

Pointer types

For values that are implemented by large structures, copying is expensive.

- Copying their *addresses* is much cheaper.

Variables' storage sizes cannot vary during execution.

- Flexible data structures can be built from records containing pointers to other records (the key is that a pointer field can be either a valid pointer to another record or *null*).

Pascal pointer types:

pointer-type constructor: $\uparrow\langle\text{type-name}\rangle$

examples:

\uparrow **integer**

\uparrow *Complex*
where

type *Complex* = **record** *x,y*: **real** **end**

pointer-value constructor: **new**(*p*)

where $p :: \uparrow T$

example:

var *pc* : \uparrow *Complex*;

...

new(*pc*); (* allocates storage for an instance of *Complex**)

(**pc*'s *r*-value is the new record's *l*-value*)

pointer-value destructor: **dispose**(*pc*)

(* Note: this is a Pascal comment. *)

pointer-value *selector*: $p\uparrow$ ("dereference" operator)
where $p :: \uparrow T$
 $\Rightarrow p :: \uparrow T \Rightarrow p\uparrow :: T$

example:

$pc\uparrow.y$ refers to the *y* field of the anonymous record to which *pc* "points"

the "empty" pointer: **nil**

other operations on pointers:

$p := q$

$p = q$

$p <> q$

(provided *p* and *q* are same type)

Pointer hazards

uninitialized pointer

dangling pointer:

pointer to a disposed object

storage leak

destroying the only pointer to an object \Rightarrow it can't be **disposed**

aliasing

$p := q$; $p\uparrow := 5$;

(* $q\uparrow$ is updated implicitly *)

double-dispose

same block of storage is disposed twice

Safety feature: Pointer values can be *copied* and *destroyed*, but they can be *created* only by **new**.

C pointer types

type constructor: $\langle\text{type-name}\rangle^*$

examples:

int*

complex*

value constructor: $p = \text{malloc}(n)$;

where $p :: T^*$
 $n = k \cdot \text{sizeof}(T)$
 $k \in \{1, 2, \dots\}$

example:

struct **complex** { **double** *x*; **double** *y*; };

...

complex* *pc* = **malloc**(**sizeof**(**complex**));

/*allocates storage for a **complex** record */

value destructor: **free**(*pc*);

another value constructor: **&***v*
where *v* has an *l*-value

examples:

&a /* address of *a* */

&b[12] /* address of *b*[12] */

value selector: $*p$ ("dereference" operator)
where $p :: T^*$

example:

$(*pc).y$ refers to field *y* of the anonymous struct to which *pc* "points"

$pc \rightarrow y$ shorthand for $(*pc).y$

$\Rightarrow p :: T^* \Rightarrow *p :: T$

the "empty" pointer: $0 :: T^*$

other operations on pointers:

$p = q$ assignment

$p = q$ comparison

$p != q$

$p \Leftarrow q$ etc.

$p - q$ $(-) :: T^* \times T^* \rightarrow \text{int}$

$p + i$ $(+) :: T^* \times \text{int} \rightarrow T^*$

$p - i$ $(-) :: T^* \times \text{int} \rightarrow T^*$

C pointer hazards

uninitialized pointer (as in Pascal)

dangling pointer (as in Pascal)

storage leak (as in Pascal)

aliasing (as in Pascal)

double-free (as in Pascal)

- + "wild" pointers:
pointer arithmetic can produce *any* address

- + freeing an address not created by **malloc** causes havoc in heap-space management

Connection (in **C/C++**) between *pointers* and *arrays*—

```
If p == &(a[i])
then p+1 == &(a[i+1]).
```

Moreover, if *a* is an array, say

```
char a[100];
its type is
char* const a
and
a == &(a[0])
so
a+i == &(a[i]),
i.e.,
a+i points to a[i].
```

The linear-search algorithm in C:

```
int A[n+1];
int i = n;
while (A[i] != x) --i;
return i;
```

The same algorithm in *pointer* style:

```
int* p = A+n;
while (*p != x) --p;
return p-A;
```

Pointers' "units" are scaled (by the compiler) according to the size of items to which they point.

Example:

```
char* pChar = 0;
int* pInt = 0;
double* pDouble = 0;
for( int i = 0; i < 6; i++ )
{
    printf( "%d %d %d %d\n",
           i, pChar + i,
           pInt + i, pDouble + i );
}
```

the output:

```
0 0 0 0
1 1 4 8
2 2 8 16
3 3 12 24
4 4 16 32
5 5 20 40
```

Pointers in Java

The claim that "Java has no pointers" is exaggerated. The term *pointer* is avoided in Java literature, but the substitute term *reference* is just another word for *pointer*.

What Java lacks (compared with C/C++) is *pointer arithmetic*—and the associated hazards.

In this respect, Java resembles Pascal, which also forbids pointer arithmetic.

Pointers are actually more prevalent in Java than in Pascal, C, or C++, because in Java values of all non-primitive types(are referred to by pointers (i.e., by "references").

The "no-pointers" claim has some justification, however, because Java's pointers lack many of the hazards present in earlier languages' handling of pointers.

- *uninitialized* pointers: Every pointer is initialized to a null value, and every use of a pointer is preceded by a run-time check for nullity (attempts to dereference null pointers cause exceptions).
- *dangling* pointers: Java has no delete operation; instead, allocated storage is reclaimed by the garbage collector, which does not reclaim any storage which is reachable from any executable code.

* Java's primitive types are boolean, char, byte, short, int, long, float, and double.

- storage *leaks*: If available storage is exhausted, the garbage collector eventually reclaims all unreachable storage.
- *aliasing*: Assignment of non-primitive values copies their references.


```
integer [] A = new ...
integer [] B;
B = A; // A and B refer to the same array
B = A.clone(); // B refers to a copy of A.
```

 Java programmers learn to think of non-primitive variables not as *containers of* values, but as *references to* values.
- "wild" pointers: Java's prohibition of pointer arithmetic, and its initialization of all references to null, mean that every non-null pointer is a valid reference.

Summary

Java's pointers are devoid of most of the hazards associated with pointers in earlier languages. By hiding such processor capabilities as the manufacture of arbitrary pointers, Java

- presents the programmer with a simpler computational model.
- closes off some avenues to greater speed.
- enables access to machine resources to be controlled and restricted.
- enhances software's portability.