COE 251

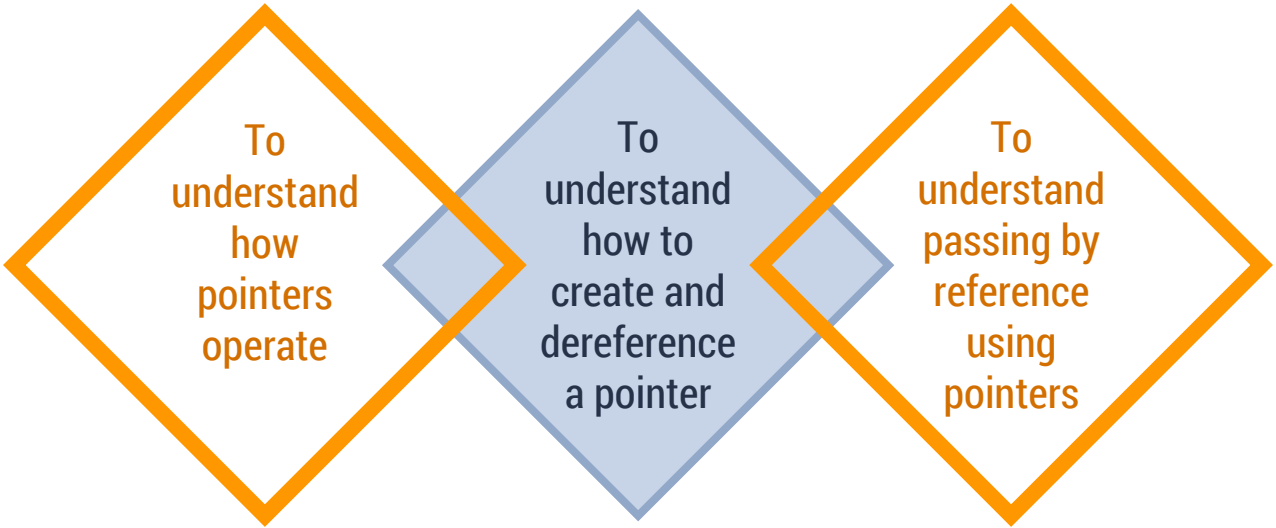
POINTERS

Dr. Eliel Keelson

OUTLINE

- DECLARING POINTERS
- DEREFERENCING POINTERS
- PASSING POINTERS TO FUNCTIONS

INTENDED LEARNING OUTCOMES (ILOs)



1

POINTERS

POINTERS

- A pointer can be defined basically as a variable that stores the address (memory location) of another variable.
- All variables that are declared reside in memory.
- And every memory space has an address (hexadecimal address).

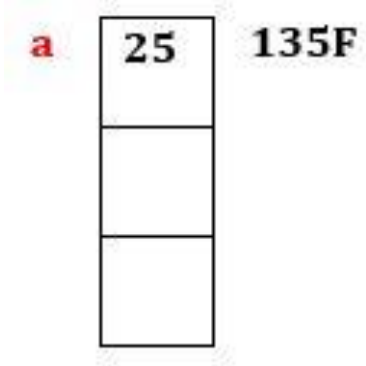
POINTERS

- Pointers are variables that would hold the hexadecimal address of another variable which resides in memory.
- Pointers could therefore be likened to an address book.
- Let's demonstrate with an example what happens when a variable is declared.

POINTERS

```
#include <stdio.h>
main()
{
int a = 25;
}
```

The variable a can be seen in memory as.



a is the name of the variable, 25 is its value and 135F is its hexadecimal address (just assuming) in the computer's memory.

POINTERS

- So when we create pointers they would store these hexadecimal addresses of other variables.

DECLARING POINTERS

POINTER DECLARATION

- The general syntax for declaring a pointer is:

data_type * pointer_name;

So some examples include:

int * ab;

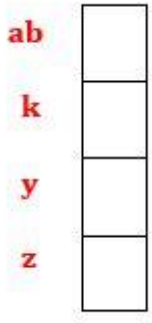
float * k;

char * y;

double * z;

POINTER DECLARATION

- Just like every other declared variable, spaces are made available for them. So the above declared pointers would be seen in memory as:



ASSIGNING POINTERS WITH ADDRESSES

- Since pointers hold the addresses of other variables, they are assigned addresses of other variables using the following syntax:

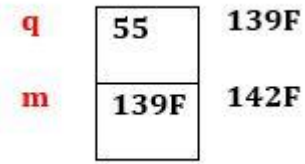
```
pointer_name = &variable;
```

- '**&**' is known as the "**address of**" operator. It produces the address of any variable it is attached to.
- An example is demonstrated on the next slide

ASSIGNING POINTERS WITH ADDRESSES

```
#include <stdio.h>
main()
{
int q = 55;
int * m;
m = &q;
}
```

- Assuming the address of q is 139F and that of m is 142F then they can be represented in memory as



ASSIGNING POINTERS WITH ADDRESSES

- It is important to note that the value of an assigned pointer is the hexadecimal address of the variable it points to.
- Also since the pointer resides in memory, it also has its own distinct memory address.
- So from the previous example, note that the value of q is 55 whiles the value of the pointer m is the address of q which is 139F. This is so be because m was assigned the address of q (i.e. m = &q;) and the pointer m is residing in memory location 142F.

ASSIGNING POINTERS WITH ADDRESSES

- The rule for assigning pointers states that: **The data type of a pointer should be the same as the variable it holds its address.**
- This is to say that only a pointer with a data type of **int** can hold the address of a variable with a data type of **int**.
- Also only a pointer with a data type of **float** can hold the address of a variable with a data type of **float**.
- As demonstrated in the next slide

ASSIGNING POINTERS WITH ADDRESSES

```
#include <stdio.h>
main()
{
int k = 90;
float j = 54.28;
int * y = &k;
float * x = &j;
}
```

- From the code we realise that y was made to point to k and not j, because y and k have the same data type.
- Also x was made to point to j and not k, because x and j have the same data type.
- This rule applies for other data types.

ASSIGNING POINTERS WITH ADDRESSES

- There is however an exception to this rule of matching data types.
- Any pointer declared as a void pointer has the capability of pointing to all kinds of variables of varying data types.
- However they cannot be ***dereferenced***. A lot more on dereferencing would be discussed later.
- Example

```
int e = 75;
float b = 98.261;
void  z = &e;
void h = &b;
```

DEREFERNCING POINTERS

DEREFERNCING POINTERS

- Since a pointer holds the location/address of another variable, which is a reference to that variable, then the pointer can be made to access/modify the value of that variable.
- This is known as **dereferencing pointers.**

DEREFERNCING POINTERS

- The syntax for dereferencing pointers is as follows:

  * pointer_name;

- Note that with dereference, a data type does not precede the pointer name.
- An example is demonstrated on the next slide.

DEREFERNCING POINTERS

- If a certain program contains the following lines of code

```
int x = 49;
int * z = &x;
```

- To dereference **z** would simply mean to access the value of **x**.
- And this is done as *** z;**
- This (***z;**) is the same as saying **x;** which is currently 49
- So whenever we dereference a pointer, it gives us access to the value of the variable it points to.

DEREFERNCING POINTERS

- If a certain program contains the following lines of code

```
int x = 49;
int * z = &x;
```

- To dereference **z** would simply mean to access the value of **x**.
- And this is done as *** z;**
- This (***z;**) is the same as saying **x;** which is currently 49
- So whenever we dereference a pointer, it gives us access to the value of the variable it points to.

PASSING POINTERS TO FUNCTIONS

PASSING POINTERS TO FUNCTIONS

- C programming allows passing a pointer to a function as an argument.
- Since pointers hold a reference (address) of variables this method of passing pointers is known as **passing by reference**
- This method allows data items within the calling portion of the program to be accessed by the function and then returned to the calling portion of the program in altered form.

PASSING POINTERS TO FUNCTIONS

- This method of passing by reference is different from the method of passing by value or variable, which was studied previously.
- In passing by value/variable, only a **copy** of the value is sent from the calling function (typically the main) to the called function.
- As such changes made to such values in called function doesn't directly affect the original values in the calling function.

PASSING POINTERS TO FUNCTIONS

- This is however not the same in passing by reference.
- In passing by reference, since the pointers passed as arguments hold a reference (address) of variables in the calling function, changes made in the called function affect directly the original values in the calling function.
- In other words, these changes are made directly to the address where the original variables are stored in memory.

PASSING POINTERS TO FUNCTIONS

- Functions accept pointers as arguments must state so in their declaration and definition.
- Such functions state clearly the data type (type) of the pointer they require as a parameter.
- An example of such a function is shown in the next slide

```c
#include <stdio.h>

void swap(int *a, int *b);

int main()
{
    int m = 10, n = 20;
    printf(“Before Swapping:\n");
    printf("m = %d\n", m);
    printf("n = %d\n\n", n);

    swap(&m, &n);    //passing pointers to function

    printf("After Swapping:\n");
    printf("m = %d\n", m);
    printf("n = %d", n);
    return 0;
}
```

```c
/*
    pointer 'a' and 'b' holds and
    points to the address of 'm' and 'n’
*/

void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

PASSING POINTERS TO FUNCTIONS

- The output of the code above is:

Before Swapping

m = 10

n = 20

After Swapping:

m = 20

n = 10

PASSING POINTERS TO FUNCTIONS

- To clearly demonstrate the difference between passing by value/variable and passing by reference, the swap code above would be tweaked.
- This tweaked code which now uses passing by variable is presented in the next slide

```c
#include <stdio.h>

void swap(int a, int b);

int main()
{
    int m = 10, n = 20;
    printf(“Before Swapping:\n");
    printf("m = %d\n", m);
    printf("n = %d\n\n", n);

    swap(m, n);    //passing pointers to function

    printf("After Swapping:\n");
    printf("m = %d\n", m);
    printf("n = %d", n);
    return 0;
}
```

```c
/*
    pointer 'a' and 'b' holds and
    points to the address of 'm' and 'n’
*/

void swap(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

After running the codes above, would you say that the two codes produced the same output? If yes why, if no why?

PASSING POINTERS TO FUNCTIONS

- The next slide also presents one more code to demonstrate these differences between passing by reference and passing by variable.
- Try and identify which method each function implements and the output of the code.

```c
#include <stdio.h>

void funct1(int u, int v);
void funct2(int *pu, int *pv);

int main()
{
    int u = 1;
    int v = 3;

    printf("\nBefore calling funct1:   u=% v=%d",  u,v);
    funct1(u,v);
    printf("\nAfter calling funct1: u=%d v=%d", u,v);

    printf("\n\nBefore calling funct2: u=%d v=%d",  u,v);
    funct2(&u,&v);
    printf("\nAfter calling funct2:  u=%d   v=%d", u,v);

    return 0;
}
```

```c
void funct1(int u, int v)
{
    u = 0;
    v = 0;
    printf("\nWithin funct1: u=%d v=%d", u,v);
    return;
}


void funct2(int *pu, int *pv)
{
    *pu = 0;
    *pv = 0;
     printf("\nWithin funct2: *pu=%d  *pv=%d",
*pu,*pv);
    return;
}
```

FUNCTIONS RETURNING POINTERS

FUNCTIONS RETURNING POINTERS

- A function can also return a pointer to the calling function.
- In this case you must be careful to have a pointer in the calling function ready to receive the returned pointer from the called function.
- This is because the local pointer within the called function would not be recognized outside the function.
- An example is demonstrated in the next slide

```c
#include <stdio.h>

int* larger(int*, int*);

void main()
{
    int a = 15;
    int b = 92;
    int *p;
    p = larger(&a, &b);
    printf("%d is larger",*p);
}
```

```c
int* larger(int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}
```

“It is also possible to have pointers point to functions often known as Function Pointers. Kindly read more on these…

THANKS!

Any questions?

You can find me at

elielkeelson@gmail.com & ekeelson@knust.edu.gh