

# 198:211 Computer Architecture

Week 3  
Fall 2010

- Topics:
  - Advanced Types
  - Pointers
  - Arrays

## 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:

```
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 {  
  
    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.

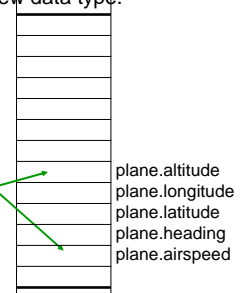
## Declaring and Using a Struct

- 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.



## Defining and Declaring at Once

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

```
• struct celebrity {  
  int height; /* in meters */  
  int weight; /* in ounces */  
  char haircolor; /* B, G, etc*/  
  double wealth; /* in MillionDollars and cents */  
} beyonce;
```

- And you can use the celebrity name to declare other structs.

```
• struct celebrity angelinajolie;
```

## typedef

- C provides a way to define a data type by giving a new name or alias to a predefined type.
- Can use any readable name for pre defined type

- **Syntax:**

```
• typedef <type> <name>;
```

- **Examples:**

```
• typedef int Color;  
• typedef struct celebrity;
```

## Using typedef

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

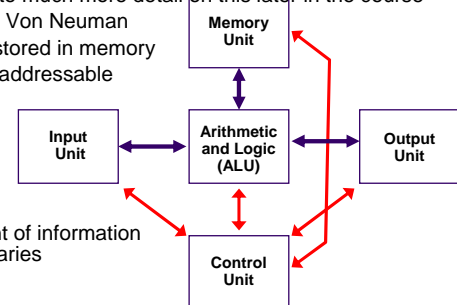
```
• Color x, y;  
• Celebrity beyonce, madonna;
```

- **Typical practice:**

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

## Quick Note on Memory...

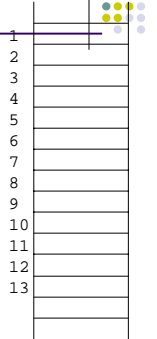
- We'll get into much more detail on this later in the course
- Remember Von Neuman
- All data is stored in memory
- Memory is addressable



- The amount of information allocated varies

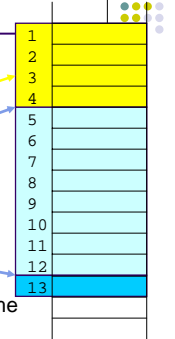
## Memory

- Think of memory as a long continuous array
- Memory allocated based on bytes



## Memory

- Think of memory as a long continuous array
- Memory allocated based on bytes
- `int x;`
- `double y;`
- `char z;`
- This memory says allocated for the lifetime of the variable

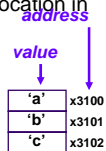


## Pointers - Introduction

- Any variable that is declared within a program is assigned to a portion of memory to store its assigned value.
- Run time systems maintains a mapping (called a **symbol table**) that store, for each variable, its assigned location in memory.

```
char char1 = 'a';
char char2 = 'b';
```

VarName	MemLoc
char1	x3100
char2	x3101



## Pointers - Intro (continued)

- In most cases, when we talk about variables, we talk about the variable name and its value
- Now we will talk about its **memory address**
- A **Pointer** is a type of variable that stores the address of another variable in memory
- Allows us to **indirectly** access variables
  - in other words, we can talk about its **address** (or where its stored) rather than its **value**

## 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 usually a pointer to a particular data type: `int*`, `double*`, `char*`, etc.
- **Pointer Operators**
- `*p` -- returns the value pointed to by p
- `&z` -- returns the address of variable z

## Pointers in C

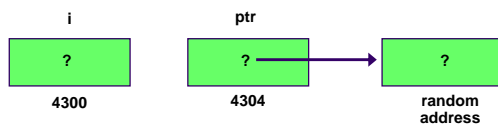
- **Declaration**
- `int a=5;` /\*a is integer with value 5 \*/
- `int *p;` /\* p is a pointer to an int \*/
- **Assignment**
- `*p = a` /\* STMT 1 make p point to integer a \*/
- `p = &a` /\* STMT 2 make p point to address of a\*/
- STMT 1 and STMT 2 are equivalent

a	5	Address=10
		Address=6
p	10	Address=2

Printf ( "... format ...", a, &a, \*p, p, &p)

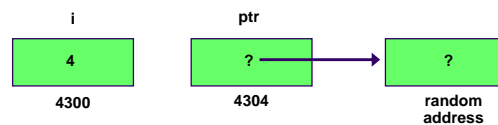
## Pointer Example

- `int i;`
- `int *ptr;`
- `i = 4;`
- `ptr = &i;`
- `*ptr = *ptr + 1;`



## Pointer Example

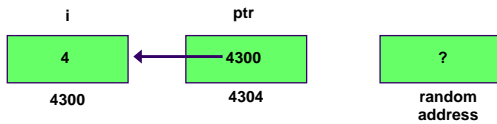
- `int i;`
- `int *ptr;` store the value 4 into the memory location associated with i
- `i = 4;`
- `ptr = &i;`
- `*ptr = *ptr + 1;`



## Pointer Example

- `int i;`
- `int *ptr;`
- `i = 4;`
- `ptr = &i;`
- `*ptr = *ptr + 1;`

store the address of i into the memory location associated with ptr



## Pointer Example

- `int i;`
- `int *ptr;`
- `i = 4;`
- `ptr = &i;`
- `*ptr = *ptr + 1;`

store the result into memory at the address stored in ptr

read the contents of memory at the address stored in ptr

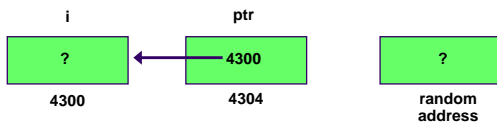


## Pointer Example

- `int i;`
- `int *ptr;`
- `i = 4;`
- `ptr = &i;`
- `*ptr = *ptr + 1;`

What would happen if the last statement was:

- `ptr = *ptr + 1;`
- `*ptr = ptr + 1;`
- `ptr = ptr + 1;`



## Pointer example

```
#include <stdio.h>

main ()
{
    int i;
    int *ia;
    i = 10;
    ia = &i;

    printf (" The address of i is %8u \n", ia);
    printf (" The value at that location is %d\n", i);
    printf (" The value at that location is %d\n", *ia);
    *ia = 50;
    printf ("The value of i is %d\n", i);
}
```

Example from: [http://www.java2s.com/Tutorial/C/C0200\\_Poiner/UsingPointers.htm](http://www.java2s.com/Tutorial/C/C0200_Poiner/UsingPointers.htm)

## Useful Examples for Pointers

- Consider the following function that's supposed to swap the values of its arguments.
- ```
void Swap(int firstVal, int secondVal)
{
    int tempVal = firstVal;
    firstVal = secondVal;
    secondVal = tempVal;
}
```
- ```
int fv = 6, sv = 10;
Swap(fv, sv);
printf("Values: (%d, %d)\n", fv, sv);
```
- Recall that functions are **pass-by-value**...

## Pointers as Arguments

- Passing a pointer into a function allows the function to read/change memory.
- ```
void NewSwap(int *firstVal, int
*secondVal)
{int tempVal = *firstVal;
*firstVal = *secondVal;
*secondVal = tempVal;
}
```
- ```
int fv = 6, sv = 10;
NewSwap(&fv, &sv);
printf("Values: (%d, %d)\n", fv, sv);
```

## 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);
```

read a decimal integer  
and store in dataIn

## Arrays

- Arrays are declared with a type and a size.
- Size **must** be known at compile time.
- ```
char fool[20]; // an array of 20 characters
```
- Arrays are accessed by referring to their elements using index numbers:
  - ```
fool[10]; // value stored at the 11th element
```
- All arrays indexes start at 0;
- There is **NO** checking on array bounds
- ```
int array[10], i =0;
for(i=0; i< 10; i++){
    array[i] = i
for(i=0; i< 15; i++){
    printf("%d\t%d\n", i, array[i]);
}
```

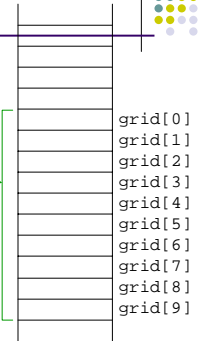
## Array Storage

Elements of an array are stored sequentially in memory

```
char grid[10];
```

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

Knowing the first element is enough to access any element



## SizeOf

- `sizeof(x)` - is a function to determine the amount of storage (in bytes) required to store an object of the type of its operand
- ```
char char_array[20];
```
- ```
int int_array[20];
```
- `sizeof(int)` is 4
- `sizeof(char)` is 1
- `sizeof(char_array)` is 20
- `sizeof(int_array)` is 80
- Number of elements in `int_array`
- `sizeof(int_array)/sizeof(int)` is 20

## Relationship between Arrays and Pointers

- An array name is essentially a pointer to the first element in the array
  - ```
.... [0]
char word[10];
char *cptr;

cptr = word; /* points to word[0] */
```
- **Difference:**
  - Can change the contents of `cptr`, as in
    - ```
cptr = cptr + 1;
```
  - (The identifier "word" is not a variable.)

## Correspondence between Ptr and Array Notation

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

```
char word[10];
char *cptr;
cptr = word;
```

|               |             |          |
|---------------|-------------|----------|
| • cptr        | word        | &word[0] |
| • (cptr + n)  | word + n    | &word[n] |
| • *cptr       | *word       | word[0]  |
| • *(cptr + n) | *(word + n) | word[n]  |

## Passing Arrays as Arguments

- C passes arrays by reference (actually... it's by value)
  - the address of the array (i.e., of the first element) is copied.
  - otherwise, would have to copy each element.

```
main() {
  int numbers[MAX_NUMS];
  ...
  mean = Average(numbers);
}
int Average(int inputValues[]) {
  for (index = 0; index < MAX_NUMS; index++)
    sum = sum + inputValues[index];
  return (sum / MAX_NUMS);
}
int Average(int *inputValues); /* Alternative Prototype */
```

This must be a constant, e.g.,  
#define MAX\_NUMS 10

## Common 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.

```
void SomeFunction(int num_elements) {
  int temp[num_elements];
  ...
}
```

## Pointer Arithmetic

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

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

allocates 10 doubles, size of 80 bytes

```
char s[10];
```

same as x[3]

```
char *t = s;
*(t + 3) = 'A';
```

How far does each pointer move?

## Array of Structs

- Can declare an array of structs:
- `Flight planes[100];`
- Each array element is a struct.
- 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;`

## Pointer to Struct

- We can declare and create a pointer to a struct:
- `Flight *planePtr;`  
`planePtr = &planes[34];`
- To access a member of the struct addressed by a Ptr:
- `(*planePtr).altitude = 10000;`
- Because the `.` operator has higher precedence than `*`, this is **NOT** the same as:
- `*planePtr.altitude = 10000;`
- C provides special syntax for accessing a struct member through a pointer:
- `planePtr->altitude = 10000;`

## Passing Structs as Arguments

- Unlike an array, a struct is always **passed by value** into a function.
  - This means the struct members are copied to the function's state and changes inside the function are not reflected in the calling routine's copy.
- Most of the time, you'll want to pass a **pointer** to a struct.
- ```
int Collide(Flight *planeA, Flight *planeB)
{
    if (planeA->altitude == planeB->altitude) {
        ...
    }
    else
        return 0;
}
```