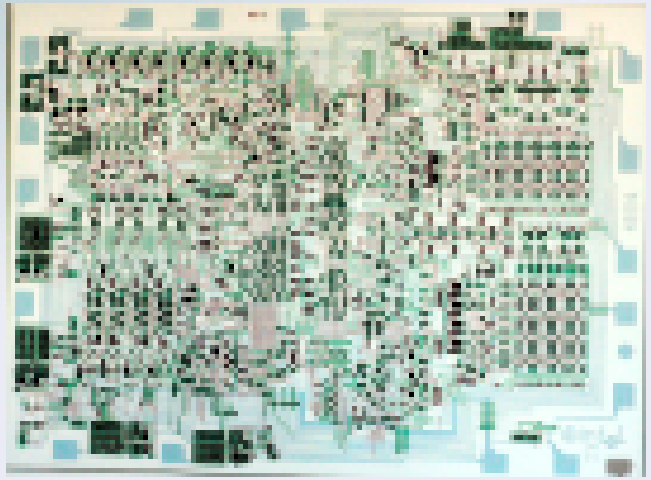

Lecture 13a: Programming in The C Language: Pointers, Arrays, Structs & Strings

CSIS11: **Computer Architecture and Organization**

Readings

- *Chapters 16* Patt and Patel: Introduction to Computing Systems...
- *The C Language* - Kernighan and Ritchie - Second Edition, strongly recommended
- Beej's Guide to C Programming (Tutorial) - Instructor Repo
- Beej's Guide to C Programming (Library Reference) - Instructor Repo
- C to LC-3 Web page - see Instructor Repository

Remaining Schedule

Day	Content	Chapter(s)
4/21	C: Introduction, Variables and Operators	11/12
4/23	C: Control Structures	13
4/28	C: Storage and Functions	14
4/30	**C: Pointers, Arrays and Structs**	**16**
5/5	C: Pointers, Arrays and Structs (cont) and I/O	18
5/7	Guest Speaker	
5/12	Review and Work on Final Project	
5/14	Review and Work on Final Project	
5/21	**Final Exam – Wed, May 21, 2025 6-8:30 PM**	

Pointers and Arrays

We've seen examples of both of these in our LC-3 programs; now we'll see them in C.

Pointer

- Address of a variable in memory
- Allows us to indirectly access variables
- In other words, we can talk about its *address* rather than its *value*

Array

- A list of values arranged sequentially in memory
- Example: a list of telephone numbers
- In other words, we can talk about its *index* (abstraction) rather than its *address*

Address vs. Value

Sometimes we want to deal with the address of a memory location, rather than the value it contains.

Recall example from Chapter 6: adding a column of numbers (*add_nums.asm*)

```
        LEA R2, DATA   ; load the starting address of the data

LOOP    LDR R3, R2, x0 ; load the next number to be added
        ADD R2, R2, #1 ; increment the pointer
        ADD R1, R1, R3 ; add the next number to the running sum
        ADD R4, R4, #-1 ; decrement the counter
```

x300B	x0001	1	DATA .FILL x01
x300C	x0002	2	.FILL x02
x300D	x0003	3	.FILL x03
x300E	x0004	4	.FILL x04
x300F	x0005	5	.FILL x05
x3010	x0006	6	.FILL x06
x3011	x0007	7	.FILL x07
x3012	x0008	8	.FILL x08
x3013	x0009	9	.FILL x09
x3014	x000A	10	.FILL x0A

Pointers in C

C lets us talk about and manipulate pointers as variables and in expressions.

Declaration:

```
int *p;  /* p is a pointer to an int */
```

- A pointer in C is **always a pointer to a particular data type**: int*, double*, char*, etc.
- And since it knows it's *type*, pointer arithmetic is always accurate

Operators:

- `*p` -- returns the value pointed to by p
- `&z` -- returns the address of variable z

Example

```
int i;
int *p_i;
i = 4;
p_i = &i;
*p_i = *p_i + 1;
```

- Store the value 4 into the memory location associated with i
- Store the address of i into the memory location associated with p_i
- Read the contents of memory at the address stored in p_i
- Store the result into memory at the address stored in p_i

Example: LC-3 Code

```
; i = 4;
AND  R0, R0, #0  ; clear R0
ADD  R0, R0, #4  ; put 4 in R0
STR  R0, R5, #0  ; store in i

; ptr = &i;
ADD  R0, R5, #0  ; R0 = R5 + 0 (addr of i)
STR  R0, R5, #-1 ; store in ptr

; *ptr = *ptr + 1;
LDR  R0, R5, #-1 ; R0 = ptr
LDR  R1, R0, #0  ; load contents (*ptr)
ADD  R1, R1, #1  ; add one
STR  R1, R0, #0  ; store result where R0 points
```

Functions and Arguments

C Function Argument Passing:

- **Arguments are passed by value -** functions receive **copies** of the values, not the original variables
- **Changes to parameters inside functions don't affect original arguments**
- **Pointers can be passed** to allow functions to modify original variables
- **Arrays are automatically passed as pointers** to their first element

Pointers as Arguments

Passing a pointer into a function allows the function to read/change memory outside its stack frame.

```
void ptrSwap(int *firstVal, int *secondVal)
{
  int tempVal = *firstVal;
  *firstVal = *secondVal;
  *secondVal = tempVal;
}
```

Arguments are integer pointers. Caller passes addresses of variables that it wants function to change.

Passing Pointers to a Function

main() wants to swap the values of valueA and valueB

Passes the addresses to ptrSwap:

```
ptrSwap(&valueA, &valueB);
```

See *newswap.c*

Null Pointer

Sometimes we want a pointer that points to nothing.

In other words, we declare a pointer, but we're not ready to actually point to something yet.

```
int *p;
p = NULL;  /* p is a null pointer */
```

NULL is a predefined macro that contains a value that a non-null pointer should never hold.

Often, NULL = 0, because Address 0 is not a legal address for most programs on most platforms.

Using Arguments for Results

Pass address of variable where you want result stored

- Useful for multiple results

Example:

- Return value via pointer
- Return status code as function result

This solves the mystery of why '&' with argument to scanf:

```
scanf("%d ", &dataIn);
```

Reads a decimal integer and store in dataIn

Syntax for Pointer Operators

Declaring a pointer

```
type *var;
type* var;
```

Either of these work -- whitespace doesn't matter. Type of variable is int* (integer pointer), char* (char pointer), etc.

Creating a pointer "&"

```
&var
```

Must be applied to a memory object, such as a variable. In other words, &3 is not allowed.

Dereferencing "*"

Can be applied to any expression. All of these are legal:

- `*var` - contents of mem loc pointed to by var
- `**var` - contents of mem loc pointed to by memory location pointed to by var
- `*3` - contents of memory location 3

Arrays

How do we allocate a group of memory locations?

- Character string
- Table of numbers

How about this?

```
int num0;
int num1;
int num2;
int num3;
```

Not too bad, but…

- What if there are 100 numbers?
- How do we write a loop to process each number?

Fortunately, C gives us a better way -- the array.

```
int num[4];
```

Declares a sequence of four integers, referenced by: num[0], num[1], num[2], num[3]

Array Syntax

Declaration

```
type variable[num_elements];
```

- All array elements are of the same type
- Number of elements must be known at compile-time

Array Reference

```
variable[index];
```

- i-th element of array (starting with zero)
- No limit checking at compile-time or run-time

Array as a Local Variable

Array elements are allocated as part of the stack frame.

```
int grid[10];
```

First element (grid[0]) is at lowest address of allocated space.

```
grid[0]
grid[1]
grid[2]
grid[3]
...
grid[6]
grid[7]
grid[8]
grid[9]
```

Passing Arrays as Arguments

C passes arrays by reference

- The address of the array (i.e., of the first element) is written to the function's stack frame
- Otherwise, would have to copy each element

There must be a constant, e.g., `#define MAX_NUMS 10`

```c
#define MAX_NUMS 10
int Average(int inputValues[MAX_NUMS]) {
    int sum = 0;
    for (int index = 0; index < MAX_NUMS; index++)
    {
        sum = sum + inputValues[index];
    }
    return (sum / MAX_NUMS);
}

int main() {
    int numbers[MAX_NUMS];
    // …
    int mean = Average(numbers);
    // …
}
```

A String is an Array of Characters

Allocate space for a string just like any other array:

```
char outputString[16];
```

Space for string **must contain room for terminating zero**.

Special syntax for initializing a string:

```
char outputString[11] = "Result is ";
```

…which is the same as:

```
outputString[0] = 'R';
outputString[1] = 'e';
outputString[2] = 's';
...
```

I/O with Strings

Printf and scanf use "%s" format character for string

Printf -- print characters up to terminating zero

```
printf("%s", outputString);
```

Scanf -- read characters until whitespace, store result in string, and terminate with zero

```
scanf("%s", &inputString);
```

Arrays and Pointers

An array name is essentially a pointer to the first element in the array

```
char word[10];
char *cptr;
cptr = word;  /* points to word[0] */
```

Difference: Can change the contents of cptr, as in

```
cptr = cptr + 1;
```

Ptr and Array Notation

Given the declarations on the previous page, each line below gives three equivalent expressions:

```
cptr                    word                    &word[0]
(cptr + n)              word + n                &word[n]
*cptr                   *word                   word[0]
*(cptr + n)             *(word + n)             word[n]
```

Pitfalls with Arrays in C

Overrun array limits

- There is no checking at run-time or compile-time to see whether reference is within array bounds.

```
int array[10];
int i;
for (i = 0; i <= 10; i++) array[i] = 0;
```

Declaration with variable size

- Size of array must be known at compile time.

```
int num_elements = 10;
void SomeFunction(int num_elements) {
  …
}
void main()
{
    ...
    int temp[num_elements];

}
```

Pointer Arithmetic

Address calculations depend on size of elements

- In our LC-3 code, we've been assuming one word per element.
- e.g., to find 4th element, we add 4 to base address
- It's ok, because we've only shown code for int and char, both of which take up one word.
- If double, we'd have to add 8 to find address of 4th element.

C does size calculations under the covers, depending on size of item being pointed to:

```
double x[10];
double *y = x;
*(y + 3) = 13;
```

- Allocates 20 words (2 per element)
- Same as x[3] -- base address plus 6

C struct

Data Structures

A data structure is a particular organization of data in memory.

We want to:

- Group related items together
- Organize these data bundles in a way that is convenient to program and efficient to execute

An array is one kind of data structure.

- a **struct** – directly supported by C, allows for multiple data types, unlike an array

Structures in C

A **struct** is a mechanism for grouping together related data items of different types.

Recall that an array groups items of a single type.

Example: We want to represent an airborne aircraft:

```
char flightNum[7];
int altitude;
int longitude;
int latitude;
int heading;
double airSpeed;
```

We can use a struct to group these data together for each plane.

Defining a Struct

We first need to define a new type for the compiler and tell it what our struct looks like.

```
struct flightType {
  char flightNum[7];  /* max 6 characters */
  int altitude;       /* in meters */
  int longitude;      /* /* in tenths of degrees */
  int latitude;       /* in tenths of degrees */
  int heading;        /* in tenths of degrees */
  double airSpeed;    /* in km/hr */
};
```

This tells the compiler how big our struct is and how the different data items ("members") are laid out in memory.

But it does not allocate any memory.

Declaring and Using a Struct

To allocate memory for a struct, we declare a variable using our new data type.

```
struct flightType plane;
```

Memory is allocated, and we can access individual members of this variable:

```
plane.airSpeed = 800.0;
plane.altitude = 10000;
```

A struct's members are laid out in the order specified by the definition.

```
plane.flightNum[0]
plane.flightNum[6]
plane.altitude
plane.longitude
plane.latitude
plane.heading
plane.airspeed
```

Defining and Declaring at Once

You can both define and declare a struct at the same time.

```
struct flightType {
  char flightNum[7];  /* max 6 characters */
  int altitude;       /* in meters */
  int longitude;      /* in tenths of degrees */
  int latitude;       /* in tenths of degrees */
  int heading;        /* in tenths of degrees */
  double airSpeed;    /* in km/hr */
} maverick;
```

And you can use the flightType name to declare other structs.

```
struct flightType iceMan;
```

typedef

C provides a way to define a data type by giving a new name to a predefined type.

Syntax: `typedef <type> <name>;`

Examples:

```
typedef int Color;
typedef struct flightType Flight;
typedef struct ab_type {
   int a;
   double b;
} ABGroup;
```

Using typedef

This gives us a way to make code more readable by giving application-specific names to types.

```
Color pixels[500];
Flight plane1, plane2;
```

Typical practice:

- Put typedef's into a header file, and use type names in main program
- If the definition of Color/Flight changes, you might not need to change the code in your main program file

Array of Structs

Can declare an array of structs:

```
Flight planes[100];
```

Each array element is a struct (7 words, in this case).

To access member of a particular element:

```
planes[34].altitude = 10000;
```

Because the [] and . operators are at the same precedence, and both associate left-to-right, this is the same as:

```
(planes[34]).altitude = 10000;
```

Example using Pointers

Review *pointers.c*