POINTERS

- A ***pointer*** is a variable that contains the memory address of another variable.
- An ***address*** is the location where the variable exists in memory.
- A variable name directly references a value in a memory location, a pointer indirectly references it.
- Pointers must be declared before they can be used. The ***** operator, named as the *indirection* or *deferencing operator*, is used in front of a variable to declare it as a pointer.

Examples:

```
int  *countptr;
char *p;
```

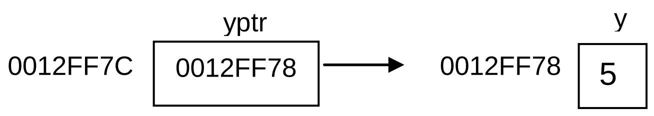
`p` is not initialized yet. Its value is whatever existed in the space reserved for `p`.

- There are three values that can be used to initialize a pointer: **0**, **NULL**, or an address.
- To initialize a pointer to an address, the **&** operator, named as the *address of operator*, is used. It returns the address of its operand. Its operand must be a variable; the address operator can not be applied to constants or expressions.

Example:

```
int *yptr;
int y = 5;
yptr = &y;
```

- In the above example, the pointer variable `yptr` is initialized to the address of the variable `y`. It is read "`yptr` points to `y`". We can show it as in the following figure:



- To display the value of `y`, we can use one of the following `printf` statements:

```
printf("%d\n", y);
printf("%d\n", *yptr);
```

- We can display the value of a pointer variable, i.e. an address value as a hexadecimal integer using **%p** placeholder in a `printf` statement.

```
printf("%p\n", &y);
printf("%p\n", yptr);
```

- If we assume that the address of the variable `y` is **0012FF78** (as a hexadecimal integer), both of the above statements will display that value. What about

  ```
  printf("%p\n", &yptr);
  ```

  It displays the address of the pointer variable `yptr`, i.e., **0012FF7C**, in our example**.**

➢ *READ Sec. 7.1, 7.2 and 7.3 (pg 250 – 254) from Deitel & Deitel.*

Pointer Expressions and Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions, assignment expressions, and comparison expressions. However, not all of the operators normally used in these expressions are valid in conjunction with pointer variables.

  1. A pointer can be assigned an address of an ordinary variable of the indicated type.

  ```
  yptr = &y;
  ```

  2. A pointer can be assigned the value of another pointer with the same data type.

  ```
  int *xptr;
  xptr = yptr;
  ```

  3. A pointer can be assigned to NULL or 0 (NULL is a pointer pointing to nowhere!).

  ```
  xptr = NULL;
  ```

  4. A pointer may be incremented or decremented.

  ```
  yptr++;   // yptr will contain  0012FF7C
  yptr--;   // yptr will contain  0012FF74
  ```

- The pointer arithmetic is not the same as normal arithmetic. Since `yptr` was **0012FF78** before the increment, we wait it to become **0012FF79** after the increment, or **0012FF77** after the decrement. However, it becomes **0012FF7C** after the increment, or **0012FF74** after the decrement.

- When we increment or decrement a pointer variable, it increases by the size of the object to which that pointer refers. Thus, the number of bytes depend on the object's data type. Since `yptr` points to an integer, and an integer is stored in 4 bytes of memory, 4 will be added to or subtracted from its value.

- If `yptr` was pointing to a character, then its value would increase or decrease by 1, thus the results would be consistent with regular arithmetic, because each character is one byte long.

5. An integer quantity can be added or subtracted from a pointer

```
yptr += 3;     // yptr will contain  0012FF84
yptr -= 2;     // yptr will contain  0012FF70
```

- When an integer is added to or subtracted from a pointer, the pointer is not simply incremented or decremented by that integer (`yptr` does not become **0012FF7B** or **0012FF76**), but by that integer times the size of the object to which that pointer refers.

  0012FF78 + 3 * 4 = 0012FF84
  0012FF78 - 2 * 4 = 0012FF70

6. One pointer variable can be subtracted from another pointer provided that both point to the objects of the same data type.

```
y = yptr - xptr;
```

The result will be equal to the difference of the two addresses divided to 4.

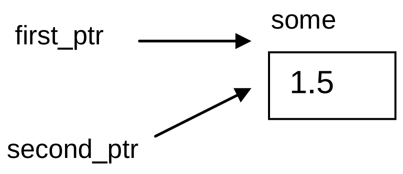
7. No other arithmetic operation is allowed. You cannot multiply a pointer with a constant. Two pointer variables cannot be added.

8. Two pointers may be compared provided that both point to the objects of the same data type. Pointer comparisons compare addresses stored in the pointers. A common usage of pointer comparison is determining whether a pointer is NULL.

```
yptr < xptr
yptr == xptr
yptr != xptr
yptr == NULL
```

Example:

```
double some;
double *first_ptr;
double *second_ptr;
some = 1.5;
first_ptr = &some;
second_ptr = first_ptr;
printf("%p %p\n", first_ptr, second_ptr);
```



- Both pointers will point to the variable `some`, and the `printf` statement will display the same address twice.

```
first_ptr++;
second_ptr -= 2;
printf("%d\n", first_ptr - second_ptr);
```

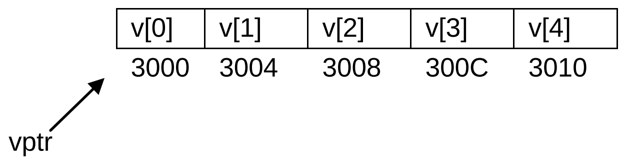
- Output will be 3 because `first_ptr` is incremented by 8, `second_ptr` is decremented by 16. Their difference is (16+8)/8 = 3.

```
if (first_ptr == second_ptr)
     printf("They are equal\n");
else if (first_ptr > second_ptr)
     printf("First is later\n");
else
     printf("Second is later\n");
```

  Output will be "`First is later`".

- Pointer arithmetic operations should only be performed on contiguous portions of memory such as an array because as you know all elements of an array are stored in subsequent locations of the memory.

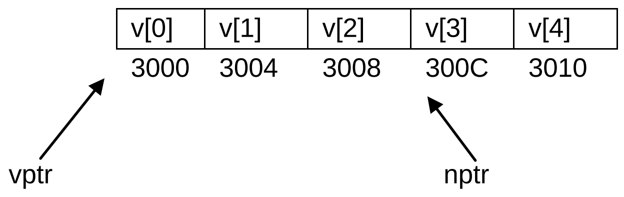
Example:



```
vptr += 2;
```

- `vptr` will be **3008** pointing to `v[2]` if it is assumed that an integer is stored in 4 bytes of memory.

- Normally, one pointer variable should only be subtracted from another pointer if both pointers point to the same array.



```
x = nptr - vptr
```

  is meaningful and `x` will become **3** which is equal to the number of array elements from `vptr` to `nptr`.

- Pointer comparisons are normally meaningful only if the pointers point to members of the same array. A comparison of two pointers pointing to the same array can show, for example, that one pointer points to a higher-indexed element of the array than the other pointer does.

```
nptr > vptr    // true
```

➢ *READ Sec. 7.7 (pg 269 – 271) from Deitel & Deitel.*

Pointers as Function Parameters

- As you have learned in 151, pointers can be used in the formal parameter list of a function, if we want the function to modify the value of the actual parameter in the calling function. In such a case, thus, when calling a function with an argument that the caller wants the called function to modify, the address of the argument is passed.

Example: Write a function that interchanges the values of its two parameters **x** and **y**.

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

- What is the output of the following main?

```
int main (void)
{    int first = 5;
     int second = 7;
     printf("first = %d second = %d\n", first, second);
     swap(&first, &second);
     printf("first = %d second = %d\n", first, second);
     return(0);
}
```

Output:

➢ *READ Sec. 7.4 (pg 254 – 258) from Deitel & Deitel.*

Home Exercise: Write a function that returns the digits of a three-digit number.