ARRAYS, POINTERS

(PART 2)

prepared by Senem Kumova Metin
modified by İlker Korkmaz

POINTERS (1)

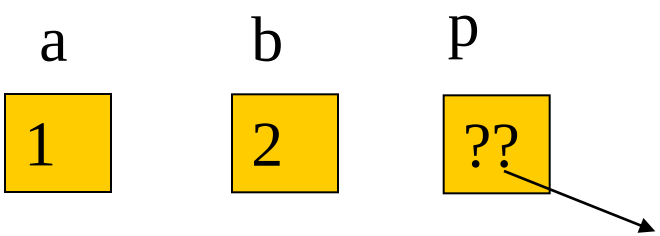
- int v = 5;
  - v is the identifier of an integer variable
  - 5 is the value of v
  - &v is the location or address of the v inside the memory
  - & means “ the address of ”

- Pointers are used in programs to access memory.

```
int v = 5;
int * p;  //  p is the identifier of a “pointer to an integer”
p=&v;  // p is assigned with the address of v
p=0;     // OR   p = NULL;
```

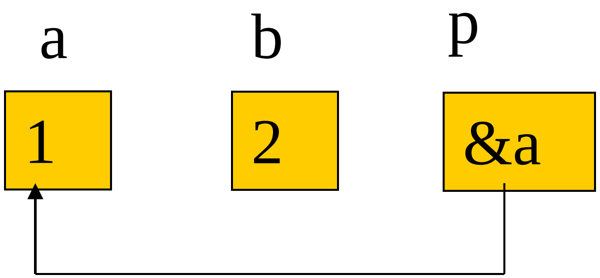
POINTERS (2)

int a=1, b=2, *p;



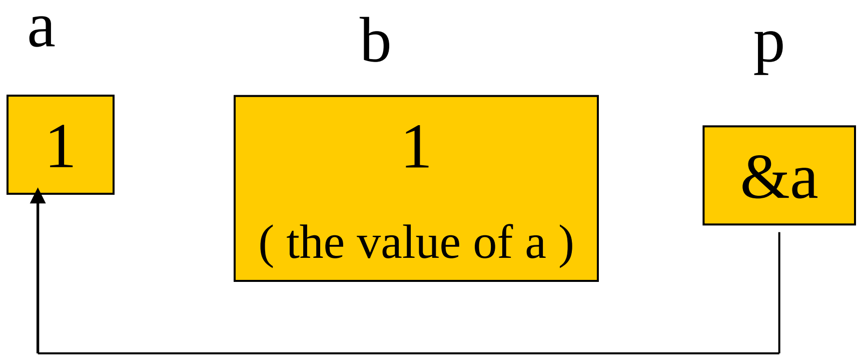
will point somewhere in memory

p=&a;



POINTERS (3)

b=*p;



// equivalent to b=a

Example: Pointers

EXAMPLE:

```
#include <stdio.h>
int main(void)
{
   int   i = 7, j , *k;

   k = &i;
   printf("%s%d\n%s%p\n", "   Value of i: ", *k, "Location of i: ",  k);
   j=*k;
   return 0;
}
```

CALL BY VALUE

```
/*  Whenever variables are passed as arguments to a function, their
    values are copied to the function parameters , and the variables
    themselves are not changed in the calling environment.*/
int main()
{   int a=20; int b=30;
    swap (a, b)
    printf(“%d  %d: “, a, b);
    return 0;
}
void swap(int x, int y)
{
    int tmp;
    tmp=x;
    x=y;
    y=tmp;
    return;
}
```

CALL BY REFERENCE

/* Whenever addresses of variables are passed as arguments to a function, their values shall be changed in the calling environment.*/

```
void main()
{   int a=20; int b=30;
    swap (&a, &b)
    printf(“%d  %d: “, a, b);
}

void swap(int *x, int *y)
{   int tmp;
    tmp = *x; // get value pointed by x.
    *x = *y; // assign value pointed by y to x
    *y = tmp;
    return;  }
```

Relationship between “pointers” and “arrays” (1)

/* The name of an array is the adress or the pointer to the first element of the array. */

```
int a[5] , *p;
p=a; // OR   p=&a[0];
```

Relationship between “pointers” and “arrays” (2)

```
&a[0] equals to a    then  a[0] equals to *a
&a[1] equals to a+1  then  a[1] equals to *(a+1)
&a[2] equals to a+2  then  a[2] equals to *(a+2)
……………………………………………………………………………………
&a[i] equals to a+i  then  a[i] equals to *(a+i)
```

- **EXAMPLE:**

```
int a [5]={1,2,3,4,5};
int *p;
printf(“%d”,a[0]); printf(“%d”,*a);
printf(“%d”,a[2]); printf(“%d”,*(a+2));
p=&a[4];
p=a+4;
```

Storage mapping

- the mapping b/w pointer values and array indices
- EXAMPLE:

```
int d [3];
d[i]  →   *(&d[0]+i)
```

- EXAMPLE:

```
int a [3] [4];
a[i][j]  →   *(&a[0][0]+4*i+j)
```

Arrays as Function Arguments

```
double sum(int [] , int);
main()
{     int x[9];
      double r;
      r=sum(x,9); //sum(&x[0],9)
}
double sum( int a[], int n)
{     int i;
      double result =0.0;
      for (i=0; i<n; i++)
            result=result+a[i];
      return result;
}
```

```
double sum(int* , int);
main()
{     int x[9];
      double r;
      r=sum(x,9); //sum(&x[0],9)
}
double sum( int *a, int n)
{     int i;
      double result =0.0;
      for (i=0; i<n; i++)
            result=result + *(a+i);
      return result;
}
```

Dynamic Memory Allocation

- calloc : **C**ontiguous memory **ALLOC**ation
- malloc : **M**emory **ALLOC**ation

calloc

- calloc(n, el_size)
  - an array of n elements, each element having el_size bytes

```
void main()
{    int *a;  //will be used as an array
     int  n; // size of array
     ....
     a=calloc(n, sizeof(int)); /* get space for a , and
                                  initialize each bit to zero */
     ....
     free(a); /* each space allocated dynamically should
                 be returned */
}
```

malloc

```
/* malloc does not initialize memory */
void main()
{     int *a;  //will be used as an array
      int  n; // size of array
      printf(“give a value for n:”);
      scanf(“%d”,&n);
      a=malloc(n*sizeof(int)); /* get space for a (allocate a)*/
      ....
      free(a);
}
```

Array Sorting Example: BUBBLE SORT

```
void swap(int *, int *);  /* swap was defined before */
void bubble(int a[ ], int n)  /* n is the size of a[] */
    {
      int  i, j;
      for (i = 0; i < n - 1; ++i)
            for (j = n -1; j > i; --j)
                  if (a[j-1] > a[j])
                       swap(&a[j-1], &a[j]);
    }
```

/* bubble sort algorithm has a worst-case complexity of ***$O(n^2)$ */***

Arguments to main() (1)

- Two arguments, conventionally called argc and argv, can be used with main() to communicate with the operating system.

- The variable argc >= 1 provides a count to the number of command line arguments, including the program’s name itself.

- The array argv is an array of pointers, each pointer pointing to a string - a component of the command line.

- main(int argc, char *argv[])

Arguments to main() (2)

```
/* echoing the command line arguments.*/

     #include <stdio.h>

      int main(int argc, char *argv[])
      {
         int i;
         printf("argc = %d\n", argc);
         for (i = 0; i < argc; ++i)
             printf("argv[%d] = %s\n", i, argv[i]);
         return 0;
      }

/* try to run the program with some arguments at the command line */
```

The Type Qualifiers
const and volatile

- They restrict, or qualify, the way an identifier of a given type can be used.

- "const" comes after the storage class (if any), but before the type, means that the variable can be initialized, but thereafter the variable cannot be assigned to or modified.

  ```
  static const int k = 3;
  ```

- "volatile" object is one that can be modified in some unspecified way by the hardware.

  ```
  extern const volatile int real_time_clock;
  ```

  - Since the storage class is "extern", the system looks for "real_time_clock" either in this file or in some other file.
  - The "volatile" qualifier indicates that the object may be acted on by the hardware; because it is "const", it cannot be modified by the program, however the hardware can change the clock.