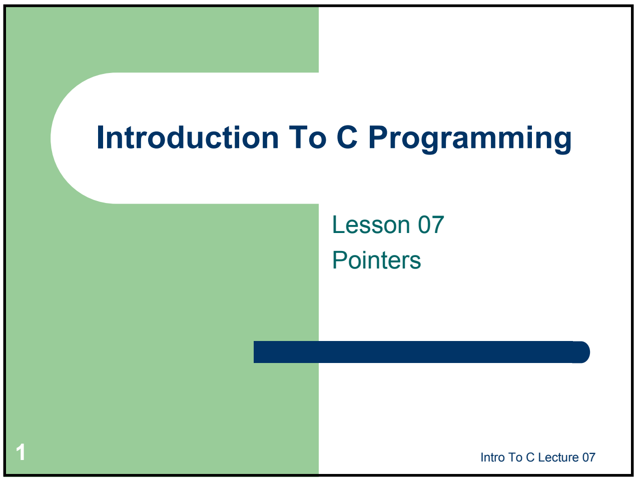
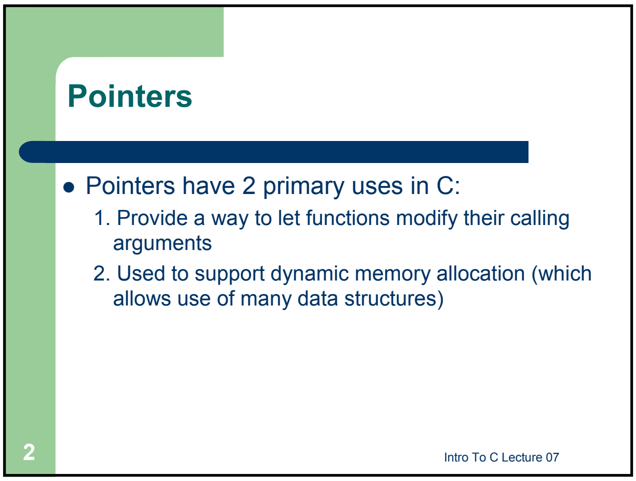


# Introduction To C Programming

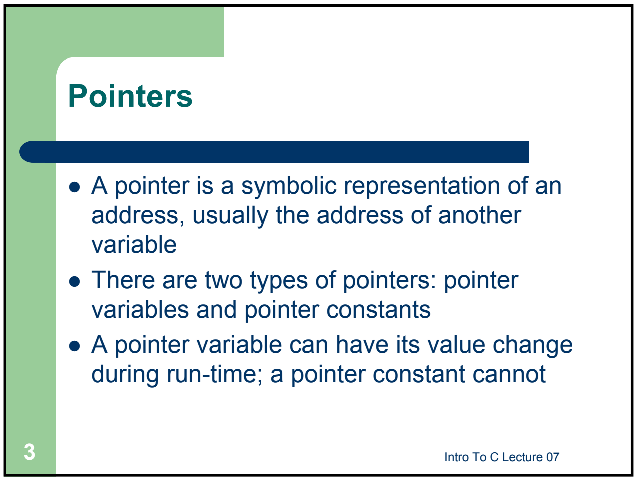
Lesson 07

Pointers



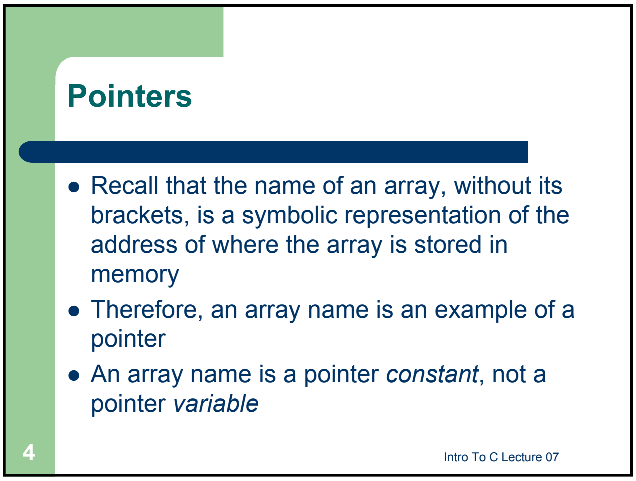
## Pointers

- Pointers have 2 primary uses in C:
  1. Provide a way to let functions modify their calling arguments
  2. Used to support dynamic memory allocation (which allows use of many data structures)



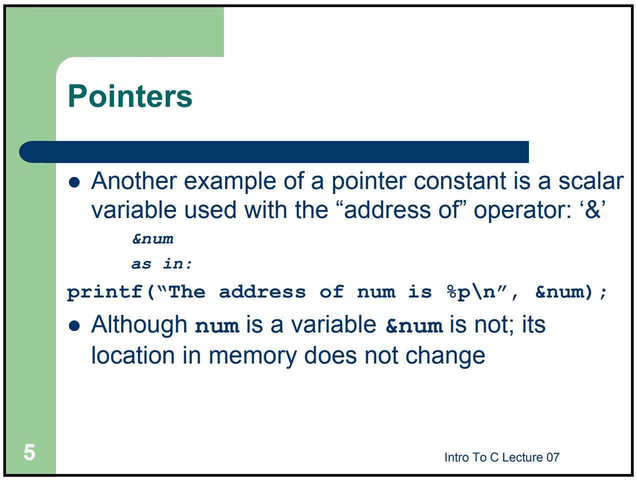
## Pointers

- A pointer is a symbolic representation of an address, usually the address of another variable
- There are two types of pointers: pointer variables and pointer constants
- A pointer variable can have its value change during run-time; a pointer constant cannot



## Pointers

- Recall that the name of an array, without its brackets, is a symbolic representation of the address of where the array is stored in memory
- Therefore, an array name is an example of a pointer
- An array name is a pointer *constant*, not a pointer *variable*

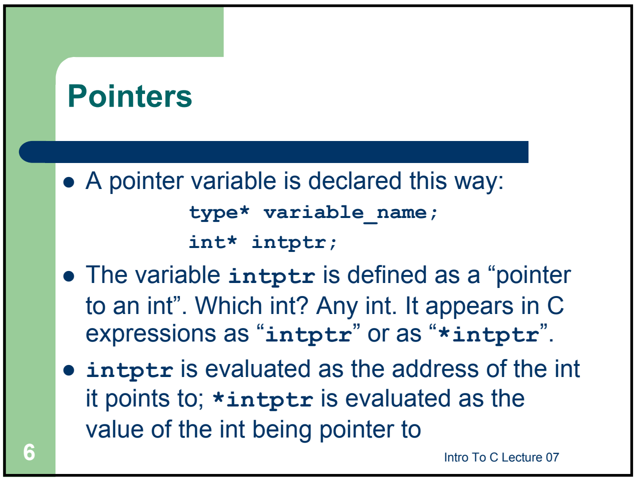


## Pointers

- Another example of a pointer constant is a scalar variable used with the “address of” operator: ‘&’

```
    &num
    as in:
printf(“The address of num is %p\n”, &num);
```

- Although **num** is a variable **&num** is not; its location in memory does not change

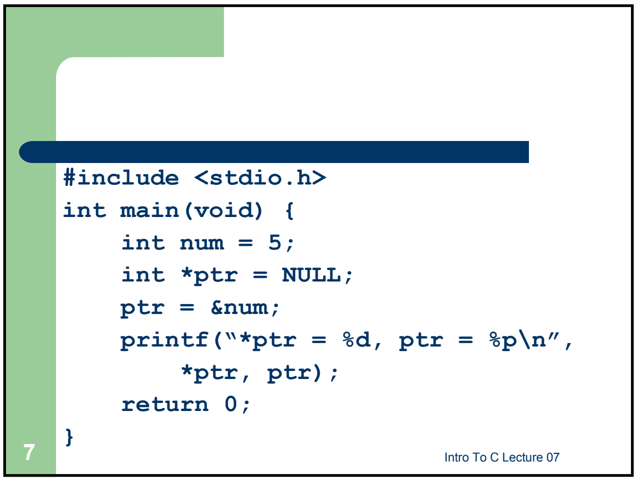


## Pointers

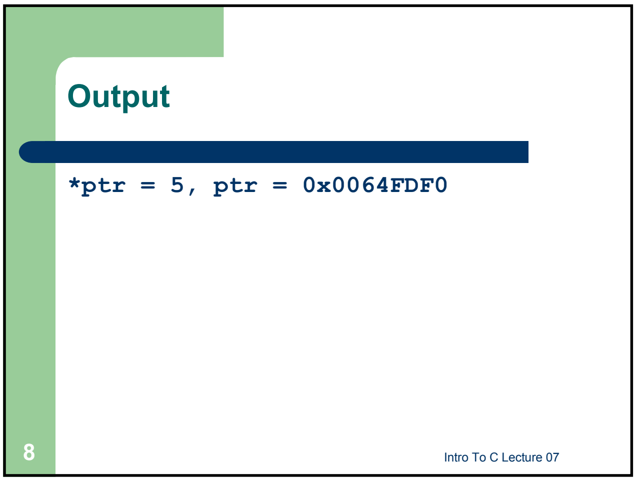
- A pointer variable is declared this way:

```
        type* variable_name;
        int* intptr;
```

- The variable **intptr** is defined as a “pointer to an int”. Which int? Any int. It appears in C expressions as “**intptr**” or as “***intptr**”.
- **intptr** is evaluated as the address of the int it points to; ***intptr** is evaluated as the value of the int being pointer to

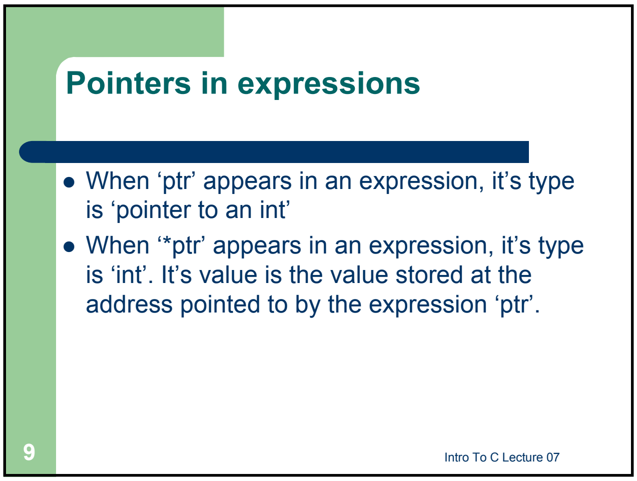


```
#include <stdio.h>
int main(void) {
    int num = 5;
    int *ptr = NULL;
    ptr = &num;
    printf(“*ptr = %d, ptr = %p\n”,
        *ptr, ptr);
    return 0;
}
```



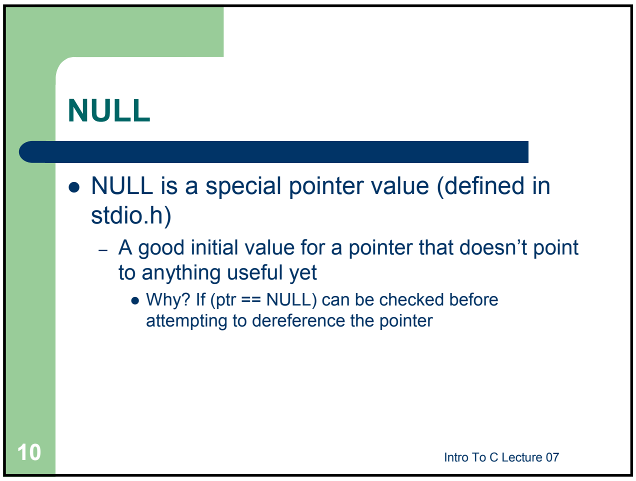
## Output

```
*ptr = 5, ptr = 0x0064FDF0
```



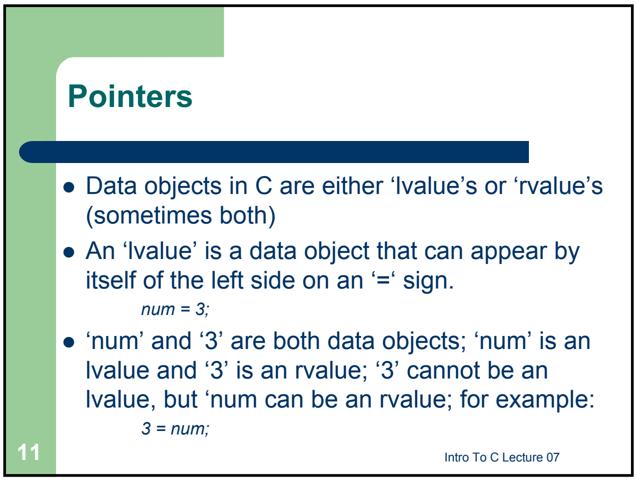
## Pointers in expressions

- When ‘ptr’ appears in an expression, it’s type is ‘pointer to an int’
- When ‘*ptr’ appears in an expression, it’s type is ‘int’. It’s value is the value stored at the address pointed to by the expression ‘ptr’.



## NULL

- NULL is a special pointer value (defined in stdio.h)
  - A good initial value for a pointer that doesn’t point to anything useful yet
    - Why? If (ptr == NULL) can be checked before attempting to dereference the pointer

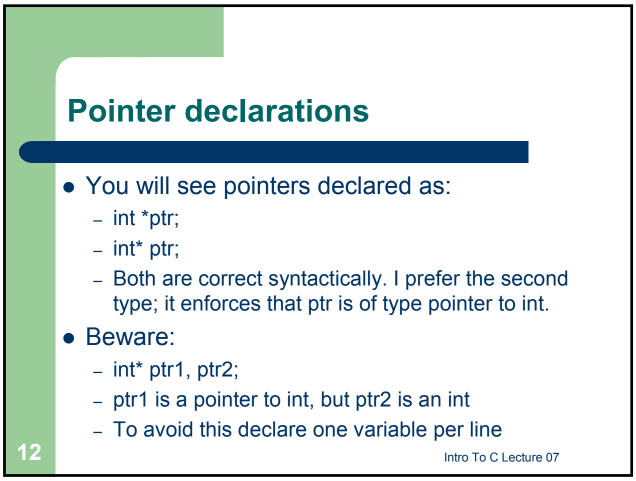


## Pointers

- Data objects in C are either ‘lvalue’s or ‘rvalue’s (sometimes both)
- An ‘lvalue’ is a data object that can appear by itself of the left side on an ‘=‘ sign.

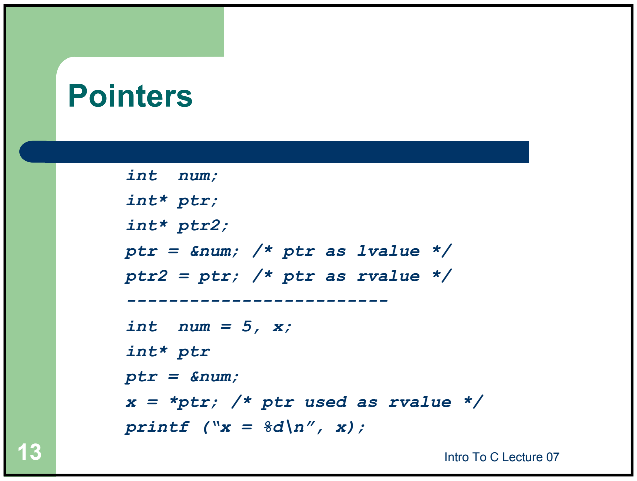
  *num = 3;*
- ‘num’ and ‘3’ are both data objects; ‘num’ is an lvalue and ‘3’ is an rvalue; ‘3’ cannot be an lvalue, but ‘num can be an rvalue; for example:

  *3 = num;*



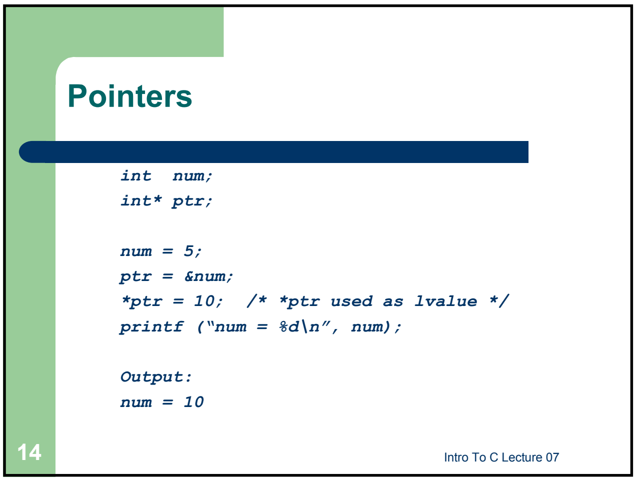
## Pointer declarations

- You will see pointers declared as:
  - int *ptr;
  - int* ptr;
  - Both are correct syntactically. I prefer the second type; it enforces that ptr is of type pointer to int.
- Beware:
  - int* ptr1, ptr2;
  - ptr1 is a pointer to int, but ptr2 is an int
  - To avoid this declare one variable per line



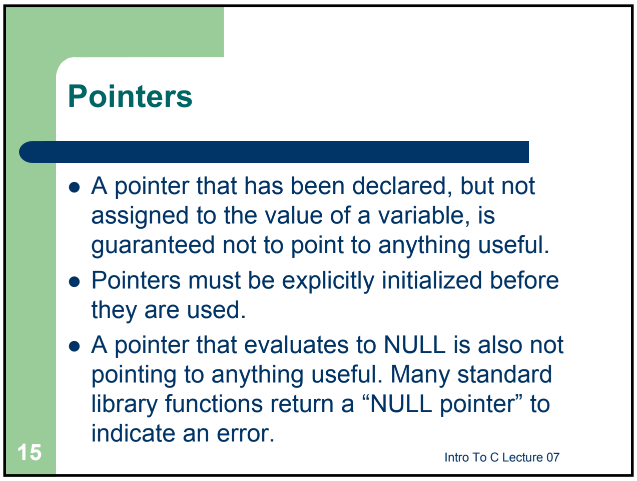
## Pointers

```
int  num;
int* ptr;
int* ptr2;
ptr = &num; /* ptr as lvalue */
ptr2 = ptr; /* ptr as rvalue */
-------------------------
int  num = 5, x;
int* ptr
ptr = &num;
x = *ptr; /* ptr used as rvalue */
printf (“x = %d\n”, x);
```



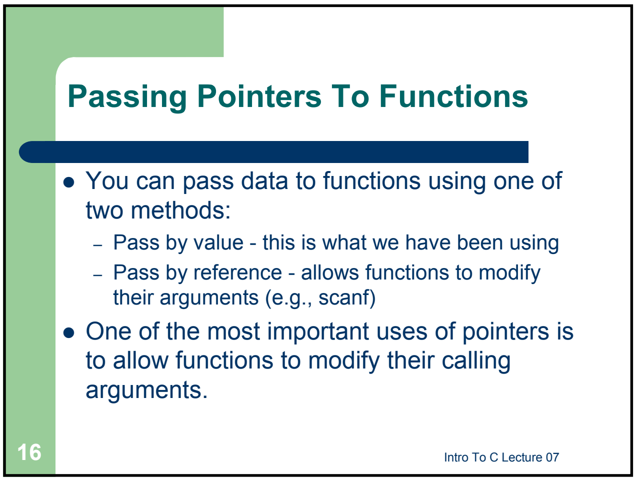
## Pointers

```
int  num;
int* ptr;

num = 5;
ptr = &num;
*ptr = 10;  /* *ptr used as lvalue */
printf (“num = %d\n”, num);

Output:
num = 10
```



## Pointers

- A pointer that has been declared, but not assigned to the value of a variable, is guaranteed not to point to anything useful.
- Pointers must be explicitly initialized before they are used.
- A pointer that evaluates to NULL is also not pointing to anything useful. Many standard library functions return a “NULL pointer” to indicate an error.



## Passing Pointers To Functions

- You can pass data to functions using one of two methods:
  - Pass by value - this is what we have been using
  - Pass by reference - allows functions to modify their arguments (e.g., scanf)
- One of the most important uses of pointers is to allow functions to modify their calling arguments.

## Passing by reference

- Consider a common ‘swap’ of two variables. This is easy when done inside a function where the variables are declared: -----------&gt;

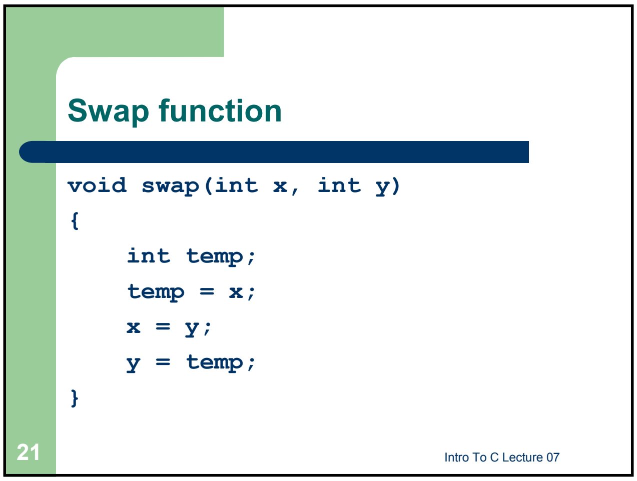
```
int main() {
    int x = 5, y = 10, temp;
    /* print original values */
    printf ("x = %d, y = %d\n", x,
  y);
    temp = x; /* swap values */
    x = y;
    y = temp;
    /* print values again */
```

## Pointers

- But what happens when swap() is a function?

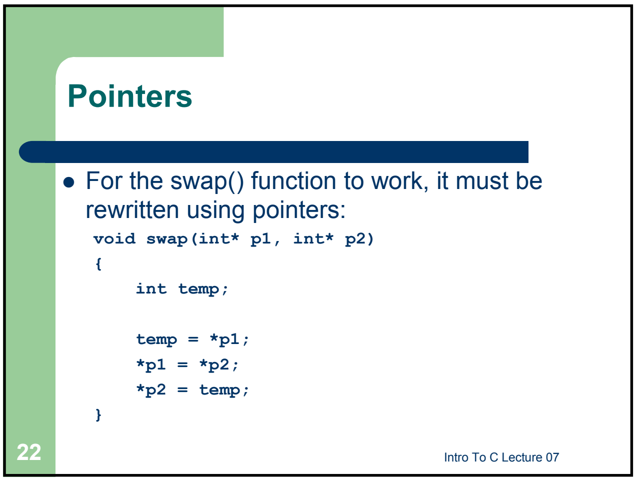
```
void swap(int x, int y)
{
    int    temp;

    temp = x;
    x = y;
    y = temp;
}
```

```
int swap(int, int);
int main()
{
    int x = 5, y = 10;
    printf ("x = %d, y = %d\n", x, y);
    swap(x, y);
    printf ("x = %d, y = %d\n", x, y);
}
```



## Swap function

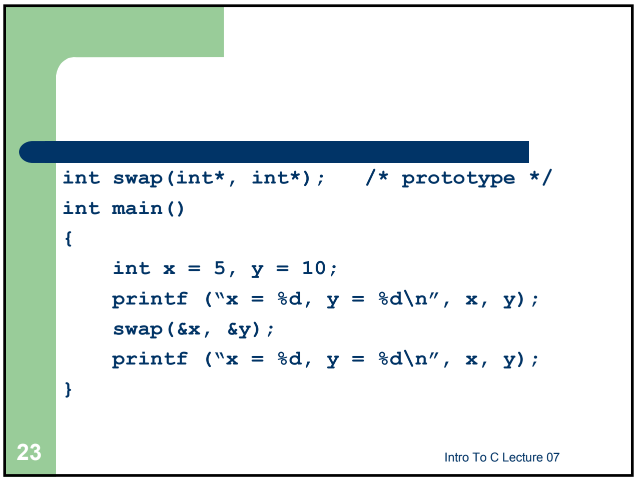
```
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```



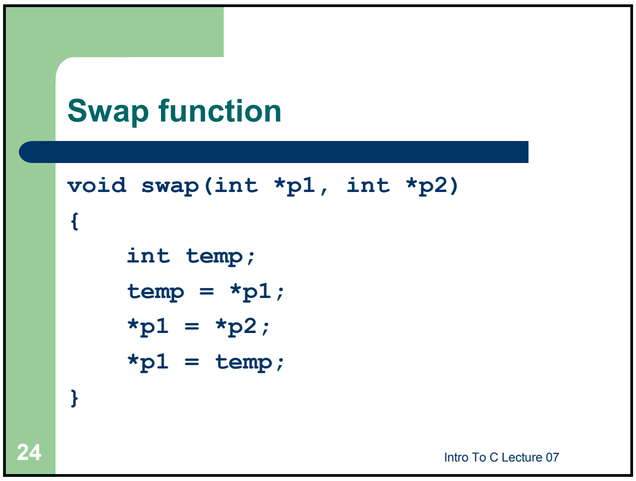
## Pointers

- For the swap() function to work, it must be rewritten using pointers:

```
void swap(int* p1, int* p2)
{
    int temp;

    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

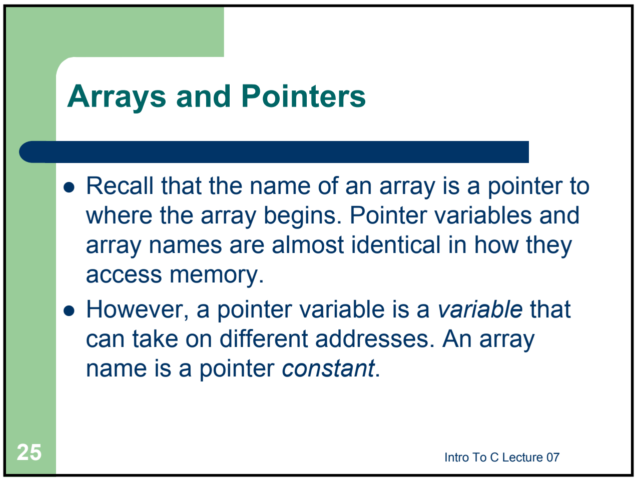


```
int swap(int*, int*);   /* prototype */
int main()
{
    int x = 5, y = 10;
    printf (“x = %d, y = %d\n”, x, y);
    swap(&x, &y);
    printf (“x = %d, y = %d\n”, x, y);
}
```



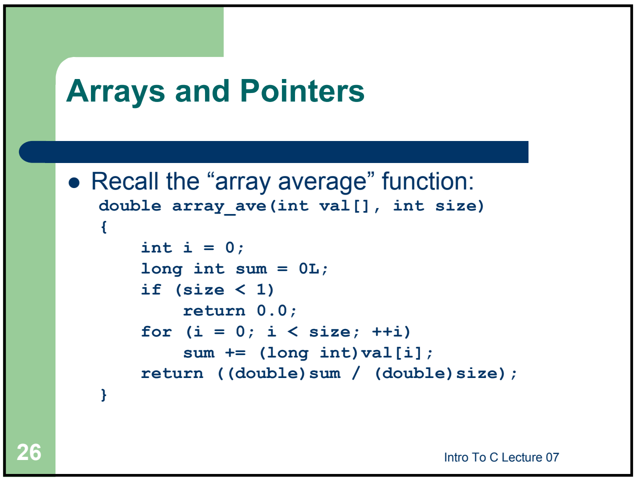
## Swap function

```
void swap(int *p1, int *p2)
{
    int temp;
    temp = *p1;
    *p1 = *p2;
    *p1 = temp;
}
```



## Arrays and Pointers

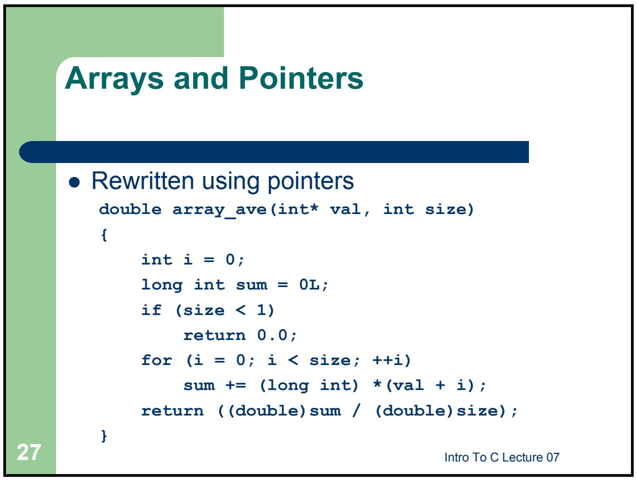
- Recall that the name of an array is a pointer to where the array begins. Pointer variables and array names are almost identical in how they access memory.
- However, a pointer variable is a *variable* that can take on different addresses. An array name is a pointer *constant*.



## Arrays and Pointers

- Recall the “array average” function:

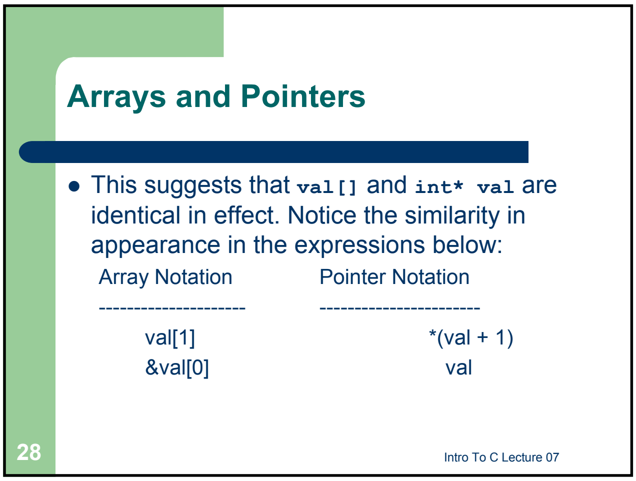
```
double array_ave(int val[], int size)
{
    int i = 0;
    long int sum = 0L;
    if (size < 1)
        return 0.0;
    for (i = 0; i < size; ++i)
        sum += (long int)val[i];
    return ((double)sum / (double)size);
}
```



## Arrays and Pointers

- Rewritten using pointers

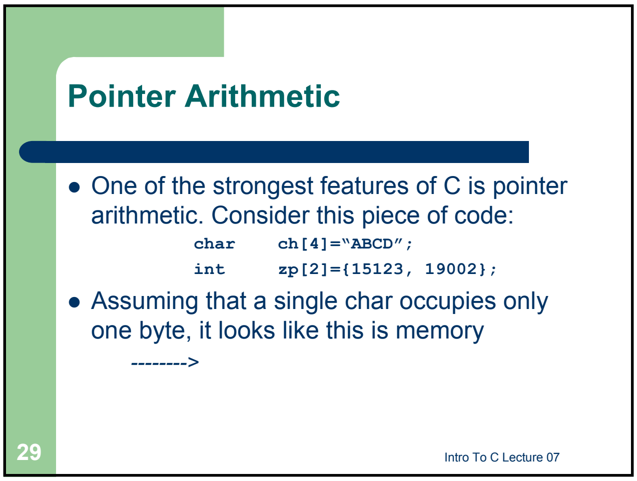
```
double array_ave(int* val, int size)
{
    int i = 0;
    long int sum = 0L;
    if (size < 1)
        return 0.0;
    for (i = 0; i < size; ++i)
        sum += (long int) *(val + i);
    return ((double)sum / (double)size);
}
```



## Arrays and Pointers

- This suggests that `val[]` and `int* val` are identical in effect. Notice the similarity in appearance in the expressions below:

| Array Notation | Pointer Notation |
|---|---|
| val[1] | *(val + 1) |
| &val[0] | val |



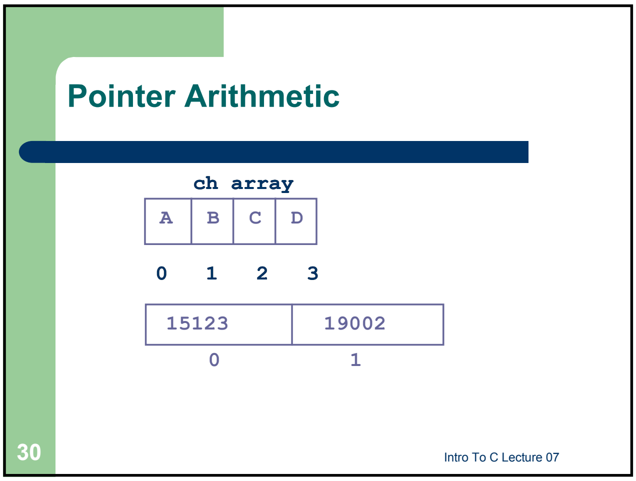
## Pointer Arithmetic

- One of the strongest features of C is pointer arithmetic. Consider this piece of code:

```
char    ch[4]=“ABCD”;
int     zp[2]={15123, 19002};
```

- Assuming that a single char occupies only one byte, it looks like this is memory

  -------->



## Pointer Arithmetic

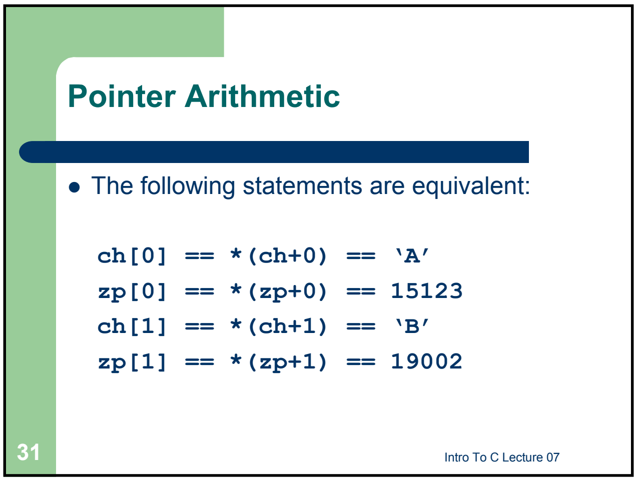
**ch array**

A B C D

0 1 2 3

15123 19002

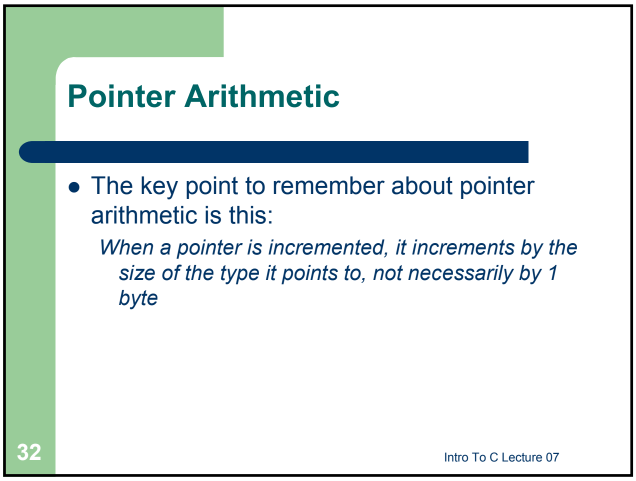
0 1



## Pointer Arithmetic

- The following statements are equivalent:

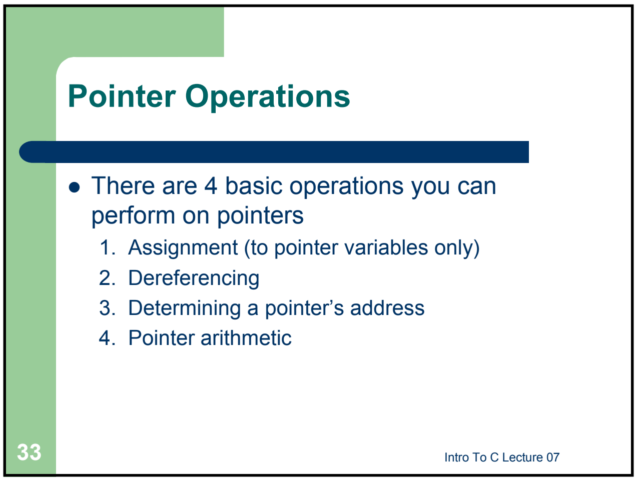
```
ch[0] == *(ch+0) == ‘A’
zp[0] == *(zp+0) == 15123
ch[1] == *(ch+1) == ‘B’
zp[1] == *(zp+1) == 19002
```



## Pointer Arithmetic

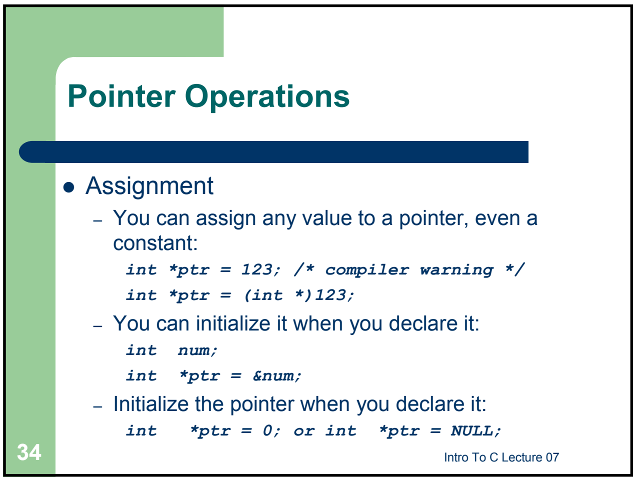
- The key point to remember about pointer arithmetic is this:

  *When a pointer is incremented, it increments by the size of the type it points to, not necessarily by 1 byte*



## Pointer Operations

- There are 4 basic operations you can perform on pointers
  1. Assignment (to pointer variables only)
  2. Dereferencing
  3. Determining a pointer’s address
  4. Pointer arithmetic



## Pointer Operations

- Assignment
  - You can assign any value to a pointer, even a constant:

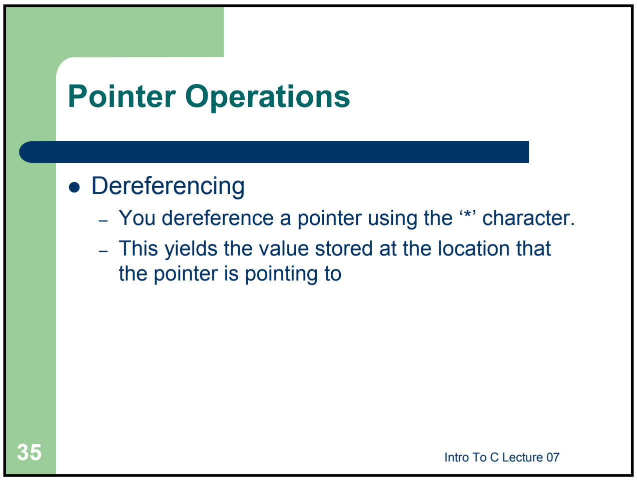
```
int *ptr = 123; /* compiler warning */
int *ptr = (int *)123;
```

  - You can initialize it when you declare it:

```
int  num;
int  *ptr = &num;
```

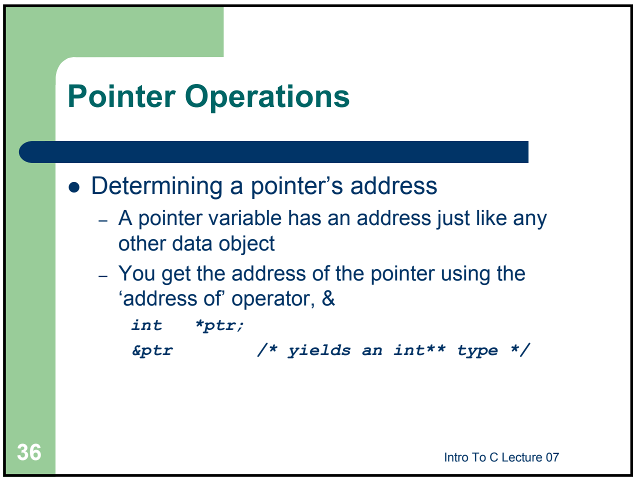
  - Initialize the pointer when you declare it:

```
int   *ptr = 0; or int  *ptr = NULL;
```



## Pointer Operations

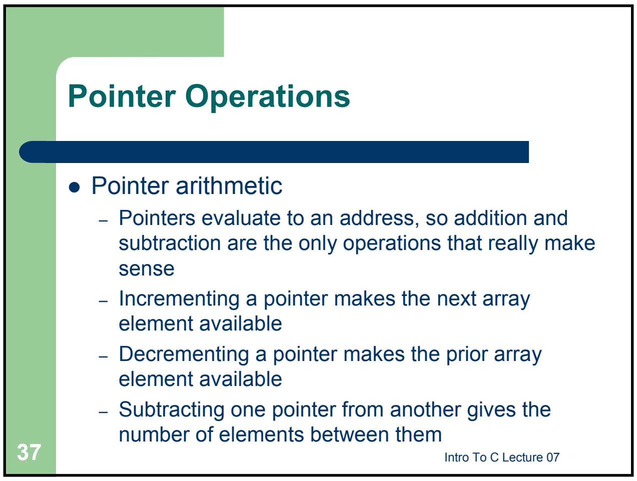
- Dereferencing
  - You dereference a pointer using the ‘*’ character.
  - This yields the value stored at the location that the pointer is pointing to



## Pointer Operations

- Determining a pointer’s address
  - A pointer variable has an address just like any other data object
  - You get the address of the pointer using the ‘address of’ operator, &

```
int   *ptr;
&ptr        /* yields an int** type */
```



## Pointer Operations

- Pointer arithmetic
  - Pointers evaluate to an address, so addition and subtraction are the only operations that really make sense
  - Incrementing a pointer makes the next array element available
  - Decrementing a pointer makes the prior array element available
  - Subtracting one pointer from another gives the number of elements between them

37

Intro To C Lecture 07