Principles of programming languages

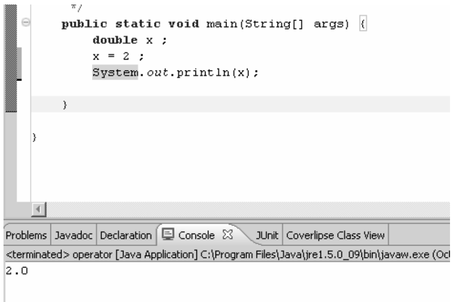
Lecture 6

http://few.vu.nl/~nsilvis/PPL/2007

Natalia Silvis-Cividjian
e-mail: nsilvis@few.vu.nl

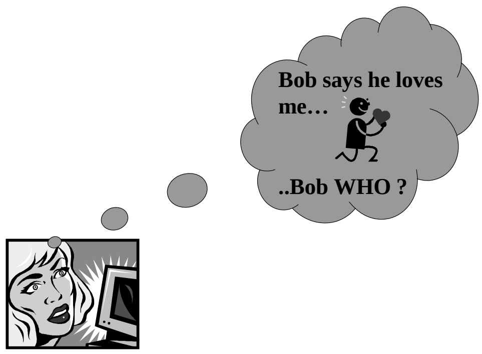
vrije Universiteit *amsterdam*

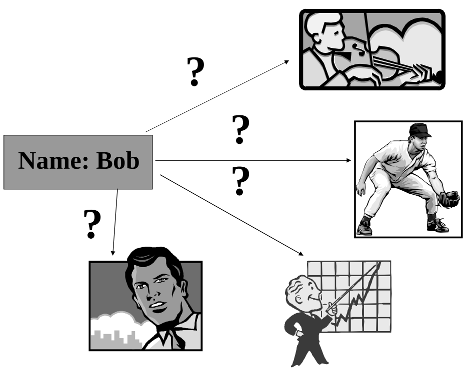
From last week



Outline

Part I. Scope
Part II. Memory management





Names are not unique

n
square

ML:

```
fun square n = n * n;
fun square square = square * square
```

C++:

temp

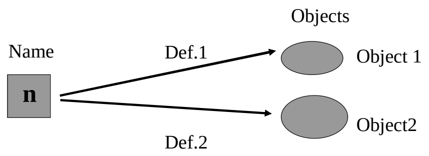
```
int temp = 50 ;

int main(){
     for (int j=1; j<10; j++)
     {
     int temp = 10*j ;
     cout << “temp=" << temp << endl ;
     }

return 0 ;
}
```

Definition

A **definition** is anything that establishes a possible binding for a name. A name can have many definitions.



The scope of definition

An occurrence of a name is *in the scope of* a given *definition* of that name whenever that definition governs the binding for that occurrence

What is *the scope of x* ? Not correct

What is *the scope of this* ***definition*** *of x* ? OK

- Problem: Names are not unique (Bob who?)
- Scope of definition must be unambiguous
- We present a few possible solutions

Approaches

- Scoping with blocks
- Scoping with labeled namespaces
- Scoping with primitive namespaces
- Dynamic scoping

1. Blocks

- A block is any language construct that contains definitions, and also contains the region of the program where those definitions apply

```
let
  val x = 1;
  val y = 2;
in
  x+y
end
```

Different ML Blocks

- The `let` is just a block: no other purpose
- A `fun` definition includes a block:

```
fun cube x = x*x*x;
```

- Multiple alternatives have multiple blocks:

```
fun f (a::b::_) = a+b
|   f [a] = a
|   f [] = 0;
```

- Each rule in a match is a block:

```
case x of (a,0) => a | (_,b) => b
```

Java Blocks

- In Java and other C-like languages, you can combine statements into one *compound statement* using braces **{ }**
- A compound statement also serves as a block:

```
while (i < 0) {
  int c = i*i*i;
  p += c;
  q += c;
  i -= step;
}
```

Nested blocks

- What happens if a block contains another block, and both have definitions of the same name?

```
let
  val n = 1
in
  let
    val n = 2
  in
    n
  end
end
```

Classic Block Scope Rule

The scope of a definition is

the block containing that definition, from the point of definition to the end of the block

minus

the scopes of any redefinitions of the same name in interior blocks

ML example

Scope of this definition is A-B

```
let
  val n = 1          A
in
  let
    val n = 2        B
  in
    n
  end
end
```

Here n is 1

Here n becomes 2

Here is n evaluated,

So the result is n=2

Scope of this definition is B

C++ example

```
#include <iostream>
using namespace std ;

int x = 50 ;

int main(){
      for (int j=1; j<10; j++)
      {
            cout << "step: " << j << endl;
            cout << “global x inside” << x ;
            int x = 10*j ;
            cout << "x=" << x << endl ;
      }
cout << "global x outside" << x ;
return 0 ;
}
```

C++ example

```
#include <iostream>
using namespace std ;

int x = 50 ;                                          A

int main(){
      for (int j=1; j<10; j++)
      {
            cout << "step: " << j << endl;
            cout << “global x inside = ” << x ;
            int x = 10*j ;
            cout << “ x = " << x << endl ;          B
      }
cout << "global x outside = " << x ;
return 0 ;
}
```

```
Output is:
step: 1
global x inside = 50 x=10
step: 2
global x inside = 50 x=20
step: 3
global x inside = 50 x=30
step: 4
global x inside = 50 x=40
step: 5
global x inside = 50 x=50
step: 6
global x inside = 50 x=60
step: 7
global x inside = 50 x=70
step: 8
global x inside = 50 x=80
step: 9
global x inside = 50 x=90
global x outside = 50
```

2. Labeled Namespaces

- A **labeled namespace** is any language construct that contains definitions and a region of the program where those definitions apply, and also has a name that can be used to access those definitions from outside the construct

Labeled Namespaces

- ML has **`structure`**
- Namespaces that are just namespaces:
  - C++ has **`namespace`**
    `Ex: using namespace std;`
  - Java has **`package`**
    `Ex: package java.util`
    `import java.util.*`
- Namespaces that serve other purposes too:
  - Class definitions in class-based object-oriented languages

ML Structures

```
structure Fred = struct
  val a = 1;
  fun f x = x + a;
end;
```

- A little like a block: **`a`** can be used anywhere from definition to the end
- But the definitions are also available outside, using the structure name: **`Fred.a`** and **`Fred.f`**

Java classes

```
public class Month {
  public static int min = 1;
  public static int max = 12;
  …
}
```

- The variables **`min`** and **`max`** would be visible within the rest of the class
- Also accessible from outside, as **`Month.min`** and **`Month.max`**
- Classes serve a different purpose too

Namespace Advantages

- Two conflicting goals:
  - Use memorable, simple names like **`max`**
  - For globally accessible things, use uncommon names like **`maxSupplierBid`**, names that will not conflict with other parts of the program
- With namespaces, you can accomplish both:
  - Within the namespace, you can use **`max`**
  - From outside, **`SupplierBid.max`**

3. Primitive namespaces

```
- val int = 3;
val int = 3 : int
```

- In ML it is legal to have a variable named **`int`**
- You can even do this (ML understands that **`int*int`** is not a type here):

```
- fun f int = int*int;
val f = fn : int -> int
- f 3;
val it = 9 : int
```

Primitive Namespaces

- ML's syntax keeps types and expressions separated
- ML always knows whether it is looking for a type or for something else
- There is a separate namespace for types

```
fun f(int:int) = (int:int)*(int:int);
```

These are in the ordinary namespace

These are in the namespace for types

Primitive Namespaces

- They are part of the language definition
- Some languages have several separate primitive namespaces
- Java: packages, types, methods, fields, and statement labels are in separate namespaces

Examples

C++ : `int int;` not allowed

ML : `fun int int = int * int` : OK

Java: this example is possible:

```
class Reuse {
     Reuse Reuse(Reuse Reuse) {
     Reuse:
       for (;;) {
     if (Reuse.Reuse(Reuse) == Reuse)
          break Reuse ;
     }
     return Reuse ;
}
```

(from Arnold&Gosling, The Java Programming language)

4. Dynamic Scoping

- Until now all scoping was static = at compile time.
- Each function has an environment of definitions
- If a name that occurs in a function is not found in its environment, its *caller's* environment is searched
- And if not found there, the search continues back through the chain of callers
- This generates a rather odd scope rule…

Classic Dynamic Scope Rule

- The scope of a definition is the function containing that definition, from the point of definition to the end of the function, along with any functions when they are called (even indirectly) from within that scope — minus the scopes of any redefinitions of the same name in those called functions

Static Vs. Dynamic Scoping

- The scope rules are similar
- Both talk about *scope holes*—places where a scope does not reach because of redefinitions
- But the static rule talks only about regions of program text, so it can be applied at compile time
- The dynamic rule talks about runtime events: “functions when they are called…”

Example

```
fun g x =
  let
    val inc = 1;
    fun f y = y+inc;
    fun h z =
      let
        val inc = 2;
      in
        f z
      end;
  in
    h x
  end;
```

What is the value of **`g 5`** using ML’s classic block scope rule?

Block (static) Scope

```
fun g x =
  let
    val inc = 1;
    fun f y = y+inc;
    fun h z =
      let
        val inc = 2;
      in
        f z
      end;
  in
    h x
  end;
```

With block scope, the reference to **`inc`** is bound to the previous definition in the same block. The definition in **`f`**’s caller’s environment is inaccessible.

`g 5` = 6 in ML

Dynamic Scope

```
fun g x =
  let
    val inc = 1;
    fun f y = y+inc;
    fun h z =
      let
        val inc = 2;
      in
        f z
      end;
  in
    h x
  end;
```

With dynamic scope, the reference to **`inc`** is bound to the definition in the caller’s environment.

`g 5` = 7 if ML used dynamic scope

Where It Arises

- Only in a few languages: some dialects of Lisp and APL
- Available as an option in Common Lisp
- Drawbacks:
  - Difficult to implement efficiently
  - Creates large and complicated scopes, since scopes extend into called functions
  - Choice of variable name in caller can affect behavior of called function

Outline

Part II. Memory management

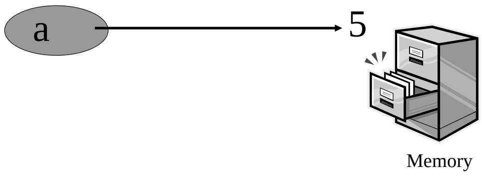
There is a connection between variables and values in memory.

Imperative languages: connection is obvious

```
Java:  a = 5
```

Functional languages: hidden, but it exists

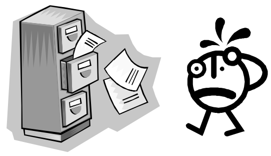
```
ML: let a = 5
```



Imagine:

Values “have to be stored” in memory.
Memory space “has to be allocated/deallocated”.

and YOU are the memory manager.



The life of a memory manager

- Variables allocation
- Activation records allocation
- Heap management
- Cleaning up the mess: garbage collection

Variables allocation

- Activation-specific (automatic) = variable lives only during one execution (activation) of a function
- Static = variable lives during all the program execution in one memory location

The life of a memory manager

- Variables allocation
- Activation records allocation
- Heap management
- Cleaning up the mess: garbage collection

Activation record

= a memory block containing all the activation-specific variables of a function, like:

arguments and returns
local variables
temporary values
return address

Activation records allocation

- Static allocation
- Dynamic allocation (stack of frames)

Static Allocation

- The simplest approach: allocate one activation record for every function, statically (at compile time), always the same address
- Older dialects of Fortran and Cobol used this system
- Simple and fast

Old style FORTRAN Example

```
      FUNCTION AVG (ARR, N)
      DIMENSION ARR(N)
      SUM = 0.0
      DO 100 I = 1, N
        SUM = SUM + ARR(I)
100   CONTINUE
      AVG = SUM / FLOAT(N)
      RETURN
      END
```

N address
ARR address
return address
I
SUM
AVG

Drawback

- Each function has one activation record
- There can be only one activation alive at a time = recursion is not possible
- Modern languages (including modern dialects of Cobol and Fortran) do not obey this restriction

Dynamic allocation:stacks

- Dynamic allocation: activation record allocated when function is called
- Allocate a new activation record for each activation
- For many languages, like C, it can be deallocated when the function returns
- A stack of activation records: *stack frames* pushed on call, popped on return
- Recursion is now possible

Current Activation Record

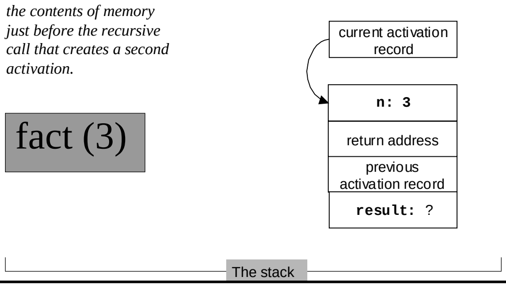
- Before, static: location of activation record was determined before runtime
- Now, dynamic: location of the *current* activation record is not known until runtime
- A function must know how to find the address of its current activation record
- Often, a machine register is reserved to hold this

C Example

```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```

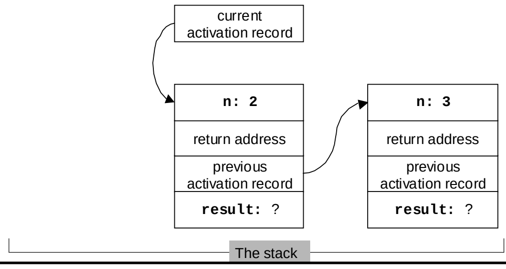
We are evaluating **`fact(3)`**. *This shows the contents of memory just before the recursive call that creates a second activation.*

fact (3)



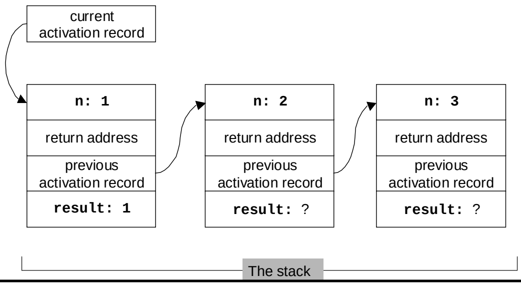
This shows the contents of memory just before the third activation.

```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



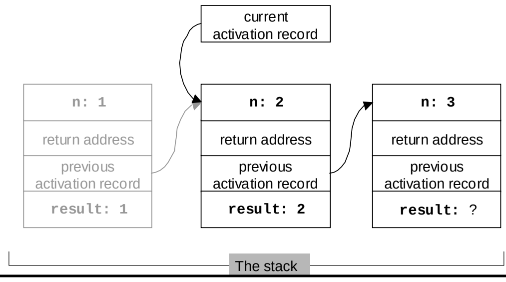
This shows the contents of memory just before the third activation returns.

```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



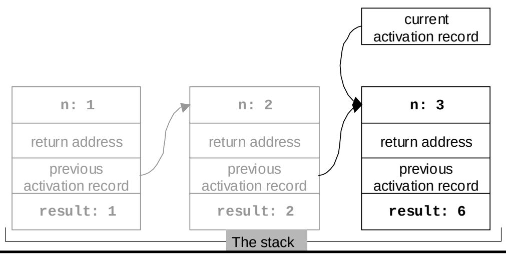
The second activation is about to return.

```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



The first activation is about to return with the result **`fact(3) = 6`**.

```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



Nesting Functions

- What we just saw is adequate for many languages, including C
- But not for languages that allow this trick:
  - Function definitions can be nested inside other function definitions
  - Inner functions can refer to local variables of the outer functions (under the usual block scoping rule)
- Like ML, Ada, Pascal, etc.
- A new item in the activation record: nesting link

The life of a memory manager

- Variables allocation
- Activation records allocation
- Heap management
- Cleaning up the mess: garbage collection

The heap

- A heap is a pool of blocks of memory used for dynamic **unordered** memory allocation/deallocation
- Used for example for `malloc` and `free` in C, `new` and `delete` in C++, dynamically resized objects like `list` and `set` in ML.
- Heap management is a trade-off between space and speed

How to manage the heap?

First-Fit mechanism

- The manager maintains a linked list of free blocks, initially containing one big free block
- To allocate:
  - Search free list for first sufficiently large block
  - If there is extra space in the block, return the unused portion at the upper end to the free list
  - Allocate requested portion (at the lower end)
- To free, just add to the front of the free list

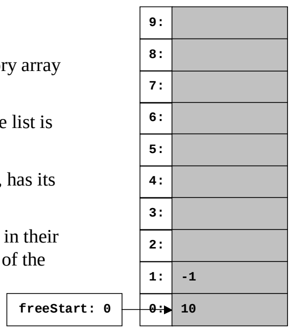
Heap example

A heap manager with a memory array of 10 words, initially empty.

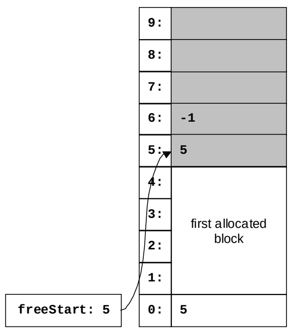
The link to the head of the free list is held in **`freeStart`**.

Every block, allocated or free, has its length in its first word.

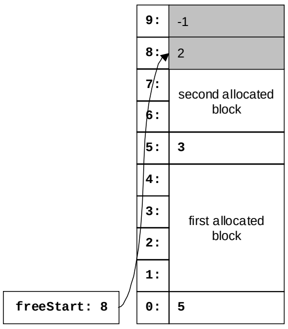
Free blocks have free-list link in their second word, or –1 at the end of the free list.



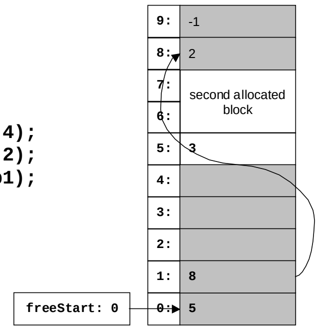
`p1=allocate(4);`



```
p1=allocate(4);
p2=allocate(2);
```

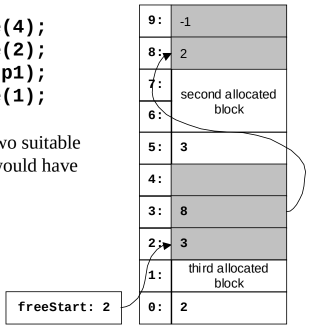



```
p1=allocate(4);
p2=allocate(2);
deallocate(p1);
```




```
p1=m.allocate(4);
p2=m.allocate(2);
m.deallocate(p1);
p3=m.allocate(1);
```

Notice that there were two suitable blocks. The other one would have been an exact fit.



Coalescing free blocks

- Consider this sequence:
  ```
  p1=allocate(4);
  p2=allocate(4);
  deallocate(p1);
  deallocate(p2);
  p3=allocate(7);
  ```
- Final **`allocate`** will fail: we are breaking up large blocks but never reversing the process
- Need to *coalesce* adjacent free blocks

Eager coalescing

- Maintain the free list sorted in address order
- When freeing, look at the previous free block and the next free block
- If adjacent, coalesce

Quick Lists

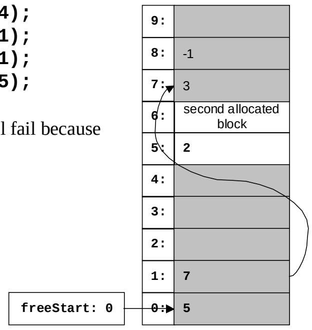
- Small blocks tend to be allocated and deallocated much more frequently
- A common optimization: keep separate free lists for popular (small) block sizes
- On these *quick lists*, blocks are one size
- *Delayed coalescing*: free blocks on quick lists are not coalesced right away (but may have to be coalesced eventually)

Fragmentation

- When free regions are separated by allocated blocks, so that it is not possible to allocate all of free memory as one block
- More generally: any time a heap manager is unable to allocate memory even though free
  - If it allocated more than requested
  - If it does not coalesce adjacent free blocks
  - And so on…

```
p1=allocate(4);
p2=allocate(1);
deallocate(p1);
p3=allocate(5);
```

The final allocation will fail because of fragmentation.



Issues in Heap Management

- Three major issues:
  - Placement—where to allocate a block
  - Splitting—when and how to split large blocks
  - Coalescing—when and how to recombine

Placement

- Where to allocate a block
- We saw first fit from a FIFO free list
- Some mechanisms use a similar linked list of free blocks: first fit, best fit, next fit, etc.
- Some mechanisms use a more scalable data structure like a balanced binary tree

Splitting

- When and how to split large blocks
- Our mechanism: split to requested size
- Sometimes you get better results with less splitting—just allocate more than requested
- A common example: rounding up allocation size to some multiple

Coalescing

- When and how to recombine adjacent free blocks
- We saw several varieties:
  - No coalescing
  - Eager coalescing
  - Delayed coalescing (as with quick lists)

Heap related defects

```
void create () {
int *p ;
New (p) ;
*p = 25 ;
}
```

Memory leak: this function allocates a block but forgets to deallocate it, runs out of scope and the pointer is lost.

```
int *p ;
p = new int;
*p =21;
delete (p);
*p = 35 ;
```

Dangling pointer: this fragment uses a pointer after the block it points to has been deallocated.

The life of a memory manager

- Variables allocation
- Activation records allocation
- Heap management
- Cleaning up the mess: garbage collection

Garbage Collection

- Since so many errors are caused by improper deallocation…
- …and since it is a burden on the programmer to have to worry about it…
- …why not have the language system reclaim blocks automatically? This is **garbage collection**

Three Major Approaches

- Mark and sweep
- Copying
- Reference counting

Mark And Sweep

- A mark-and-sweep collector uses current heap links in a two-stage process:
  - *Mark*: find the live heap links and mark all the heap blocks linked to by them
  - *Sweep*: make a pass over the heap and return unmarked blocks to the free pool
- Blocks are not moved

Copying Collection

- A copying collector divides memory in half, and uses only one half at a time
- When one half becomes full, find live heap links, and copy live blocks to the other half
- Compacts as it goes, so fragmentation is eliminated
- Moves blocks

Reference Counting

- Each block has a counter of heap links to it (counts the pointers that refer to an object)
- Incremented when a heap link is copied, decremented when a heap link is discarded
- When counter goes to zero, block is garbage and can be freed (object not useful)

Reference Counting

- Problem with cycles of garbage:
- Problem with performance generally, since the overhead of updating reference counters is high
- One advantage: naturally incremental, with no big pause while collecting

Garbage Collecting Languages

- Some require it: Java, ML
- Some encourage it: Ada
- Some make it difficult, but possible : C, C++

Garbage collection

- Pro
- Contra

Trends

- An old idea whose popularity is increasing
- Good implementations are within a few percent of the performance of systems with explicit deallocation
- Programmers who like garbage collection feel that the development and debugging time it saves is worth the runtime it costs

Conclusion

- We described 3 storage possibilities for objects: static and dynamic (stack and heap).
- Static object live during all the program.
- Stack objects live as long a subroutine is active.
- Heap objects have a less well defined lifetime.