Lectures 10-11

Memory management

Iztok Savnik, FAMNIT

May, 2023.

Literature

- John Mitchell, Concepts in Programming Languages, Cambridge Univ Press, 2003 (Chapter 7)
- Michael L. Scott, Programming Language Pragmatics (3rd ed.), Elsevier, 2009 (Chapters 3 and 8)
- Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, Second Edition, Prentice Hall, Inc., 1988.

Outline

1. Binding time
2. Lifetime and storage management
3. Static allocation
4. Stack allocation
5. Blocks, scope and activation records
6. Heap allocation
7. Explicit memory management
8. Garbage collection

Introduction

- We did talk about these topics but not in a organized way and not in detail
- How and where memory for variables are allocated?
  - Static and dynamic variables
  - Stack allocation and heap allocation
- Implementation view of scope and lifetime of objects
  - Activation records, stack allocation,
  - Local and global variables
  - How to access global variables?

Introduction

- How the chains of function calls are implemented?
  - Stack activation records
  - Structures formed by the activation records
- Manual and automatic allocation of heap storage
  - We did not talk much about this
  - Memory for objects, records, arrays, lists, ...
  - Memory leaks, dangling references, possible bugs
- Heap management strategies and algorithms
  - Price for automatic storage allocation
  - Garbage collection

Binding Time

- A binding is an association between two things, such as a name and the thing it names
- Binding time is time at which a binding is created
  - The time at which any implementation decision is made
  - Binding between question and answer
- Important binding times in SW systems
  - Compile time: Mapping of high-level constructs to machine code, including the layout of statically defined data in memory
  - Link time: Name in one module refers to an object in another module, the binding between the two is not finalized until link time
  - Run time: Bindings of values to variables and other run-time decisions

Binding time

- Early binding times are associated with greater efficiency, while later binding times are associated with greater flexibility
  - Compiler-based language implementations tend to be more efficient than interpreter-based implementations because they make earlier decisions
  - Interpreters must analyze the declarations every time the program begins execution
- The terms static and dynamic are generally used to refer to things bound before run time and at run time

Lifetime and Storage Management

- It is important to distinguish between names and the objects to which they refer
- Key events in object / binding lifetime:
  - Creation of objects
  - Creation of bindings
  - References to variables, subroutines, types, and so on, all of which use bindings
  - Deactivation and reactivation of bindings that may be temporarily unusable
  - Destruction of bindings
  - Destruction of objects

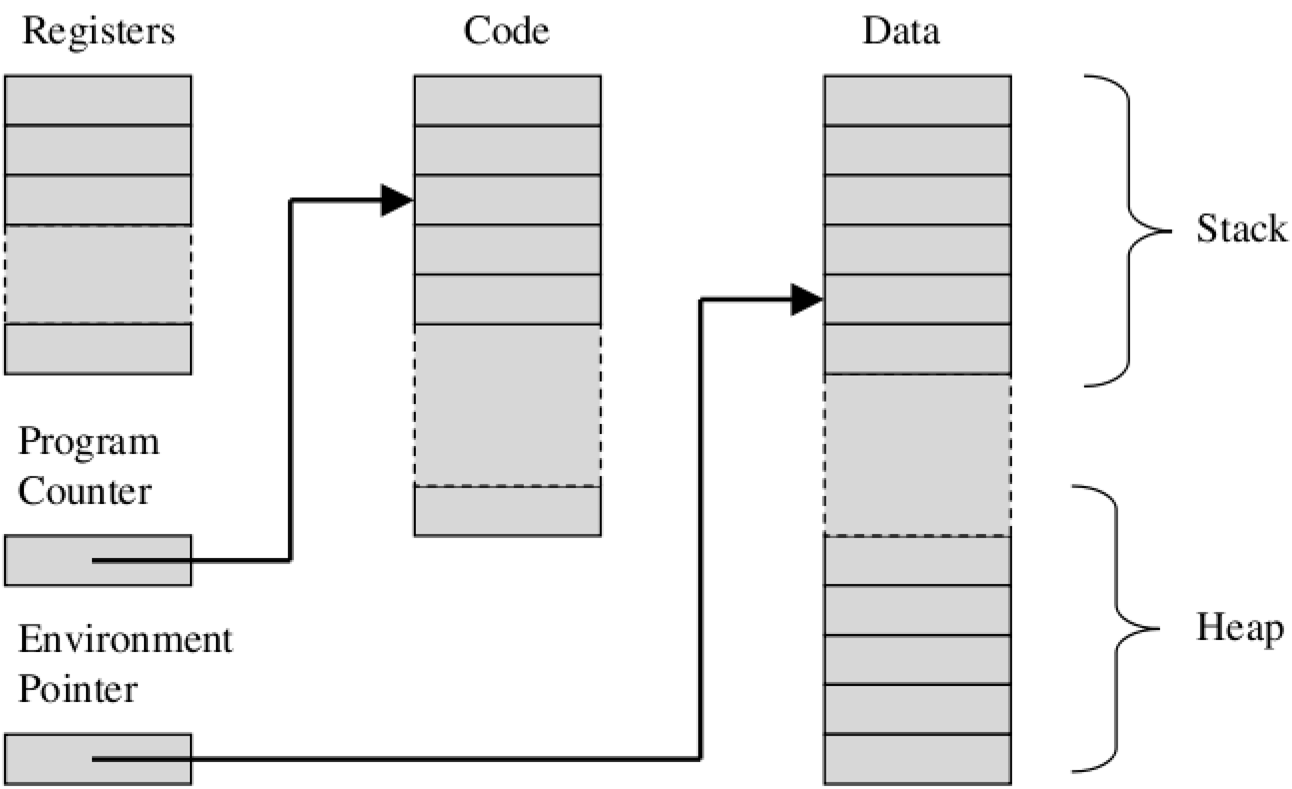
Lifetime

- The period of time between the creation and the destruction of a name-to-object binding is called the binding's lifetime
- The time between the creation and destruction of an object is the object's lifetime
  - Object may retain its value and the potential to be accessed even when a given name can no longer be used to access it
    - When a variable is passed to a subroutine by reference
    - Fortran, var in Pascal, or '&' in C++
  - Lifetime of name-to-object binding is longer than that of object
    - Generally, a sign of a program bug – dangling references

Storage allocation mechanisms

- Object lifetimes generally correspond to one of three principal storage allocation mechanisms, used to manage the object's space:
  1. Static objects are given an absolute address that is retained throughout the program's execution.
  2. Stack objects are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns.
  3. Heap objects may be allocated and deallocated at arbitrary times. They require a more general (and expensive) storage management algorithm.

Program memory



Static allocation

- Global variables are static objects, but not the only ones
- Other static objects:
  - Instructions that constitute a program's machine language translation
  - Variables that are local to a single subroutine, but retain their values from one invocation to the next
  - Numeric and string-valued constant literals, such as A = B/14.7 or printf("hello, world\n")
  - Tables that are used by run-time support routines for debugging, dynamic-type checking, garbage collection, exception handling, and other purposes

Static allocation

- Statically allocated objects are often allocated in protected, read-only memory
- Static activation records
  - Storing variables in blocks, subrutines
  - One activation record for one subrutine
  - Only one activation of subrutine can live at a given time
  - Location of activation record is determined in compile time
  - Simple and very fast
- Static activation records store:
  - Local variables + Temporary values
  - Subrutine arguments, return address, return value
  - Reference to activation record of the caller

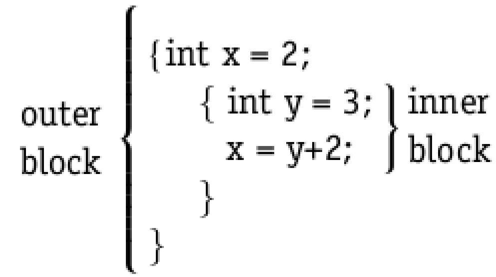
Static allocation

- Problems with static allocation records
  - Recursion can not be used
  - Fortran did not have recursion until very late version
  - Multi-threading can also not be implemented statically

Stack allocation

- Activation record or stack frame is allocated when block or subrutine are activated
  - Natural nesting of blocks and subroutine calls makes it easy to allocate space for locals on a stack
- Maintenance of the stack is the responsibility of the subroutine calling-sequence
  - The code executed by the caller immediately before and after the call (prologue/epilogue)
- Access to higher name spaces
  - Chain of activation records following structure of blocks and subrutines

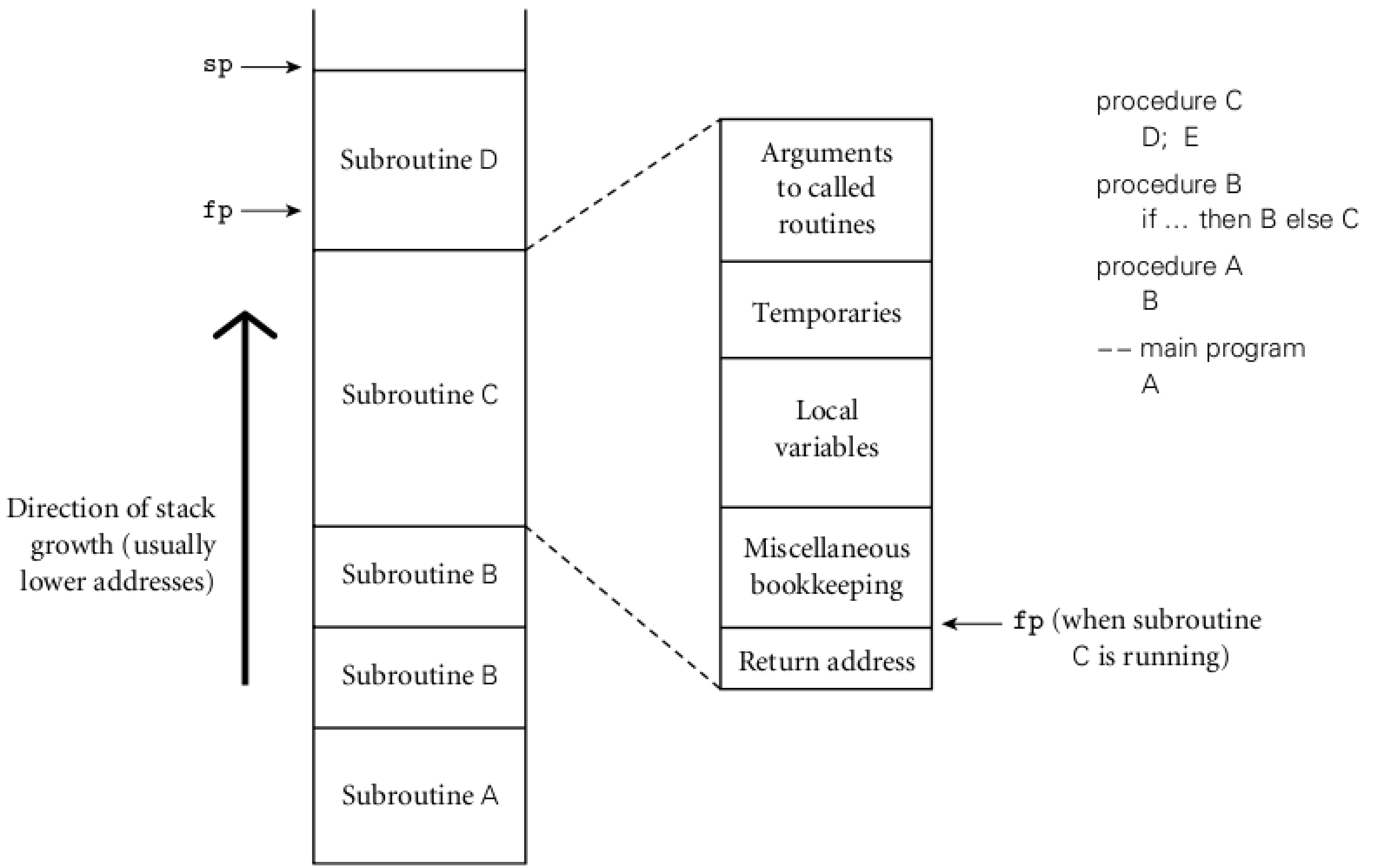
Block-structured languages



- Storage management mechanisms associated with block structures
  - A variable declared within a block is said to be local to that block
  - A variable declared in an enclosing block is said to be global to the block
- Properties of block-structured languages
  - May define new variables anywhere in block
  - Blocks may be nested, but cannot partially overlap
  - When entering, memory is allocated for variables declared in block
  - When exiting, some or all of the memory allocated to variables declared in that block will be deallocated

Recall!

Stack-based allocation of space for subroutines



Memory for block-variables

- Memory management mechanisms for three classes of variables
- Local variables
  - Stored on the stack in activation record associated with block
- Parameters
  - Parameters to subrutine stored in activation record
- Global variables
  - Accessed from an activation record that was placed on the run-time stack before activation of the current block

Example

```
{ int x=0;
  int y=x+1;
     { int z=(x+y)*(x-y);
     };
};
```

- When the outer block is entered, an activation record containing space for x and y is pushed onto the stack
- On entry into the inner block, a separate activation record containing space for z will be added to the stack
- After the value of z is set, the activation record containing this value will be popped off the stack
- Finally, on exiting the outer block, the activation record containing space for x and y will be popped off the stack

Intermedate results

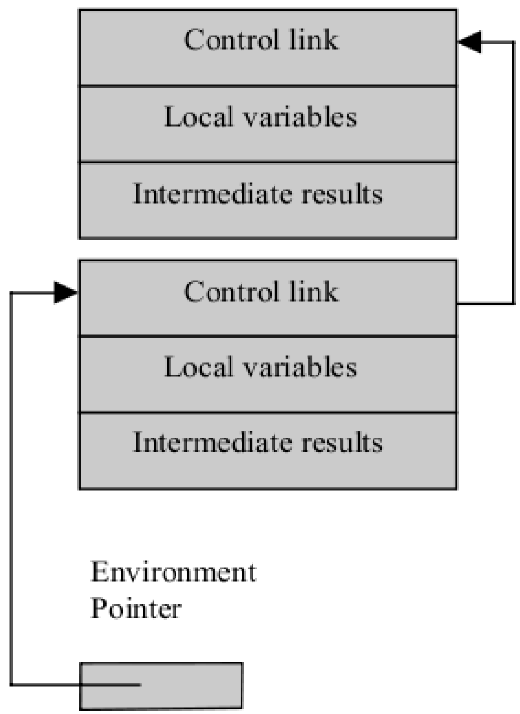
```
{ int z = (x+y)*(x-y);
}
```

Space for z
Space for x+y
Space for x-y

Space for global variables	Space for global variables	Space for global variables	Space for global variables
	Space for x,y	Space for x,y	Space for x,y
		Space for z	

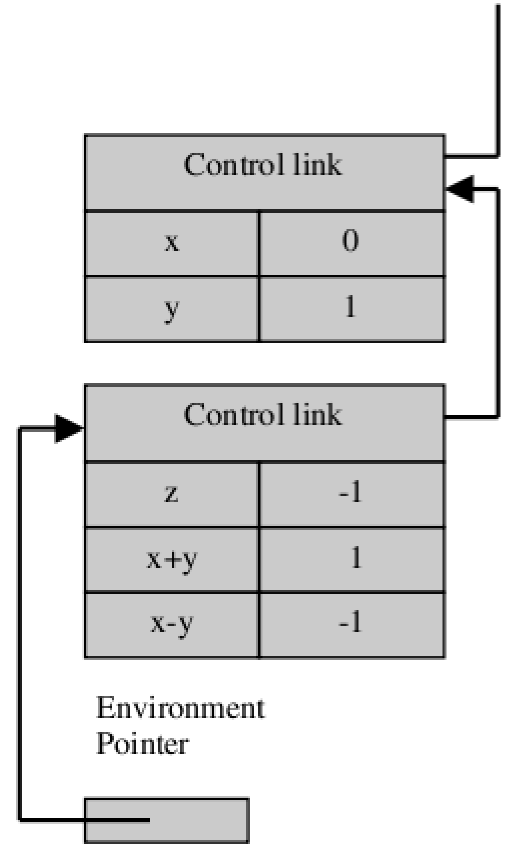
Control link

- Different activation records have different sizes
  - Operations that push and pop activation records from the run-time stack store a pointer to the top of the preceding activation record
- Control link (dynamic link)
  - The pointer to the top of the previous activation record
- When a new act. record is added
  - Control link of the new activation record is set to the previous value of the environment pointer
  - Environment pointer is updated



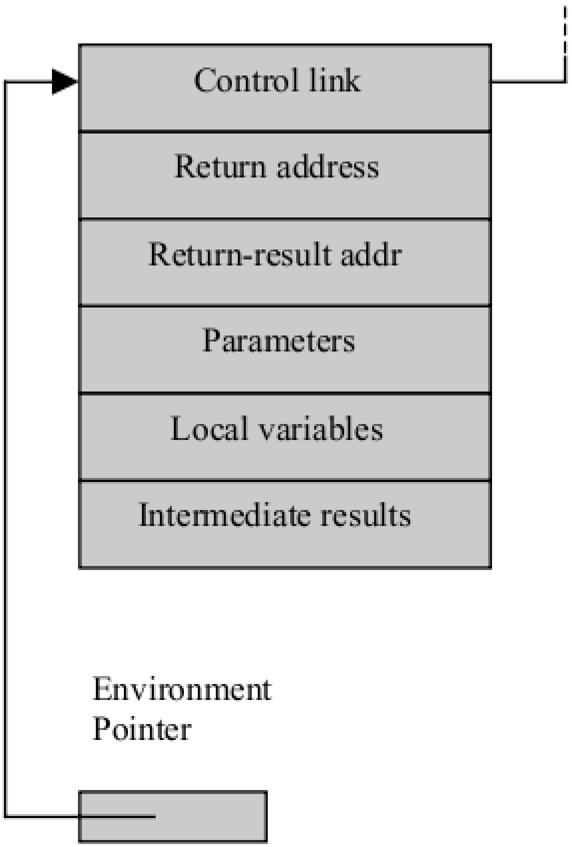
Example

```
{ int x=0;
  int y=x+1;
      { int z=(x+y)*(x-y);
       };
       };
```



Activation Records for Functions

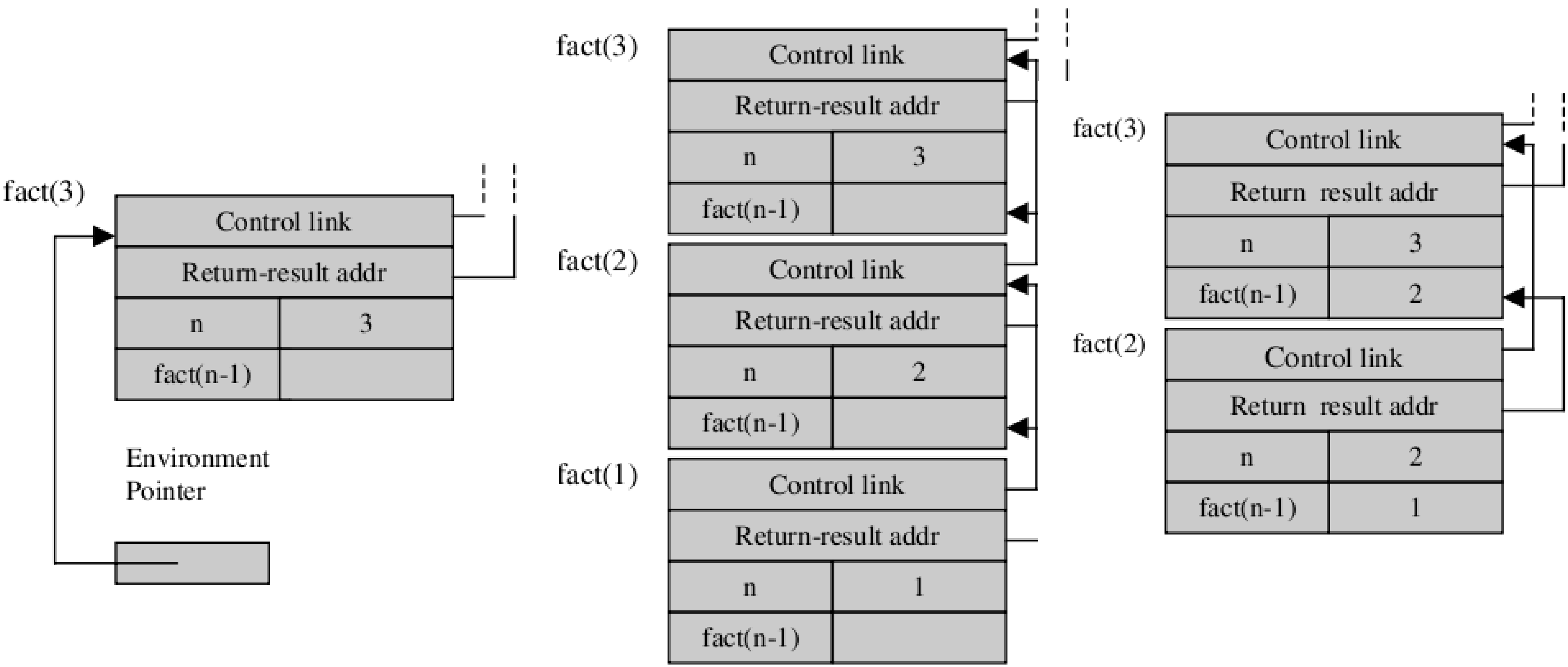
- control link, pointing to the previous activation record on the stack,
- access link (static link), pointer to structurally subsuming block
- return address, giving the address of the first instruction to execute when the function terminates,
- return-result address, the location in which to store the function return value,
- actual parameters of the function,
- local variables declared within the function,
- temporary storage for intermediate results computed with the function executes



Example

```
fun fact(n) = if n <= 1 then 1 else n * fact(n-1);
```

- Activation records are added and removed from the run-time stack when tracing the execution of the familiar factorial function



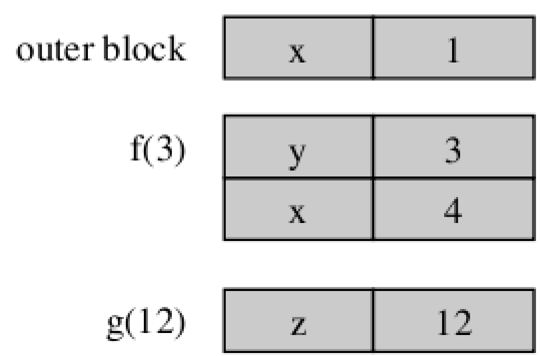
Global Variables

- Identifier x appears in the body of a function, but x is not declared inside the function
- Access to a global x involves finding an appropriate activation record elsewhere on the stack
- There are two main rules for finding the declaration of a global identifier
  - Static Scope:
    - A global identifier refers to the identifier with the name that is declared in the closest enclosing scope of the program text
  - Dynamic Scope:
    - A global identifier refers to the identifier associated with the most recent activation record

Static and dynamic scope

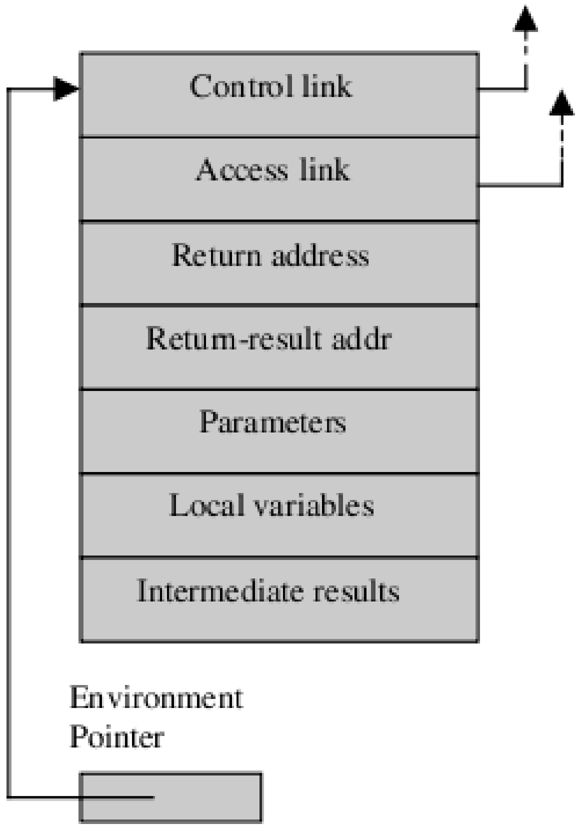
- Difference between static and dynamic scope:
  - Finding declaration under static scope uses the static (unchanging) relationship between blocks in program text.
  - Actual sequence of calls that are executed in the dynamic (changing) execution of the program.
- Example:

```
int x=1;
function g(z) = x+z;
function f(y) = {
      int x = y+1;
      return g(y*x)
};
f(3);
```



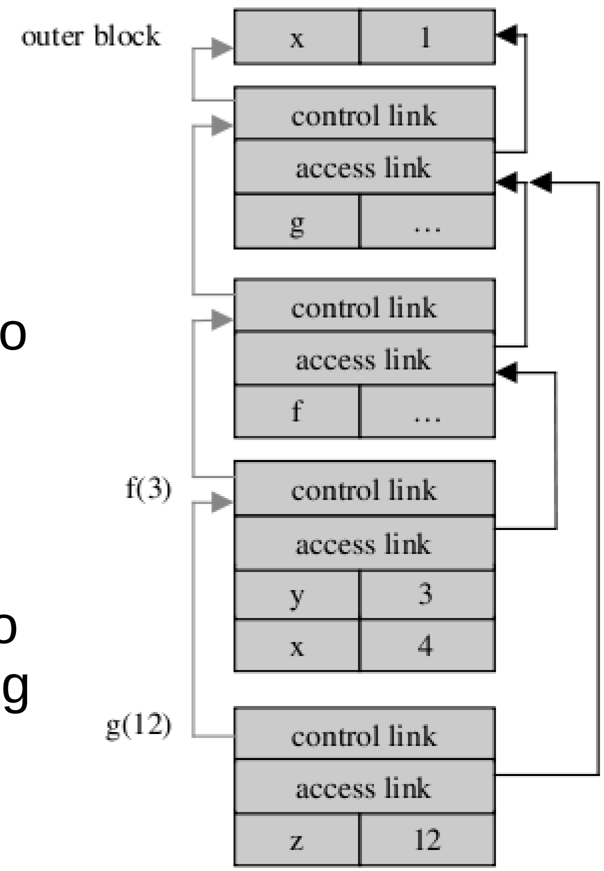
Access link

- How to implement Static scope?
- Access link (static link) of an activation record points to the activation record of the closest enclosing block in the program
- In-line blocks do not need an access link, as the closest enclosing block will be the most recently entered block
- Access link will generally point to a different activation record than the control link



Example

```
int x=1;
function g(z) = x+z;
function f(y) = {
      int x = y+1;
      return g(y*x)
};
f(3);
```



- Declaration of g occurs inside the scope of declaration of x
    - access link for declaration of g points to activation record for declaration of x
- Declaration of f is similarly inside the scope of the declaration of g
    - Access link for declaration of f points to activation record for the declaration of g
- Calls f(3) and g(12) cause activation record to be allocated for scope associated with body of f and body of g, respectively

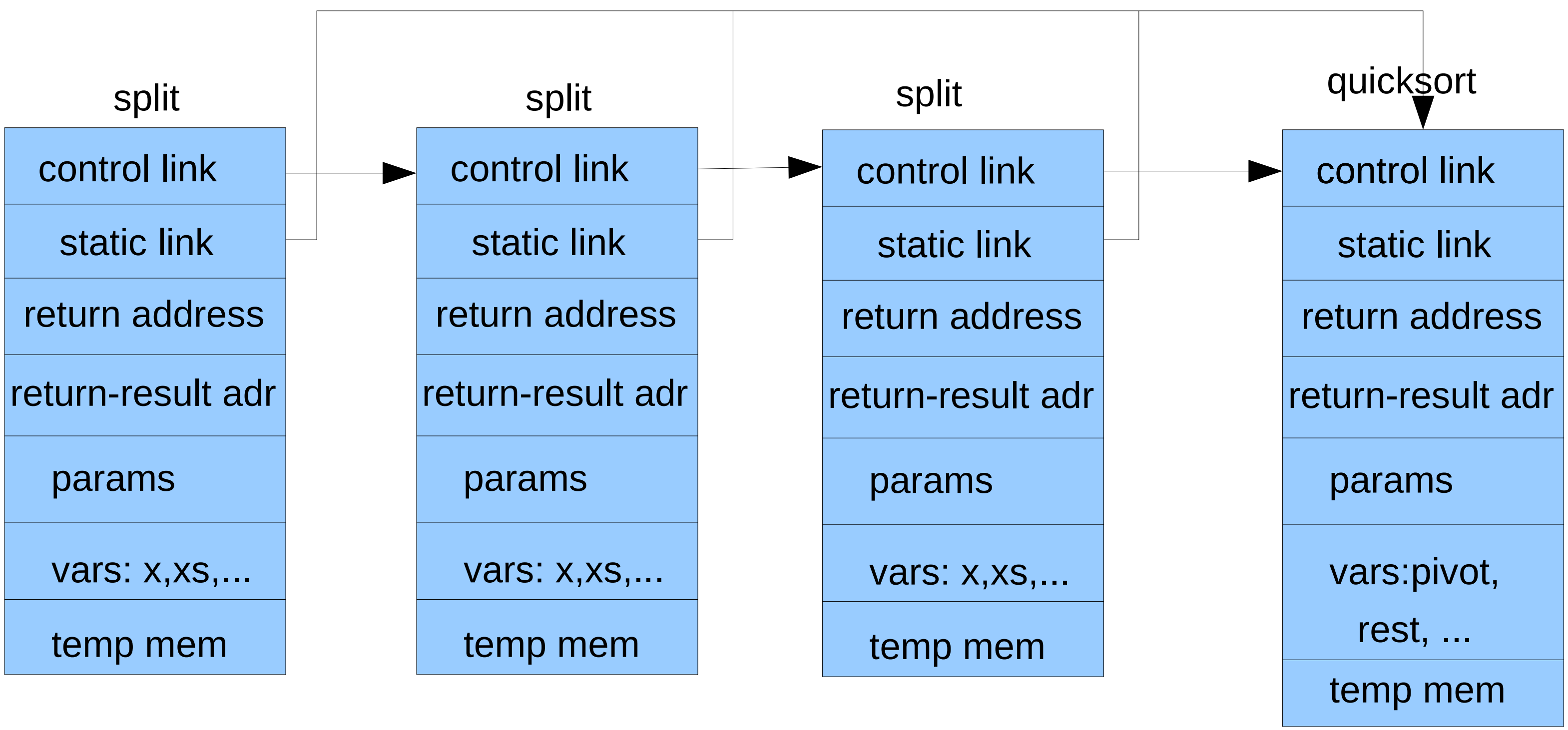
Control and access links

- To summarize:
  - Control link is a link to the activation record of the previous (calling) block
  - Access link is a link to the activation record of the closest enclosing block in program text
  - Control link depends on the dynamic behavior of program
  - Access link depends on only the static form of the program text
  - Access links are used to find the location of global variables in statically scoped languages with nested blocks at run time

Example: quicksort

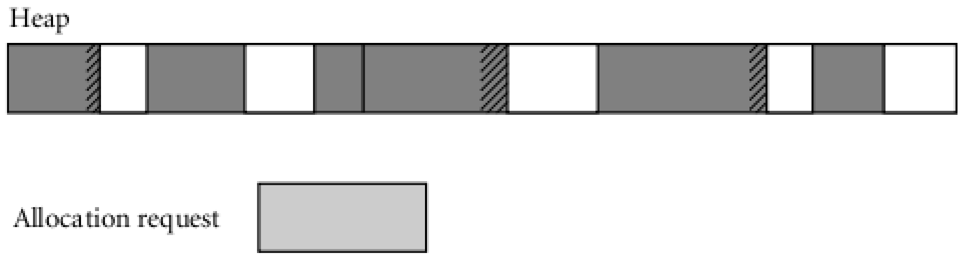
```
# let rec quicksort = function
      [] -> []
    | pivot::rest ->
         let rec split = function
             [] -> ([],[])
           | x::tail ->
               let (below, above) = split tail
               in
                 if x<pivot then (x::below, above)
                 else (below, x::above)
         in let (below, above) = split rest
             in quicksort below @ [pivot] @ quicksort above;;
val quicksort : 'a list -> 'a list = <fun>
```

Example: quicksort



Heap allocation

- A heap is a region of storage in which subblocks can be allocated and deallocated at arbitrary times
- Heaps are required for the dynamically allocated pieces of linked data structures
  - Character strings, lists, and sets, whose size may change on update
  - Arrays, records, objects, recursive data structures, ...
- Strategies to manage space in a heap
  - Tradeoffs between speed and space
  - Internal and external fragmentation



Storage-management algorithms

- Single linked list—the free list
  - Heap blocks not currently in use
  - Initially list consists of a single block comprising the entire heap
  - Allocation request searches list for a block of appropriate size
  - First fit algorithm
  - Best fit algorithm
  - Unneeded portion below some min threshold is left in block as internal fragmentation, or, inserted back to list

Single linked list

- One would expect the best-fit algorithm to do a better job
- Time complexity
  - Best-fit has higher allocation cost than first-fit algorithm
    - Always goes through all candidates
    - In recent applications, we may have a huge number of blocks!
  - The concept of “current” block in first-fit algorithm
    - Travels in a round-robin fashion
- Any algorithm is linear in the number of free blocks
  - In worst case the algorithm has to inspect all blocks

Single linked list

- Space complexity
  - First-fit tends to behave better then best-fit
  - Best-fit results in a larger number of very small “left-over” blocks
    - Internal as well as external fragmentation?
  - Score depends on the distribution of size requests
    - Distribution depends on the application type

Single linked list

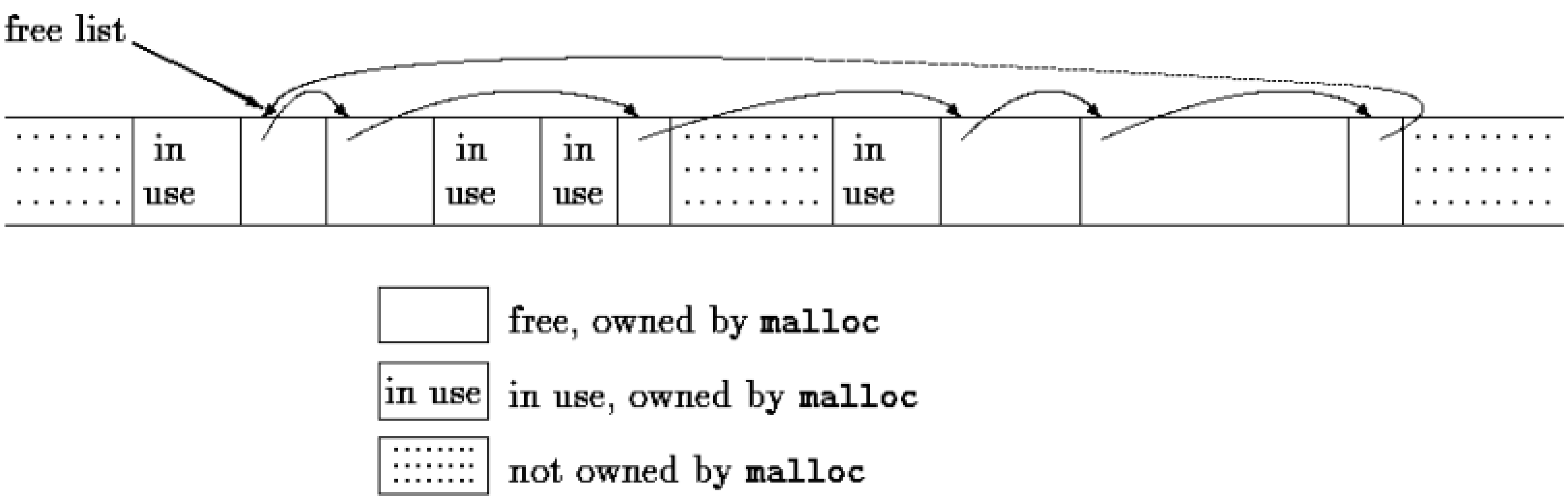
- How to reduce the cost of the algorithm?
  - Maintain separate free lists for blocks of different sizes
  - If block is not found in the appropriate list then the list with larger blocks is searched
    - The leftover is stored in a list with smaller blocks
  - Cost can be reduced to a constant
- We first consider a heap in the C prog. language
- Then we will go through some solutions that use more lists

Heap in C

- C originally implemented heap based on linked list of free blocks
  - Calls to malloc() and free() may occur in any order
  - malloc() calls upon the operating system to obtain more memory when needed
  - Space that malloc() manages may not be contiguous
  - Free storage is kept as a list of free blocks
  - Each block contains a size, a pointer to the next block, and the space itself
  - Blocks are kept in order of increasing storage address
  - Last block (highest address) points to the first

Heap in C

- When a request is made, the free list is scanned until a big-enough block is found, i.e. “first fit”
- If the block is too big, it is split, and the proper amount is returned to the user while the residue remains on the free list

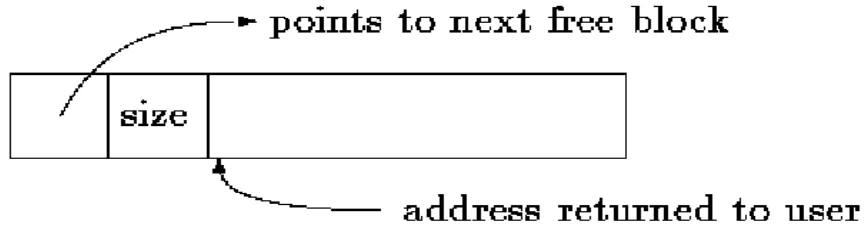


Heap in C

- Proper alignment for the objects stored
  - The most restrictive type can be stored at a particular address, then all other types may be also
  - On some machines, the most restrictive type is a double; on others, int or long suffices

```
typedef long Align;     /* for alignment to long boundary */
union header {          /* block header */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned size;      /* size of this block */
    } s;
    Align x; /* force alignment of blocks */
};
typedef union header Header;
```

Heap in C



A block returned by **malloc**

- Requested size in characters is rounded up to the proper number of header-sized units
  - Block that will be allocated contains one more unit, for the header itself
- Search for a free block of adequate size begins where the last block was found (at `freep`)
  - This strategy helps keep the list homogeneous
  - If a too-big block is found, the tail end is returned to user
  - Pointer returned to the user points to free space within the block, which begins one unit beyond the header

malloc()

```
static Header base;       /* empty list to get started */
static Header *freep = NULL;     /* start of free list */
/* malloc: general-purpose storage allocator */
void *malloc(unsigned nbytes)
{
   Header *p, *prevp;
   Header *morecore(unsigned);
   unsigned nunits;

   nunits = (nbytes+sizeof(Header)-1)/sizeof(header) + 1;
   if ((prevp = freep) == NULL) {
       /* no free list yet */
       base.s.ptr = freeptr = prevptr = &base;
       base.s.size = 0;
   }
```

⇩

⇩

```
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {        /* big enough */
            if (p->s.size == nunits)       /* exactly */
                prevp->s.ptr = p->s.ptr;
            else {                          /* allocate tail end */
               p->s.size -= nunits;
               p += p->s.size;
               p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep)      /* wrapped around free list */
            if ((p = morecore(nunits)) == NULL)
                return NULL;    /* none left */
        }
}
```

morecore()
free()

```
#define NALLOC 1024  /* minimum #units to request */
/* morecore: ask system for more memory */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;
    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1)   /* no space at all */
       return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *)(up+1));
    return freep;
}
```

```
/* free: put block ap in free list */
void free(void *ap)
{
    Header *bp, *p;
    bp = (Header *)ap – 1;    /* point to block header */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
          if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
               break; /* freed block at start or end of arena */
    if (bp + bp->size == p->s.ptr) {   /* join to upper nbr */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->size == bp) {  /* join to lower nbr */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

Dynamic pools

- Heap is divided into “pools,” one for each standard size
  - Request is rounded up to the next standard size
  - Division into ranges may be static or dynamic
- Two common mechanisms for dynamic pool:
  - Buddy system
  - Fibonacci heap
- Buddy system
  - Standard block sizes are powers of two
  - Block of size 2^k is needed
    - none ⇒ block of size 2^{k+1} is split in two (one is put into 2^k free list)

Dynamic pools

  - When a block is deallocated, it is coalesced with its "buddy"—if that buddy is free
- Fibonacci heap
  - Fibonacci numbers for the standard sizes
  - Slightly more complex
  - Leads to slightly lower internal fragmentation
- Problems with external fragmentation
  - The ability of the heap to satisfy requests may degrade over time
  - It is always possible to devise a sequence of requests that cannot be satisfied (while enough space ∃)
  - Compact the heap, by moving already-allocated blocks
    - update all outstanding references

Manual/automatic memory management

- Explicit (manual) memory management
  - Explicit allocation and deallocation of objects
  - Programer has total control over memory management
  - C, C++, Pascal, ...
- Automatic memory management
  - Compiler and run-time system manage memory
  - Garbage collection
  - Java, Scala, Go, Haskell, Erlang, Python, Perl, ...

Explicit memory management

- Program allocates memory blocks and has full control over them
  - After block is not needed it is reclaimed
- Usual malloc-free cycle known from C
  - Functions malloc() and free() implement heap in C
- Problems:
  - If pointer is “lost” then we have memory leak—performance decreases and memory is filled-up eventually
  - If object is reclaimed by mistake we have dangling pointer
- Programmer has to be very careful and design all allocations in pairs.

Example

```
void function_which_allocates() {
    // allocate an array of 50 floats
    float* a = new float[50];
     // additional code making use of ’a’
     // return to caller, having forgotten
     // to delete the memory we allocated
}
```

```
int main() {
    function_which_allocates();
    // the pointer ’a’ no longer exists, and therefore cannot be freed,
    // but the memory is still allocated. a leak has occurred.
    int* p = new(1024);
    int* q = p;
    delete p; // q is dangling pointer by now
    // main continues
    *q = 2048; // memory corruption: write to garbage memory
    delete q; // memory corruption: double free of memory
}
```

Explicit memory management

- Many reclaims are automatic
  - On function return, space for local variables and parameters is reclaimed
- Disciplined allocation/deallocation of memory can lead to efficient programs
  - In reality, all fast programs are implemented in languages that allow explicit memory management
  - Performance of PL with GC is comparable to explicit memory management if there is enough memory :-|
  - Language without GC can perform orders of magnitude better than languages with GC
  - If memory is a problem languages with explicit control are always better

Garbage collecton

- Allocation of heap-based objects is always triggered by some specific operation in program:
  - Instantiating an object, appending to the end of a list, assigning a long value into previously short string, ...
- Deallocation can be done in two ways:
  - It is explicit in some languages
    - e.g., C, C++, and Pascal
  - In many languages objects are deallocated implicitely
    - After they can not be reached from any program variable
    - Such language must then provide a garbage collection mechanism to identify and reclaim unreachable objects

Garbage collection

- Languages with garbage collection
  - Most functional and scripting languages require garbage collection
  - Many more recent imperative languages (including Modula-3, Java, and C) use garbage collectors
- Arguments in favor of explicit deallocation!
  - Implementation simplicity
    - Even naive implementations of automatic garbage collection add significant complexity to the implementation
  - Execution speed
    - Even the most sophisticated garbage collector can consume nontrivial amounts of time in certain programs

Garbage collection

- Argument in favor of automatic garbage collection
  - Manual deallocation errors are among the most common and costly bugs in real-world programs
  - Object is deallocated too soon
    - Program may follow a dangling reference
    - Accessing memory now used by another object
  - Object is not deallocated at the end of its lifetime
    - Program may “leak memory,” eventually running out of heap space
    - Deallocation errors are notoriously difficult to identify and fix

Garbage collection

- Automatic garbage collection is an essential language feature
  - Conclusion of both language designers and programmers
  - Many times we do not want to spend many days debugging but want solution »at once«
  - The cost of garbage collection is compensated by faster hardware
- In many cases it is not possible to implement system efficiently without explicit control of memory allocation
  - Most compilers, DBMSs, OS routines, ... are written in C or C++

Reference counts

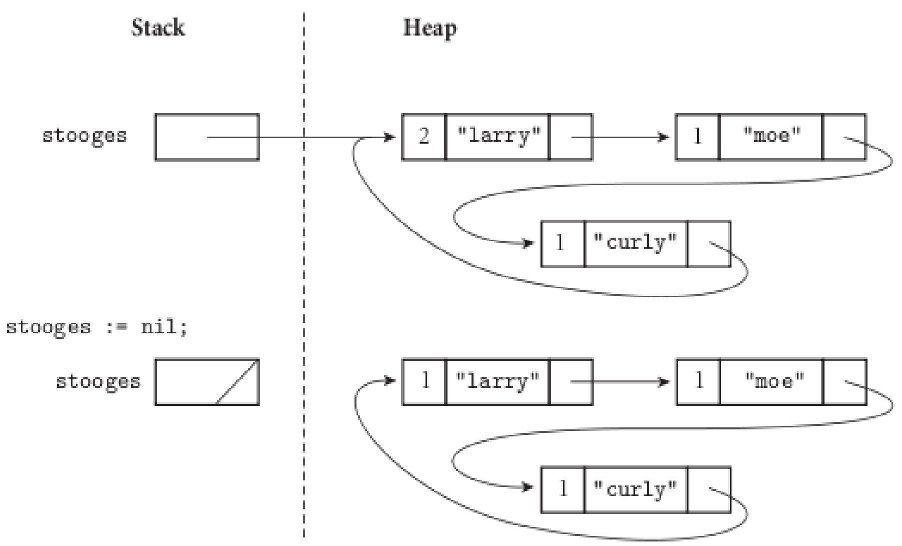
- When is an object no longer useful?
  - There are no pointers to object
- Simple solution:
  - Place the counter of pointers referencing the object in object itself
    - Initially this counter is 1
  - When pointer a is assigned to pointer b:
    1. dec_rc(object(b))
    2. Make assignment b := a
    3. inc_rc(object(a))
  - On subrutine return
    - calling-sequence epilogue has to decrement reference counts of all objects referred to by parameters and local variables

Reference counts

  - When reference count is 0 object can be reclaimed
    - Recursively, run-time system must decrement counts for any objects referred to by pointers within the reclaimed object
- In order for reference counts to work
  - Language implementation must be able to identify the location of every pointer
    - Which words in object or stack frame represent pointers?
  - Type descriptors (offsets of components) generated by compiler are used
  - In general languages with strong typing can use such algorithms
    - Solutions for languages not strongly typed also exist

Reference counts

- The most important problem is definition of "useful object".
  - Object may be useless despite there are references to it
  - RCs fail to collect circular structures
- Many languages use RC for var-length strings
  - They do not contain refs
- Perl uses RCs for all dynamically allocated data
  - Programmer is warned to break cycles



Mark-and-sweep

- Better definition of a “useful” object
  - Can be reached by following a chain of valid pointers starting from something that has a name
  - Circularly referenced objects do not stay in heap
- Recursively exploring the heap to determine what is useful
  - Starting from external pointers (very expensive...)
- Mark-and-sweep
  - Classic mechanism to identify useless blocks, under this more accurate definition
  - When amount of free space remaining in heap falls below some minimum threshold
  - It proceeds in three main steps

Mark-and-sweep

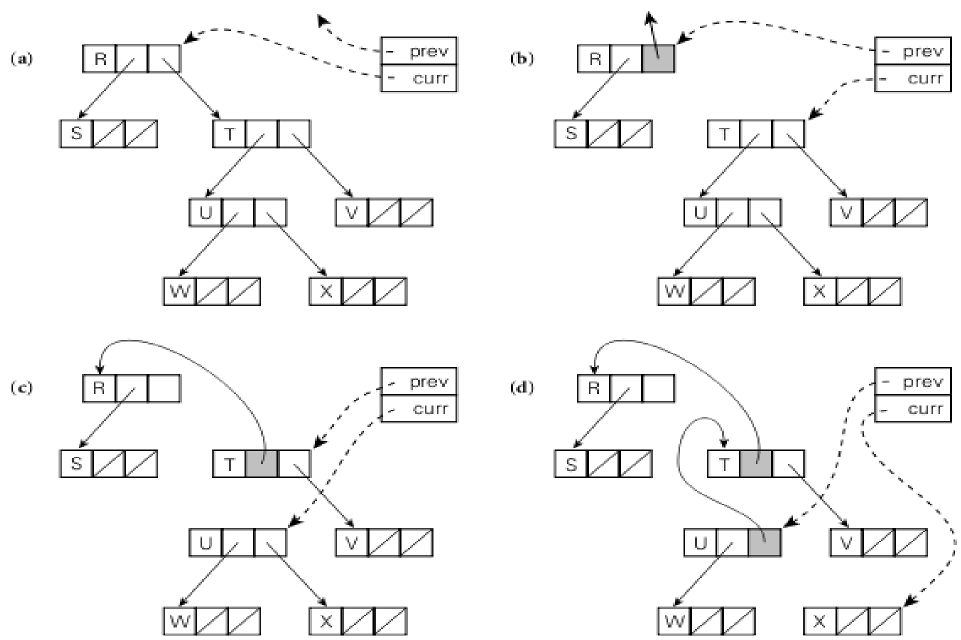
1. Collector walks through the heap, tentatively marking every block as “useless.”
2. Beginning with all pointers outside the heap, collector recursively explores all linked data structures in the program, marking each newly discovered block as “useful.”
3. The collector again walks through the heap, moving every block that is still marked “useless” to the free list.

Mark-and-sweep

- Problems with algorithm:
  - We must know where every block in-use begins and ends
  - Every block must begin with an indication of its size, and of whether it is currently free
  - Collector must be able in Step 2 to find the pointers contained within each block
    - Solution: put pointer to (block) type descriptor near the beginning of block

Improvements of Mark-and-sweep

- Pointer reversal
  - Recursive exploration of heap requires storage
    - Heap could be used to track the path
  - As collector explores the path to a given block, it reverses the pointer to the block
    - Collector keeps track of current block and the block from whence it came
  - Search can be implemented without stack



Improvements of Mark-and-sweep

- Stop-and-copy
  - Reduce external fragmentation by performing storage compaction
  - Eliminating Steps 1 and 3
  - Divide the heap into two regions of equal size
    - All allocations happen in first part
  - Each reachable block is copied into second half of the heap, with no external fragmentation
    - Old copy is marked “useful”
    - Pointers to old block are corrected to point to new
  - When collector finishes, all useful blocks are stored in the second part of heap
    - First part of heap is empty!
    - Collector swaps its notion of first and second halves

Generational Collection

- Observation: most dynamically allocated objects are short-lived
- Heap is divided into multiple regions (often two)
  - When space runs low the collector first examines the youngest region (the “nursery”)
    - It is likely to have the highest proportion of garbage
  - If it is unable to reclaim sufficient space in this region the collector examines the next-older region
  - In worst case collector has to examine complete heap
- In most cases, the overhead of collection will be proportional to the size of youngest region only

Generational Collection

- Object that survives few collections (often one) is promoted (moved) to the next older region
  - Reminiscent of stop-and-copy
  - Promotion requires that pointers reflect new locations
  - At each pointer assignment, the compiler generates code to check whether new value is old-to-new pointer
  - This instrumentation on assignments is known as a write barrier