Imperative Programming — Data Structures

Steven Zeil

Sep. 20, 2001

Contents

Imperative Languages

1. Statements
2. Data Structures
3. Procedure Activation

2 Data Structures

1. What is a Type?
2. Type Systems
3. Basic Types
4. Arrays
5. Records
6. Unions
7. Pointers
8. References

2.1 What is a Type?

- A **data object** (object) is any separately identifiable unit of data manipulated by a program.
- The **representation** of an object is a description of how that object is mapped into memory.

2.1.1 Intent versus Possibility

- Every object has intended uses.
- Every representation implies possible operations for manipulating the data.

Sometimes these match well. Other times, they clash.

Consider, for example:

Object	Representation	Operations
boardLength	`float`	+, ∗
thisMonth	`integer`	+, ∗
dictionary	`array[<>] of string`	indexing

Types help eliminate mismatches.

2.1.2 Types

Different writers have different views of what constitutes a type:

- a type is a set of values
- a type is a set of operations

- a type is a representation

Sethi says a **type expression** is a description of how a data representation is built up.
e.g., `array[0.99] of char`
He uses "type" as an abbreviation for "type expression".

I'm not entirely happy with Sethi's definition.

- In some languages, data with the same representation can still have different types.

---

e.g., in Ada

```
type Mass is digits 8;
type Voltage is digits 8;

m: Mass;
v: Voltage := 1.0;

m := v;  -- illegal!
```

2.1.3 Types as Algebras

A **type** is a collection of values with an associated representation and a set of permitted operations.

- this set of operations may be bigger than, smaller than, or identical to the set allowed on the representation.

Type	Representation	Operations
Lengths	`float`	+, $*$
Months	`integer`	+
WordSets	`array[<>] of string`	lookup

One advantage of viewing types as algebras is that we can use ideas from set theory to describe the semantics of type expressions.
Given the types (sets):

bool $\{true, false\}$

color $\{red, blue, green\}$

int $\{0, -1, 1, -2, 2, \ldots\}$

What do the following sets resemble?

- bool $\times$ color
- int^{20}
- bool $\cup$ int
- int $\rightarrow$ int

2.2 Type Systems

A **type system** is a set of rules for associating types with expressions.

E.g.,

1. A numeric constant has type `real` if it contains a decimal point. Otherwise it has type `integer`.
2. A variable has the type it is declared with.
3. If expressions E and F have the same type, then $E+F$, $E-F$, $E*F$, and E/F have that same type.

What does this system say about `1.2 + 3*X`?

2.2.1 Type Equivalence

A common rule in most language type systems is

- If `f` has type $S \rightarrow T$ and `a` has a type *equivalent to* S, then `f(a)` has type T.

- Operators can be treated under this rule (replacing rule 3), e.g.:

  $$+ : \text{int} \times \text{int} \rightarrow \text{int}$$

  if we regard the infix form as "syntactic sugar" for the prefix form of function calls.

There are two general approaches to determining when one type is "equivalent to" another:

- structural equivalence
- name equivalence

Most languages use a mixture, but lean hard to one side or the other.

Structural Equivalence
In this approach, types are equivalent if they have the same representation.

SE1. A type name is equivalent to itself.

SE2. Two types are equivalent if they are formed by applying the same type constructor to structurally equivalent types.

SE3. After a type declaration giving name n to type expression T, types n and T are equivalent.

C and C++ function mainly under structural equivalence rules:

```
typedef int Integer;
typedef const Integer* IntegerPointer;

void foo (const int* p);

IntegerPointer pI;
foo (pI);  /*  This is legal. */
```

(Exception for C and C++: structs and classes are judged under name equivalence).

Name Equivalence

Under name equivalence, declaring a name for a type expression produces a new type that is not equivalent to any existing type.

Pascal, Modula 2, and Ada function largely under name equivalence:

```
type Arr1 is array[1..10] of integer;
type Arr2 is array[1..10] of integer;

procedure Foo (a: Arr1); FORWARD;

  b: Arr2;
begin
  Foo (b);  -- Illegal!
```

In practice, the philosophy of name equivalence leaves several options open, e.g.,

- How to treat equivalence between type expressions?
- ...between names and type expressions?
- Is a user-declared type name a type expression?

Name Equiv. Variations

Pure Name Equiv: No two type expressions are identical to each other.

```
  x, y: array[1..10] of integer;
  z: array[1..10] of integer;
begin  -- Ada
  x := y;  -- OK
  z := y;  -- type error
```

Name Eq. Variations (cont.)

Transitive Name Equiv: Type names can be declared equivalent to other types. Type equivalence among names is transitive.

```
TYPE  /* Modula 2 */
    S = INTEGER;
    T = S;
    U = INTEGER;
    /* All 3 are equivalent */
```

Name Eq. Variations (cont.)

Type Expression Equiv: A name is equiv only to itself. Expressions are equiv. if they are formed by applying the same constructor to equiv. expressions.

Modula 2's type system is based on Name Equiv, but shows the compromises that get made to provide utility.

Two Modula 2 types are "compatible" if

- They are the same name, or
- They are names `s` and `t` and `s=t` is a type declaration, or
- One is a subrange of the other, or
- Both are subranges of the same basic type.

```
TYPE
  S = INTEGER 1..10;
  T = INTEGER 1..10;
  U = S RANGE 1..5;
  V = S RANGE 1..5;
```

Equivalent?

	S	**T**	**U**	**V**
S	y	y	y	y
T	y	y	n	n
U	y	n	y	n
V	y	n	n	y

2.2.2 Common Special Cases

- Overloading
- Coercion
- Polymorphism

Overloading

$$+ : \text{int} \times \text{int} \to \text{int}$$

$$+ : \text{real} \times \text{real} \to \text{real}$$

Same name, different functions.

Coercion

Some languages have rules similar to

> Every `int` can be implicitly converted to a `real` if doing so will make an otherwise invalid expression legal.

So $2 * 3.14159$ is treated as `intToReal(2)*3.14159`.

Polymorphism

Polymorphic types carry parameters that can be assigned as needed to make an expression legal:

$$push : \text{stack}(\alpha) \times \alpha \to \text{stmt}$$

```
s1: stack(real);
s2: stack(int);
  ⋮
push(s1, 3.0);
push(s2, 32);
```

2.2.3 Strong versus Weak Typing

- A **type error** occurs when a function f expects objects of type T as a parameter but is passed an object of a different type.
- A program that executes without type errors is called **type safe**.

Strong versus Weak Typing (cont.)

- A type system is **strong** if it only accepts type safe expressions. Otherwise it is called a **weak** system.

- Strong typing provides
  - portability and
  - protection from programmer errors.
- Weak typing provides
  - flexibility and
  - the ability to take advantage of special machine-dependent data properties

---

Type Checking

Types can be checked

- statically (at compile time)
- dynamically (at run time)

Static checking is usually more efficient.

Dynamic checking of strong type rules can provide more flexibility without compromising safety.

2.3 Basic Types

Enumerations:

`Color = (red, blue, green);` (Pascal)
`enum Color {red, blue, green};` (C++)

Booleans: `LOGICAL` in FORTRAN, `Boolean` in Pascal and Ada, `bool` in C++

Booleans:

`LOGICAL` in FORTRAN, `Boolean` in Pascal and Ada, `bool` in C++

- often equivalent to an enumeration `(false, true)`
- some languages use integers for this purpose

Characters: `Char` in Pascal, `char` in C and C++

Integers: `Integer` in Pascal and FORTRAN, `int` in C and C++

Floating Point: `Real` in Pascal and FORTRAN, `float` in C and C++

2.3.1 Layout of Basic Types

- Enumerations, booleans, and characters are encoded as machine integers of various sizes.
- Integers pose a portability problem
  - how many bytes per integer?
  - how are bytes laid out (big-endian versus little-endian)
- Floating points offer similar problems
  - somewhat alleviated by IEEE floating point standard

Integer Portability - C and C++

- A variety of integer types are defined: `char`, `short int`, `int`, `long int`
  - Also signed and unsigned versions of each
  - Length relation is char $\leq$ short $<$ int $\leq$ long
  - `char` must be large enough to hold a character in the "implementation's" basic character set. There are no other restrictions on size (in C++).

Integer Portability - Pascal

- No standard for what can fit into an `Integer`
- Programmers can gain some safety by using **subranges**:

  ```
  TYPE
      Int = -100000..100000;
  ```

  Error messages will result if the underlying `Integer` type does not allow numbers in this range.

Numeric Portability - Ada

Ada extended the idea of subrange:

- Although there are "default" integer and real types, programmers are strongly encouraged to define their own:

```
type TableIndices is 0..100000;
   -- integer
type Mass is digits 8;
   -- floating point
type Dollars is delta 0.01
   range -1_000_000_000 ..
         1_000_000_000;
   -- fixed point
```

2.4 Arrays

Arrays hold sequences of elements.

`array[`⟨*subrange*⟩`] of T`, Pascal, Ada

`T` ⟨*id*⟩`[numElements]`, C, C++

`A[i]` names the element corresponding to index i in almost all languages.

- FORTRAN used () instead of []
- C and C++ arrays always have 0 as lowest index
- C++ allows programmers to write ADTs that simulate arrays, even down to using [] for indexing
  - variable length arrays
  - arbitrary index types

- Pascal, Ada, allow programmer to specify lower and upper index bounds.
  - default is 1
- Pascal & Ada allow enumerations as indices as well.
- A few languages (Perl, AWK) allow strings as indices

2.4.1 Layout of Arrays

Single-dimensioned arrays are laid out as a linear sequence in memory:

`A: array[Low..High] of T;`

A[Low]	A[Low+1]	A[Low+2]	...	A[High]

The address of `A[i]` is computed as

$$A_{\text{base}} + (i - \text{Low}) * w$$

where A_{base} is the starting address of the array and w is the width of a single array element.

Padding

Many machines have **alignment** requirements that require certain kinds of data to begin at addresses that are evenly divisible by 2, or 4, or ....

Compilers will then insert "padding" into each array element, so that elements are properly aligned.

Example:

`a: array[1..NumEmployees] of PersonelRecord;`,

where each `PersonelRecord`

- requires 61 bytes,
- begins with a floating point number,
- on a machine that requires 8-byte alignment of floats

A[1]	p	A[Low+1]	p	A[Low+2]	p	...	A[High]

where each p represents 3 bytes of padding.
The address of `A[i]` is computed as

$$A_{\text{base}} + (i-1) * 64$$

Multi-dimension Arrays

`A: array[1..3] of [1..2] of T;`
`A[i]` is an array (`array[1..2] of T`).

Visualize `A` as

$A[1,1]$	$A[1,2]$
$A[2,1]$	$A[2,2]$
$A[3,1]$	$A[3,2]$

or as

$A[1,1]$	$A[2,1]$	$A[3,1]$
$A[1,2]$	$A[2,2]$	$A[3,2]$

.

- Most languages that allow multi-dimensioned arrays do not actually allow you to access a "slice" of the array — you must supply all indices. (exception: Ada)

Linearization

Multi-dimensioned arrays are laid out by linearizing them:

Row-major: $k_{A[i,j]} = j + (i-1) * n_j$, where n_j is the range of the j dimension.

A[1,1]	A[1,2]	A[2,1]	A[2,2]	A[3,1]	A[3,2]

Linearization (cont.)

Column-major: $k_{A[i,j]} = i + (j-1) * n_i$, where n_i is the range of the i dimension.

A[1,1]	A[2,1]	A[3,1]	A[1,2]	A[2,2]	A[3,2]

Does row- or column-major ordering really make a difference?

- On occasion.

Given: 8-byte floats, 4096 byte virtual memory pages,

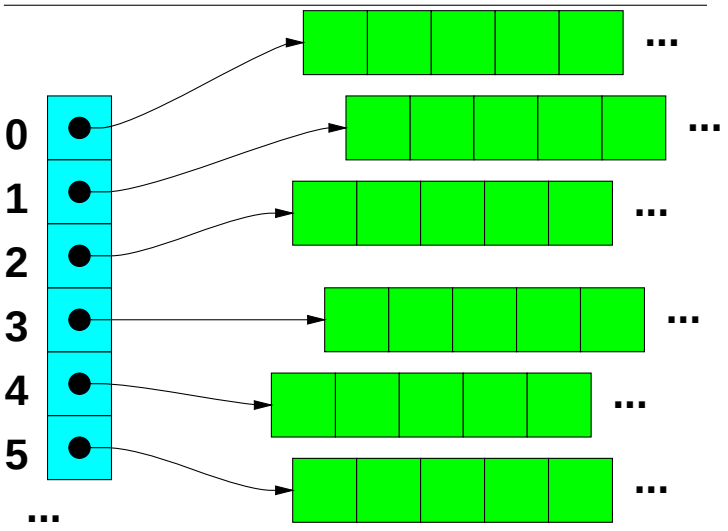
```
VAR
  A: array[1..512] of [1..1000]
   of float;

FOR I := 1 TO 512 DO
  FOR J := 1 TO 1000 DO
    A[I,J] := I + J;
```

could cause 1000 page faults (disk reads) under column-major, but 512,000 page faults under row major ordering.

Arrays in C

In C and C++, the type `T a[...];` is identical to the type `T*` (pointer to `T`). The value of the variable `a` is the base address of the array.

- The lower bound on all arrays is 0.
- a[i] is therefore equivalent to $a + i * w$.
  - In fact, C and C++ define addition of integers to pointers `a+i` as meaning $a + i * w$.
- Multi-dimensioned arrays are not supported directly. But you can use arrays of pointers to similar effect.



Array Bounds

In Pascal, array bounds were part of the type.
Early Pascal did not allow you to write something like

```
procedure Sort (
   A: in out array[<>] of T;
   N: in integer); — Ada
```

Instead, you had to do

```
procedure Sort (
   A: var array[1..1000] of T;
   N: Integer);  {Pascal}
```

Array Bounds Checking

An access `A[i]`, where `i` lies outside the declared array bounds, can access/alter arbitrary blocks of memory.

Different languages deal with this in different ways.

C, C++: Let the programmer deal with it.

Pascal, Ada: Run-time code checks the index for legality. Good compilers will recognize when these checks can be eliminated during optimization:

```
procedure Sort (
    A: in out array[<>] of T;
    N: in int) is  -- Ada
begin
   for i in A'range loop
     ... A[i] ...
   end loop;
end Sort;
```

Assignment of Arrays

Given two arrays, A and B, of the same type and size, what does A := B (or A = B) do?

Pascal not allowed

Ada B is copied into A

C, C++ Usually not allowed. When it is, then it alters the base address of A to share the same memory as B.

2.5 Records

Records (**structs** in C/C++) gather together a set of named **fields** (**members**).

```
TYPE
  PersonelRecord = RECORD
    Name      : STRING;
    LocalAddr : Address;
    HomeAddr  : Address;
    Salary    : Dollars;
  END;
```

The '.' operator is used to select fields:

```
My.Salary := 1.1 * Jones.Salary;
```

'.' is a somewhat odd operator - we'll revisit it later when we talk about "scope".

2.5.1 Record Layout

Records are generally laid out, field by field, sequentially in memory.

- Alignment rules may force padding between fields, as with arrays.
- Some compilers may try to rearrange fields to reduce padding.
  - Some programmers try hard to arrange fields in decreasing order by size.

2.6 Unions

Unions or **variant records** allow an object to contain different types of data within the same address.

E.g., we want to write a lexical analysis program for a typical HLL. Each call to `lex()` returns a `Token`. Every `Token` contains an identifying code, plus other information:

- identifiers have a pointer to lexeme string
- integer and floating point constants have the numeric value
- string constants have a pointer to the string value
- reserved words have no other data.

In C, we can do the following:

```
union TokenData {
  char*  id;
  int    integerConst;
  double floatConst;
  char*  stringConst;
};

struct Token {
  int       code;
  TokenData data;
};
```

Code to use this might look like:

```
Token t;

t = lex();
switch t.code {
  case 1 /* id */:
    strcpy (lexeme, t.data.id);
    break;
  case 2 /* integer */:
    value = t.data.integerConst;
    break;
  ⋮
}
```

A union, by itself, is hard to work with.

- Usually packaged into a struct, one field of which is a **tag**
  - Tag indicates which option of the union is in effect.

Other languages made the union-within-a-struct a single construct: the variant record.

```
TYPE    {Pascal}
   TokenCodes = 1..35;
   Token = RECORD
     CASE code: TokenCodes OF
        1: (id: Lexeme);
        2: (IntegerConst: Integer);
        3: (FloatConst: Real);
        4: (StringConst: String)
     END
   END;
```

Ada has a very similar syntax.

Unions and Type Safety

Unions and variant records can be a major weakness in the type system. For example,

```
  T: Token;
begin
  T.code := 3;
  T.FloatConst := 3.14159;
  T.code := 2;
  writeln (T.IntegerConst);
```

Ada made variant records type safe by requiring:

- Every variant record must have a tag.
- A tag can be changed only if the variant fields are reassigned in the same statement.

2.7 Pointers

A **pointer** provides indirect access to other objects. Key operations on pointers are

allocation: Obtain a block of memory from the heap, returning a pointer to it. (`new` in Pascal, Ada, and C++, `malloc` in C)

deallocation: Signal that we are done with a previously allocated block of memory. (`delete` in Pascal and C++, `free` in C)

dereferencing: Obtain the object that is referenced by the pointer. (ˆ in Pascal, ∗ in C and C++)

A typical memory structure for a program is

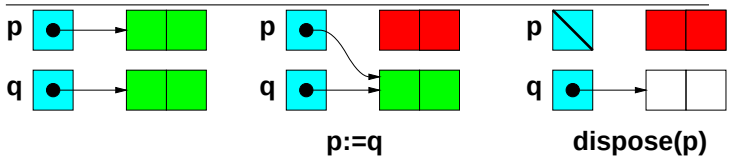
- The **static** memory area holds objects whose lifetime is equivalent to that of the program execution. Typically these are global variables.
- The **dynamic stack** holds objects whose lifetime is bound to some function/procedure activation.
- The **heap** is a data area holding items created dynamically and whose lifetime may extend beyond the subroutine activation in which it was created.

Allocation and deallocation usually occur only in the heap.
Most languages allow pointers to reference only objects in the heap. (C and C++ are exceptions.)

Pointers allow for data structures whose capacity can be determined dynamically, up to the limits of available memory.

They also lead to some well-known programming problems.

- **Dangling pointers** are pointers to memory areas that have been deallocated. Attempts to reference these (or to deallocate them a second time) have unpredictable results.
- **Memory leaks** occur when areas of heap memory are no longer reachable via any pointer but are not deallocated.
  - Such areas are called **garbage**.



Some languages (e.g., LISP, Java) use automatic **garbage collection** to collect garbage.

- Such languages usually do not have a `delete` command, so both garbage and dangling pointers are eliminated.
- Many programmers regard garbage collection as prohibitively expensive.
  - But most programmers only know garbage collection algorithms that are over 30 years old.
  - Modula 3 provides garbage collection by default, but programmers can turn it off for selected pointer types.

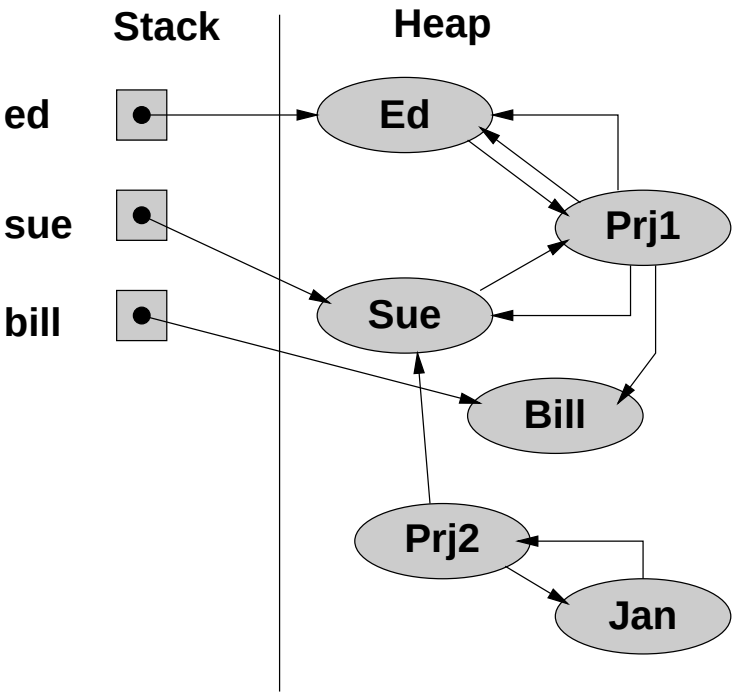
A Basic Garbage Collection Algorithm

Assume that

- we can find all pointers in the static and dynamic stack areas of memory.
- given a pointer to an object on the heap, we can find all pointers that it contains
- each object on the heap has a "mark" bit available for our use

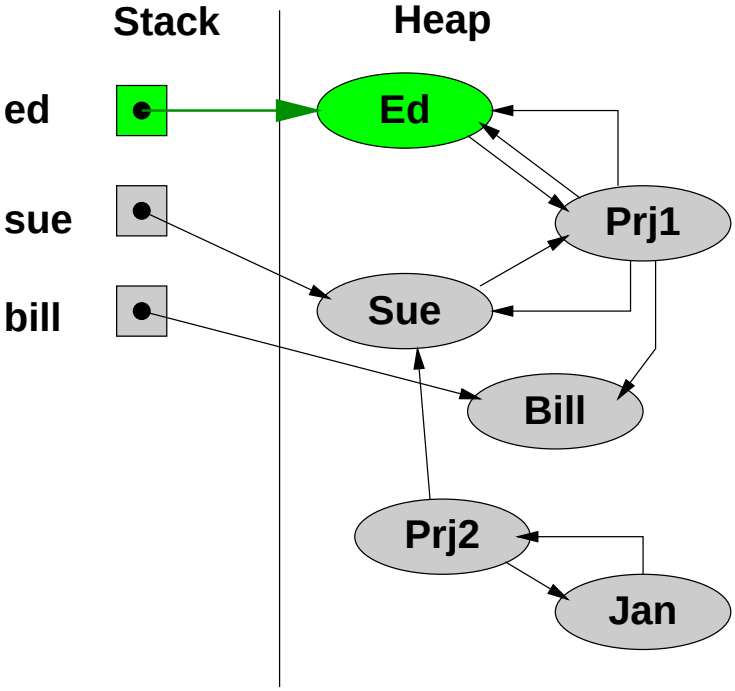
```
procedure mark (pointer p)
  if *p is not already marked then
    set mark bit for *p;
    for all pointers q inside *p loop
      mark (q);
    end loop;
  end if;
end mark;
```

```
procedure collector ()
  // Mark phase
  for all pointers p on the
      run-time stack or in the
      static data area do
    mark(p);
  end loop;
  // Sweep phase
  for all objects x on the heap do
    if x is not marked then
      delete &x;
    else
      clear x's mark bit;
    end if;
  end loop;
end collector;
```

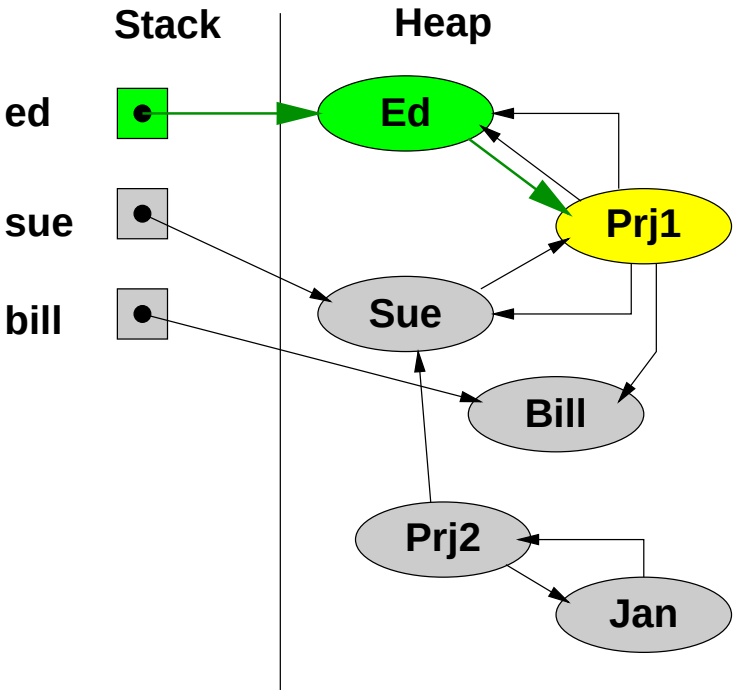


Move down the stack, `mark`ing each pointer, starting with `ed`.

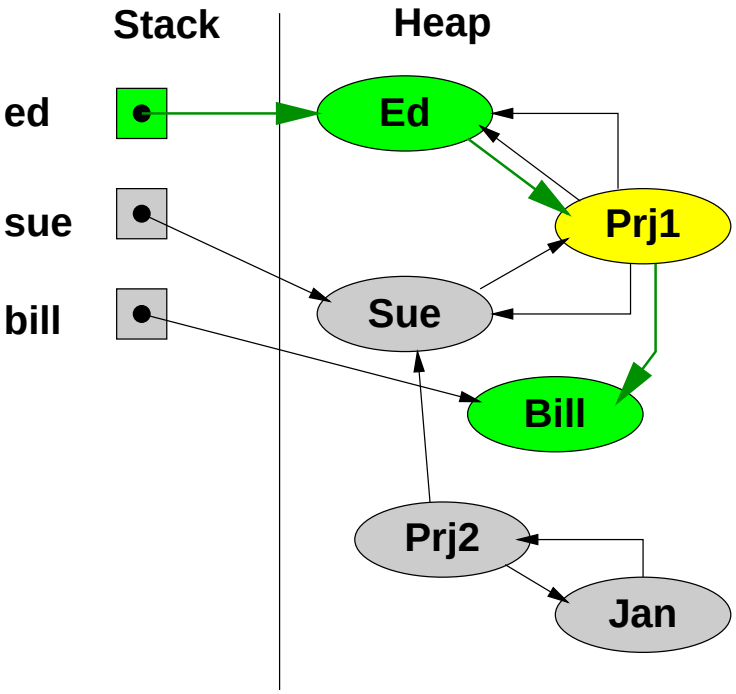
```
mark(ed)
```

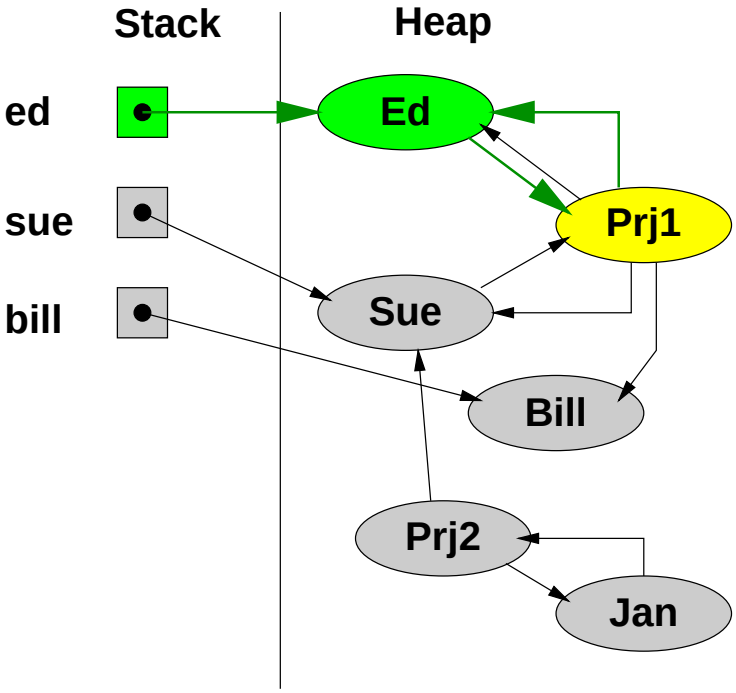


`Ed` is marked.

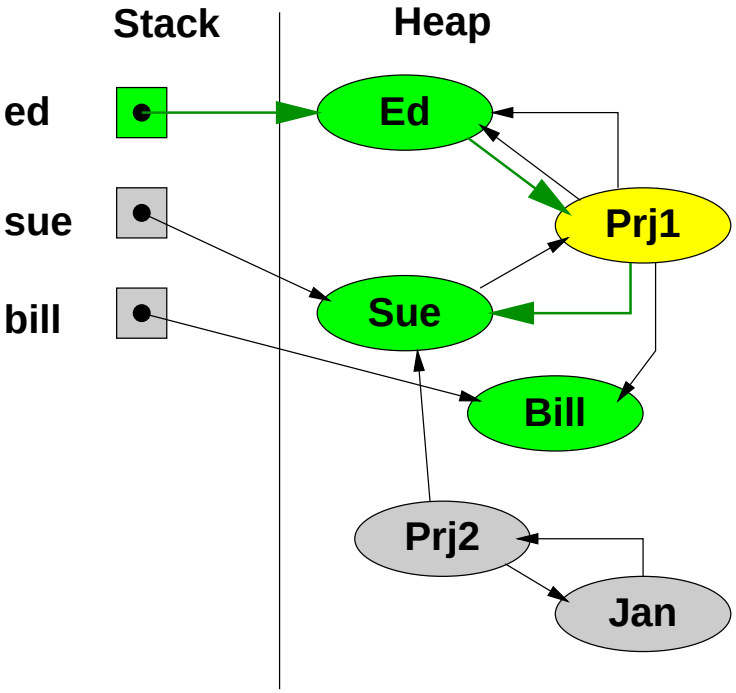


Then we recursively mark each pointer in `Ed`. `Prj1` is marked, and we proceed to mark each of its pointers.

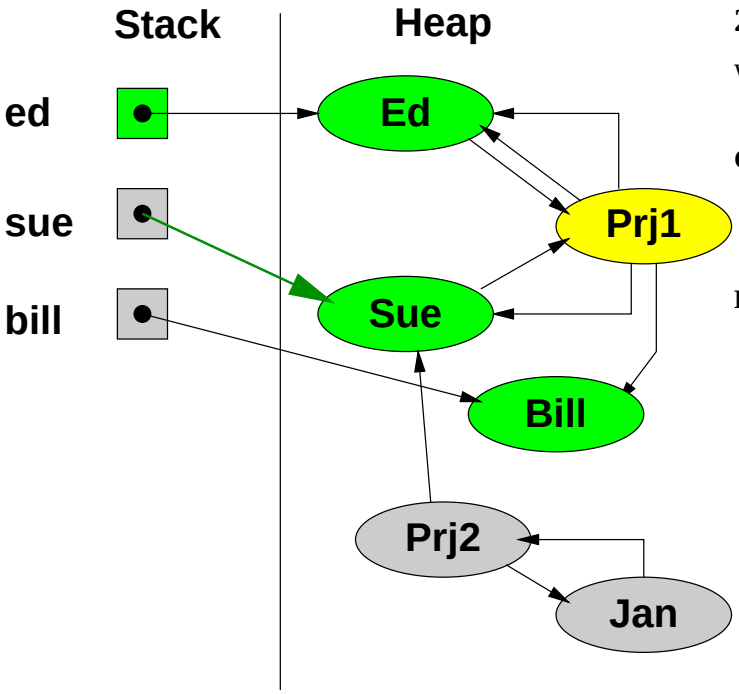




`Ed` is already marked, so `mark(p)` for the pointer from `Prj1` to `Ed` has no effect.



Eventually, we return from the recursive calls and move on to the next pointer on the stack.



2.8 References

What does the assignment `x = y` mean?
There are two common answers:

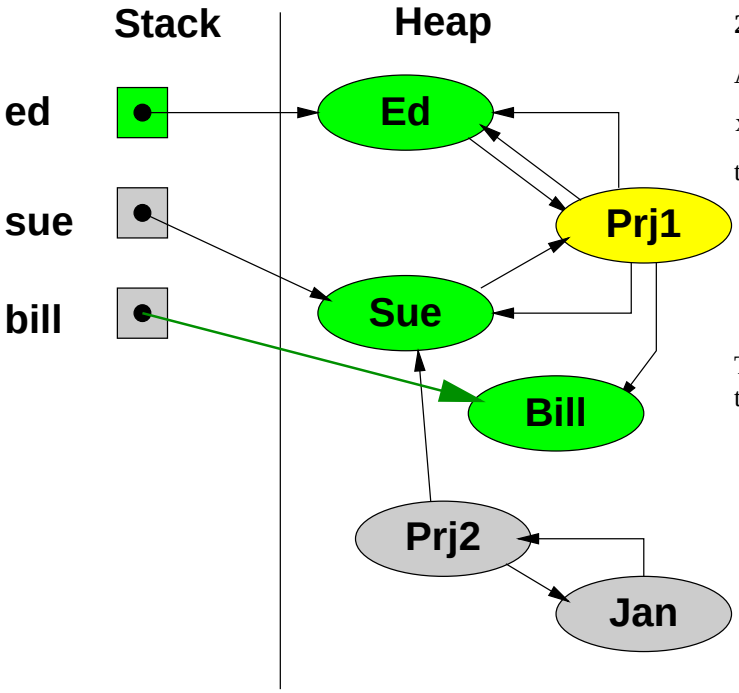
Copy Semantics: The value of `y` is copied into the location of `x`

- A subsequent change to either `x` or `y` will not affect the other.

Reference Semantics: `y` is altered so that it refers to the same location as `x`.

- A subsequent change to either `x` or `y` may affect the other as well.

- Pascal, C, and Ada use copy semantics.
- Copy semantics are the default in C++.
- LISP uses reference semantics.
- Java uses copy semantics for basic types, reference semantics for all non-basic types.



With the mark phase complete, we sweep the heap, collecting unmarked objects (`Prj2` & `Jan`).

2.8.1 Copy Semantics

Although the typical assignment statement looks symmetrical:

```
x = y;
```

the two operands are treated very differently.

- The left operand is used to provide a *location* where a value can be stored.
- The right operand is used to provide a *value* to store there.

These are sometimes referred to as **l-values** and **r-values**, respectively.

Some expressions can yield either an r-value or an l-value, e.g.:

- `x`
- `A[i]`
- `rec.field`

Others can only produce an r-value, e.g.:

- `x+y`
- `23`

- `foo(x)`
  - In most languages any user-written function can only return r-values.
  - C++ is an exception. The "reference type" in C++ is an l-value.

2.8.2 Reference Semantics

A language that uses reference semantics for assignment tends to view all variables as "implicit" pointers.

(Such languages often do not have an explicit pointer type.)

```
/* Java */
class A {
  int i;
}


// C++
struct A {
  int i;
};
```

What are `x.i` and `y.i` after each of the following assignments?

```
A x, y;

x.i = 1; y.i = 2;
x = y;
x.i = 3;
y.i = 4;
```

	Java		C++	
	x.i	y.i	x.i	y.i
`x.i = 1; y.i = 2;`	1	2	1	2
`x = y;`	2	2	2	2
`x.i = 3;`	3	3	3	2
`y.i = 4;`				

	Java		C++	
	x.i	y.i	x.i	y.i
`x.i = 1; y.i = 2;`	1	2	1	2
`x = y;`	2	2	2	2
`x.i = 3;`	3	3	3	2
`y.i = 4;`	4	4	3	4

2.8.3 Copying Variations

Related to the copy/reference semantics is the question of what to do when copying an object that contains pointers:

Shallow copy copy the pointer value

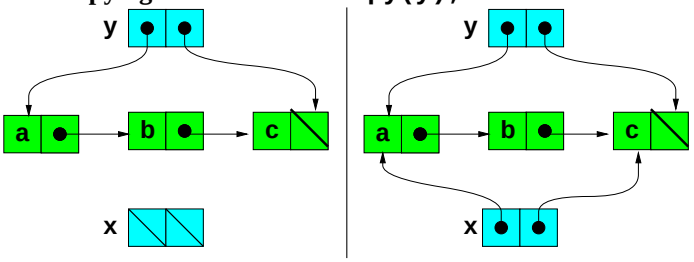
Deep copy Copy the object pointed to, placing a pointer to the new copy in this object.

```
struct Node {
  string data;
  Node* next;
  ⋮
};

class List {
  Node* first;
  Node* last;
public:
  ⋮
};
```

Shallow Copying

```
struct Node {
  string data;
  Node* next;
  ⋮
};

class List {
  Node* first;
  Node* last;
public:
  ⋮
  List shallowCopy (List x)
  {
    first = x.first;
    last = x.last;
  }
};
```

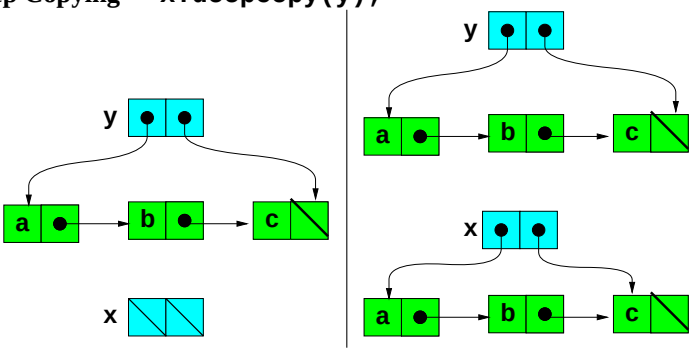
Shallow Copying — `x.shallowCopy(y);`



Shallow Copying

```
class List {
  Node* first;
  Node* last;
public:
  ⋮
  List deepCopy (List x)
  {
    eraseAll();
    first = last = 0;
    for (Node* p = x.first; p != 0;
         p = p->next)
      {
        Node* q = new Node;
        q->data = p->data;
        q->next = 0;
        if (first = 0)
          first = last = q;
        else
          last->next = q;
        last = q;
      }
  }
};
```

Deep Copying — `x.deepCopy(y);`



Languages with copy semantics all perform shallow copying, which is not always what is desired.

- One reason why C++ allows programmers to implement their own assignment operators.
- Deep copying is not always a simple enough algorithm for automatic compiler generation.