ECE 131 – Programming Fundamentals

Dr. Daryl O. Lee

University of New Mexico

Pointers

- The "P" word strikes fear into the hearts of many beginning programming students. Many hopes, dreams and aspirations have been dashed on the rocky shores of C pointers.
- Nevertheless, you can't call yourself a C programmer unless you understand the concept of pointers — they're fundamental to understanding how arrays and more advanced data structures are created and used in C.
- If you have the right programming model in mind, and work to comprehend a few simple concepts, then you too can learn to declare, manipulate, and in general work magic with pointers.
- So let's all take a deep breath ..... and prepare to enter the world of .....

pointers!

THE UNIVERSITY of NEW MEXICO

Pointers

- Pointers implement the concept of indirect addressing that you will see again in your microprocessors course.
- Consider the following C code:

```
int x = 200; // declare an int variable
int* y; // declare a pointer-to-int variable
y = &x; // assign the address of x to y
```

- Variable `y` has type pointer-to-int. I.e., `y` is a variable that can store addresses, and it is assumed that if we go to the memory location whose address is stored in `y` we will find an integer.
- The assignment:

```
y = &x
```

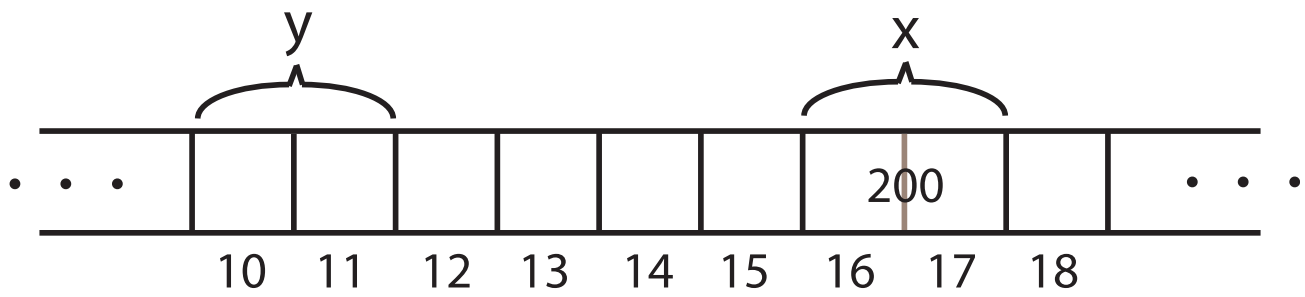
uses the address-of operator (`&`) to obtain the address where variable `x` is stored. This address is then assigned to the variable `y`. Thus, after this assignment, `y` "points to" `x`.

THE UNIVERSITY of NEW MEXICO

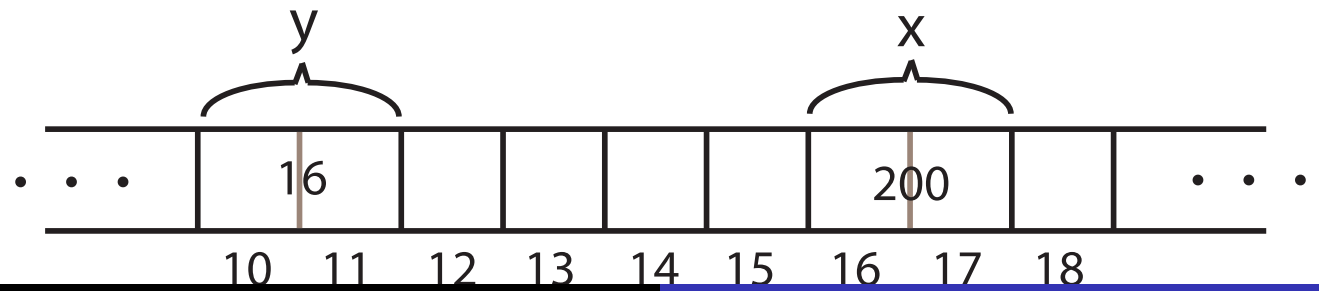
Pointers

Pictorially, here's what's happening when the compiler executes the previous C statements:

- First, the compiler allocates space to store the variables `x` and `y`.



  In this example, we're assuming that the compiler stores these variables in the memory locations whose addresses are 10 and 16, respectively. Notice also that we're assuming both `int` and pointer-to-`int` data types are stored in two bytes.
- Next, the assignment causes the address of `x`, i.e., 16, to be assigned to the variable `y`.



Pointers

- In the previous example, the pointer was declared with the `*` attached to the type, i.e., `int* y`; however, the syntax in C also allows for `int *y`, where the `*` is attached to the identifier. So which form should you use?
- Either. The first form emphasizes the pointer's type, while the second form allows multiple declarations on one line. E.g., the previous example could be rewritten as:

```
int x = 200, *y;
y = &x; // assign the address of x to y
```

- Be careful! What are the types of `y` and `z` in the following declaration statement?

```
int* y, z;
```

Variable `y` has type pointer-to-`int`, while `z` has type `int`.

THE UNIVERSITY of NEW MEXICO

Special Pointers

- C has a special "generic pointer" whose type is `void*`. It's called a void pointer.
- Any pointer can be cast to `void*`, and any pointer may be converted to `void*`. I.e., pointers may be assigned to and from pointers of type `void*`, and they may be compared with them. We'll see void pointers when we go over dynamic memory allocation.
- The null pointer is used to indicate that a pointer does not point to any valid memory location.
- E.g., if a function fails to allocate memory, this is typically indicated by returning a null pointer.
- C defines the constant NULL to represent the null pointer, and normally a compiler will use the value 0 for NULL.

THE UNIVERSITY of NEW MEXICO

Pointer Dereferencing (The * Operator)

- A pointer is only useful if you have some way of getting at the thing it points to.
- In order to access the memory location a pointer points to, the pointer must be dereferenced. Pointer dereferencing is accomplished using the unary * operator.
- Notice in the C Operator Precedence chart , pg. 440 in Kochan, that the unary * operator associates right-to-left, and that it has higher precedence than the binary * operator (multiplication).
  Ex:

```
int x = 200, *y;
y = &x;
printf("x = %d", x); // this prints:  x = 200
printf("x = %d", *y); // so does this
```

THE UNIVERSITY of NEW MEXICO

Pointer Dereferencing (The * Operator)

You can think of the unary * operator as "undoing" the & operator.

Ex:

```
int x = 200;
printf("x = %d", x); // this prints:  x = 200
printf("x = %d", *&x); // so does this
```

- Both the unary * and & operators associate right-to-left.
- So *&x is the same as *(&x).
- Thus, the address-of x will first be determined as a result of applying the & operator, and then this address will be dereferenced by the * operator.

THE UNIVERSITY of NEW MEXICO

Pointers and Arrays

- In C, there is a very strong relationship between pointers and arrays.
- Important: In C, any pointer can be indexed like an array, and any array name can be manipulated as a pointer.
- The name of an array (by itself, with no index) is a synonym for the address of the first element in the array.

  Ex. Consider the declaration: `int x[3];`
  Then the following two expressions yield the same result:

  `x` and `&x[0]`

  Namely, they both evaluate to the address of the first element in array `x`.

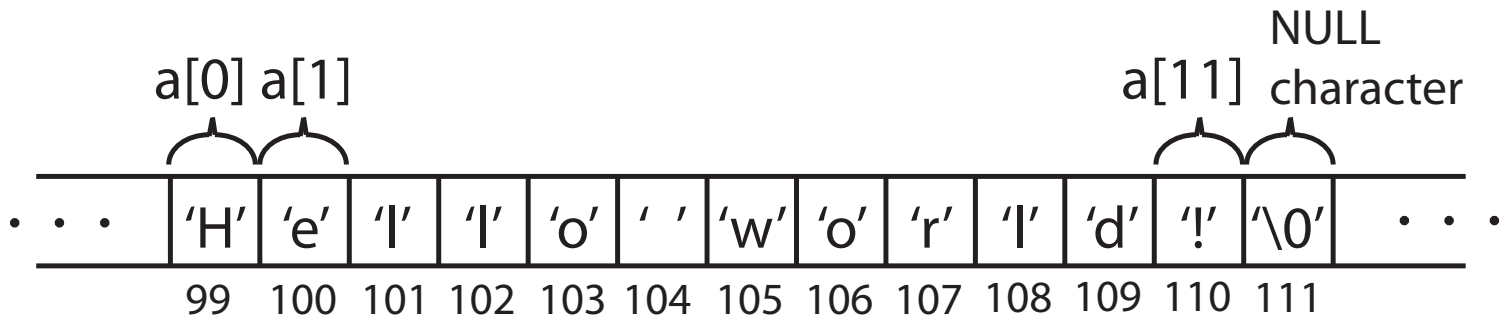
THE UNIVERSITY of NEW MEXICO

Pointers and Arrays

Let's consider the declaration:

```
char* a = "Hello World!"
```

- The identifier named `a` is a variable of type pointer-to-char. This variable can store addresses that are assumed to point to a `char` variable or constant.
- We've already seen how C compilers handle string constants: Ex:



Pointers and Structs

Let's consider the declarations:

```
struct time {
    int hh;
    int mm;
    int ss;
} t1 = {12, 30, 00} , *pT;

pT = &t1;
printf("t1 = %d:%d:%d", t1.hh, t1.mm, t1.ss);
printf("t1 = %d:%d:%d", (*pT).hh, (*pT).mm, (*pT).ss);
printf("t1 = %d:%d:%d", pT->hh, pT->mm, pT->ss);
```

THE UNIVERSITY of NEW MEXICO

Pointers and Structs

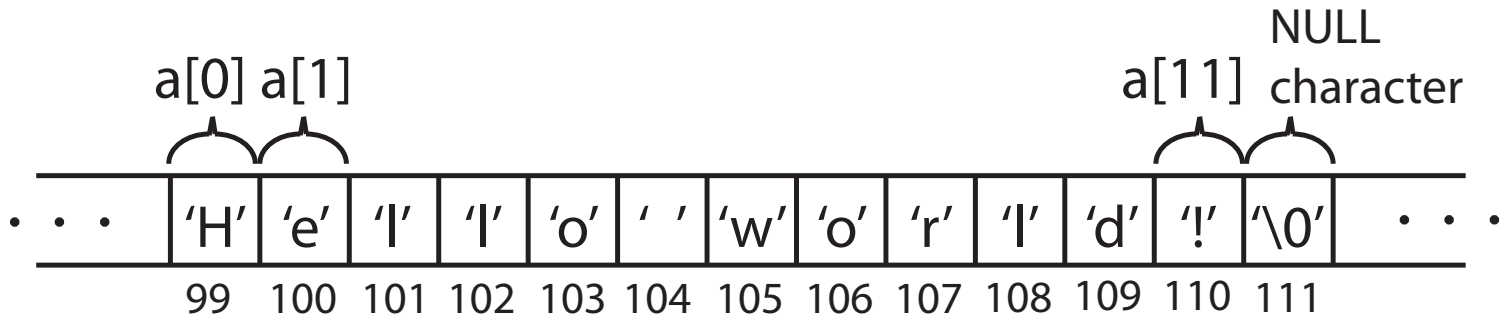
- In the second printf, the parens are needed because the . operator has higher precedence than the * operator. *t1.hh would cause the compiler to try to interpret t1.hh as pointer.
- In the third printf, the -> operator is "syntactic sugar", making the pointers easier to read and to use.

Pointer Arithmetic

- In the previous example, after the assignment statement, the pointer variable a will store 99.
- Furthermore, we can apply certain arithmetic operations to pointer variables, namely +, −, ++, −−, +=, and −=.
- Pointer addition and subtraction can involve:
  - a pointer and an integer, or
  - two pointers, but only if they point to the same data type.
- All other pointer arithmetic is illegal, e.g., you can't multiply two pointers.
- Two pointer variables can also be compared using the ==, !=, <, <=, etc. operators.

Pointer Arithmetic

Ex:



expression	evaluates to
a	99
a+1	100
a−1	98
a+10	109
a++	100 (a is changed)
a+=10	110 (a is changed again)

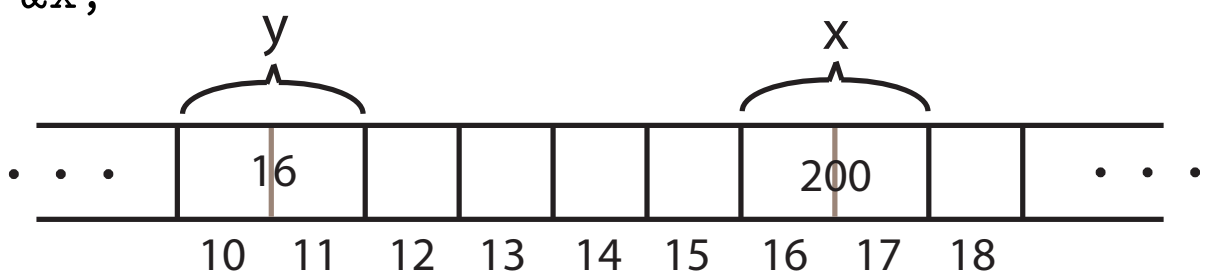
THE UNIVERSITY of NEW MEXICO

Pointer Arithmetic

Important: When performing pointer arithmetic, the C compiler must take into account the type of the pointer.

Ex. Consider again the example:

```
int x = 200, *y;
y = &x;
```



expression	evaluates to
y	16
y+1	18 (not 17!)
y+2	20
x	200
y[0]	200
*y	200
&y[0]	16
&y	10

THE UNIVERSITY of NEW MEXICO

Example of Pointer Usage

The following function will compute, and return, the length of a string supplied as input:

```
// return the length of string s
int strlen(char* s) {
   int n = 0;
   while (*s++)
      n++;
   return n;
}
```

This function does the same thing:

```
// return the length of string s
int strlen(char* s) {
   char* p=s;
   while(*p != '\0')
      p++;
   return p-s;
}
```

THE UNIVERSITY of NEW MEXICO

Memory Allocation

- We have seen that when you declare variables and constants, the compiler (at compile-time) will allocate storage in memory for those variables and constants. When the variable or constant is an array, the compiler allocates enough storage for the entire array.
- Sometimes, however, we don't know how many elements will be in an array until the program is running. E.g., User input may determine how many array elements are needed.
- In these cases, we need to allocate storage for the array at run-time rather than at compile-time.
- The allocation of memory at run-time is referred to as dynamic memory allocation.

Dynamic Memory Allocation

- Different programming languages handle dynamic memory allocation differently.
- Some programming languages take an implicit approach to dynamic memory management. E.g., Java. In these languages, memory management functions are provided by the language's run-time environment.
  - You simply declare variables and arrays as you need them, and the size of the array can be a variable.
  - The system tries to figure out when memory is no longer being used, so that it can reclaim it. In programming parlance, this is called garbage collection.
- Other programming languages take an explicit approach to dynamic memory management, leaving the management of dynamically allocated memory up to the programmer.
- Which approach do you think C takes?

Pointers and Dynamic Memory Allocation

- C takes the explicit approach to dynamic memory management, you (the programmer) are responsible for setting aside the right amount of storage during dynamic memory allocation, and you must reclaim that memory when it is no longer needed.
- In C, there are two functions that can be used to dynamically allocate memory:
  - `void* malloc(size_t size)` — Allocates `size` bytes of storage, and returns a pointer to the allocated memory. If the storage cannot be allocated, a null pointer is returned.
  - `void* calloc(size_t n, size_t size)` — Allocates `n` elements, each containing `size` bytes of storage, initializes each byte to the value 0, and returns a pointer to the allocated memory. If the storage cannot be allocated, a null pointer is returned.

  If either of these functions fails to allocate the requested memory, a null pointer is returned.

THE UNIVERSITY of NEW MEXICO

Reclaiming Memory

- To reclaim dynamically allocated storage in C, use:
  - `void free(void* ptr)` — `ptr` is a pointer that was previously returned by `malloc()` or `calloc()`.
- So the memory allocated by malloc() or calloc() is special in that it contains information about how many bytes were allocated. The free() function uses this information to reclaim the memory.
- What happens if dynamically allocated memory is no longer needed, but we forget to reclaim it? The memory remains allocated as long as the program that allocated it is still running — this is referred to as a memory leak.
- If a program runs long enough with a persistent memory leak, it can eventually use up all available memory, and the program will "crash".

Dynamic Memory Allocation – Example

```
#include <stdio.h>
#include <stdlib.h> // needed to use malloc()
main() {
   int* ary; // will point to a dynamically allocated array
   int i, num;
   printf("How many integers would you like to enter?  -> ");
   scanf("%d", &num); getchar();
   ary = malloc(sizeof(int) * num); // allocate memory
   if (ary) { // Check pointer for validity
      // do something with pointer
   }
   free(ary); // reclaim the memory; free(NULL) is okay
}
```

THE UNIVERSITY of NEW MEXICO