CSE 1320 - Intermediate Programming

Pointers

Alex Dillhoff

University of Texas at Arlington

Addressing

A program keeps track of memory using an addressing system.

Some systems address each byte and are called **byte-addressable** computers.

Others are **word-addressable**.

Addressing

When a program is executed and loads into memory, the loader determines where in memory the values of each variable are stored.

These memory locations in which the values are stored are called **addresses**.

In C, we can determine the address of any variable using the `&` operator.

Pointers

Pointers in C are addresses to some location in memory that contains an object or function.

This allows data to be accessed from anywhere, as long as you have a pointer to it.

Pointers

The `&` operator, when applied to a variable, produces a **pointer** value.

A **pointer** variable stores the address of a memory location.

Note: This address is considered a value.

Pointers

In C, every data type has a corresponding **pointer-to** type.

The pointer type is derived from the **referenced type** – the object or function type.

- `pointer-to-int`
- `pointer-to-char`
- `pointer-to-double`
- etc.

This implies that we can have **pointers to pointers**.

Pointers

To declare a pointer, add an asterisk before the **identifier**.

```
int *intptr;
char *charptr;
```

Pointers

Example: Create a pointer-to-int and assign it the address of an existing integer.

Pointers

Consider the following code.

```
int a = 10;
int a_ptr = &a;
```

Pointers

Both variables have an **address** AND a **value**.

Type	Name	Address	Value
`int`	`a`	`0xFF0`	`10`
`int *`	`a_ptr`	`0xFF4`	`0xFF0`

The **type** indicates what kind of value is stored at that address.

Pointer Arithmetic

Pointer arithmetic permits addition between a pointer and an integer.

```
int *ptr = 0xFF0;
// increases the address by 3 `int`s
ptr += 3;
```

What will be the result?

Pointer Arithmetic

```
int *ptr = 0xFF0;
// increases the address by 3 `int`s
ptr += 3;
```

What will be the result?

If an `int` takes up 4 bytes, then `FF0 + 3 * sizeof(int) = FFC`.

Pointer Arithmetic

It also allows subtraction between two pointers or a pointer and an integer.

```
int *ptr1, *ptr2;
// size between the pointers
int diff = ptr1 - ptr2;
```

Pointer Arithmetic

Example: `pointer_arithmetic.c`

Dereferencing Pointers

The address is useful for knowing where the value is stored, **but how do we get the value stored at a particular address?**

Dereferencing Pointers

The address is useful for knowing where the value is stored, **but how do we get the value stored at a particular address?**

C permits this through **dereferencing**.

Dereferencing Pointers

The syntax for dereferencing a pointer is *.

```
int a = 5;
int *ptr = &a;
printf("%d", *ptr);
```

Output: 5

Understanding the Syntax

Declare a **pointer-to-int** named `ptr`.

```
int *ptr;
```

The variable `ptr` is a **pointer-to-int** and `*ptr` is an `int`.

Pointer Examples

Understand the difference between the following:

- `*ptr`
- `*ptr + 1`
- `*(ptr + 1)`
- `(*ptr) + 1`
- `*&ptr`
- `&ptr`
- `&ptr + 1`

Assigning Manual Locations

It is possible to assign a memory location to a pointer manually.

```
int *ptr = (int *) 4;
```

However, the operating system may not allow the program to alter the contents at that memory location.

Default Assignment

It is good practice to assign `NULL` to pointer declarations.

```
int *ptr = NULL;
```

`NULL` is defined in most of the standard library headers, including `stdio.h`.

Testing Pointers

Example: Testing pointers for valid addresses

Arrays and Pointers

The name of an array points to the address of the first object in the array.

We can use pointer arithmetic to move to subsequence addresses.

```
char arr[] = { 'a', 'b', 'c', 'd' };
char *c_ptr = arr + 2;
char c = *c_ptr; // 'c'
```

Arrays and Pointers

Example: Pointer arithmetic on arrays

Strings and Pointers

Using pointer notation with strings is very similar to using pointers with arrays.

The identifier of the string is a pointer to the first character in the string.

Strings and Pointers

Example: print_string.c

This example also showed the usage of the `const` keyword. When added at the start of a variable declaration, this qualifier prevents the variable from being modified.

Strings and Pointers

The string functions provided in `string.h` require pointers to `char`.

Compare the input to the function declarations listed at `https://www.cplusplus.com/reference/cstring/`

Example: String tokenization and string search.

String Literals vs. Character Arrays

In the previous example, we saw that the following initializations produced different results:

```
// Character Array
char arr[] = "char array.";

// String Literal
char *arr_ptr = "String literal.";
```

String Literals vs. Character Arrays

They seem very similar, but the C standard has different rules regarding them.

See Section 6.7.8 Example 32 `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf`

String Literals vs. Character Arrays

The first declaration

```
char arr[] = "char array.";
```

Creates a `char` array object `arr` and initializes it with the string literal `"char array."`

String Literals vs. Character Arrays

The second declaration

```
char *arr_ptr = "String literal.";
```

Points to an object with type "array of `char`" whose elements are initialized with a string literal.

Any attempt to modify the array pointed to by `arr_ptr` is undefined.

Pointer Arithmetic for 2D Arrays

Compare and understand the following examples of pointer arithmetic with a 2D array.

```
char arr2d[10][10];
char *ptr1 = *arr2d; // &arr2d[0][0]
*ptr1 = *(arr2d + 1); // &arr2d[1][0]
*ptr1 = *(*arr2d + 1); // &arr2d[0][1]
```

Double Indirection

A pointer to another pointer is referred to as **double indirection**.

```
int a = 10;
int *b = &a;
int **c = &b;
```

Double Indirection and Arrays

It might seem intuitive at this point to think of the following as a possibility:

```
int arr[2][2] = { 0 };
int **arr_ptr = arr;
```

We have already seen how the identifier of the array is the address.

```
int arr[2] = { 0 };
int *arr_ptr = arr;
```

Double Indirection and Arrays

Example: `array2d_static.c` **and** `array2d_pointers.c`

Compare access of 2D array statically versus one with double indirection.

Memory Layout of Static Arrays

This is similar to a 2D array in some respects, but the memory layout between pointers-to-pointers and a static 2D array is different.

Recall that when an array is created in C, the values of the array are guaranteed to be contiguous in memory.

Memory Layout of Static Arrays

A static 2×2 array in C would have the following memory layout.

Location	Value
0	1
4	2
8	3
12	4

Memory Layout of Pointer Arrays

An 2×2 array of pointers-to-`int` might have the following layout.

Location	Value
0	1000
8	2000
16	3000
24	4000

The values are addresses of each integer.

Memory Layout of Arrays

Example: ptrptr.c

Pointers to Functions

Just as each variable identifier has an address associated with it, the identifier of a function also has an address.

This address represents the location of the execution code for that function.

Pointers to Functions

The fact that the identifier is an address means that we can declare a pointer to a function returning any type.

The value of such a pointer would be the address pointing to the execution code of the function.

Pointers to Functions

Consider the following declaration.

```
int (*fn_ptr)(int a);
```

First, what is `(*fn_ptr)(int a)`?

Pointers to Functions

Consider the following declaration.

```
int (*fn_ptr)(int a);
```

First, what is `(*fn_ptr)(int a)`?

This pointer to a function is an `int`.

Pointers to Functions

The parentheses around `*fn_ptr` indicate that the identifier is bound to the dereference operator before the argument list.

Pointers to Functions

Removing the parenthesis yields

```
*fn_ptr(int a);
```

This is a function taking a single `int`. We have already established that it returns `int`.

Pointers to Functions

Removing the dereference operator leaves a pointer to a function that returns an `int` and accepts an `int` as input.

This is analogous to the following:

```
int *ptr;
```

`*ptr` will return an `int`. Removing the dereference operator returns an address.

Pointers to Functions

Example: operator_ptr.c

Function Pointers: Another Example

```
int (*fn (char *c)) (int a, int b);
```

`fn` is a function that takes one string and returns a pointer to a function that returns a `int` and accepts two `int` values.

Function Pointers: Another Example

Example: operator_ptr_return.c

Function Pointers: qsort

qsort is a function from the standard C library which implements the quick sort algorithm.

The declaration of the function is:

```
void qsort(void *base, size_t num, size_t width,
  int(*compare)(const void *elem1, const void *elem2));
```

Function Pointers: `qsort`

The first three parameters relate to the values or objects to be sorted.

- `base` - The array of elements.
- `num` - The number of elements in the array.
- `width` - The byte size of each element.

Function Pointers: `qsort`

The fourth argument is a function pointer which is used to compare two elements in the array. This will be defined depending on the application.

The input type for this comparison function is `void *` so that it can handle any data type.

Function Pointers: `qsort`

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two members are equal, their relative order is undefined.

Function Pointers: `qsort`

Example: `qsort_basic.c`

qsort and structs

It is also useful to complex elements such as an array of `struct` depending on some member value.

This requires that the elements of the comparison function be cast to the desired `struct` and member.

qsort and structs

Example: qsort_struct.c

Command Line Arguments

Thus far, we have accepted `void` as a formal parameter to `main`.

Our programs can become more general by accepting parameters from the command line.

Command Line Arguments

In C, the `main` function accepts two formal parameters:

```
int main(int argc, char **argv) {
    return 0;
}
```

Command Line Arguments

The first argument `argc` represents the number of command line arguments passed via `stdin`, including the name of application.

Command Line Arguments

Source

```c
#include <stdio.h>
int main(int argc, char **argv) {
    printf("%d\n", argc);
    return 0;
}
```

Output

```
$ ./a.out arg1 arg2 arg3
4
```

Command Line Arguments

The second argument is a `pointer-to-pointer-to-char`.

It stores each individual command line argument, where an argument is separated by a space.

Command Line Arguments

Example: Print Arguments

Command Line Arguments

Example: Command Line Operators