# 18ES611 Embedded System Programming

*Sarath tv*

# Direct Access

Access objects directly – using variable names.
Variables local to a function -within that function.
During function call- called function and calling function.
Arguments are passed to another function only the values are passed.
The called function may use these values ,but cannot affect the variable in the calling function.
A need to have direct access to the cells in the another function.- Through INDIRECT ACCESS –USING pointer.

# Pointer

A variable that represents the location (not the value) of a data item.

Applications

1.To pass information back and forth between a function and its reference point.

2.An alternate way to access individual array elements

3.Represent multidimensional array -to be replaced by lower dimensional array of pointers.

.

## Fundamentals

Every stored data –occupies one or more contiguous memory cells.

V is a variable

Compiler assigns memory cells for this data item.

location/address of the memory cell.

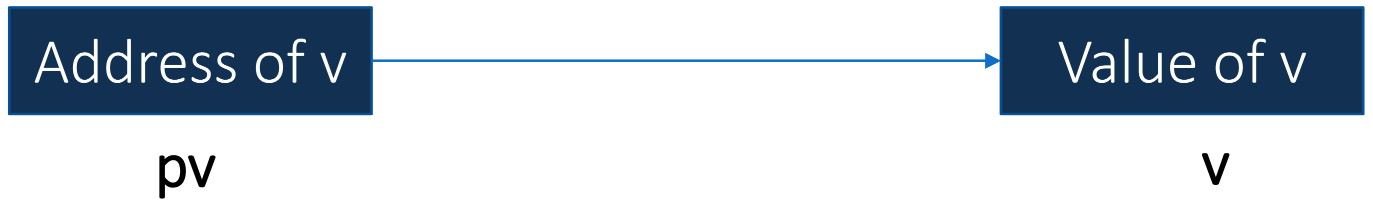
Address operator **&** -> &v gives address of its operand.

Another variable and assign the address of v to it

**pv =&v**

Pv is called a pointer to v since it points to the location where v is stored in memory

Pv represents address not value.



Data from the pointer variable can be obtained by **indirection operator** *pv.

Both *pv and v represent the same data item.

Pv=&v

U=*pv

Implies both u and v have same value.

When a variable is defined, it is allocated a portion of memory.

Thus, the variable has a value and an address for where that value resides.

*A pointer is a variable whose value is the address of another variable.*

Let `x` be defined and initialized to the value 3.

```
char x = 3;
```

variable is stored at address 62.

A pointer **`px`** is subsequently defined, assume it is stored at address 25, and initialized with the address of `x` as follows.

```
char *px = &x;
```

The value of `px`, therefore, is 62.

A pointer is just another type of variable;

pointed-to by a pointer-to-a-pointer variable

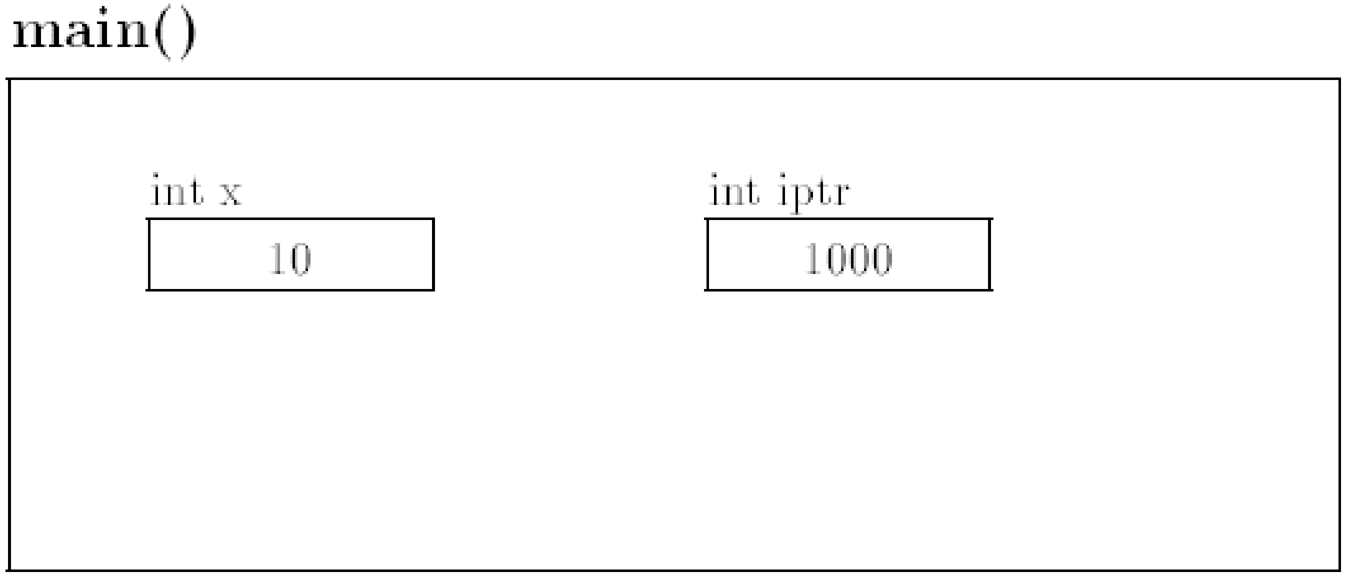
## POINTE

```
main()
{     int x;
      int iptr;

      printf("***Testing Pointer Variables***\n");
      x = 10;
      iptr = &x;
      printf("%d\n",iptr);

}
```

We have declared two integers `x` intended to hold an integer value, and `iptr` which is intended to hold a pointer to an integer ie  and address of an integer.

We then assign a value to `x` and the address of `x` to the variable `iptr` using the address of operator.

The address of a variable is simply the byte address of the cell which was allocated by the declaration .



To print the value of the cell pointed to by `iptr` and not the value of `iptr` itself ??

The **indirection operator *** accesses the object pointed to by its operand.

**The value of** `iptr` **is an address of some object** ie `iptr` points to some object located at address

able to **access that object with an expression** like

`*iptr.`

However there is **no way to know how many bytes to access at address nor how to interpret the data unless the type of object at address is known**

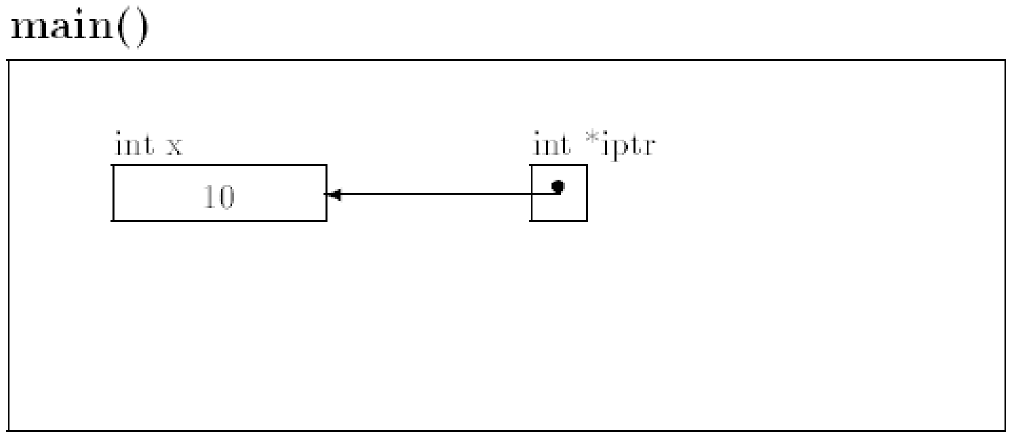
*is it an int a float a char etc*

In order for the **compiler** to **know how to access an object indirectly** it **must know the type** of that object

specify the type of object to access by indicating to the compiler the **type of objects a pointer refers to when we declare the pointer**

So in our example we should declare the variable `iptr` as a pointer to an integer as follows

`int *iptr`



The declaration specifies a variable `iptr` of type int  ie **integer pointer the type is read directly from the declaration.**

So **int  is the type of** `iptr` **- the thing it points to .**This statement declares an integer pointer variable iptr and allocates memory for a pointer variable

Similarly we can declare float pointers or character pointers

```
float  *pa ,*pb;char  *pc;
```

These statements declare variables `pa` and `pb` which can point to float type objects and `pc` which can point to a char type object.

*All pointer variables store addresses which are unsigned integers and so need the same amount of memory space regardless of the pointer types*

**The address of an object is called a pointer to that object** since the address tells one where to go in order to access the object

**The address by itself does not provide sufficient information to access an object.**

In other words **pointers must be specified to be** `int` **pointers pointing to an integer type object, float pointers pointing to a floating point type object char pointers**

## Indirect access of values

The **indirection operator** accesses an object of a specified type at an address accessing an object by its address is called indirect access.

Thus `iptr` indirectly accesses the object that `iptr` points

The **indirection operator** is also called the **contents of operator** or the **dereference operator .**

***The indirection operator to a pointer variable is referred to as dereferencing the pointer variable***

Consider this

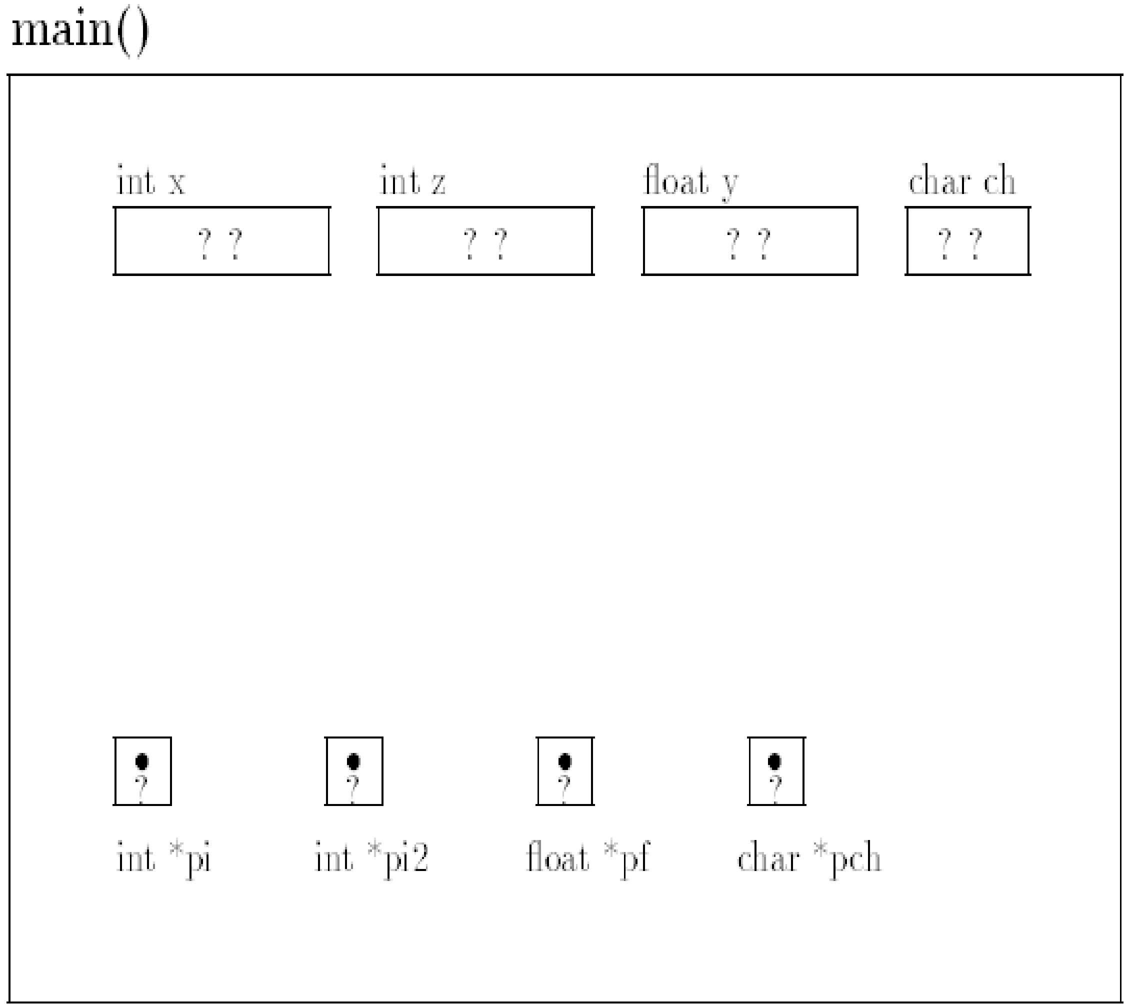
```
int x,z;
float y;
fhar ch, *pch;
int *pi,*pi2;
float *pf;
```

Variables `x` and `z` are int types `y` is float and `ch` is char

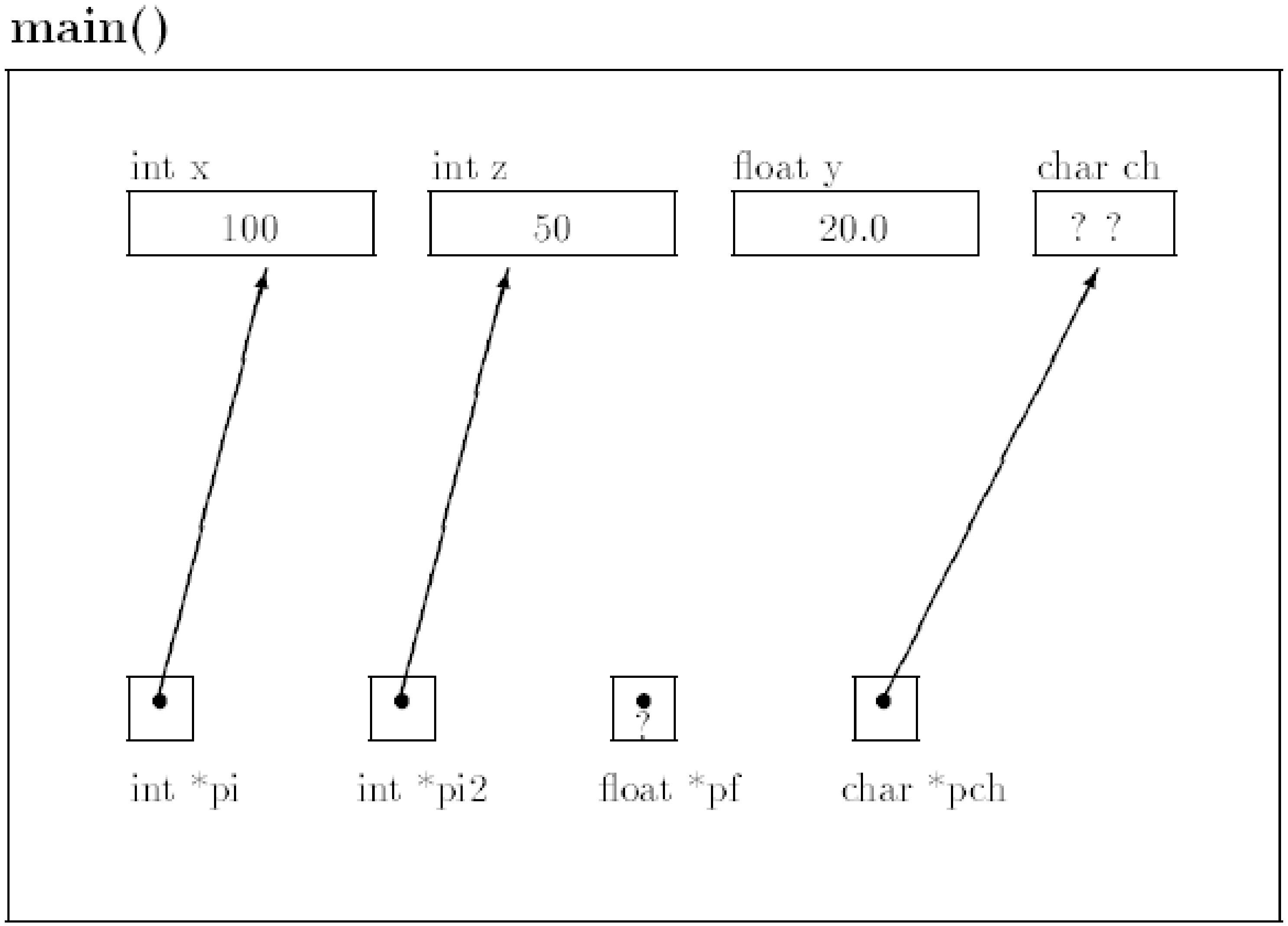
Pointer variables `pi` and `pi2` are variables that can point to integers `pf` is a float pointer and `pch` is a character pointer

Note that the **initial values of all variables including pointer variables are unknown**

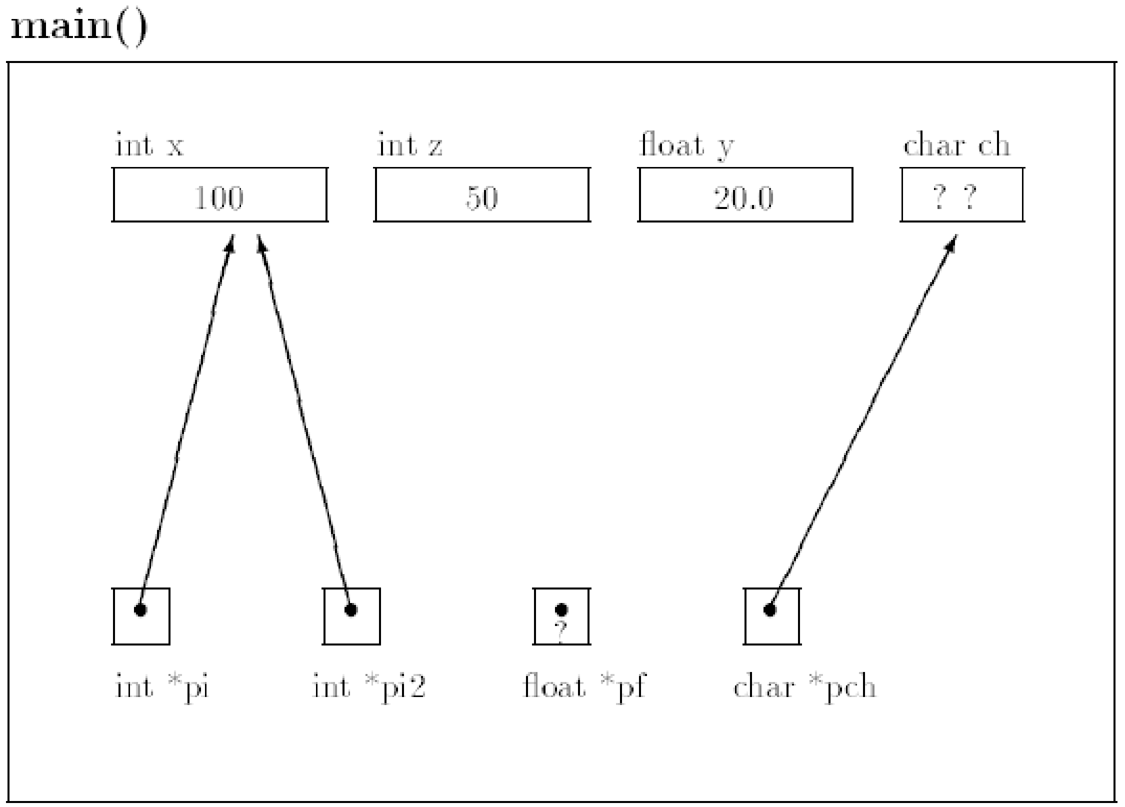
Just as we must initialize int and float variables we must also initialize pointer variables



```
X=100;
y =20.0;
Z=50;
pi =&x;// pi points to x
Pi2=&z; // pi2 points to
Pch=&ch;// pch points to
```

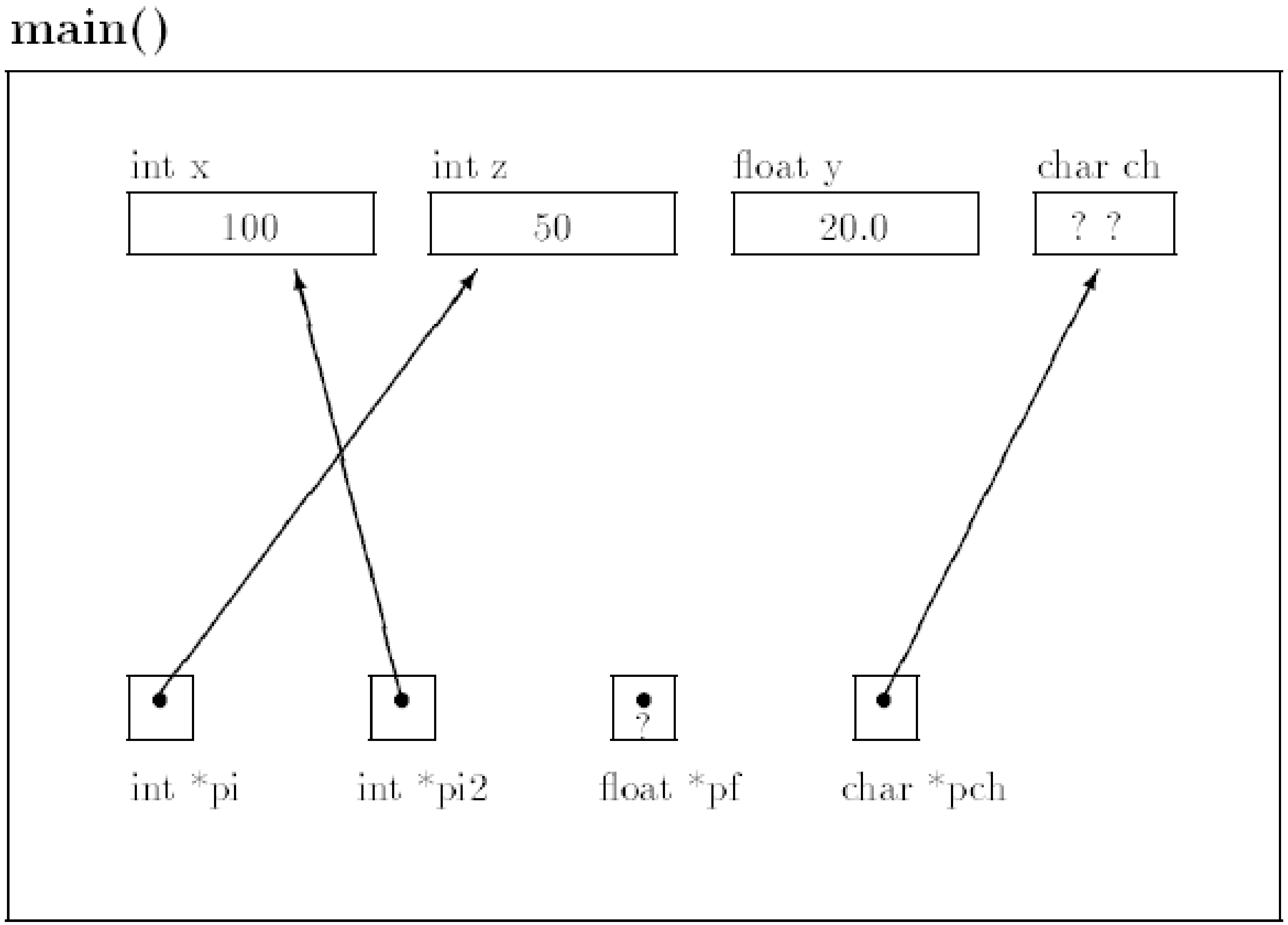


`Pi` points to the cell for the variable `x` `pi2` points to `z` `pch` points to `ch` and `pf` still contains garbage



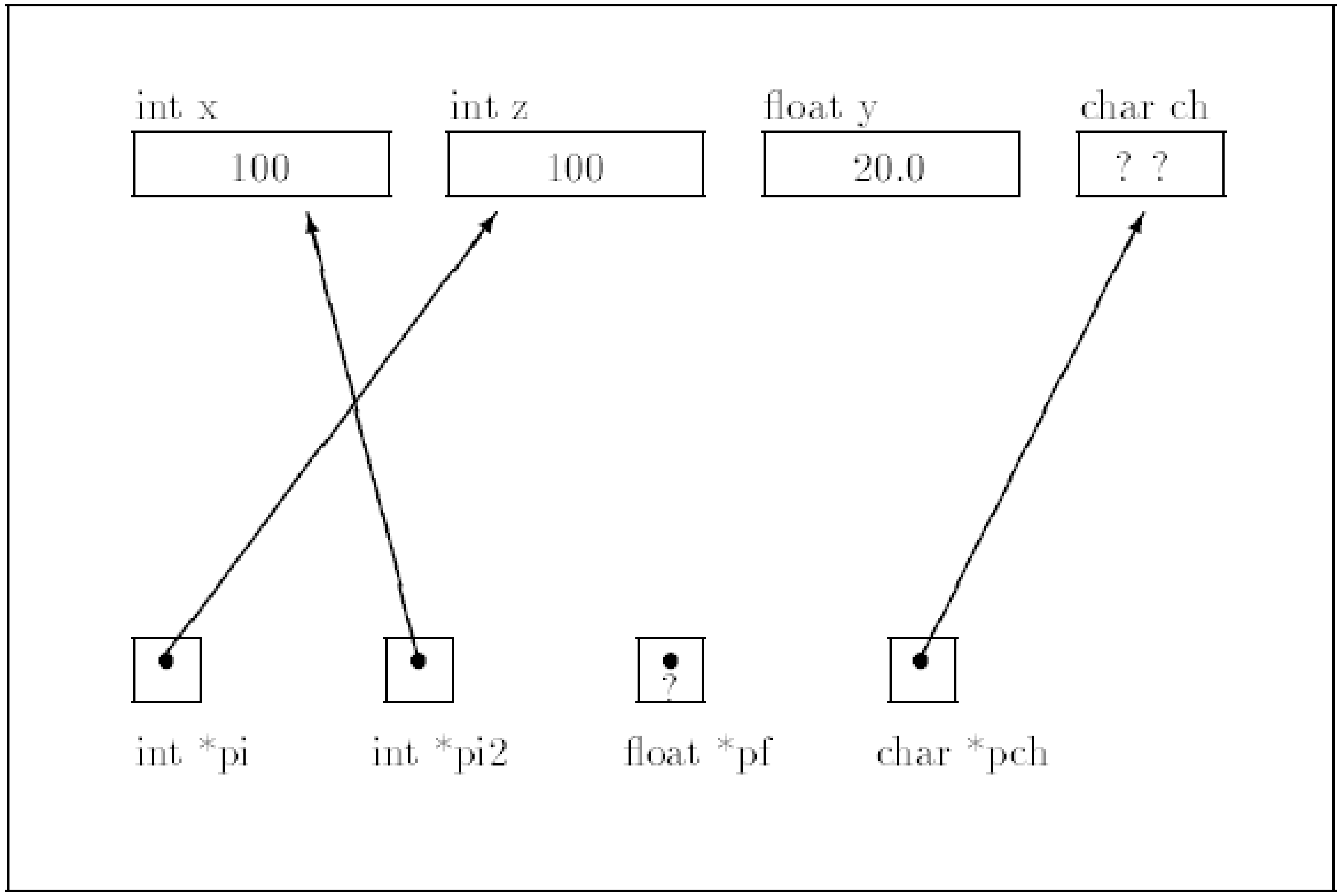
Here are some examples of statements and how they change things for the above memory organization

```
1:   pi2 = pi;                   /* pi2 points to where pi points             */
                                 /* i.e. pi2 ==> x                            */
```



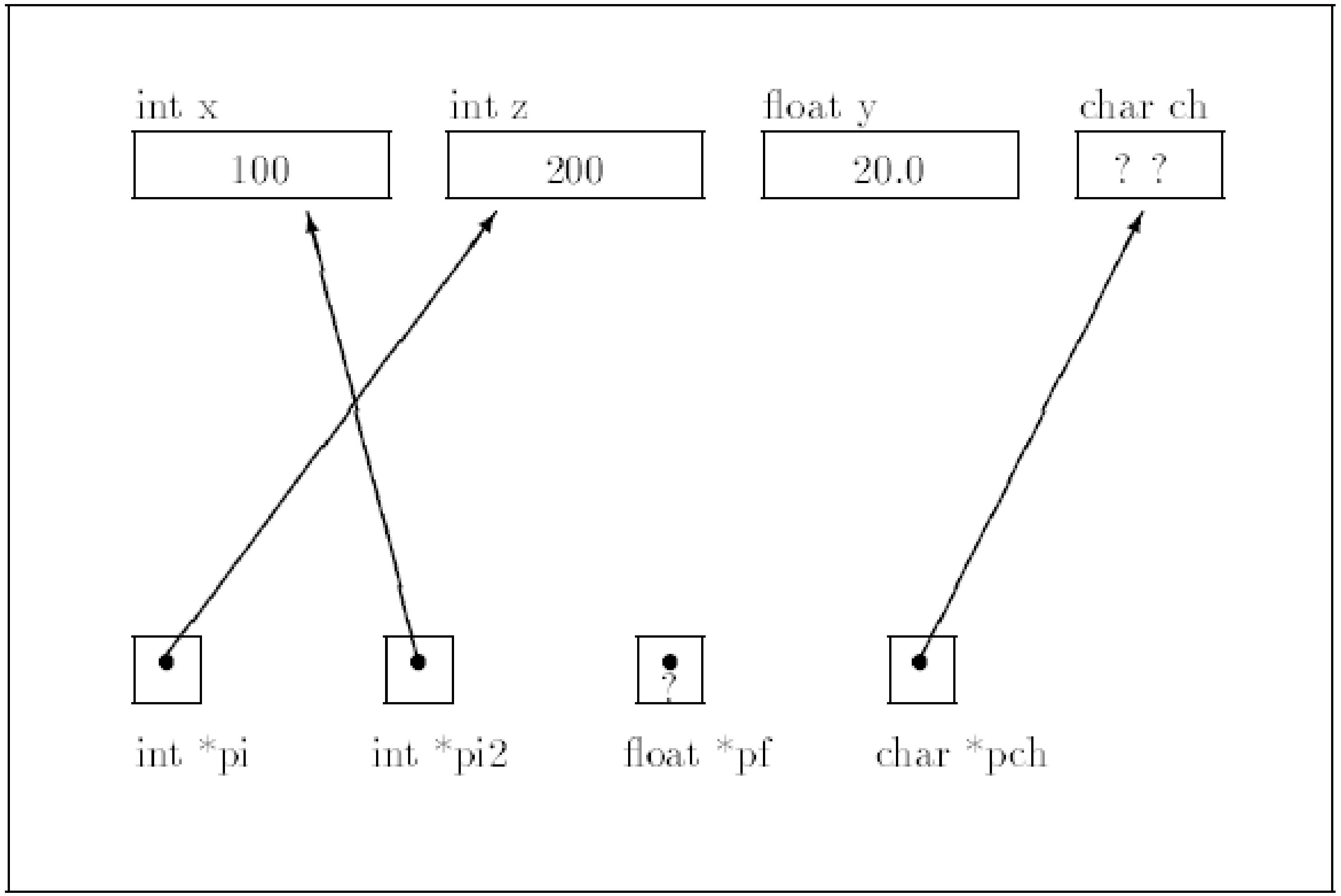
```
2:   pi = &z;                    /* pi now points to z, pi2 still points to x */
                                 /* i.e. pi ==> z, pi2 ==> x                  */
```

**main()**



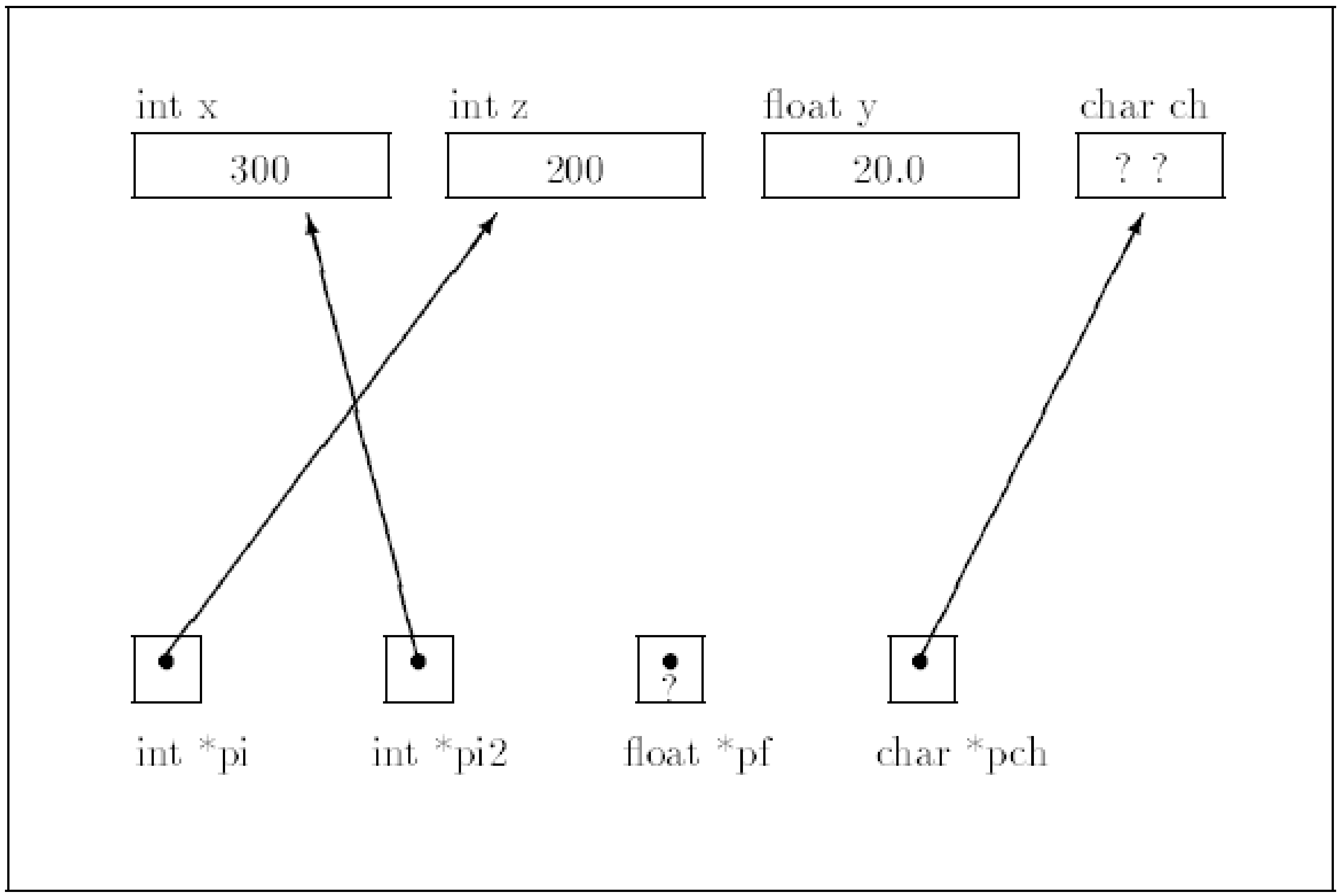
```
3:    *pi = *pi2;                    /* z = x, i.e, z = 100                          */
```

**main()**



```
4:   *pi = 200;                    /* z = 200, x is unchanged                 */
```

**main()**



```
5:   *pi2 = *pi2 + 200;         /* x = 300, z is unchanged                    */
```

## THANK YOU!!!!!