Chapter 7

Pointers and References

Pointers

Values are memory addresses plus NULL/NIL

Uses

- Addressing flexibility
- Dynamic storage management
- Aliasing

Pointers (or references) are necessary for dynamic data structures

Pointers and Recursive Types

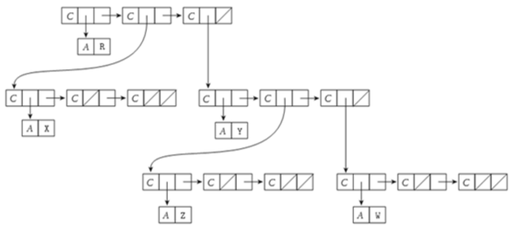


Figure 7.11 Implementation of a tree in Lisp. A diagonal slash through a box indicates a null pointer. The *C* and *A* tags serve to distinguish the two kinds of memory blocks: cons cells and blocks containing atoms.

Pointers – Design Issues

What is the scope and lifetime of a pointer variable?

What is the lifetime of the object it points to?

- A heap-dynamic variable

Any restrictions to reference only certain types?

How to manage dynamic storage?

Pointers – Operations

(1) Assignment of an address to a pointer

(2) Referencing

- Implicit (array elements, like a[5])
- Explicit (dereferencing operator, like *p in C)

Arithmetic operations (pointer arithmetic)

Comparison

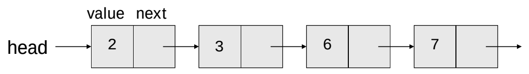
- Complete (>, <, >=, <=, ==, !=)
- Partial (== or !=)

Pointers in C

```
int x, y, *z;
x = 10;
y = 20;
z = &y;
x = y + *z;
z = &x;
x = y + *z;
z++;
x = y + *z;
```

Address	Contents	
0999		
1000		
1001		x
1002		
1003		
1004		
1005		y
1006		
1007		
1008		
1009		
1010		z
1011		
1012		

Pointers in C



```
struct Node {
   int value;
   struct Node *next;
}
struct Node *head;
```

```
struct Node *p = head;
while(p != NULL && p->value != x) {
   p = p->next;
}
```

More Pointer Examples

C, C++

Arrays, records, unions, function names are all pointers in disguise

```
double a[10];
double * aptr = a;

*(aptr+3) ≡ aptr[3] ≡ a[3]
```

`void *p;` can point to any type!

More Pointer Examples

C++ reference types

Used for reference parameters

```
void double(int &x) {
   x = 2*x;
}

int y = 10;
double(y);
```

Java
- Only has references
- No explicit deallocation, uses garbage collection

Dynamic Data

Two basic memory operations with dynamic data
- Object *creation* allocates heap storage for an element.
- Object *destruction* returns heap storage to the OS for later use.

Example in C

`void *malloc(int number_of_bytes)`

Allocates a contiguous amount of memory of the specified type and returns the address of the first byte.

`void free(void *p)`

De-allocates the memory referenced by **p**

Dynamic Data in C

```
// Dynamic data and pointer arithmetic
char *makeHelloString() {
  char *str = (char*)malloc(6 * sizeof(char));

  *str = 'h';
  *(str+1) = 'e';
  *(str+2) = 'l';
  *(str+3) = 'l';
  *(str+4) = '0';
  *(str+5) = (char)0;
  return str;
}
```

Pointer Efficiency

Pointers can be used to efficiently pass lots of data!
- C always uses pass by value
- When a pointer is passed, how much data is copied?

```
char *str =
       (char *)malloc(5000 * sizeof(char));

someStringFunction(str);
```

Pointers – Problems

Dangling pointers

- Pointer points to a heap object that has been explicitly deallocated or has gone out of scope

Memory leaks

- Pointer points to a heap object, then gets reassigned without deallocating the memory of the first one

Aliasing

- More than one reference to the same memory location

Java and Garbage Collection

Object creation uses **new**

Object *destruction*?

- Java doesn’t let programmers deallocate memory because they’ll probably create dangling pointers and memory leaks
- Memory that's been allocated but no longer in use is "garbage collected"
  - Reference counting
  - Mark-and-sweep