LECTURE 34

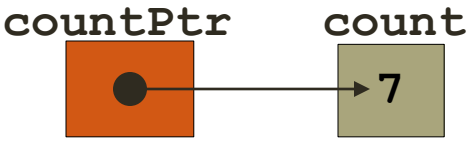
POINTERS

Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)

Pointers contain address of a variable that has a specific value (indirect reference)

- Indirection – referencing a pointer value



Declaring pointer variables

In C every variable must be declared by its type. Since pointer variables contain addresses that belong to a separate type, they must be declared as pointers before we use them.

The declaration of a pointer variable takes the following form :

data_type *pt_name;

This tells the compiler three things about the variable pt_name :

1. **The asterisk (*) tells that the variable pt_name is a pointer variable.**
2. **pt_name needs a memory location.**
3. **pt_name points to a variable of type data type.**

Pointer Variable Declarations and Initialization

- Pointer declarations
  - **`*`** used with pointer variables

    ```
    int *myPtr;
    ```

  - Declares a pointer to an **`int`** (pointer of type **`int *`**)
  - Multiple pointers require using a **`*`** before each variable declaration

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

  - Can declare pointers to any data type
  - Initialize pointers to **`0`**, **`NULL`**, or an address
    - **`0`** or **`NULL`** – points to nothing (**`NULL`** preferred)

Pointers are flexible. We can make the same pointer to point to different data variables in different statements.

We can also use different pointers to point to the same data variable.

Accessing variable through its pointer

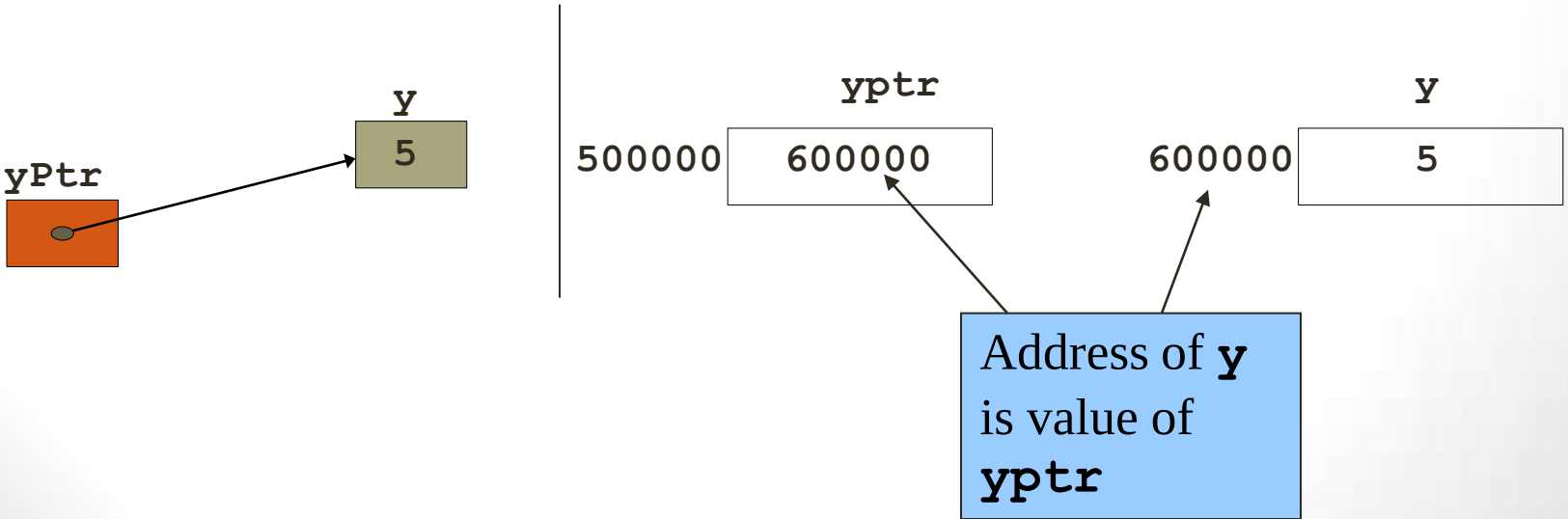
Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer?

This is done by using a unary operator asterisk (*), usually known as the indirection operator, another name for this operator is the dereferencing operator or it can be remembered as value at address.

Pointer Operators

- **&** (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;      // yPtr gets address of y
yPtr "points to" y
```



Pointer Operators

- **`*`** (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - **`*yptr`** returns **`y`** (because **`yptr`** points to **`y`**)
  - **`*`** can be used for assignment
    - Returns alias to an object

      ```
      *yptr = 7;  // changes y to 7
      ```
  - Dereferenced pointer (operand of **`*`**) must be an lvalue (no constants)
- **`*`** and **`&`** are inverses
  - They cancel each other out

LECTURE 35

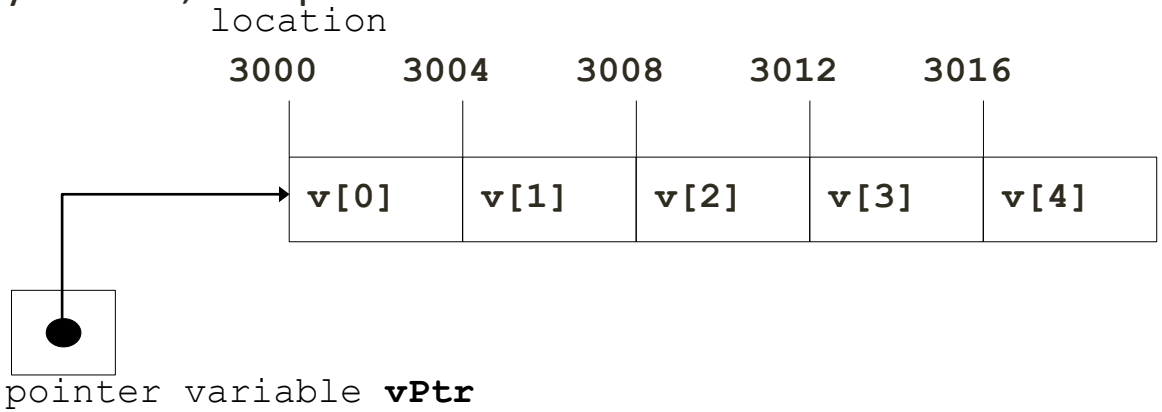
POINTER EXPRESSIONS, INCREMENT AND SCALE FACTOR

Pointer Expressions and Pointer Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (++ or −−)
  - Add an integer to a pointer(+ or += , − or −=)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array

Pointer Expressions and Pointer Arithmetic

- 5 element **int** array on machine with 4 byte **int**s
  - **vPtr** points to first element **v[0]**
    - at location **3000** (**vPtr = 3000**)
  - **vPtr += 2;** sets **vPtr** to **3008**
    - **vPtr** points to **v[2]** (incremented by 2), but the machine has 4 byte **int**s, so it points to address **3008**



Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns number of elements from one to the other. If

```
vPtr2 = v[ 2 ];
vPtr = v[ 0 ];
```

  - **vPtr2 - vPtr** would produce 2
- Pointer comparison (**<**, **==** , **>**)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to **0**

Pointer Expressions and Pointer Arithmetic

- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used
  - Exception: pointer to **`void`** (type **`void *`**)
    - Generic pointer, represents any type
    - No casting needed to convert a pointer to **`void`** pointer
    - **`void`** pointers cannot be dereferenced

LECTURE 36

POINTERS AND ARRAYS

The Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Declare an array **`b[ 5 ]`** and a pointer **`bPtr`**
  - To set them equal to one another use:

    ```
    bPtr = b;
    ```

    - The array name (**`b`**) is actually the address of first element of the array **`b[ 5 ]`**

      ```
      bPtr = &b[ 0 ]
      ```

    - Explicitly assigns **`bPtr`** to address of first element of **`b`**

The Relationship Between Pointers and Arrays

- Element `b[ 3 ]`
  - Can be accessed by `*( bPtr + 3 )`
    - Where `n` is the offset. Called pointer/offset notation
  - Can be accessed by `bptr[ 3 ]`
    - Called pointer/subscript notation
    - `bPtr[ 3 ]` same as `b[ 3 ]`
  - Can be accessed by performing pointer arithmetic on the array itself
    `*( b + 3 )`

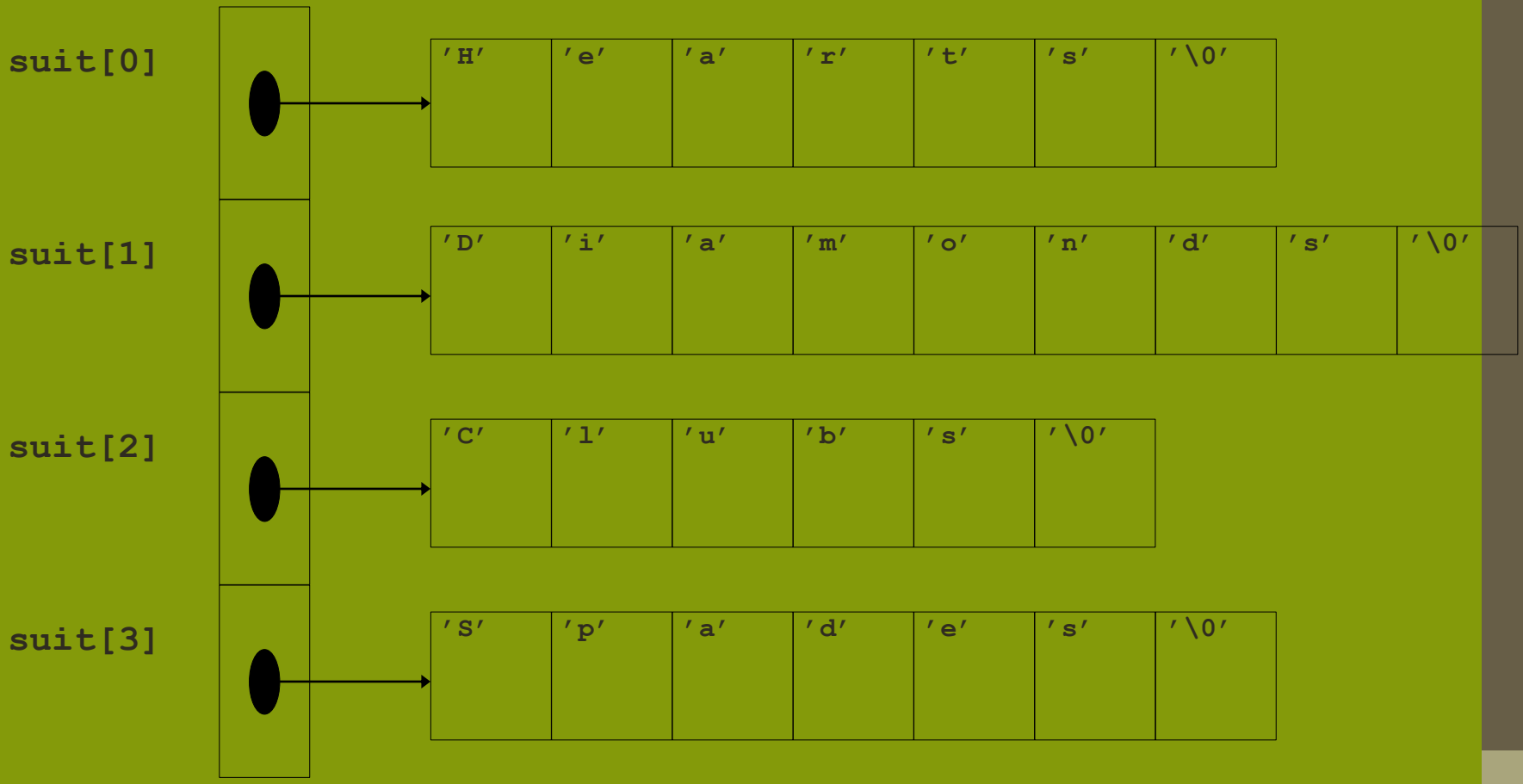
Arrays of Pointers

- Arrays can contain pointers
- For example: an array of strings

```
char *suit[ 4 ] = { "Hearts", "Diamonds",
   "Clubs", "Spades" };
```

  - Strings are pointers to the first character
  - **`char *`** – each element of **`suit`** is a pointer to a **`char`**
  - The strings are not actually stored in the array **`suit`**, only pointers to the strings are stored

  - **`suit`** array has a fixed size, but strings can be of any size



LECTURE 37

POINTERS AND CHARACTER STRINGS

POINTERS AND CHARACTER STRINGS

We can use pointer to access the individual characters in a string.

```
char str[5]="good";

char *str="good";

printf("%s",str);

puts(str);
```

```
static char *name[3]={
                        "abc" ,  "xyz" ,   "def"
                      }
```

Declares name to be an array of three pointers to characters, each pointer pointing to a particular name

name[0] -> abc

name[0] ->xyz

name[0] ->def

To access j^{th} character of i^{th} name

***(name[i]+j)**

The character arrays with the rows of varying length are called ragged arrays and are better handled by pointers.

LECTURE 38

POINTER AND FUNCTION

Pointer and Functions

Let us now examine the close relationship between pointers and C's other major parts. We will start with functions.

When C passes arguments to functions it passes them by value.

There are many cases when we may want to alter a passed argument in the function and receive the new value back once to function has finished. C uses pointers explicitly to do this. Other languages mask the fact that pointers also underpin the implementation of this.

The best way to study this is to look at an example where we must be able to receive changed parameters.

Let us try and write a function to swap variables around?

The usual function *call*:

swap(a, b) WON'T WORK.

Pointers provide the solution: *Pass the address of the variables to the functions and access address of function.*

Thus our function call in our program would look like this:

```
swap(&a, &b)
```

The Code to swap is fairly straightforward:

```
void swap(int *px, int *py)
{
        int temp;
        temp = *px;     /* contents of pointer */
       *px = *py;
       *py = temp;
}
```

We can return pointer from functions. A common example is when passing back structures. *e.g.*:

```
  typedef struct
{
   float x,y,z;
} COORD;
  main()
 {
    COORD p1, *coord_fn();/* declare fn to return ptr of COORD typ
*/
     ...
   p1 = *coord_fn(...);/* assign contents of address returned */
   ....
 }
COORD *coord_fn(...)
{  COORD p;
.....  p = ....;   /* assign structure values */
 return &p; /* return address of p */}
```

Here we return a pointer whose contents are immediately *unwrapped* into a variable. We must do this straight away as the variable we pointed to was local to a function that has now finished. This means that the address space is free and can be overwritten. It will not have been overwritten straight after the function has quit though so this is perfectly safe.

LECTURE 39

POINTERS AND STRUCTURES

Pointers and structures

```
struct allowance
                {
                   int basic_pay’
                    int da;
                    int hra;
                    int city_allowance;
                 };

struct employee
            {
                char name[10];
                char department[10];
                struct allowance a1;
          }*ptr;
```

ptr->name, ptr->department, ptr->a1.basic_pay, ptr->a1.da,

ptr->a1.hra, ptr->a1.city_allowance

```
struct employee
{
  int eid;
 char name[10];
 char dept[10];
}emp[5],*ptr;
ptr=emp;
ptr->eid
ptr-->name
ptr->dept
for(ptr=emp;ptr<emp+5;ptr++)
printf(“%d%s%s”,ptr->eid,ptr->name,ptr->dept);
```

We could also use the notation :

(*ptr).eid

(*ptr).name

(*ptr).dept

While using structure pointers, we should take care of the precedence of operators

The operators ->, . , ( ), [ ] enjoy the highest priority among the operators.

```
struct
{
   int count;
    float *p;
  } *ptr;
  ++ptr->count;
```

Increments the count not ptr, however

```
(++ptr)->count;
```

First increments the ptr then links count.

```
struct employee
{
    int eid;
    char name[10];
    char dept[10];
 } emp,*ptr;
```

Now the members can be accessed in three ways :

1. Using dot operator : emp.eid, emp.name, emp.dept
2. Using indirection : (*ptr).eid, (*ptr).name, (*ptr).dept
3. Using selection notation : ptr->eid, ptr->name, ptr->dept

Arrays vs structures

Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner.

But they differ in a number of ways :

1. An array is a collection of related data elements of same type. Structures can have elements of different types.

2. An array is derived data type whereas a structure is a programmer defined one.

3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in case of a structure, first we have to design and declare a data structure before the variable of that type are declared and used.