Intro to pointers and memory

Main memory is just a linear array of bits. Bytes are groups of 8 bits and are the minimal addressable units in most current architectures.

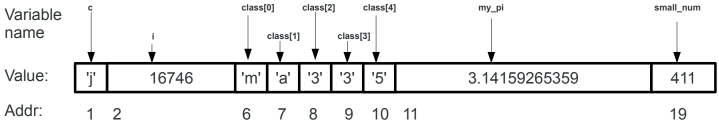
Value:	1	0	0	0	1	0	1	1	0	1	1	1	0	1	0	1	0	0	0	1
	0	0	1	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	0	0
	1	0	1	0	1	1	0	1	0	0	1	1	0	1	0	1	1	1	0	1
	0	0	0	1	0	0	1	1	0	1	1	1	1	1	0	0	0	0	0	1
	0	0	1	1	1	1	1	0	1	1	0	0	0	1	0	1	1	1	0	0
	1	0	1	1	0	1	0	1	0	0	1	1	1	1	0	0	1	1	1	0
	0	0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	0	0	1	1
	1	0	0	1	0	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0
Addr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

- Different variable types take different amounts of memory to store.
- A `char` is 1 byte
- A `short` is 2 bytes
- A `int` is 4 bytes
- A `float` is 4 bytes
- A `double` is 8 bytes

Imagine the code

```
int main(){
    int i;
    char c;
    double my_pi;
    char class[5];
    short small_num;
      .
      .
      .
    do some stuff with these
    variables
}
```

Memory might well look like this



The & returns the **address** of the variable that follows it. In the previous example, what are the following?

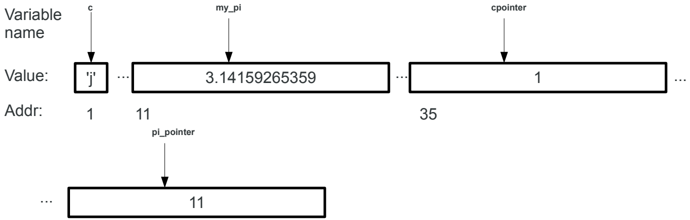
```
&i
&c
&my_pi
&small_num
&(class[0])
&(class[1])
```

- An address to a spot in memory is called a **pointer**.
- An address is a number just like any other (takes 32 or 64 bits to store, depending), if we wish to store the address of a given variable we use a pointer.
- Pointer variables are denoted with a `*`.

Consider the code

```
int main(){
    int i;
    char c;
    double my_pi;
    char class[5];
    short small_num;
    char* cpointer=&c;
    double* pi_pointer=&my_pi;
      .
      .
      .
    do some stuff with these
    variables
}
```

Memory might look like this



- Because different variables have different storage patterns and require different amounts of memory, knowing what type of variable a pointer points to is important.
- We can ask for the value of the variable stored in a given address using the * operator.

Consider the following code:

```
int main(){
    int i;
    i=100;
    int* i_pointer=&i;
    int j;
    j=*i_pointer;
    printf("%d\n",j);
}
```

Consider the following code:

```
int main(){
    int i;
    i=100;
    int* i_pointer=&i; //(i_pointer will have some crazy
                       // value,say 11234568)
    int j;
    j=*i_pointer; //(j will have the value of the integer
                  // stored at position 11234568, which
                  // happens to be 100.)
    printf("%d\n",j);
}
```

We can change the value stored at a given position also using the $*$ operator.

Consider the following code:

```
int main(){
    int i;
    i=100;
    int* i_pointer=&i;
    *i_pointer=5901;
    printf("%d\n",i);
}
```

Consider the following code:

```
int main(){
    int i;
    i=100;
    int* i_pointer=&i; //(i_pointer will have some crazy
                       // value,say 11234568)
    *i_pointer=5901;   // change the value of the data
                       // stored at 11234568 to be 5901.

    //since i is stored at location 11234568
    //the value of i will be changed.
    printf("%d\n",i);
}
```

We can use pointers to allow functions to change argument values.

```
void mod2mod3(int j,int* jmod2_p,int* jmod3_p){
    *jmod2_p=j % 2;//put the value of j mod 2 in
                   //to the memory address in jmod2p
    *jmod3_p=j % 3;//put the value of j mod 3 in
                   //to the memory address in jmod3_p
}
```

How should we call this function?

```
void mod2mod3(int j,int* jmod2,int* jmod3){
    *jmod2=j % 2;
    *jmod3=j % 3;
}

int main(){
   int k=100;
   int kmod2; //memory has been allocated
   int kmod3; //for kmod2 and kmod3, but
              //there has been no value put
              //int that memory, it is
              //garbage.
   mod2mod3(k,&kmod2,&kmod3); //give the function the
   //address of kmod2 and kmod3.The function will put the
   //correct answer in those addresses.

   printf("%d \% 2= %d\n",k,kmod2);
   printf("%d \% 3= %d\n",k,kmod3);
}
```

Pointer Arithmetic

- Pointer arithmetic is pretty smart. If you have a `int* foo`, then `foo+1` is the value in `foo` incremented by the size of an `int`.
- Suppose that we have `int nums[3]` and that `&(nums[0])` is 1000.
- `nums[0]=20`, `nums[1]=2`, `nums[2]=-4`
- Suppose an in gets stored in 4 bytes.

expression	value
`nums`	
`&(nums[0])`	
`nums+1`	
`nums+2`	
`*nums`	
`*(nums+1)`	
`*(nums+2)`	

- Pointer arithmetic is pretty smart. If you have a `int* foo`, then `foo+1` is the value in `foo` incremented by the size of an `int`.
- Suppose that we have `int nums[3]` and that `&(nums[0])` is 1000.
- `nums[0]`=20, `nums[1]`=2, `nums[2]`=-4
- Suppose an in gets stored in 4 bytes.

expression	value
`nums`	1000
`&(nums[0])`	1000
`nums+1`	1004
`nums+2`	1008
`*nums`	20
`*(nums+1)`	2
`*(nums+2)`	-4

- It is true that nums[i] is just a shorthand for *(nums+i).

```
int main(){

    int nums[3];
    *(nums+0)=11;
    *(nums+1)=6;
    *(nums+2)=0;
    printf("%d,%d,%d\n",nums[0],nums[1],nums[2]);

}
```

Command line arguments

- `main` takes two arguments, an `int` and a `char**`
- You will see this written as `int main(int argc,char** argv)`.
- Note that `argv[0]`, `argv[1]` etc are all of type `char*`. They are the locations in memory where the command line option strings are stored.

```
int main(int argc,char** argv){

  cout<<"You supplied "<<argc<<" options."<<endl;
  int i;
  for (i=0;i<argc;i++){
    printf("argument %d:%s\n"i,argv[i]);
  }
}
```

Convert string to int

- There are multiple ways to convert a string that happens to represent an integer (or float or whatever) to an integer.
- My favorite is `atoi` for which you will need the `stdlib.h` library.

```
#include<stdio.h>
#include<stdlib.h>

int main(int argc,char** argv){

  printf("One plus the input is %d\n",
 atoi(argv[1])+1);
}
```

- Similar to `atoi` is `atof`
- To go the reverse direction try `itoa`
- The `NULL` pointer is a special value (basically 0). It is used in initialization, error states, etc. You can do `int* unused=NULL`