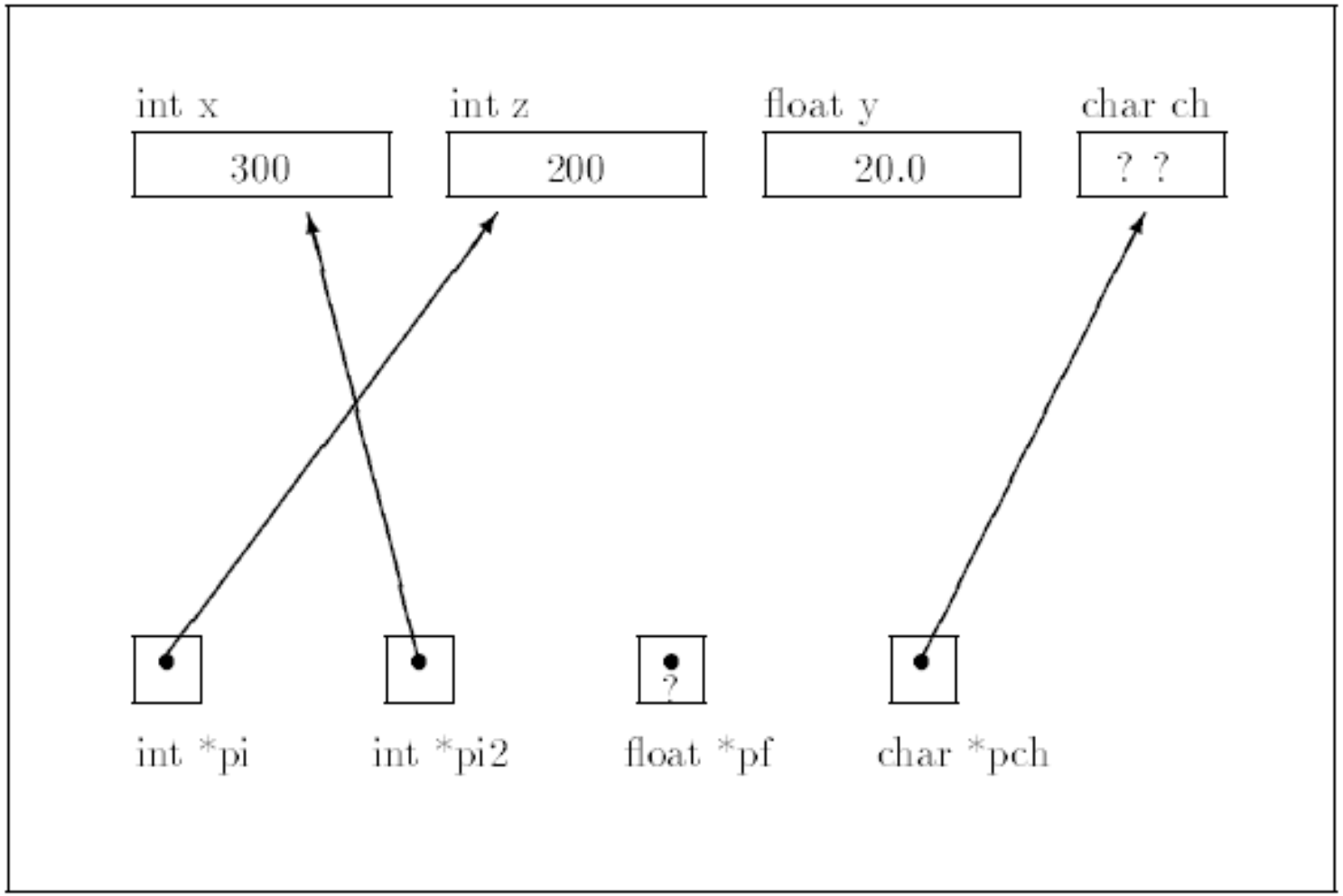
18ES611 Embedded System Programming

Sarath tv

main()


int x
300
int z
200
float y
20.0
char ch
? ?
?
int *pi
int *pi2
float *pf
char *pch

- **Objects on the left and right hand side of the assignment operators**.
- Assigned an *integer value to a cell pointed to by an integer pointer* and when assigning pointers.
- Statement such **as pi=x is a legal statement in c**.
- The value of x will be placed in the pointer cell pi and subsequent dereferencing of pi(*pi) will use that value as a pointer(an address)to find the cell to indirectly access.

```
*pi = x;        or        pi = &x;
```

- **Use the uninitialized variable pf**.

- The value of *pf is garbage*.

- The garbage value of pf may be an invalid memory address
- The program will be aborted due to a memory fault a run time error.
- Even more unfortunate **if the value in pf is a valid memory address we would access a value**
- From some unknown place in memory
- The situation is even worse when an uninitialized pointer is used indirectly.
    - *pf =3.5;
- We do not know where pf is pointing if it happens to be a legal address.
- Placed the value in some unknown location in memory possible a cell belonging to a variable in another part of the program
- **When using pointers particularly ensuring that pointers are properly initialized**

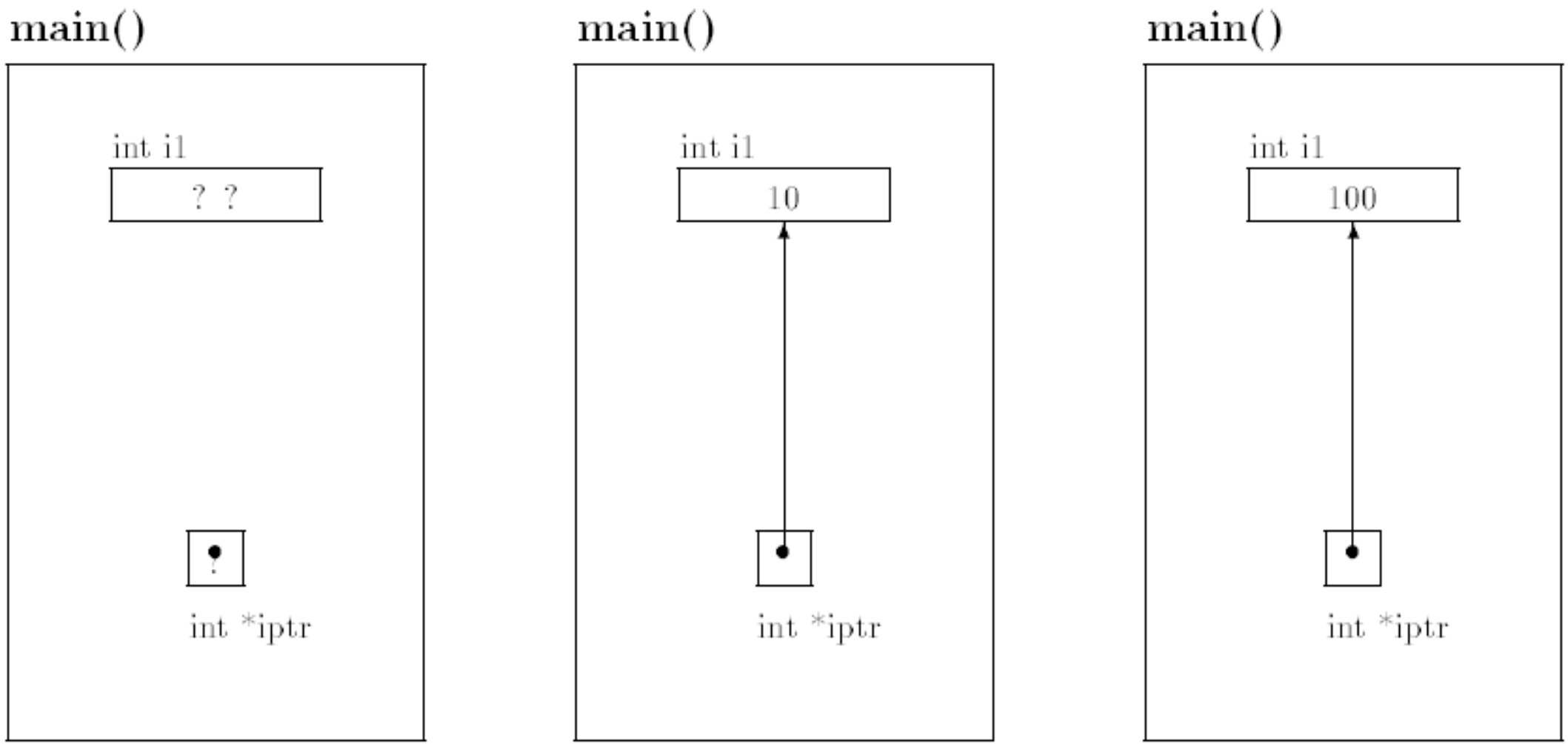
- The character variable ch is not initialized
- But the pointer variable pch is initialized to point to ch
- The expression *pch will access the object ch correctly
- If the value of pch is accessed it will be garbage but a value can be stored in pch correctly

```
/*   File: access.c
     This program prints out the values of pointers and values of
     dereferenced pointer variables.
*/
#include <stdio.h>
main()
{    int *iptr,     /* integer pointer */
          i1;

     printf("Pointers: Direct and Indirect Access\n\n");
     /* initializations */
     i1 = 10;
     iptr = &i1;    /* iptr points to the object whose name is i1 */

     /* print value of iptr, i.e., address of i1 */
     printf("iptr = %u\n", iptr);
     /* print value of the object accessed indirectly and directly */
     printf("*iptr = %d,  i1 = %d\n", *iptr, i1);

     *iptr = *iptr * 10;       /* value of *iptr changed */
     /* print values of the object again */
     printf("*iptr = %d,  i1 = %d\n", *iptr, i1);
}
```


main()
int i1
? ?
int *iptr
main()
int i1
10
int *iptr
main()
int i1
100
int *iptr

Passing Pointers to Functions

- Arguments are passed to functions by value ie only the values of argument expressions are passed to the called function
- Some programming languages allow arguments passed by reference which allows the called function to make changes in argument objects.
- If a called function is to change the value of an object defined in the calling function it can be passed a value which is a pointer to the object. The called function can then dereference the pointer to access the object indirectly.

- A c function can return a single value as the value of the function.
- By indirect access a called function can effectively return several values.
- This use of pointer variables is one of the most common in c.

Indirectly Incrementing a Variable

- program which uses a function to increment the value of an object defined in main.

```
/*    File: indincr.c
      Program illustrates indirect access
      to x by a function indirect_incr().
      Function increments x by 1.
*/
#include <stdio.h>
void indirect_incr(int * p);

main()
{     int x;

      printf("***Indirect Access***\n");
      x = 7;
      printf("Original value of x is %d\n", x);
      indirect_incr(&x);

      printf("The value of x is %d\n", x);
}


/*    Function indirectly accesses object in calling function.  */
void indirect_incr(int * p)
{
      *p = *p + 1;
}
```

Computing the Square and Cube

- Sometimes whether a value should be returned as the value of a called function or indirectly stored in an object is a matter of choice.
- Function that has to return two values.

```
/*   File: sqcube.c
     Program uses a function that returns a square of its argument and
     indirectly stores the cube.
*/
#include <stdio.h>
double sqcube(double x, double * pcube);

main()
{    double x, square, cube;

     printf("***Directly and Indirectly Returned Values***\n");
     x = 3;

     square = sqcube(x, &cube);
     printf("x = %f, square = %f, cube = %f\n",
               x, square, cube);
}

/* Function return square of x, and indirectly stores cube of x */
double sqcube(double x, double * pcube)
{
     *pcube = x * x * x;
     return (x * x);
}
```

function returns the square as its value and returns the cube by indirection

Importance of type

Need to identify the ***type*** of variable that a pointer points
Once ptr "points to" something, and we write *ptr = 2; the compiler will know how many bytes to copy into that memory location pointed to by **ptr**.

If **ptr** was declared as pointing to an integer, 2 bytes would be copied, if a long, 4 bytes would be copied.

Consider a block in memory consisting if ten integers in a row. That is, 20 bytes of memory are set aside to hold 10 integers.
Point our integer pointer **ptr** at the first of these integers.

Lets say that integer is located at memory location 100 (decimal).
What happens : **ptr + 1;**

Because the compiler "knows" this is a pointer (i.E. Its value is an address) and that it points to an integer (its current address, 100, is the address of an integer), ***it adds 2*** *to* ***ptr*** *instead* ***of 1,*** *so the pointer "points to" the* ***next integer****, at* ***memory location 102***.

Similarly, were the **ptr** declared as a pointer to a long, it would add 4 to it instead of 1.

- since **++ptr** and **ptr++** are both equivalent to **ptr + 1** (though the point in the program when **ptr** is incremented may be different***), incrementing a pointer using the unary ++ operator, either pre- or post-, increments the address it stores by the amount sizeof(type) where "type" is the type of the object pointed to***. (i.e. 2 for an integer, 4 for a long, etc.).

- Exp Pointers
- Reorder a one-dimensional, integer array from smallest to largest, using pointer notation

Note

Ask the user for number of elements to be entered.

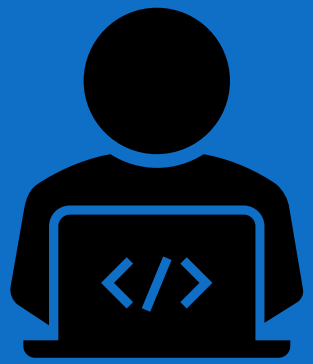
Dynamically allocate the requested amount of memory.

Save the numbers entered by user.

Call a function reorder- which takes two arguments – length and pointer to starting location.-

```
void reorder( int n, int *x)
```

Print the number in ascending order



THANK YOU!!!!!