

# Computer Programming 143 – Lecture 17

## Pointers I

Faculty of Engineering



# What pointers will help us understand

- Why did we need & in scanf("%d", &a)?
- Why, when you passed an array into a function, and then changed the array in the function, did the original array change?

```
int main()
{
    int b[] = {1, 2, 3, 4};

    printf("%d\n", b[3]);
    foo(b);
    printf("%d\n", b[3]);

    return 0;
}
```

```
void foo(int array[])
{
    array[3] = 20;
}
```

## Three types of variables

- Scalar : `int a;`
- Array : `int b[5];`
- Pointer : Stores memory location of another variable

# Pointer example

```
int x, y;
```

```
int *fooPtr;
```

```
x = 12;
```

```
y = 5;
```

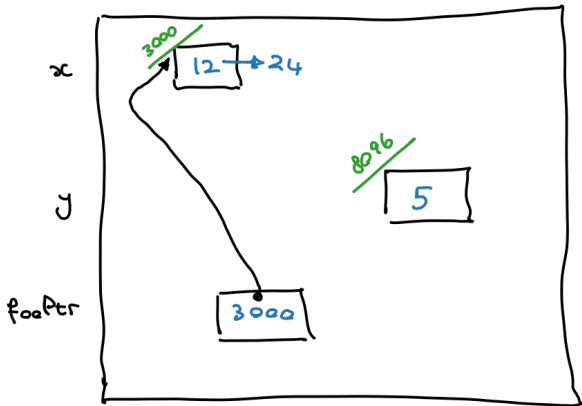
```
fooPtr = &x;
```

```
*fooPtr = 24;
```

```
int b[5];
```

```
b[4] = 3;
```

Computer memory -



## 7.2 Pointer Variable Declaration and Initialisation III

### Pointer declaration

- \* used to declare pointer variables

```
int *myPtr;
```

Defines a pointer to an int value (pointer of type `int`)

- Multiple pointers require using a \* before each variable definition

```
int *myPtr1, *myPtr2, myInt;
```

- Pointer and scalar variables can be declared on the same line
- Can define pointers to any data type
- Initialise pointers to 0, NULL, or an address
  - 0 or NULL – points to nothing (NULL preferred)

```
int *myPtr1 = NULL;
```

```
int *myPtr2;
```

```
myPtr2 = 0x22FF7C; /* Hardware address */
```

## 7.3 Pointer Operators I

### & (address operator)

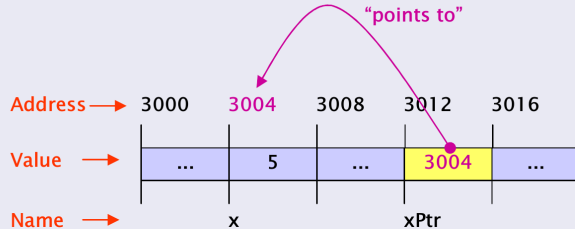
- Returns address of operand

```
int x = 5;
```

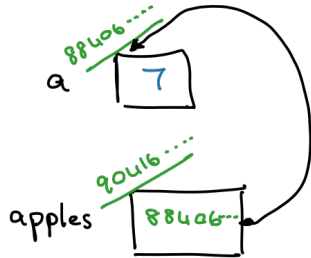
```
int *xPtr;
```

```
xPtr = &x;      /* value of xPtr becomes address of x */
```

Variable xPtr is then said to “point to” x



# ~~Indirection / dereferencing~~



## 7.3 Pointer Operators II

### \* (indirection/dereferencing operator)

- Referencing a value through a pointer is called indirection
- \* is called the indirection or dereferencing operator
- \* not limited to pointer declaration, also used as operand
- \* provides access to memory/variable that its operand points to
- Access can be to read the value or to change the value

```
int x = 5; // assign 5 to x (direct reference)
int *xPtr; // declare an integer pointer
xPtr = &x; // assign the address of x to xPtr
printf("%d", *xPtr); // indirect reference to x, prints 5
*xPtr = 7; // indirect reference, same effect as x = 7
```

- Note: The names of pointer and variable are independent

## 7.3 Memory Concepts

### Memory visualisation

| address | value | variable |
|---------|-------|----------|
|         | ⬇     | ⬇        |
|         | ⋮     |          |
| 3000    | 1     | x        |
| 3004    | 5     | y        |
| 3008    | 8     | z        |
| 3012    |       |          |
| 3016    |       |          |
|         | ⋮     |          |

```
int x = 1;  
int y = 5;  
int z = 8;
```

## 7.3 Memory Concepts

### Memory visualisation

| address | value | variable |
|---------|-------|----------|
|         | ↓     | ↓        |
|         | ⋮     |          |
| 3000    | 1     | x        |
| 3004    | 5     | y        |
| 3008    | 8     | z        |
| 3012    |       | intPtr1  |
| 3016    |       | intPtr2  |
|         | ⋮     |          |

```
int x = 1;           int * apples ;  
int y = 5;  
int z = 8;  
int *intPtr1, *intPtr2;
```

## 7.3 Memory Concepts

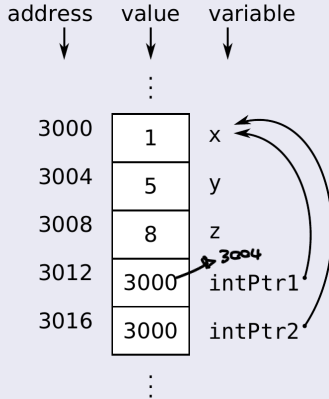
### Memory visualisation

| address | value | variable |
|---------|-------|----------|
|         | ⋮     |          |
| 3000    | 1     | x        |
| 3004    | 5     | y        |
| 3008    | 8     | z        |
| 3012    | 3000  | intPtr1  |
| 3016    | 3000  | intPtr2  |
|         | ⋮     |          |

```
int x = 1;  
int y = 5;  
int z = 8;  
int *intPtr1, *intPtr2;  
intPtr1 = &x;  
intPtr2 = intPtr1;
```

## 7.3 Memory Concepts

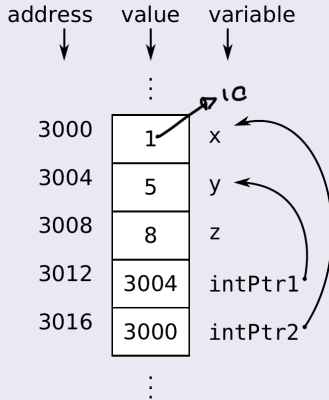
### Memory visualisation



```
int x = 1;
int y = 5;
int z = 8;
int *intPtr1, *intPtr2;
intPtr1 = &x;
intPtr2 = intPtr1;
intPtr1 = &y;
```

## 7.3 Memory Concepts

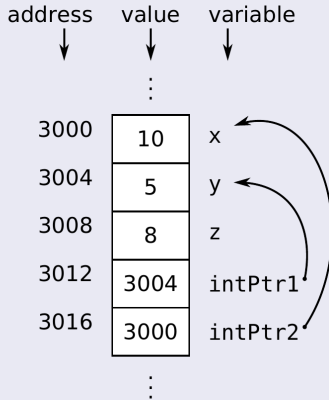
### Memory visualisation



```
int x = 1;
int y = 5;
int z = 8;
int *intPtr1, *intPtr2;
intPtr1 = &x;
intPtr2 = intPtr1;
intPtr1 = &y;
*intPtr2 = 10;
```

## 7.3 Memory Concepts

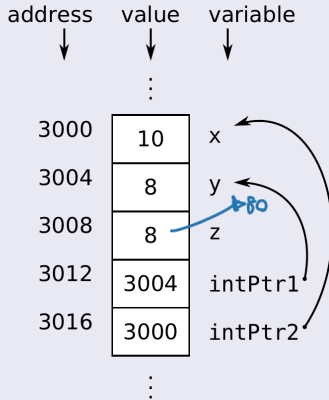
## Memory visualisation



```
int x = 1;
int y = 5;
int z = 8;
int *intPtr1, *intPtr2;
intPtr1 = &x;
intPtr2 = intPtr1;
intPtr1 = &y;
*intPtr2 = 10;
```

## 7.3 Memory Concepts

### Memory visualisation



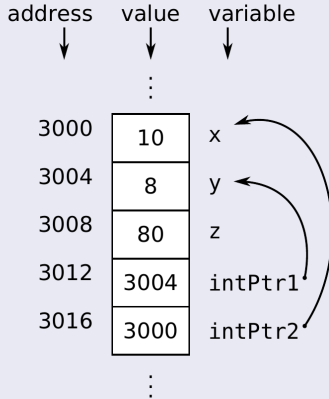
```
int x = 1;
int y = 5;
int z = 8;
int *intPtr1, *intPtr2;
intPtr1 = &x;
intPtr2 = intPtr1;
intPtr1 = &y;
*intPtr2 = 10;
*intPtr1 = z;
z = (*intPtr1) * (*intPtr2);
intPtr1 = &intPtr2;
```

Handwritten annotations in blue:

- Under `*intPtr2` in the assignment `*intPtr2 = 10;`, the value `8` is written.
- Under `*intPtr1` in the assignment `*intPtr1 = z;`, the value `10` is written.
- Under the expression `(*intPtr1) * (*intPtr2)` in the assignment `z = (*intPtr1) * (*intPtr2);`, the values `8` and `10` are written below the respective terms.

## 7.3 Memory Concepts

### Memory visualisation



```
int x = 1;
int y = 5;
int z = 8;
int *intPtr1, *intPtr2;
intPtr1 = &x;
intPtr2 = intPtr1;
*intPtr2 = 10;
*intPtr1 = z;
z = (*intPtr1) * (*intPtr2);
```