CS 330 Lecture 18

- Chapter 5 Louden
- Outline
  - The symbol table
  - Static scoping vs dynamic scoping

Symbol table

- Dictionary associates names to attributes
- In general: hash tables, tree and lists (assignment 3) can be used
- Lexically scoped language with block structure
  - C, Pascal, Ada, (Java, C++) etc.
  - Needs stack like operation (entry-exit) from block

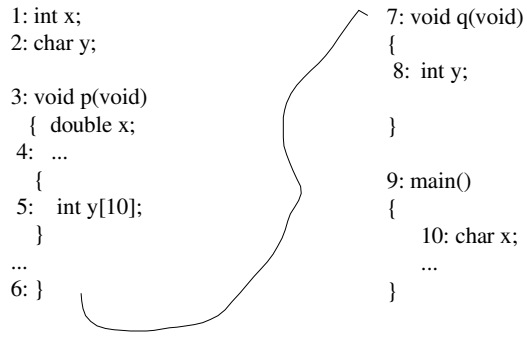
Declarations

- Establishing binding
  - int x; double f(int x);
- compound statements
  - { }
  - begin end
  - class declarations
- scope of binding is the region of program were a binding is maintained

C scope rules

- lexical score
  - scope of binding is limited to the block in which the associated declaration appears
  - declaration before use rule
- scope hole
- visibility vs score
- scope resolution operator

Lexical Scope An example



Symbol Table

Label 4:
x -> double local to p -> int global
y -> char global
p -> void function

Label 5:
x -> double local to p -> int global
y -> int array local to nested block in p -> char global
p -> void function

Label 6:
x -> int global
y -> char global
p -> void function

Dynamic vs Static Scoping

```
1: int x = 1;
2: char y = 'a';
3: void p(void) {
4:  double x = 2.5;
5:  printf(%c\n", y);
6:  { int y[10]; }}

7: void q(void) {
8:   int  y = 42;
9:   printf("%d\n", x); p();}
10:
11: main() { char x = 'b';
12:         q(); return 0;}
```

Line 11:
x -> char = b local to main -> int = 1 global
y -> char = a global

Line 12:
x -> char = b local to main -> int = 1 global
y -> int = 42 local to q -> char = a global

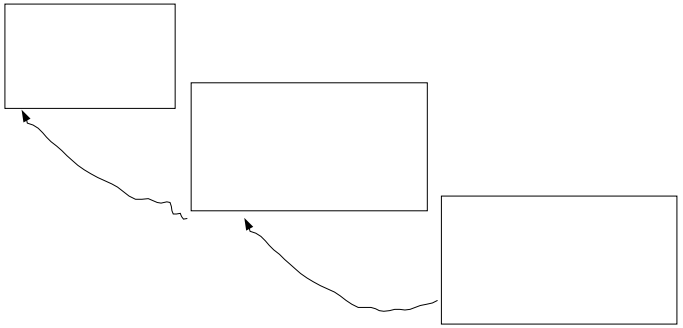
Line 9:
x -> double = 2.5 -> char = b local to main -> int 1 global
y -> int 42 -> char a global

Static scoping output: 1 a
Dynamic scoping : 92(b) *(42)

Problems with dynamic scoping

- Semantics are based on program execution not reading
- static typing and dynamic scoping can't coexist
- Maintaining lexical scope in interpreter hard
  - Scheme, ML
- Dynamic scoping easier to implement
  - APL, Snobol, (old Perl), (old LISP)

Nested symbol tables



Name resolution & overloading

- ad-hoc polymorphism (the + operator) vs parametric polymorphism (the list length function)
- C++, Ada = overloading of operators, functions
- Java = overloading of functions
- Haskell = overloading of operators, functions plus new operators

How can overloading be done ?

- Extend lookup with calling context
- Still complex situations can arise max(2.5, 3) ?
- Java : only lossless coersion
- Different namespaces (Java, ML)

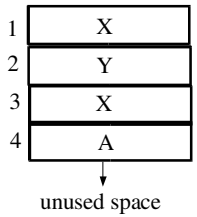
The environment

- Bindings of names to locations
- Fortran – static environment
- Lisp - dynamic environment
- Most languages – combination
- Some names don't need location
  - const int MAX = 10;

Compilers vs Interpreters

- Compilers: symbol table what allocation code to generate as declaration is processed
- Interpreters: symbol table and environment are combined
- Typically globals are allocated statically, locals dynamically

Stack

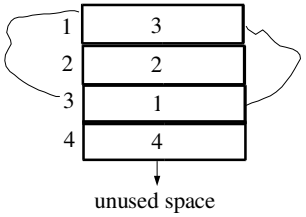


Environment = linear sequence of memory cells

what about if I call a function p many times ?
Activation records

Pointers

- Is a storage location whose stored value is a reference to another object



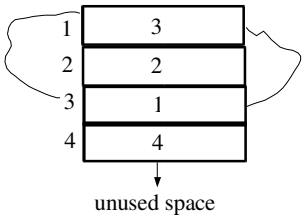
In C: int *x;

causes allocation of a pointer variable, but NOT the allocation of a object to which x points

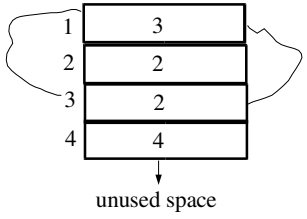
Convention: 0 or NULL
Java: null, Pascal nil

More pointers

- *x = 2;
  - the value pointed by x (a pointer variable) is 2



int *x;
x = 3;
*x = 2;



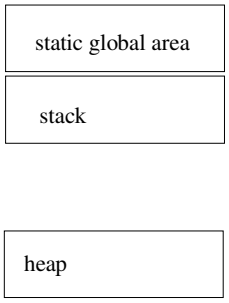
Anonymous pointers

- void* x; (anonymous pointer variable x)
- x = (int) **malloc**(sizeof(int));
- Allocate a block of memory that fits an integer
- Dereferencing operator * (*x)
- Pointer type is also confusingly * (for example int* or float*)
- free(x);

Dynamic allocation – Heap

- Memory used for calls to malloc, free is called the heap
- In C, C++ manual allocation is possible
- Java and ML don't allow allocation
- Static, Dynamic, Stack-based and Heap allocation

Memory layout

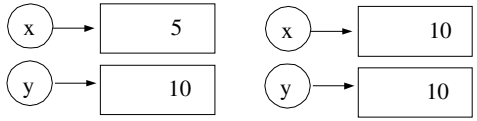


Heap storage can be released anywhere leaving "holes". Simple stack doesn't work. Functional languages automatically manage the heap. Java allows heap allocation but not deallocation.

Manual control of the heap results in very few cases in more efficient code but invites all kinds of unsafe operations.

Variables – Storage semantics

- Value can be changed during execution
- name – location – value
- x = y



l-value, r-value

- x = y
- x is the name of a location of a variable
- y is the value of the variable named y
- In ML distinction explicit:
  - x := !x +1;
  - x := !y;

"Address of"operator in C

- int x;
- &x is the address of x and can be assigned to a pointer;
- For example:

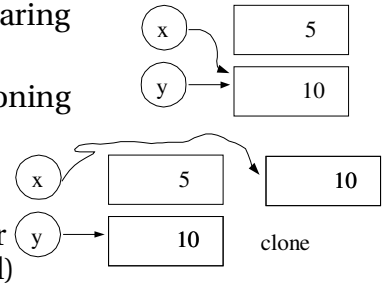
```
int x;
x = 10;
int *y = &x;
int z = *y;
int k = &y; (what does this one do ?)
```


The swamp of C

- Address arithmetic (pointers can be added subtracted like integers)
- mixing dereferencing and address of operators expressions and assignment can lead to some very confusing and complex situations

Pointer semantics

- Assignment by sharing
- Assignment by cloning done in Java by implicit pointers
  *x = *y
  (pointers under under the hood)



Value semantics – constants

- No location just a value
- Not necessarily known at compile time once computed never updated
- Examples: ML, Single assignment C
- In Java, keyword final is used for constants (gets only one final value) and static can be used when value can be computed prior to execution.

Function Definitions

- In virually all languages functions are essentially constants whose values are functions
- In ML: val square = fn(x:int) => x * x;

Function Pointers in C

```
int gcd(int u, int v)
{
    if (v == 0) return u;
    else return gcd(v, u % v);
}

/* function variable – pointer syntax necessary otherwise prototype */
int (*gcdv)(int, int) = gcd;

/* can be called */

gcdv(15,10)
```


Aliases

- Same thing bound to two different names at the same time

```
int *x, *y;
x = (int *) malloc(sizeof(int));
*x = 1;
y = x;
y = 2;    /* changes x although x doesn't appear in the assignment */
printf("%d\n", *x);
```

Dangling references

- Location that has been dellocated from the environment but can still be accessed
- pointer to a deallocated object:

```
int *x, *y;
x = (int *) malloc(sizeof(int));
*x = 2;
y = x;
free(x);
printf("%d\n", *y);
```


Garbage

- Eliminate dangling reference by never deallocating
- Garbage only wastes memory doesn't corrupt the program behavior

```
int *x;
...
x = (int *) malloc(sizeof(int));
x = NULL;
```


Garbage collection

- Lisp, Smalltalk, Java
- ML has a very efficient garbage collector
- There is a lot of interesting work in how to implement garbage collectors – some of you may learn about it when you write a Compiler