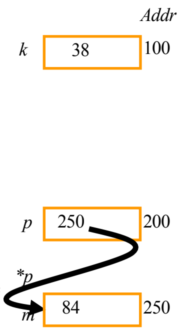
CS1100
Introduction to Programming

Introduction to Pointers

What is a Pointer?

- *Recap*: a variable ***int*** *k*
  - Names a memory location that can hold one value at a time
  - Memory is allocated statically at compile time
  - One name ποιντσ το one location
- A pointer variable ***int*** **p*
  - Contains the address of a memory location that contains the actual value
  - Memory can be allocated at runtime
  - One name ποιντσ το many locations



l-value and *r*-value

- Given a variable *k*
  - Its *l*-value refers to the address of the memory location
  - *l*-value is used on the left side of an assignment
    - Ex. *k* = expression
  - Its *r*-value refers to the value stored in the memory location
  - *r*-value is used in the right hand side of an assignment
    - Ex. *var* = *k* + …

Pointer Variables

- Pointer variables are variables that store the address of a memory location
- Memory required by a pointer variable depends upon the size of the memory in the machine
  - one byte could address a memory of 256 locations
  - two bytes can address a memory of 64K locations
  - four bytes can address a memory of 4G locations
  - modern machines have RAM of 1GB or more…
- The task of allocating this memory is best left to the system

Declaring Pointers

- Pointer variable – precede its name with an asterisk
- Pointer type - the type of data stored at the address
  - For example, ***int** *p;*
  - *p* is the name of the variable. The '*' informs the compiler that *p* is a pointer variable
  - The ***int*** says that *p* is used to point to an integer value

Ted Jenson's tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

Random Q

```
int q = 40;

int* p = &q;

q = 45;

printf("%d\n", *p);
```

Random Q2

```
int q = 40; // q's address is 1008

int* p = &q; // p's address is 1028
int *s = NULL;
int *r = &p; // r's address is 1048
q = 45;
// r 1028; *r 1008; **r 45
printf("%d\n", *p);
```

Contents of Pointer Variables

- In ANSI C, if a pointer is declared outside any function, it is initialized to a *null* pointer
  - For example,

```
int k;
int *p, *q = NULL;
p = &k;               //assigns the address of int k to p
if (q == NULL)        //tests for a null pointer
    q = malloc(sizeof(int));  //dynamic allocation,
                              //creates an anonymous
int                            // in memory at runtime
```

Dereferencing Operator

- The asterisk symbol is the "dereferencing operator" and it is used as follows

  *ptr = 7;

  - Will copy 7 to the memory location whose address is pointed to by *ptr*
  - Thus, since *p* "points to" (contains the address of) *k*, the above statement will set the value of *k* to 7
- Using '*' is a way of referring to the value in the location which *ptr* is pointing to, but not the value of the pointer itself
  - ***printf***("%d\n",**ptr*); --- prints the number 7

Random Q.

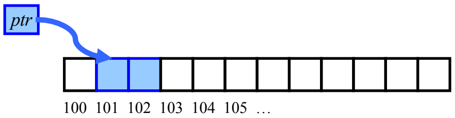
```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int num;
  double *darray = NULL;  /* initialize */
  scanf("%d", &num);
  // Creating a dynamic sized array of length num;
  // Each element in the array is of type double.
  darray = malloc(num * __sizeof (double)____________);
}
```

malloc and free

- malloc() system call allocates memory on demand
  - Dynamic memory allocation
  - Needed when we do not know the memory requirements at the time of program compilation
  - More efficient way to utilize memory space
  - Allocates space in program **Heap** memory
- free() system call releases memory that is not needed anymore
  - Eliminates memory leaks in program

short int Pointer

- ***short*** **ptr*;
  - says that *ptr* is the address of a short integer type
- ***short*** – allocates **two** bytes of memory



  - **ptr* = 20; //store the value 20 in the above **two** bytes
- if we had said "***int*** **ptr*"
  - it would have allocated **four** bytes of memory

Memory Needed for a Pointer

- A pointer requires two chunks of memory to be allocated:
  - Memory to hold the pointer (address)
    - Allocated statically by the pointer declaration
  - Memory to hold the value pointed to
    - Allocated statically by a variable declaration
    - OR allocated dynamically by ***malloc***( )
- One variable or pointer declaration → allocation of one chunk of memory

Accessing Arrays with Pointers

```
#include <stdio.h>
int myArray[ ] = {1,24,17,4,-5,100};
int *ptr;
int main(void){
 int i;
 ptr = &myArray[0]; // myArray, &myArray are also same.
 printf("\n");
 for (i = 0; i < 6; i++){
        printf("myArray[%d] = %d ", i, myArray[i]);
        printf("value at ptr + %d is %d\n", i, ptr[i]);
 }
 return 0;
}
```

Arrays

The name of the array is the address of the first element in the array

Given

int myArray[10];

In C, we can replace

int *ptr = &myArray[0];

with

ptr = myArray;

to achieve the same result

Arrays Names Are Not Pointers

While we can write

ptr = myArray;

we cannot write

myArray = ptr;

The reason:

While ptr is a variable, myArray is a constant

That is, the location at which the first element of myArray will be stored cannot be changed once myArray has been declared

Pointer Types

C provides for a pointer of type void. We can declare such a pointer by writing:

```
void *vptr;
```

A void pointer is a generic pointer

For example, a pointer to any type can be compared to a void pointer

Type casts can be used to convert from one type of pointer to another under proper circumstances

Trying Out Pointers

```
#include <stdio.h>
int j = 1, k = 2; int *ptr;
main( ) {
ptr = &k;
printf(“\n j has the value %d and is stored at %p”,j,(void*)&j);
printf(“\n k has the value %d and is stored at %p”,k,(void*)
                                        &k);
printf(“\n ptr has the value %p stored at %p”, ptr, (void *)
                                        &ptr); printf(“\nThe value of the
integer pointed to by ptr is %d\n”,
*ptr);
}
```

Generic address of j

Dereferencing – will print r-value of k

Random Q

```
#include <stdio.h>
#include <stdlib.h>
int main()  {
 int *p1 = NULL, *p2 = NULL;
 p1 = (int *) calloc(1, sizeof(int)); //initi. To 0
 printf(“Value stored in p1 is %d\n”, *p1);
 *p1 = 27;
 p2 = p1;
 printf(“Value stored in p2 is %d\n”, *p2);
}
```