number of bytes. (You can think of the number of addresses occupied as the number of bytes occupied)

chars are 1 byte, 8 bits
ints are 4 bytes, 32 bits
floats are 4 bytes, 32 bits
doubles are 8 bytes, 64 bits (exactly DOUBLE what is taken up by a float).

The size of a type may vary from system to system. To be absolutely certain how large a type is you can use the sizeof(type) operator. For example:

sizeof(int) == 4;

C has a few bits of syntax that allow us to work with addresses directly.

& is the reference operator. If we apply it to a variable it returns the address of the variable. For example:

```
int i=5;
&i; //this is the address of the variable i.
```

* is the indirection operator. If we apply it to an address it returns the value located at that address. For example:

```
int i=5;
*&i==5; //first we get the address of the variable, then we get the value at that address.
*&i==i; //of course this is effectively the same as just calling the variable.
```

* has another use. If we put it after a type it allows us to declare a variable that holds an address. This sort of variable is called a pointer. For example:

```
int i=5;
int * iaddr=&i;
```

In C the relationship between memory and arrays is very close. In fact, arrays are just a short hand for working with memory directly.

To declare an array in C we use:

```
int myArray[5]; // this declares an array named myArray that holds 5 ints.
```

In C we do not put [] directly after int, like we would in Java (int[] myArray= new int[5];) This is because in C int[] is not a type. In fact, when we declare an array what we are really declaring is a constant pointer to the beginning of the array. For example:

```
int myArray[5]==const int (* myArray)[5];
```

Let me explain the right side of the ==. const tells C that the variable cannot be assigned a value, this is much like a final variable in Java. int (* myArray)[5] is the C syntax for declaring a pointer named myArray that points to the beginning of 5 ints in memory.

Because of this we can treat any array as though it is actually a pointer. In fact, the [x] syntax that we are familiar with in Java is actually C shorthand for pointer math. Allow me to explain with an example:

```
int myArray[3]={1,2,3}; //we can initialize arrays just like in Java
myArray[0]==*(((int *)myArray)+0);
myArray[1]== *(((int *)myArray)+1);
myArray[2]== *(((int *)myArray)+2);
```

The left hand side is the array syntax we know from Java (which is perfectly valid in C). The right hand side is called pointer math. Let me break the syntax down in steps.

((int *)myArray) // This casts myArray from a const int (*) [5] to an int *. (so it is the address of one int)
((int *)myArray)+1 // This tells C to look one address past the beginning of myArray.
*(((int *)myArray)+1) // This tells C to get the value at this address.

Why not just do *(myArray+1) ? We can't. C will multiply the 1 by how many addresses myArray takes up. In this case, it will multiply 1 by 5 (since myArray is a const int (*) [5]), causing us to jump all the way to the end of the array. Actually, since we are talking about int * s C will actually multiply 1 by 4 and *then* by 5. This is because ints take up 4 bytes, and we are moving by 5 ints (which take up 20 addresses).

All of this can be very hard to keep in our heads. Fortunately we can print out addresses using printf and the %p format specifier. For example:

```
int i;
printf("%p",&i);
```

C also provides support for matrices (AKA two dimensional arrays). For example, we can represent the matrix:

	Column 0	Column 1
Row 0	A	B
Row 1	C	D

In C as follows:

```
char myMatrix[2][2];
myMatrix[0][0]='A';
myMatrix[0][1]='B';
myMatrix[1][0]='C';
myMatrix[1][1]='D';
```

We can shorten this considerably:

```
char myMatrix[2][2]={{'A','B'},{'C','D'}};
```

Or even:

```
char myMatrix[2][2]={'A','B','C','D'};
```

C also has support for strings. In C string is not a type, instead a string consists of two things, a char * that points to the first letter in the string, and a '\0' character that tells C where the String ends. '\0' is the NULL character (it has ascii code 0).

There are a few different ways to produce strings. The easiest is to use a string literal. For example:

char * myString = “hi”;

This syntax will tell C to put ‘h’ at *myString, ‘i’ at *(myString+1) and ‘\0’ at *(myString+2).

Of course we can do this manually using an array.

char myString[3]={‘h’,’i’,’\0’};

The value of having a string is that we can use printf and the %s format specifier to print it. For example:

```
char * myString = “hi”;
printf(“%s”,myString);
```

Keep in mind, this only works because printf stops printing characters when it reaches the ‘\0’ that C puts at the end of the string (thanks to the string literal we used). If there were no ‘\0’ character printf would just keep printing characters until it encountered a Segmentation Fault. (I will explain segmentation faults in the next lecture)

At the risk of beating a dead horse indulge me in another more complicated example. What if we want an array of strings? Here is how we would do this:

char * myArrayOfStrings[2]={“hi”,”ho”};

This may not seem that complicated, but it can be confusing to work with. For example, how would we print out just the second letter of the second word?

```
char * myArrayOfStrings[2]={“hi”,”ho”};
printf(“%c”,*(myArrayOfStrings[1]+1));
```

Let me break that last bit down in steps.

```
myArrayOfStrings //This is a char (*) [3] to the beginning of “hi”
myArrayOfStrings[1] // This is a char *to the beginning of “ho”
myArrayOfStrings[1]+1 // This is a char * to the second letter of “ho”
*(myArrayOfStrings[1]+1) // This is the second letter of “ho”
```

Since we’re looking to access individual letters we would actually be better served by a matrix of characters.

```
char myMatrixOfCharacters[2][3]={“hi”,”ho” };
printf(“%c”,myMatrixOfCharacters[1][1]);
```

Broken down into steps that is

myMatrixOfCharacters // This is a char (*)[3]
myMatrixOfCharacters[1]// This is a char * to the beginning of “ho”
myMatrixOfCharacters[1][1]// This is the second character in “ho”

References

Operators

* The Dereference operator
When used in a declaration it denotes that the type is an address, not a value.

Examples

```
int * anAddr;
char * argv[];
```

When used with a variable it denotes that the variable holds an address, and that the program should work with the value stored at that address, not the address itself.

Examples

```
*anAddr=5;
printf("%p:%i\n",anAddr,*anAddr);
```

& The Reference operator
Used to denote that the program should work with the address of a variable, not the value of the value. Often used when passing by reference.

Examples

```
int * anAddr = &i;
scanf("%i",&i);
```