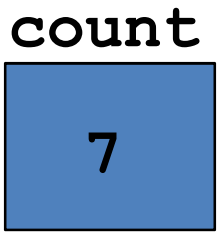
Pointers

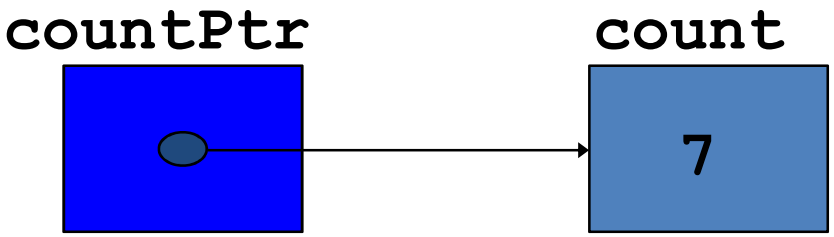
Introduction

- Powerful, but difficult to master
- Simulate call-by-reference
- Close relationship with arrays and strings
- every variable has a memory location and every memory location has its address defined which can be accessed using **ampersand (&)** operator, which denotes an address in memory.
- A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address

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)





Cont..

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

    ```
    int *myPtr;
    ```
  - Declares a pointer to an **`int`** (pointer of type **`int *`**)
  - Multiple pointers, multiple **`*`**

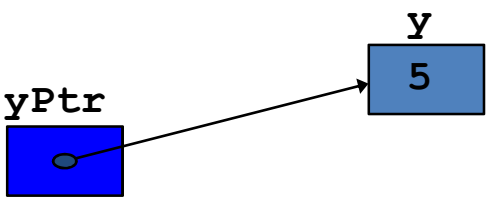
    ```
    int *myPtr1, *myPtr2;
    ```
  - Can declare pointers to any data type
  - Initialize pointers to **0**, **`NULL`**, or an address
    - **0** or **`NULL`** - points to nothing (**`NULL`** preferred)

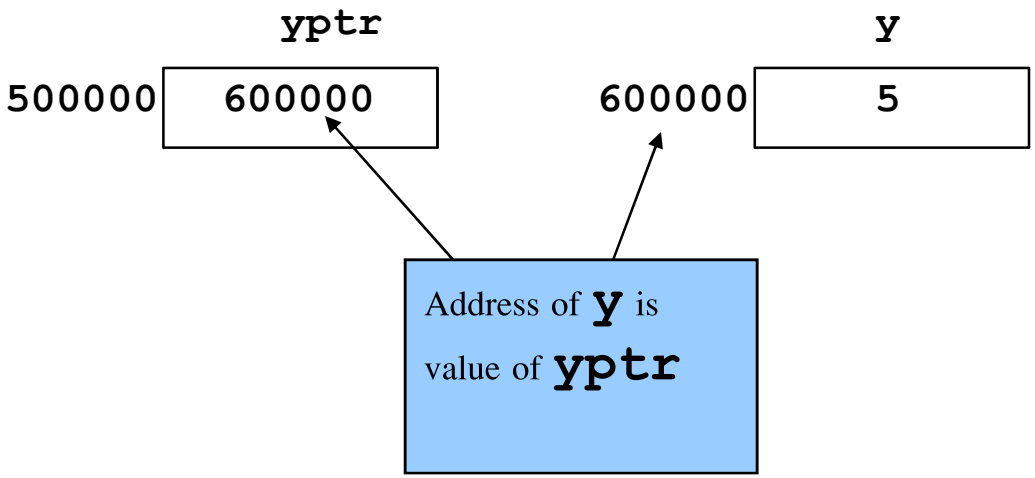
Pointer Operators

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

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

  - **yPtr** "points to" **y**





Cont..

- **`*`** (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)

Cont..

- ***** and **&** are inverses
  - They cancel each other out

```
*&yptr  ->  * (&yptr) -> * (address of yptr)->
  returns alias of what operand points to -> yptr

&*yptr -> &(*yptr) -> &(y) -> returns address of y,
  which is yptr -> yptr
```

- The general form of a pointer variable declaration is:

type *var-name;

- Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk (*) used to declare a pointer is the same asterisk that is used for multiplication. Following are the valid pointer declaration:

```
int  *ip;          /* pointer to an integer */
double *dp;        /* pointer to a double */
float  *fp;        /* pointer to a float */
char  *ch ;        /* pointer to a character */
```

Operations used in pointers

Following example makes use of these operations:

```
#include <stdio.h>

int main ()

{

int  var = 20;                          /* actual variable declaration */

int  *ip;                               /* pointer variable declaration */

ip = &var;                              /* store address of var in pointer variable*/

printf("Address of var variable: %u\n", &var  );       /* address stored in pointer variable */

printf("Address stored in ip variable: %u\n", ip );     /* access the value using the pointer */

printf("Value of *ip variable: %d\n", *ip );

return 0;

}
```

Cont..

- When the above code is compiled and executed, it produces result something as follows:

  Address of var variable: bffd8b3c

  Address stored in ip variable: bffd8b3c

  Value of *ip variable: 20

NULL Pointers in C

- It is always a good practice to assign a **NULL** value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned **NULL** is called a **null** pointer.

- The **NULL** pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>

int main ()

{

int  *ptr = NULL;

printf("The value of ptr is : %d\n", &ptr  );

return 0;

}
```

Cont..

- When the above code is compiled and executed, it produces the following result:

The value of ptr is 0

- On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system.

- However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

Pointer arithmetic

- C pointer is an address which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can have a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and –

- To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

ptr++

- Now, after the above operation, the ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to next memory location without impacting actual value at the memory location.

- If ptr points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

Incrementing a Pointer

- We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer.
- The following program increments the variable pointer to access each succeeding element of the array:

```
#include <stdio.h>

int MAX = 3;

int main ()

{

 int  var[] = {10, 100, 200};

 int  i, *ptr;                    /* let us have array address in pointer */

 ptr = &var;
```

Cont..

```
for ( i = 0; i < MAX; i++)
{
printf("Address of var[%d] = %u\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );                    /* move to the next location */
ptr++;
}
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer

- The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

```
#include <stdio.h>

int MAX = 3;

int main ()

{

int  var[] = {10, 100, 200};

int  i, *ptr;                          /* let us have array address in pointer */

ptr = &var;

for ( i = MAX; i > 0; i--)
```

Decrementing a Pointer Cont..

```
 {
printf("Address of var[%d] = %u\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );                    /* move to the previous location */
ptr--;
}
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[3] = bfedbcd8
Value of var[3] = 200
Address of var[2] = bfedbcd4
Value of var[2] = 100
Address of var[1] = bfedbcd0
Value of var[1] = 10
```

Pointer Comparisons

- Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.
- The following program modifies the previous example one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1]:

```
#include <stdio.h>

const int MAX = 3;

int main ()

{

   int  var[] = {10, 100, 200};

   int  i, *ptr;                    /* let us have address of the first element in pointer */

   ptr = var;

   i = 0;
```

Cont..

```
while ( ptr <= &var[MAX - 1] )
{
printf("Address of var[%d] = %u\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );          /* point to the previous location */
ptr++;
i++;
}
return 0;
}
```

- When the above code is compiled and executed, it produces result something as follows:

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

Array of pointers

- Before we understand the concept of arrays of pointers, let us consider the following example, which makes use of an array of 3 integers:

```
#include <stdio.h>

const int MAX = 3;

int main ()

{

int  var[] = {10, 100, 200};

int i;

for (i = 0; i < MAX; i++)

{

printf("Value of var[%d] = %d\n", i, var[i] );

}

return 0;

}
```

Cont..

- When the above code is compiled and executed, it produces the following result:

  Value of var[0] = 10

  Value of var[1] = 100

  Value of var[2] = 200

- There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:

  int *ptr[MAX];

- This declares ptr as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value.

Cont..

- Following example makes use of three integers, which will be stored in an array of pointers as follows:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
int  var[] = {10, 100, 200};
int i, *ptr[MAX];
for ( i = 0; i < MAX; i++)
{
ptr[i] = &var[i];                    /* assign the address of 10,100,200. */
}
for ( i = 0; i < MAX; i++)
{
printf("Value of var[%d] = %d\n", i, *ptr[i] );
}
return 0;
}
```

Cont..

- When the above code is compiled and executed, it produces the following result:

Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200

- You can also use an array of pointers to character to store a list of strings as follows:

```
#include <stdio.h>
const int MAX = 4;
int main ()
{
   char *names[] = {
                        “FCP SVNIT”,
                        “SVNIT Surat",
                        “Surat India",
                        “India Asia"
} ;
```

Array of pointers Cont..

```
int i = 0;

for ( i = 0; i < MAX; i++)

{

    printf("Value of names[%d] = %s\n", i, names[i] );

}

return 0;

}
```

- When the above code is compiled and executed, it produces the following result:

```
Value of names[0] = FCP SVNIT

Value of names[1] = SVNIT Surat

Value of names[2] = Surat India

Value of names[3] = India Asia
```

Passing pointers to functions/Call by reference

```
#include <stdio.h>
int main( )
{
int  v1 = 11, v2 = 77 ;
printf("Before swapping:");
printf("\nValue of v1 is: %d", v1);
printf("\nValue of v2 is: %d", v2);

swapnum( &v1, &v2 );

printf("\nAfter swapping:");

printf("\nValue of v1 is: %d", v1);

printf("\nValue of v2 is: %d", v2);

}
```

Cont..

```
void swapnum(int *num1, int *num2)
{
int tempnum;
tempnum = *num1;
*num1 = *num2;
*num2 = tempnum;
}
```

Output:

Before swapping:

Value of v1 is: 11

Value of v2 is: 77

After swapping:

Value of v1 is: 77

Value of v2 is: 11