# Variables and Location Bindings

A CS 329 Lecture

*Rushikesh K Joshi*
*Department of Computer Science & Engineering*
*IIT Bombay*

# *Variables and Type Bindings (an overview of what we did already)*

int i  char c  Bool b

Person p  Student s  Faculty f

In all the declarations above, we have types bound to variable names.

We know that variables are bound to types in statically typed languages. In dynamically typed languages, variables do not have types bound to them, but values do have types.

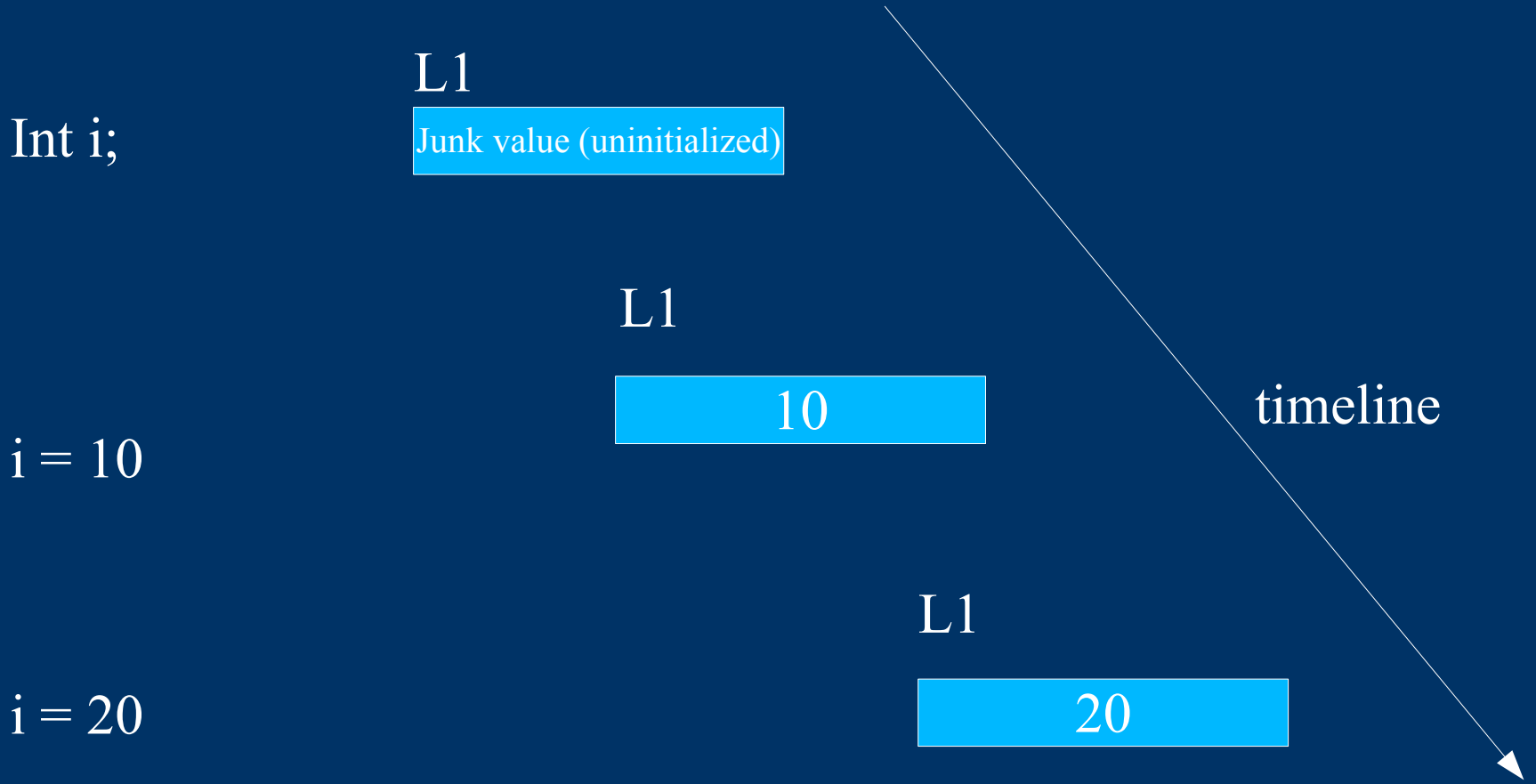
Values have types just as variable have types. In statically typed languages, the rules of what values can be assigned to given variables are governed by subtyping (reflexive, ...and we have seen these rules already)

# *Variables and space bindings*

- Variables are also bound to locations, i.e they are allocated space (locations) in which the value of the variable is stored.
- Storage can be allocated in different storage areas such as:
- statically allocated (predefined) as for globals
- dynamically allocated on heap (as for allocations through malloc or new)
- dynamically allocated on stack for parameters passed or for variables local to function invocations.
- (once the scope of a variable is over, we can free its location for another use)

# A snapshot of location bindings and variable updates

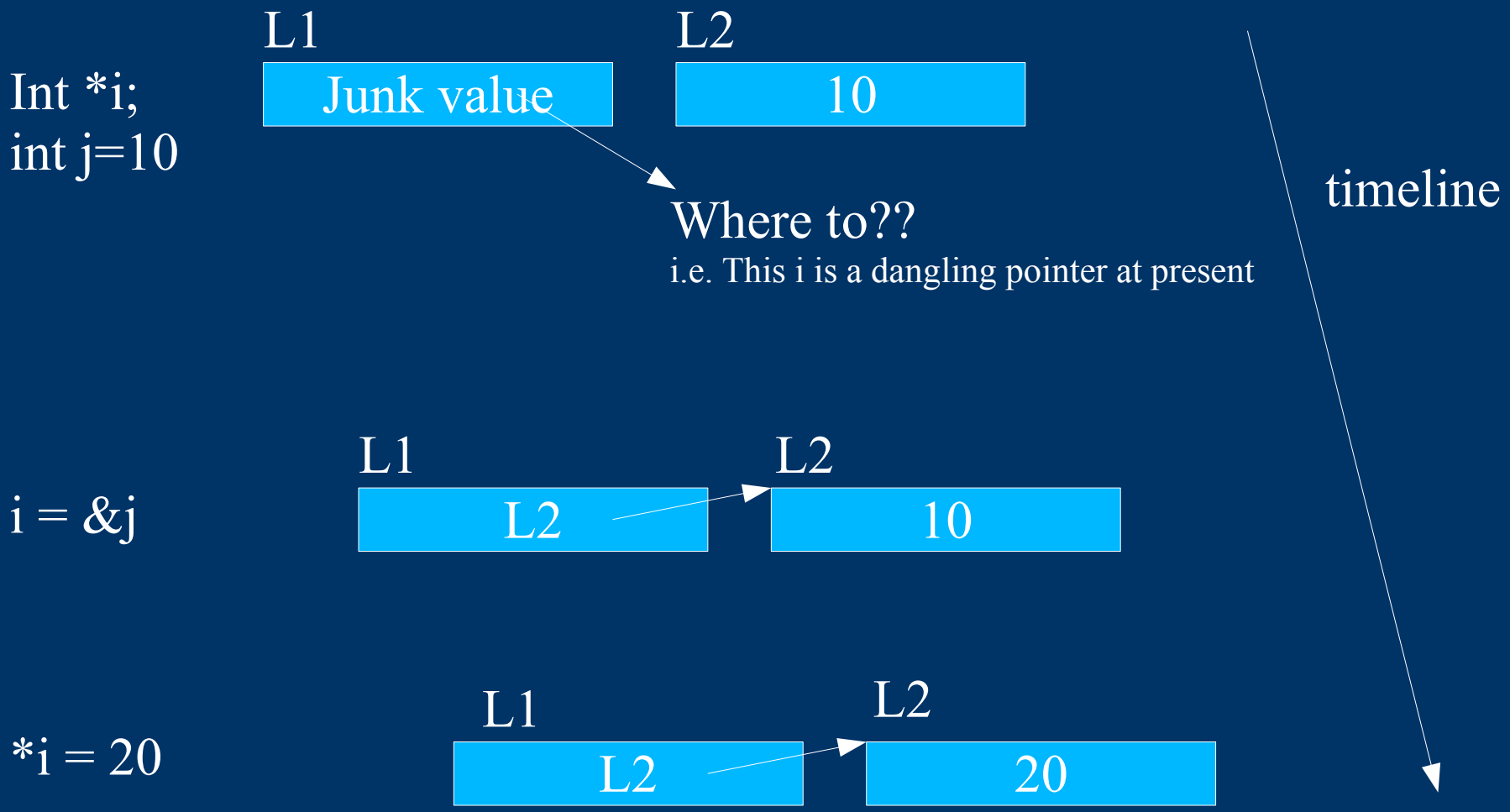
*address of the variable is written on top of the box and value of the variable is written inside*



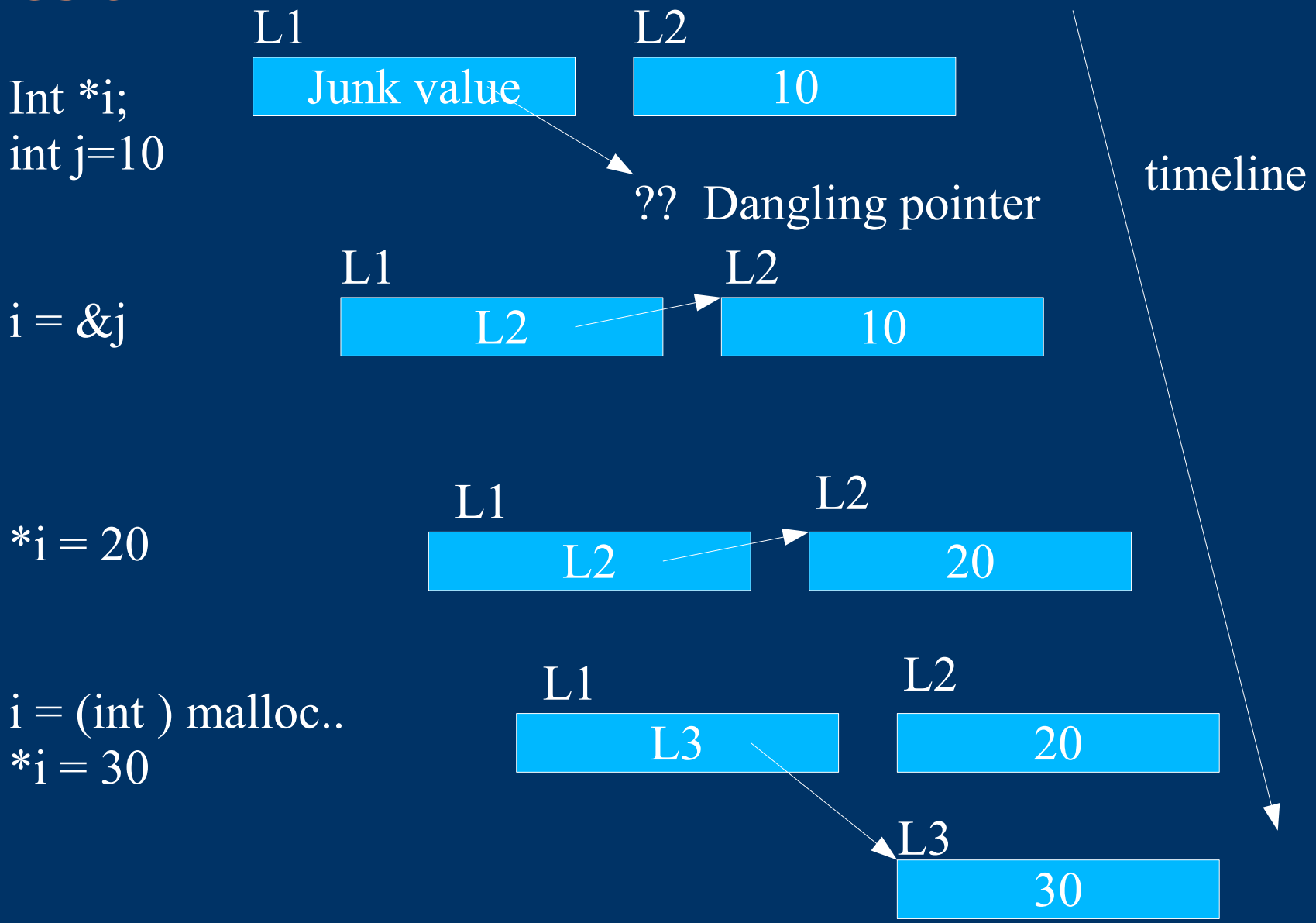

L1 is the location for variable i

# A snapshot of location bindings and variable updates for pointer variables

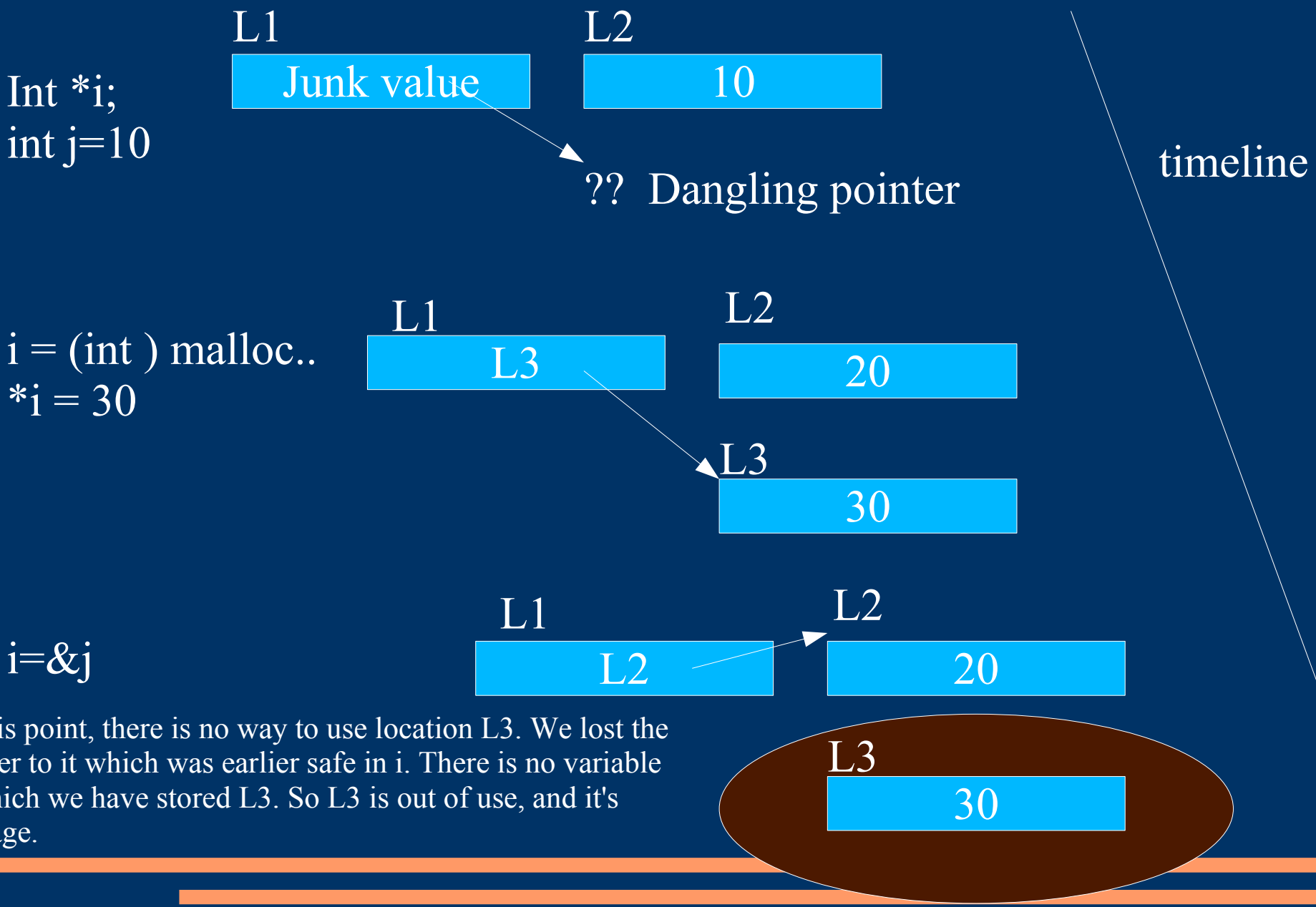
*L1 is location of i, L2 of j , see how their values, i.e. Values at these locations change*



# Changing the value of the pointer itself!



# We may leave garbage in memory




At this point, there is no way to use location L3. We lost the pointer to it which was earlier safe in i. There is no variable in which we have stored L3. So L3 is out of use, and it's garbage.

# *What happens to garbage?*

Garbage is memory locations which were earlier allocated and used, but have gone completely out of use as no one knows its location.

Garbage may get created when you use the same pointer to store different locations.

When a specific call to a function, all its local variables are garbage.

Garbages are freed automatically at runtime if the runtime can detect that specific locations are turned into garbages... But this does not happen with pointers in C/C++....

# *Clearing Garbage by yourself (needed in some languages and for some specific features only)*

If you use pointers in C or C++, its your responsibility to free the allocated memory when you donot need it. If not done so, you may run out of memory soon.

The primitives used to free dynamic allocations are free and delete. The latter is used for objects.

Example:

int **i = (int ) malloc (sizeof (int)***10000);
... use the array ..
free (i);
i = (int) ....  a shorter array... a fresh allocation...

# Automatic Garbage Collection

If a language does not give you a primitive to get yourself some bare memory allocated, it usually detects all locations that turn into garbages.
Some garbages are easy to handle (e.g. Local variables and parameters allocated on an activation stack of a function invocation)

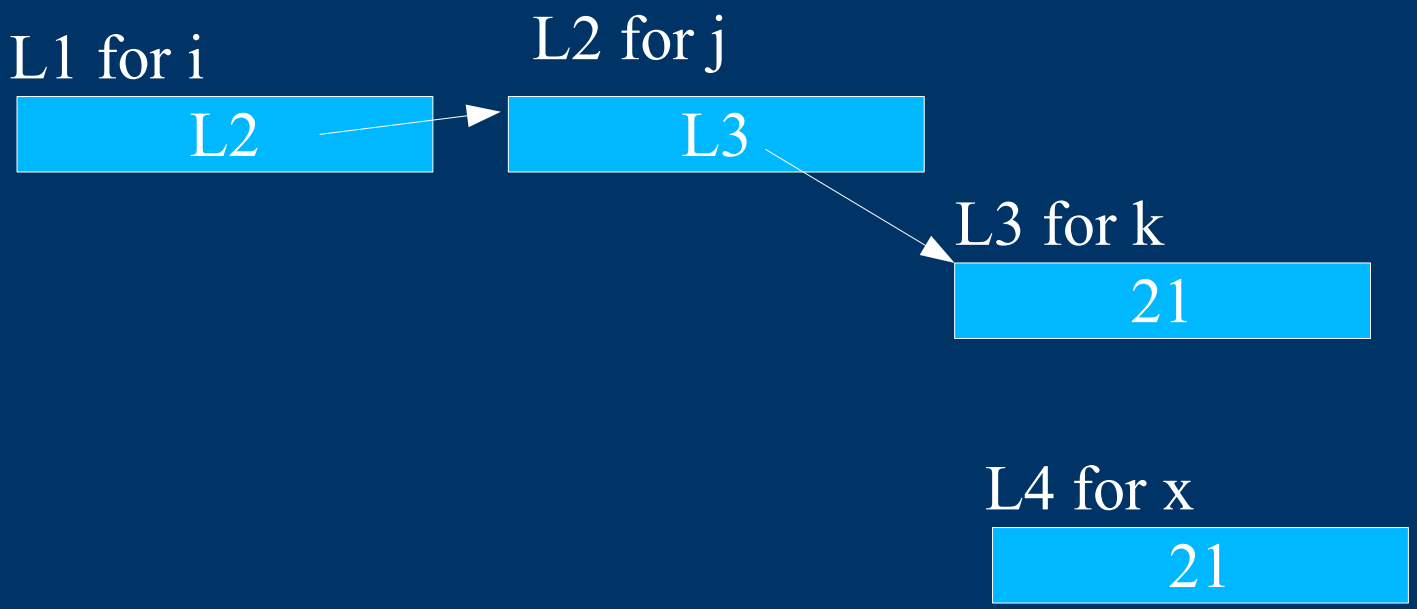
But to track dynamically allocated objects, such languages have their own garbage collectors.
e.g. Java has a runtime garbage collector. And it does not have a free statement.

A garbage collector should not consume as less time as possible and it should keep memory as clean as possible.

# Pointer to Pointer

```
Int **i;
int *j;
int k=21
j = &k
i = &j
int x = **i
```

The snapshot after the last statement



# Bulk Allocations and Pointer Arithmetic

```
Int *A = (int *) malloc (sizeof (int) * 10);
for (j=0; j<10; j++) A[j] = j;
```

or you can say

```
for (j=0; j<10; j++) *(A+j)=j;
```

since the pointer is of type int * , an increase of 1, takes us to the location of the next int. The location of the next int may be some k bytes away from the earlier when k bytes are needed to store an integer.

# *Using a variable on the left and right sides of an assignment statement*

Int i;
int j;
i = 10 is ok
i = j  is also ok
but 10 = j is not ok, though value of i on the right hand side is actually just 10, and i=j was ok.

Which only means that there are two types of values associated when a variable is used.

# *Lvalue and rvalue*

The value of a variable on the right hand side of the assignment statement refers to the value contained in the location: rvalue

The value of the variable when used on the left hand side of the assignment statement, it means the location of the variable: lvalue

And the assignment statement then means, at the location value appearing on the left side, write the value appearing at the right side.

Not all values are valid lvalues in a language. The language applies its rules (of safety and subtyping).