“To iterate is human, to recurse divine.”

-L. Peter Deutsch

CSE341
Programming Languages

Lecture 6 – November 5, 2015

Exceptions

Chapter 6 slides are adapted from R.W. Sebesta, C. Li & W. He and V. Shmatikov

Exceptions: Structured Exit

- Terminate part of computation
  - Jump out of construct
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
  - Unnecessary activation records may be deallocated
    - May need to free heap space, other resources
- Two main language constructs
  - Declaration to establish exception handler
  - Statement or expression to raise or throw exception

Often used for unusual or exceptional condition, but not necessarily

ML Example

```
exception Determinant;  (* declare exception name *)
fun invert (M) =           (* function to invert matrix *)
      …
      if …
           then raise Determinant    (* exit if Det=0 *)
           else …
 end;
…
invert (myMatrix) handle Determinant => … ;
```

Value for expression if determinant of myMatrix is 0

C++ Example

```
Matrix invert(Matrix m) {
    if … throw Determinant;
    …
};

try { … invert(myMatrix); …
}
catch (Determinant) { …
    // recover from error
}
```

C Example

```
#include <setjmp.h>
#include <stdio.h>
#include <stdlib.h>

enum { SOME_EXCEPTION = 1 } exception;
jmp_buf state;

int main(void)
{
  if(!setjmp(state))                    // try
  {
    if(/* something happened */)
    {
      exception = SOME_EXCEPTION;
      longjmp(state, 0);    // throw SOME_EXCEPTION
    }
  }
else switch(exception)
  {
    case SOME_EXCEPTION: // catch SOME_EXC...
      puts("SOME_EXCEPTION caught");
      break;
    default:                             // catch ...
      puts("Some strange exception");
  }
  return EXIT_SUCCESS;
}
```

C++ vs ML Exceptions

- C++ exceptions
  - Can throw any type
  - Stroustrup: "I prefer to define types with no other purpose than exception handling. This minimizes confusion about their purpose. In particular, I never use a built-in type, such as int, as an exception."

    The C++ Programming Language, 3rd ed.
- ML exceptions
  - Exceptions are a different kind of entity than types
  - Declare exceptions before use

Similar, but ML requires what C++ only recommends

Termination Semantics

- Exception handling mechanisms in contemporary languages are typically non-resumable ("termination semantics") as opposed to hardware exceptions, which are typically resumable.
  - This is based on experience of using both, as there are theoretical and design arguments in favor of either decision; these were extensively debated during C++ standardization discussions 1989–1991, which resulted in a definitive decision for termination semantics.
  - On the rationale for such a design for the C++ mechanism, Stroustrup notes: At the Palo Alto meeting in November 1991, we heard a brilliant summary of the arguments for termination semantics backed with both personal experience and data from Jim Mitchell. Jim had used exception handling in half a dozen languages over a period of 20 years and was an early proponent of resumption semantics as one of the main designers and implementers of Xerox's Cedar/Mesa system. His message was

    “termination is preferred over resumption; this is not a matter of opinion but a matter of years of experience. Resumption is seductive, but not valid.”

  - He backed this statement with experience from several operating systems. The key example was Cedar/Mesa: It was written by people who liked and used resumption, but after ten years of use, there was only one use of resumption left in the half million line system – and that was a context inquiry. Because resumption wasn't actually necessary for such a context inquiry, they removed it and found a significant speed increase in that part of the system. In each and every case where resumption had been used it had – over the ten years – become a problem and a more appropriate design had replaced it. Basically, every use of resumption had represented a failure to keep separate levels of abstraction disjoint.

Source: Wikipedia

Opposing Views…

- A contrasting view on the safety of exception handling was given by C.A.R Hoare in 1980, described the Ada programming language as having "...a plethora of features and notational conventions, many of them unnecessary and some of them, like exception handling, even dangerous. [...] Do not allow this language in its present state to be used in applications where reliability is critical[...]. The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus: It may be a nuclear warhead exploding over one of our own cities." [14]
- Citing multiple prior studies by others (1999–2004) and their own results, Weimer and Necula wrote that a significant problem with exceptions is that they "create hidden control-flow paths that are difficult for programmers to reason about".

Exception Handling

- We distinguish between two such classes of events:
  - Those that are detected by hardware: e.g., disk read errors, end-of-file
  - Those that are software-detectable: e.g., subscript range errors
- ***Definition:*** An ***exception*** is an unusual event that is detectable by either hardware or software and that may require special processing.
- ***Terminology:*** The special processing that may be required when an exception is detected is called ***exception handling***. The processing is done by a code unit or segment called an ***exception handler***. An exception is ***raised*** when its associated event occurs.

User-Defined Exception Handling

- When a language does not include specific exception handling facilities, the user often handles software detections by him/herself.
- This is typically done in one of three ways:
  - Use of a ***status variable*** (or flag) which is assigned a value in a subprogram according to the correctness of its computation. [Used in standard C library functions]
  - Use of a ***label parameter*** in the subprogram to make it return to different locations in the caller according to the value of the label. [Used in Fortran].
  - Define the handler as a ***separate subprogram and pass its name*** as a parameter to the called unit. But this means that a handler subprogram must be sent with every call to every subprogram.

Advantages to Built-in Exception Handling

- Without built-in Exception Handling, the code required to detect error conditions can considerably clutter a program.
- Built-in Exception Handling often allows exception propagation. i.e., an exception raised in one program unit can be handled in some other unit in its dynamic or static ancestry. A single handler can thus be used in different locations.
- Built-in Exception Handling forces the programmer to consider all the events that could occur and their handling. This is better than not thinking about them.
- Built-in Exception Handling can simplify the code of programs that deal with unusual situations. (Such code would normally be very convoluted without it).

Illustration of an Exception Handling Mechanism

```
void example ( )  {
   …
   average = sum / total;
   …
   return;
  /* Exception handlers */
  When zero_divide  {
    average = 0;
    printf(“Error-divisor (total) is zero\n”);
  }
  …
}
```

The exception of division by zero, which is implicitly raised causes control to transfer to the appropriate handler, which is then executed

Design Issues: Exception Binding

- Binding an exception occurrence to an exception handler:
  - At the unit level: how can the same exception raised at different points in the unit be bound to different handlers within the unit?
  - At a higher level: if there is no exception handler local to the unit, should the exception be propagated to other units? If so, how far? [Note: if handlers must be local, then many need to be written. If propagation is permitted, then the handler may need to be too general to really be useful.]

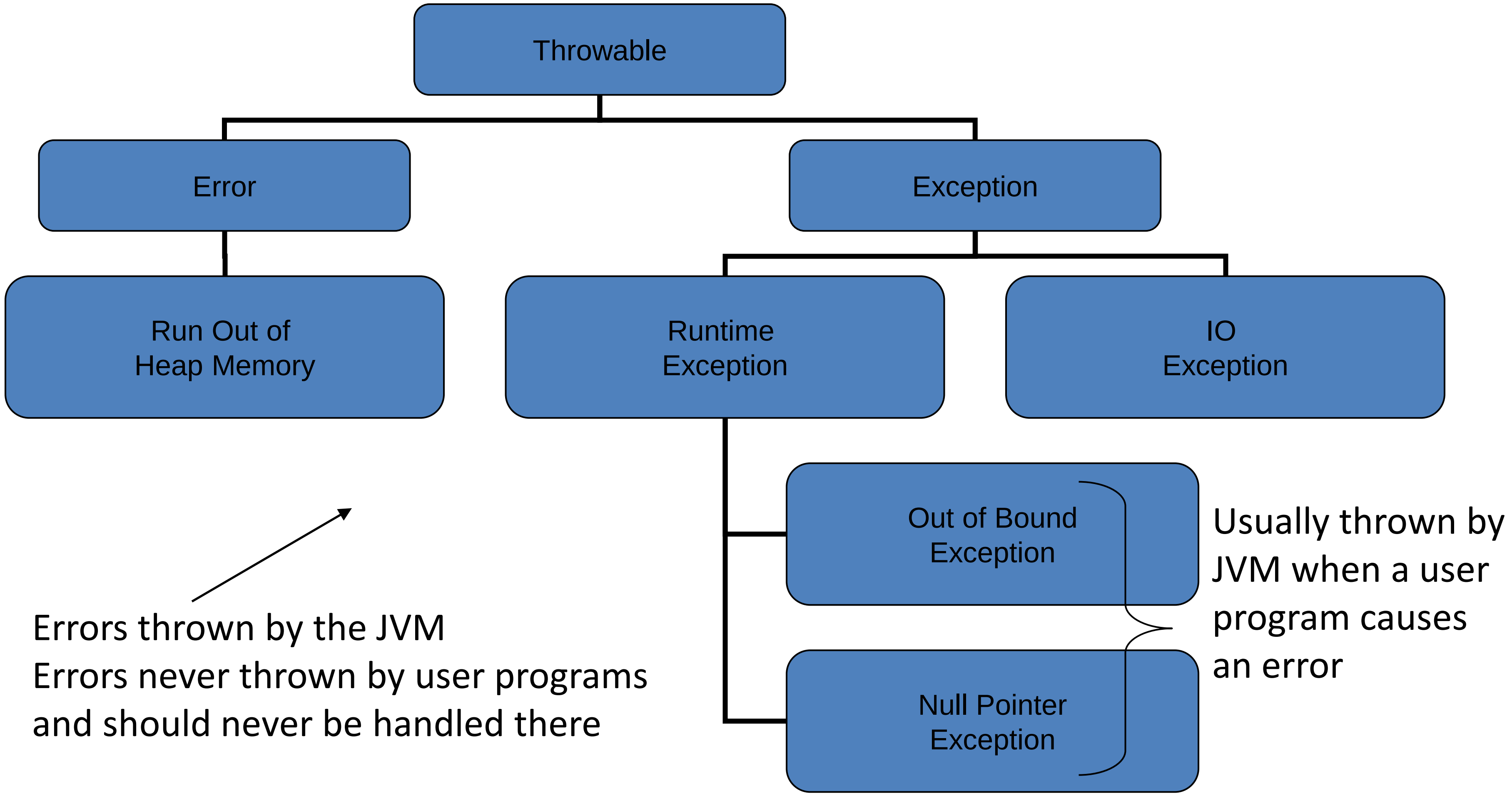
Design Issues: Continuation

- After an exception handler executes, either control can transfer to somewhere in the program outside of the handler code, or program execution can terminate.
  - Termination is the simplest solution and is often appropriate.
  - Resumption is useful when the condition encountered is unusual, but not erroneous. In this case, some convention should be chosen as to where to return:
    - At the statement that raised the exception?
    - At the statement following the statement that raised the exception?
    - At some other unit?

Design Issues: Others

- Is finalization—the ability to complete some computations at the end of execution regardless of whether the program terminated normally or because of an exception—supported?
- How are user-defined exceptions specified?
- Are there pre-defined exceptions?
- Should it be possible to disable predefined exceptions?
- If there are pre-defined exceptions, should there be default exception handlers for programs that do not provide their own?
- Can pre-defined exceptions be explicitly raised?
- Are hardware-detectable errors treated as exceptions that may be handled?

Exception Handling in Java: Class Hierarchy



Usually thrown by JVM when a user program causes an error

Errors thrown by the JVM
Errors never thrown by user programs and should never be handled there

Exception Handling in Java: Exception Handlers

- A try construct includes a compound statement called the try clause and a list of exception handlers:

```
try {
//** Code that is expected to raise an exception }
catch (formal parameter) {
//* A handler body
}
…
catch (formal parameter) {
//** A handler body
}
```

Exception Handling in Java: Binding Exceptions to Handlers

- An exception is thrown using the **throw** statement.

  E.g.: **throw** new MyException (“a message to specify the location of the error”)

- **Binding:** If an exception is thrown in the compound statement of a try construct, it is bound to the first handler (catch function) immediately following the try clause whose parameter is the same class as the thrown object, or an ancestor of it. If a matching handler is found, the throw is bound to it and it is executed.

Exception Handling in Java: The ***finally*** clause

- Sometimes, a process must be executed regardless of whether an exception is raised or not and handled or not.
- This occurs, for example, in the case where a file must be closed or an external resource released, regardless of the outcome of the program.
- This is done by adding a ***finally*** clause at the end of the list of handlers, just after a complete try construct.
- The finally clause is executed in all cases whether or not try throws an exception, and whether or not it is caught by a catch clause.

Event Handling I

- Event handling is similar to exception handling.
- The difference is that while exceptions can be created wither explicitly by user code or implicitly by hardware or a software interpreter, events are created by external actions, such as user interactions though a graphical user interface (GUI)
- In event-driven programming, parts of the program are executed at completely impredictable times, often triggered by user interactions with the executing program.

Event Handling II

- An event is a notification that something specific has occurred, such as a mouse click on a graphical button.
- An event handler is a segment of code that is executed in response to the appearance of an event.
- Event handling is useful in Web applications such as:
  - commercial applications where a user clicks on buttons to select merchandise or
  - Web form completion, where event handling is used to verify that no error or omission has occurred in the completion of a form.
- Java supports two different approaches to presenting interactive displays to users: either from application programs or from applets.

ML Exceptions

- Declaration: exception ⟨name⟩ of ⟨type⟩
  - Gives name of exception and type of data passed when this exception is raised
- Raise: raise ⟨name⟩ ⟨parameters⟩
- Handler: ⟨exp1⟩ handle ⟨pattern⟩ => ⟨exp2⟩
  - Evaluate first expression
  - If exception that matches pattern is raised, then evaluate second expression instead

General form allows multiple patterns

Dynamic Scoping of Handlers

```
exception Ovflw;
fun reciprocal(x) = if x<min  then  raise Ovflw  else  1/x;
(reciprocal(x) handle Ovflw=>0)  /  (reciprocal(y) handle Ovflw=>1);
```

- First call to reciprocal() handles exception one way, second call handles it another way
- Dynamic scoping of handlers: in case of exception, jump to most recently established handler on run-time stack
- Dynamic scoping is not an accident
  - User knows how to handle error
  - Author of library function does not

Exceptions for Error Conditions

```
- datatype ‘a tree = LF of  ‘a | ND of  (‘a tree)*(‘a tree)
- exception No_Subtree;
- fun lsub (LF x) = raise No_Subtree
   |     lsub (ND(x,y)) = x;
> val lsub = fn : ‘a tree -> ‘a tree
```

- This function raises an exception when there is no reasonable value to return
  - What is its type?

Exceptions for Efficiency

- Function to multiply values of tree leaves

```
fun prod(LF x) = x
|   prod(ND(x,y)) = prod(x) * prod(y);
```

- Optimize using exception

```
fun prod(tree) =
    let exception Zero
        fun p(LF x) = if x=0 then (raise Zero) else x
        |   p(ND(x,y)) = p(x) * p(y)
    in
        p(tree) handle Zero=>0
    end;
```

Scope of Exception Handlers

scope

```
exception X;
(let fun f(y) = raise X
    and g(h) = h(1) handle X => 2
in
    g(f) handle X => 4
end) handle X => 6;
```

handler

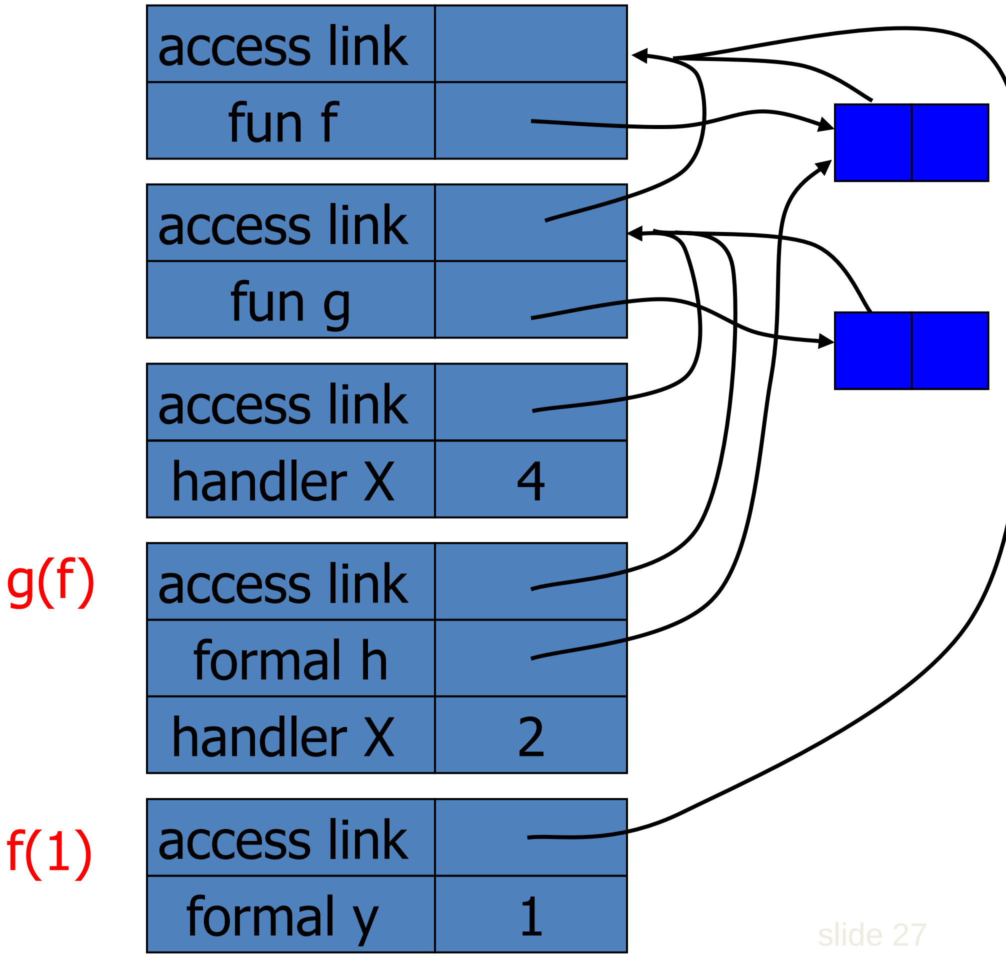
Which handler is used?

Dynamic Scope of Handlers (1)

```
exception X;
fun f(y) = raise X
fun g(h) = h(1) handle X => 2
g(f) handle X => 4
```

Dynamic scope:

find first X handler, going up the dynamic call chain leading to "raise X"



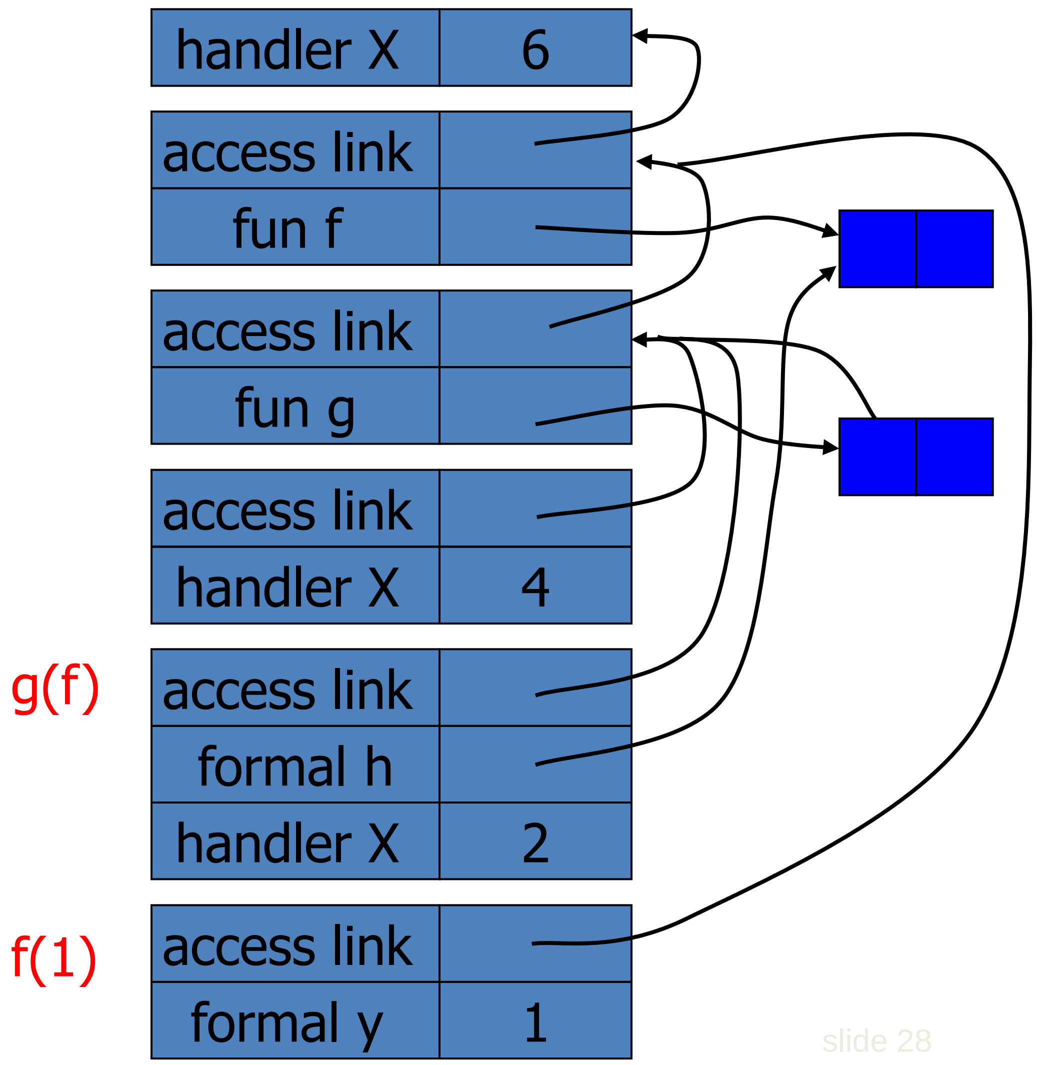
slide 27

Dynamic Scope of Handlers (2)

```
exception X;
(let fun f(y) = raise X
    and g(h) = h(1) handle X => 2
in
    g(f) handle X => 4
end) handle X => 6;
```

Dynamic scope:

find first X handler, going up the dynamic call chain leading to "raise X"



slide 28

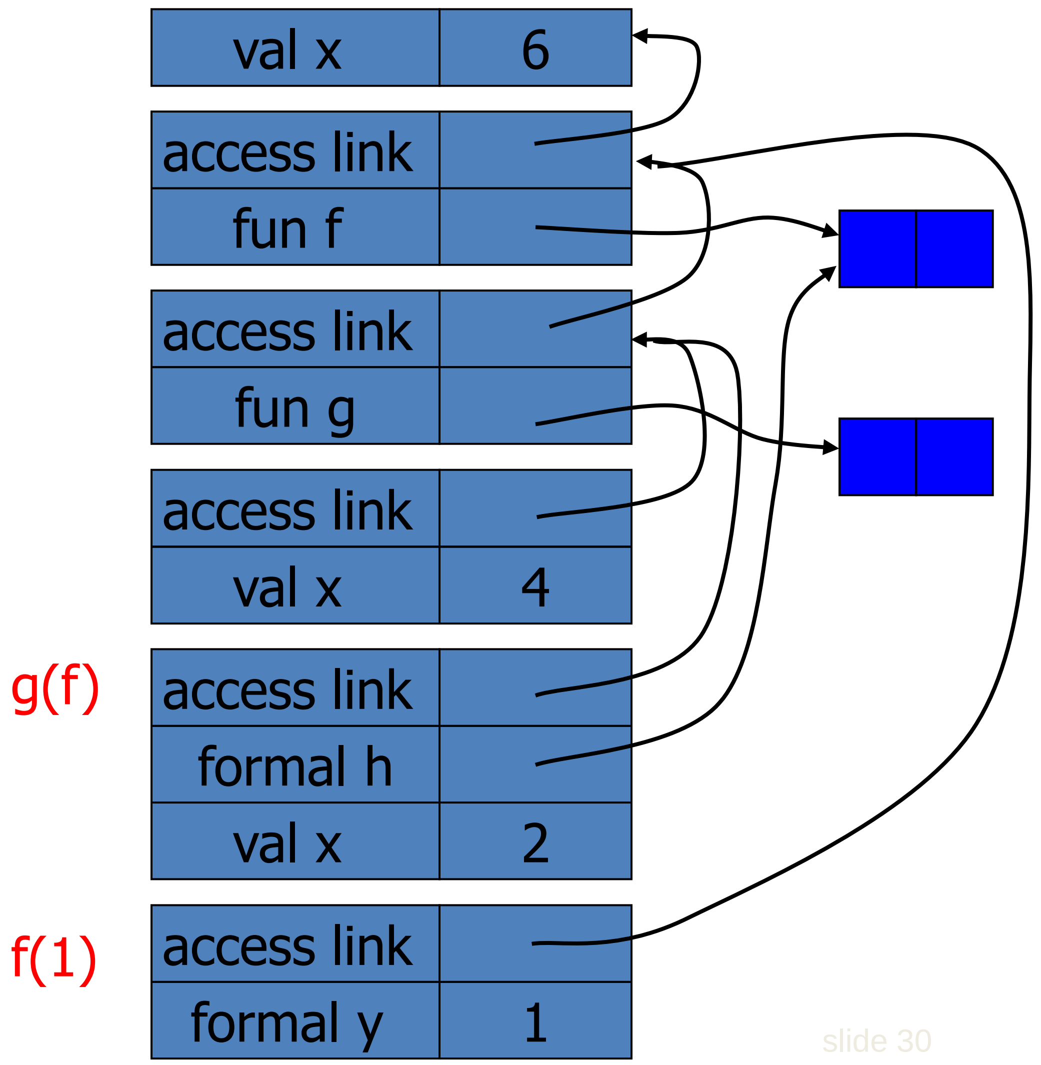
Scoping: Exceptions vs. Variables

```
exception X;
(let fun f(y) = raise X
    and g(h) = h(1)
                handle X => 2
in
    g(f) handle X => 4
end) handle X => 6;
```

```
val x=6;
(let fun f(y) = x
     and g(h) = let val x=2 in
                    h(1)
  in
     let val x=4 in g(f)
end);
```

Static Scope of Declarations

```
val x=6;
(let fun f(y) = x
    and g(h) = let val x=2 in
                  h(1)
 in
    let val x=4 in g(f)
end);
```



Static scope:

find first x, following access links from the reference to X

Typing of Exceptions

- Typing of raise ⟨exn⟩
  - Definition of typing: expression e has type t if normal termination of e produces value of type t
  - Raising an exception is not normal termination
    - Example: 1 + raise X
- Typing of handle ⟨exception⟩ => ⟨value⟩
  - Converts exception to normal termination
  - Need type agreement
  - Examples
    - 1 + ((raise X) handle X => e) Type of e must be int (why?)
    - 1 + (e_1 handle X => e_2) Type of e_1,e_2 must be int (why?)

Exceptions and Resource Allocation

```
exception X;
(let
   val x = ref [1,2,3]
 in
   let
       val y = ref [4,5,6]
   in
      … raise X
    end
end);  handle X => ...
```

- Resources may be allocated between handler and raise
  - Memory, locks on database, threads …
- May be "garbage" after exception

General problem,
no obvious solution

Garbage Collection

Major Areas of Memory

- Static area
  - Fixed size, fixed content, allocated at compile time
- Run-time stack
  - Variable size, variable content (activation records)
  - Used for managing function calls and returns
- Heap
  - Fixed size, variable content
  - Dynamically allocated objects and data structures
    - Examples: ML reference cells, malloc in C, new in Java

Cells and Liveness

- Cell = data item in the heap
  - Cells are "pointed to" by pointers held in registers, stack, global/static memory, or in other heap cells
- Roots: registers, stack locations, global/static variables
- A cell is live if its address is held in a root or held by another live cell in the heap

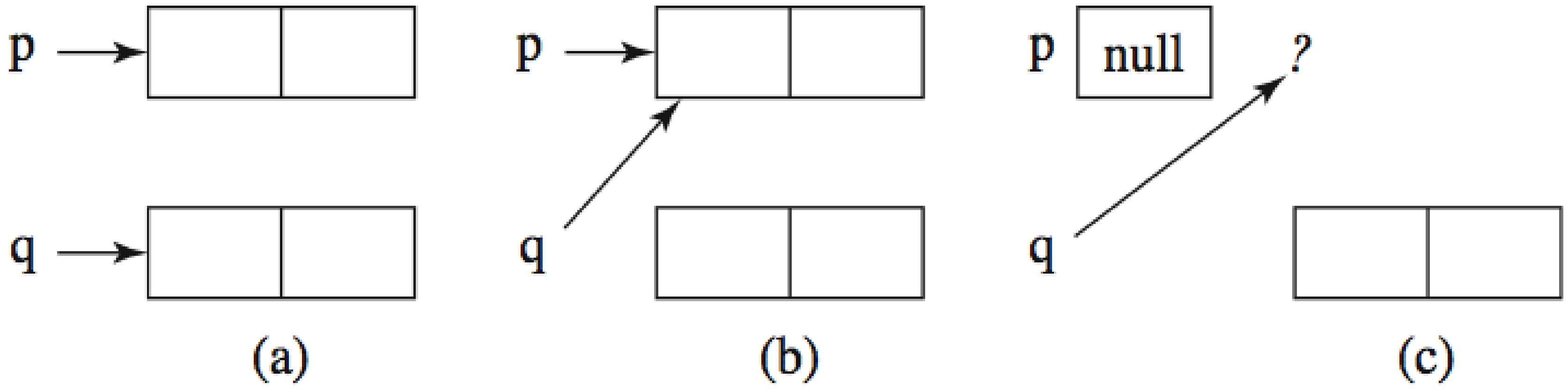
Garbage

- Garbage is a block of heap memory that cannot be accessed by the program
  - An allocated block of heap memory does not have a reference to it (cell is no longer “live”)
  - Another kind of memory error: a reference exists to a block of memory that is no longer allocated
- Garbage collection (GC) - automatic management of dynamically allocated storage
  - Reclaim unused heap blocks for later use by program

Example of Garbage

```
class node {
    int value;
    node next;
}
node p, q;
```

```
p = new node();
q = new node();
q = p;
delete p;
```



Why Garbage Collection?

- Today's programs consume storage freely
  - 8-16 GB laptops, 16-64GB desktops, 64-512GB servers
  - 64-bit address spaces (SPARC, Itanium, Opteron)
- … and mismanage it
  - Memory leaks, dangling references, double free, misaligned addresses, null pointer dereference, heap fragmentation
  - Poor use of reference locality, resulting in high cache miss rates and/or excessive demand paging
- Explicit memory management breaks high-level programming abstraction

GC and Programming Languages

- GC is <u>not</u> a language feature
- GC is a pragmatic concern for automatic and efficient heap management
  - Cooperative langs: Lisp, Scheme, Prolog, Smalltalk …
  - Uncooperative languages: C and C++
    - But garbage collection libraries have been built for C/C++
- Recent GC revival
  - Object-oriented languages: Modula-3, Java
    - In Java, runs as a low-priority thread; System.gc may be called by the program
  - Functional languages: ML and Haskell

The Perfect Garbage Collector

- No visible impact on program execution
- Works with any program and its data structures
  - For example, handles cyclic data structures
- Collects garbage (and only garbage) cells quickly
  - Incremental; can meet real-time constraints
- Has excellent spatial locality of reference
  - No excessive paging, no negative cache effects
- Manages the heap efficiently
  - Always satisfies an allocation request and does not fragment

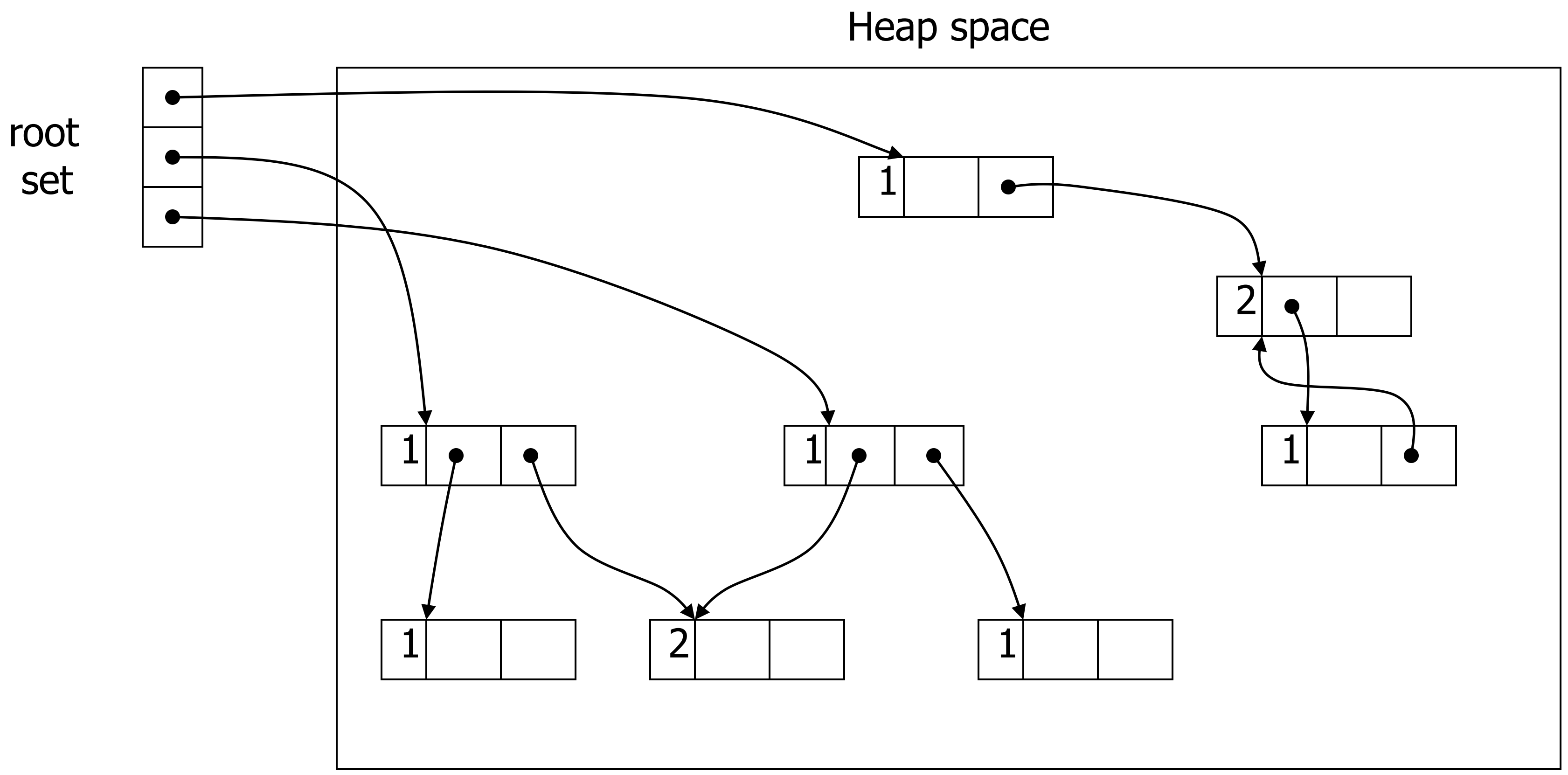
Summary of GC Techniques

- Reference counting
  - Directly keeps track of live cells
  - GC takes place whenever heap block is allocated
  - Doesn’t detect all garbage
- Tracing
  - GC takes place and identifies live cells when a request for memory fails
  - Mark-sweep
  - Copy collection
- Modern techniques: generational GC

Reference Counting

- Simply count the number of references to a cell
- Requires space and time overhead to store the count and increment (decrement) each time a reference is added (removed)
  - Reference counts are maintained in real-time, so no “stop-and-gag” effect
  - Incremental garbage collection
- Unix file system uses a reference count for files
- C++ “smart pointer” (e.g., auto_ptr) use reference counts

Reference Counting: Example



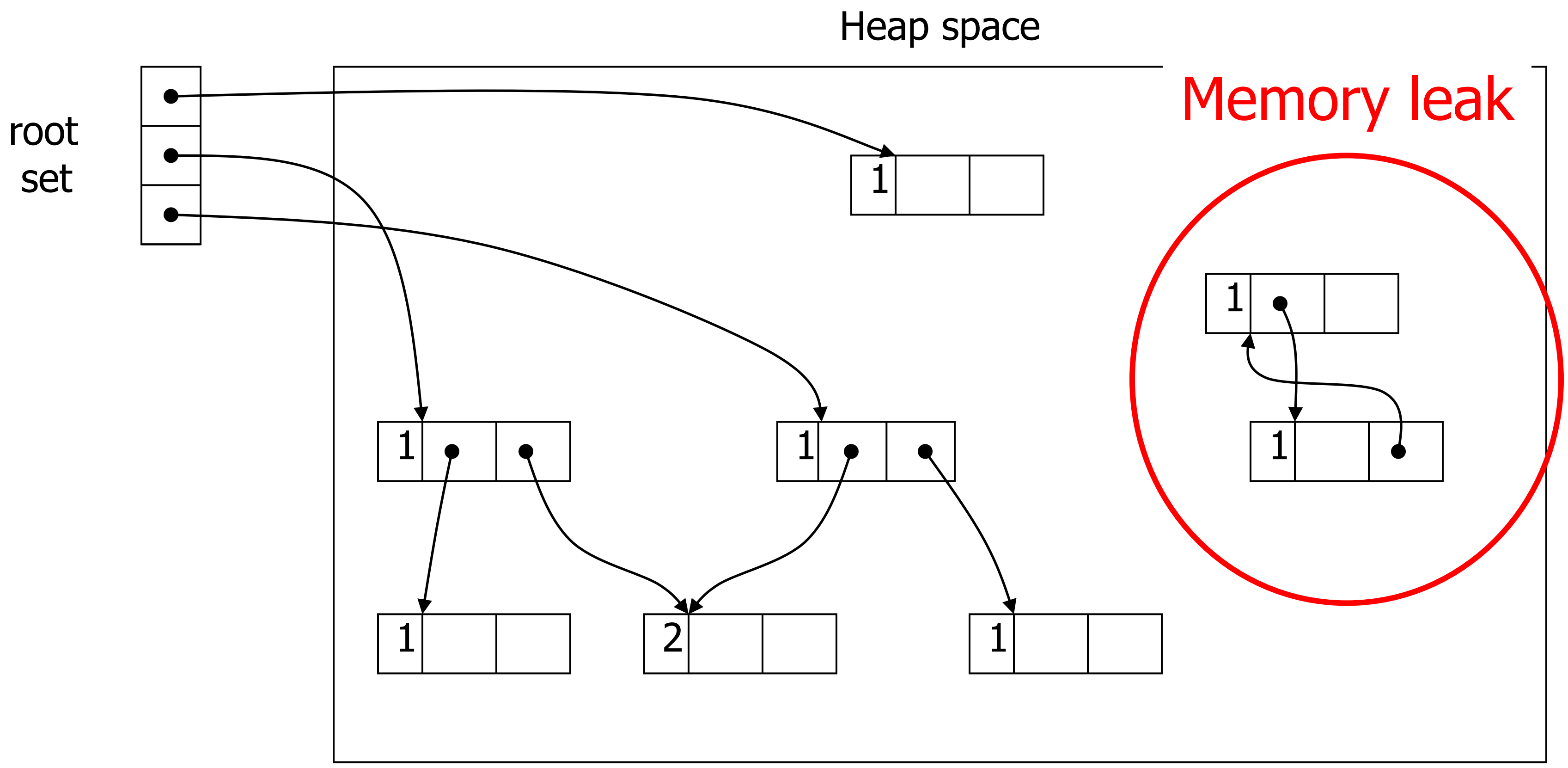
Reference Counting: Strengths

- Incremental overhead
  - Cell management interleaved with program execution
  - Good for interactive or real-time computation
- Relatively easy to implement
- Can coexist with manual memory management
- Spatial locality of reference is good
  - Access pattern to virtual memory pages no worse than the program, so no excessive paging
- Can re-use freed cells immediately
  - If RC == 0, put back onto the free list

Reference Counting: Weaknesses

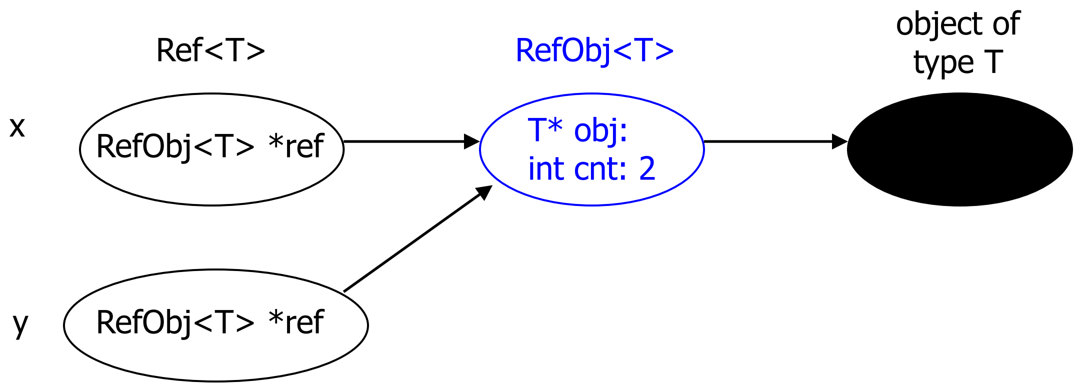
- Space overhead
  - 1 word for the count, 1 for an indirect pointer
- Time overhead
  - Updating a pointer to point to a new cell requires:
    - Check to ensure that it is not a self-reference
    - Decrement the count on the old cell, possibly deleting it
    - Update the pointer with the address of the new cell
    - Increment the count on the new cell
- One missed increment/decrement results in a dangling pointer / memory leak
- Cyclic data structures may cause leaks

Reference Counting: Cycles



"Smart Pointer" in C++

- Similar to std::auto_ptr<T> in ANSI C++



sizeof(RefObj<T>) = 8 bytes of overhead per reference-counted object

sizeof(Ref<T>) = 4 bytes

Fits in a register

Easily passed by value as an argument or result of a function

Takes no more space than regular pointer, but much "safer" (why?)

Smart Pointer Implementation

```
template<class T> class Ref {
    RefObj<T>* ref;
    Ref<T>* operator&() {}
public:
    Ref() : ref(0) {}
    Ref(T* p) : ref(new RefObj<T>(p)) { ref->inc();}
    Ref(const Ref<T>& r) : ref(r.ref) { ref->inc(); }
    ~Ref() { if (ref->dec() == 0) delete ref; }

    Ref<T>& operator=(const Ref<T>& that) {
      if (this != &that) {
        if (ref->dec() == 0) delete ref;
          ref = that.ref;
          ref->inc(); }
       return *this; }
    T* operator->() { return *ref; }
    T& operator*() { return *ref; }
};
```

```
template<class T> class RefObj {
     T* obj;
     int cnt;
public:
    RefObj(T* t) : obj(t), cnt(0) {}
    ~RefObj() { delete obj; }

    int inc() { return ++cnt; }
    int dec() { return --cnt; }

    operator T*() { return obj; }
    operator T&() { return *obj; }
    T& operator *() { return *obj; }
};
```

Using Smart Pointers

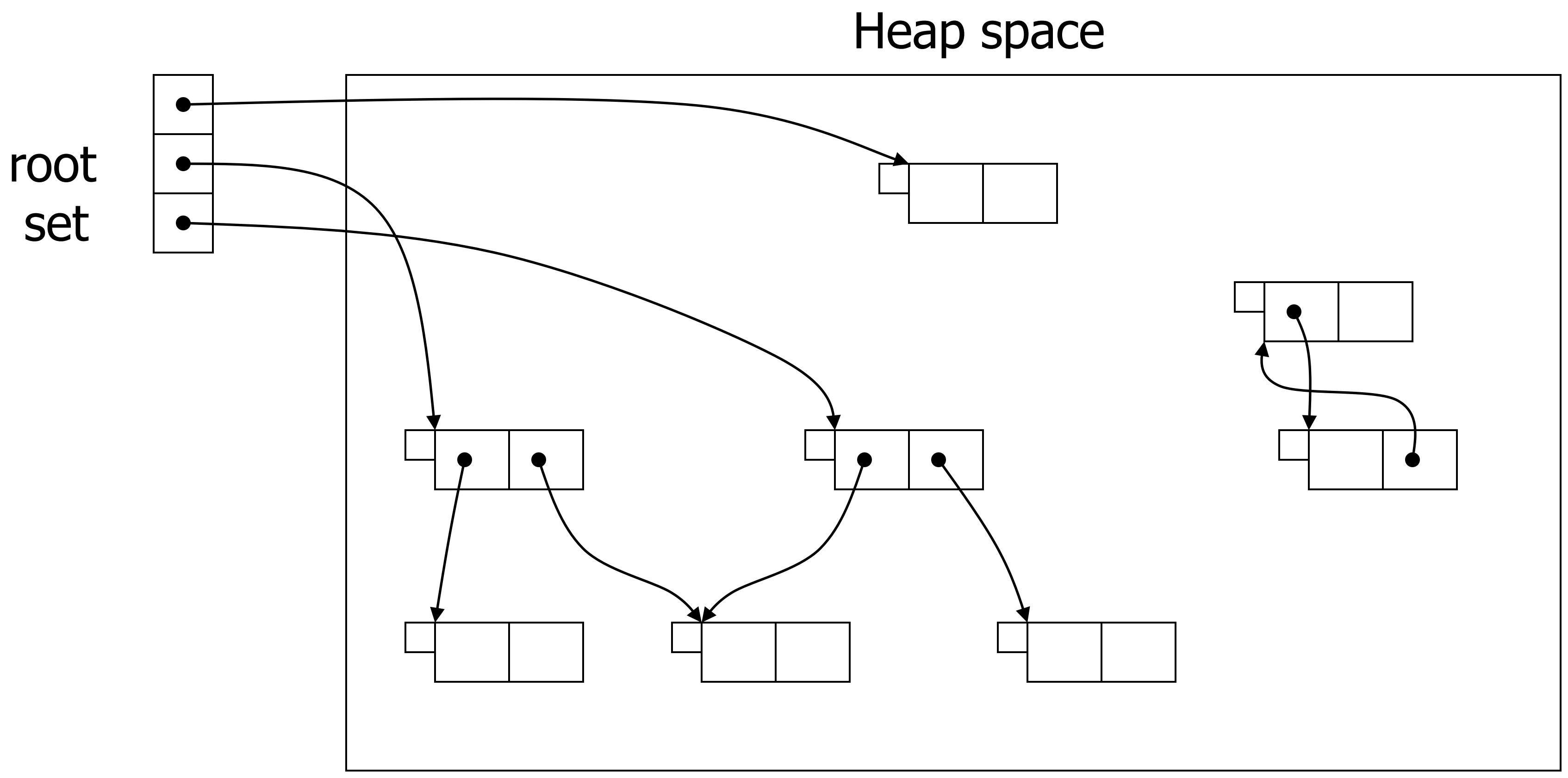
```
Ref<string> proc() {
      Ref<string> s = new string(“Hello, world”); // ref count set to 1
      …
      int x = s->length();  // s.operator->() returns string object ptr
      …
      return s;
} // ref count goes to 2 on copy out, then 1 when s is auto-destructed

int main()
{
      …
      Ref<string> a = proc();  // ref count is 1 again
      …
} // ref count goes to zero and string is destructed, along with Ref and RefObj objects
```

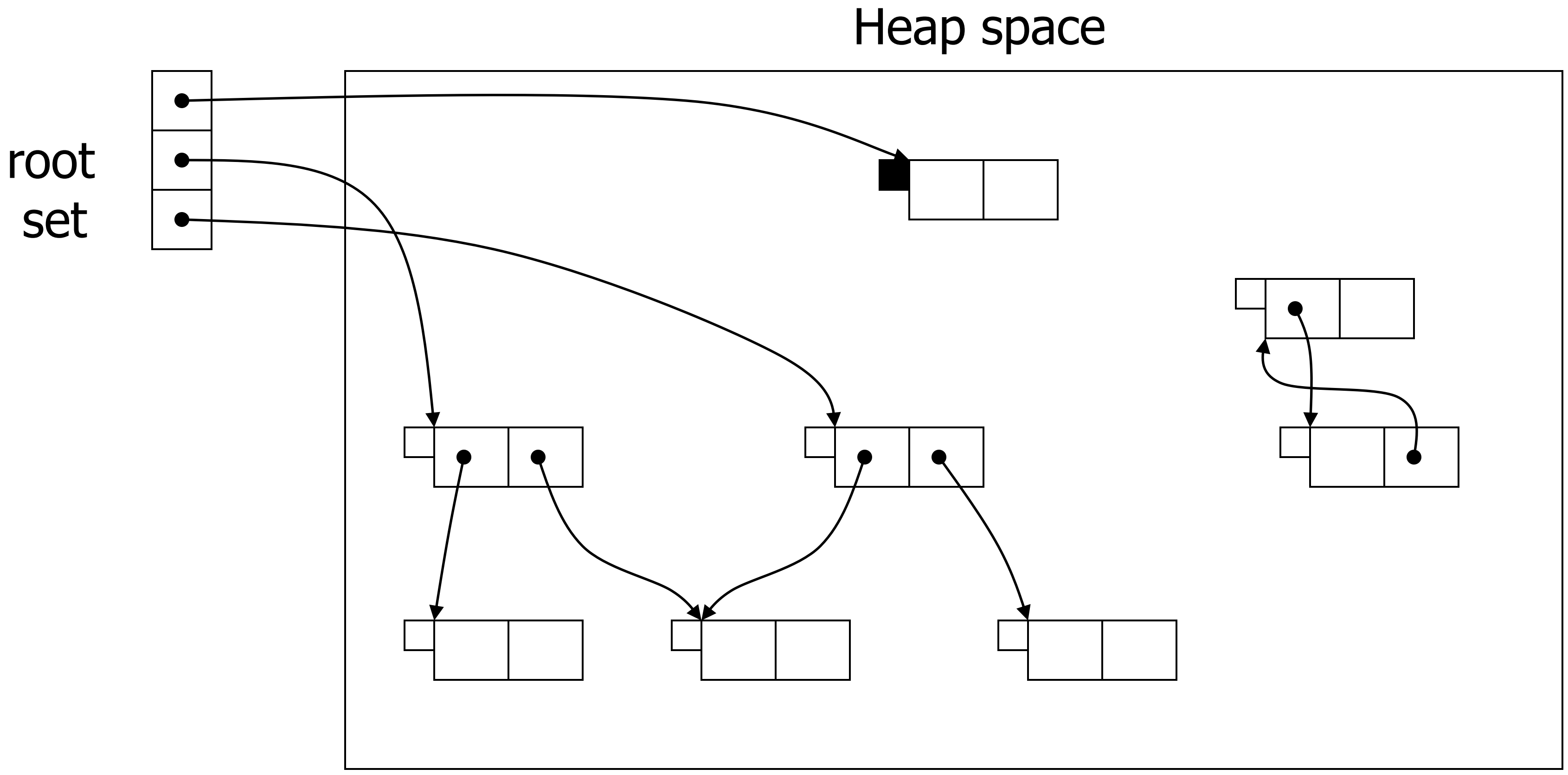
Mark-Sweep Garbage Collection

- Each cell has a mark bit
- Garbage remains unreachable and undetected until heap is used up; then GC goes to work, while program execution is suspended
- Marking phase
  - Starting from the roots, set the mark bit on all live cells
- Sweep phase
  - Return all unmarked cells to the free list
  - Reset the mark bit on all marked cells

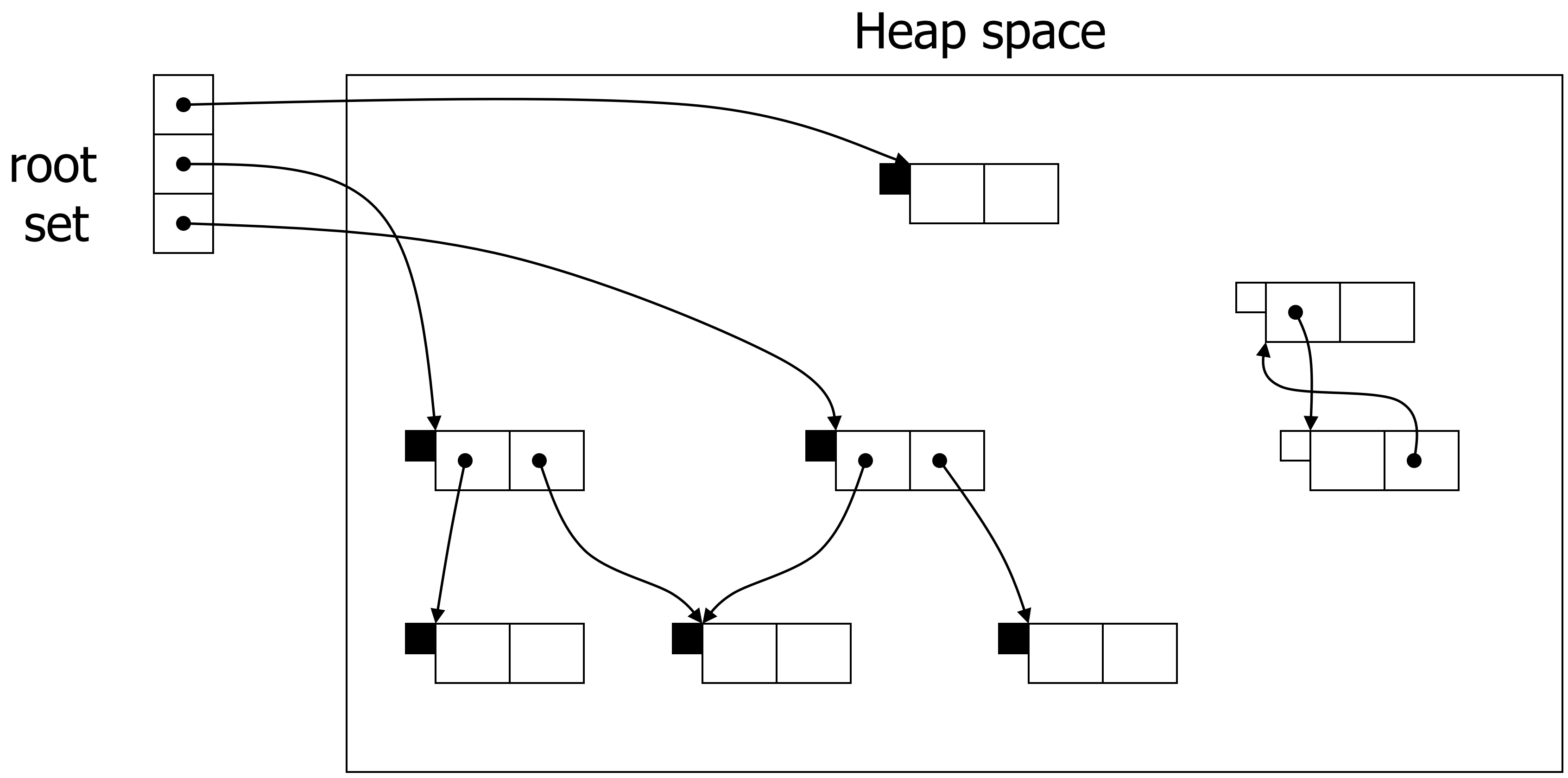
Mark-Sweep Example (1)



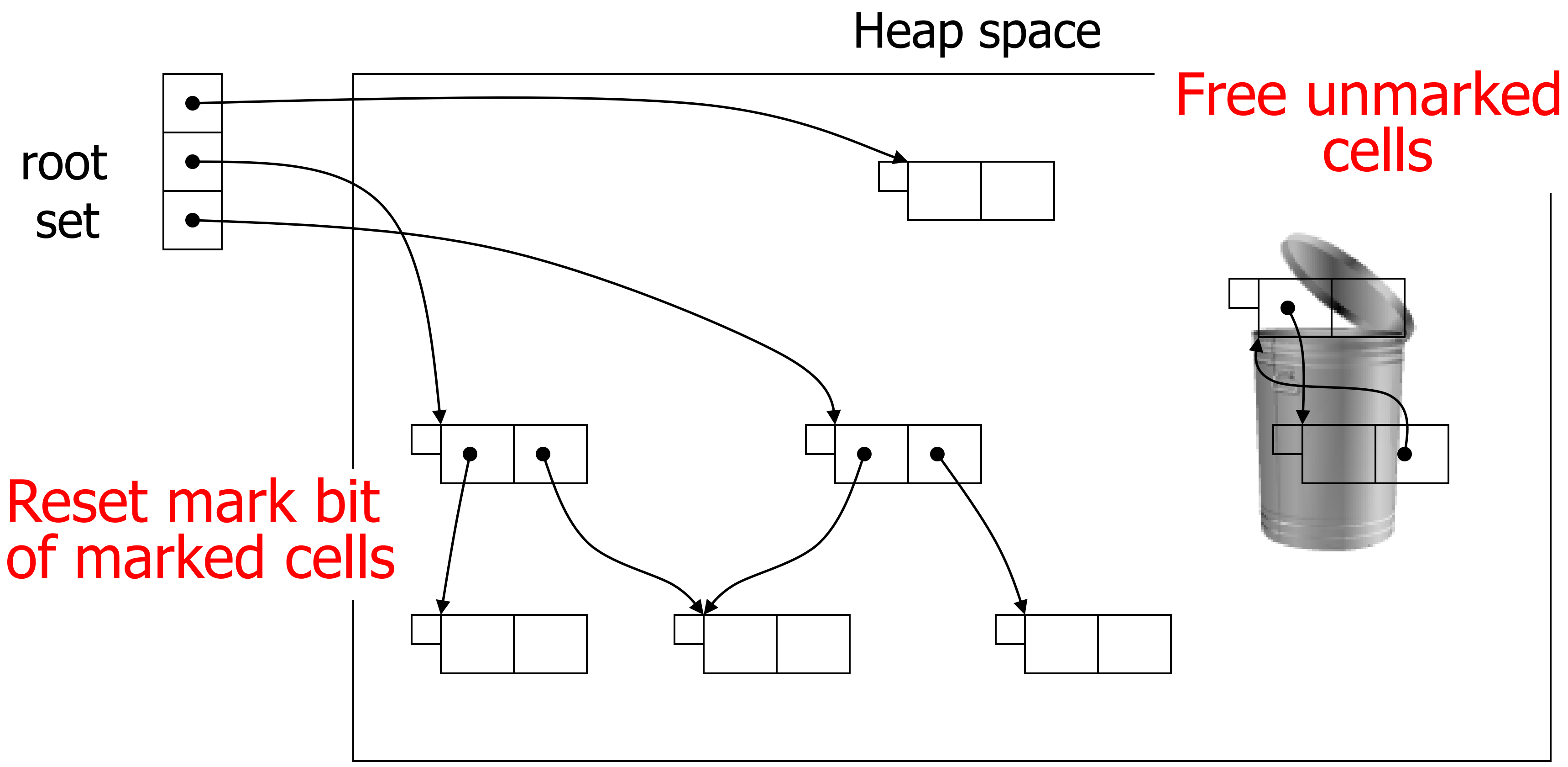
Mark-Sweep Example (2)



Mark-Sweep Example (3)



Mark-Sweep Example (4)



Mark-Sweep Costs and Benefits

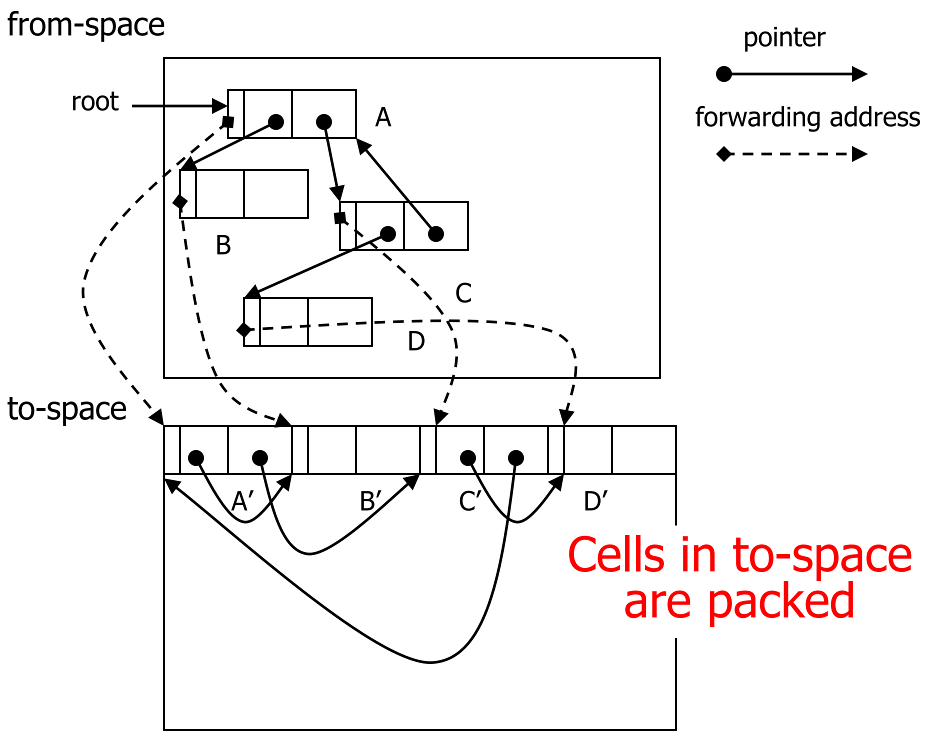
- Good: handles cycles correctly
- Good: no space overhead
  - 1 bit used for marking cells may limit max values that can be stored in a cell (e.g., for integer cells)
- Bad: normal execution must be suspended
- Bad: may touch all virtual memory pages
  - May lead to excessive paging if the working-set size is small and the heap is not all in physical memory
- Bad: heap may fragment
  - Cache misses, page thrashing; more complex allocation

Copying Collector

- Divide the heap into “from-space” and “to-space”
- Cells in from-space are traced and live cells are copied (“scavenged”) into to-space
  - To keep data structures linked, must update pointers for roots and cells that point into from-space
    - This is why references in Java and other languages are not pointers, but indirect abstractions for pointers
  - Only garbage is left in from-space
- When to-space fills up, the roles flip
  - Old to-space becomes from-space, and vice versa

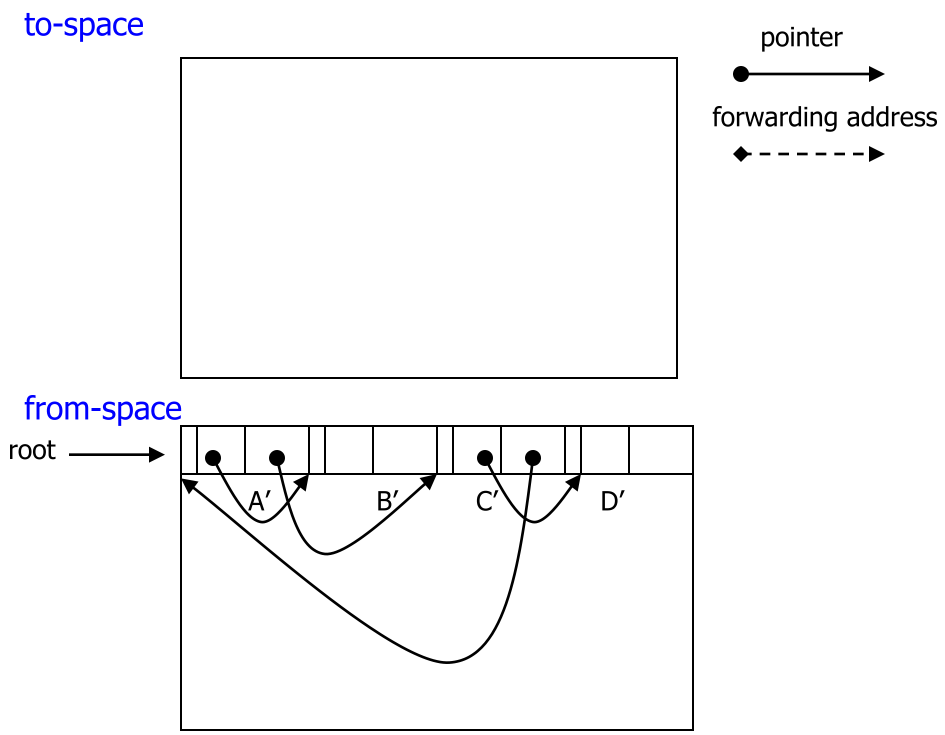
Copying a Linked List

[Cheney's algorithm]



Cells in to-space are packed

Flipping Spaces



Copying Collector Tradeoffs

- Good: very low cell allocation overhead
  - Out-of-space check requires just an addr comparison
  - Can efficiently allocate variable-sized cells
- Good: compacting
  - Eliminates fragmentation, good locality of reference
- Bad: twice the memory footprint
  - Probably Ok for 64-bit architectures (except for paging)
    - When copying, pages of both spaces need to be swapped in. For programs with large memory footprints, this could lead to lots of page faults for very little garbage collected
    - Large physical memory helps

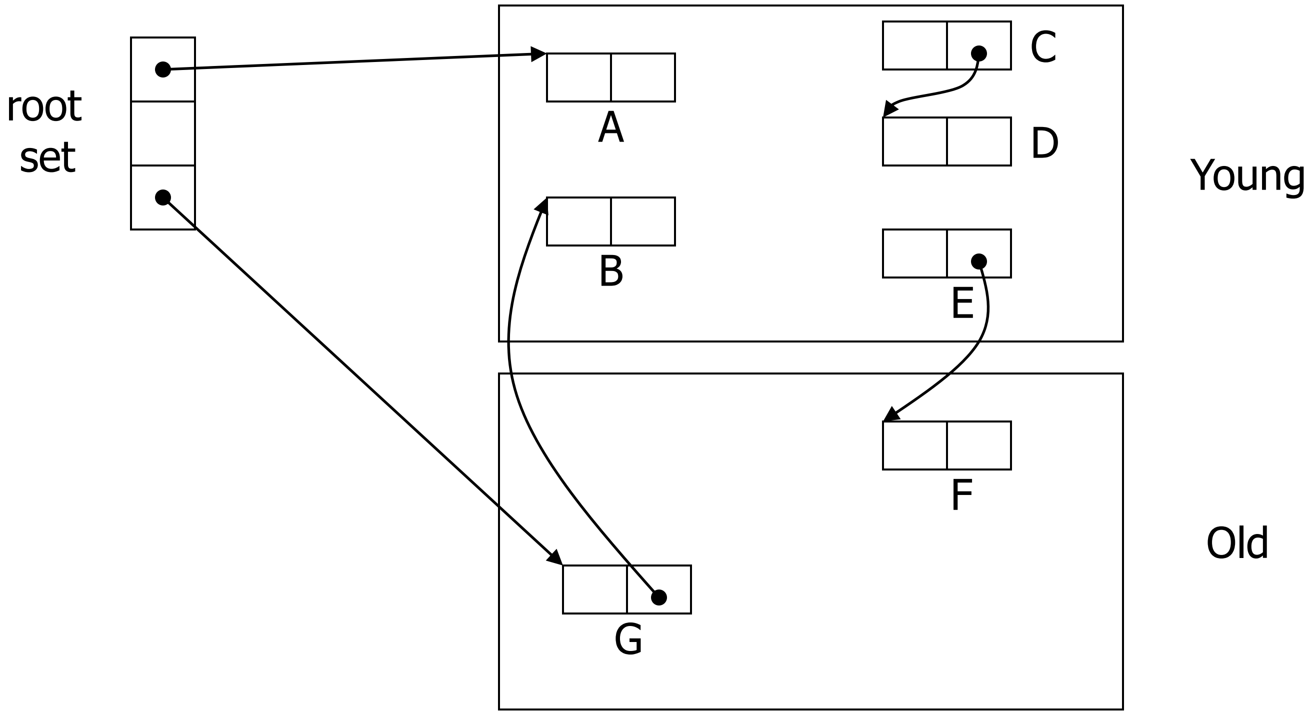
Generational Garbage Collection

- Observation: most cells that die, die young
  - Nested scopes are entered and exited more frequently, so temporary objects in a nested scope are born and die close together in time
  - Inner expressions in Scheme are younger than outer expressions, so they become garbage sooner
- Divide the heap into generations, and GC the younger cells more frequently
  - Don’t have to trace all cells during a GC cycle
  - Periodically reap the “older generations”
  - Amortize the cost across generations

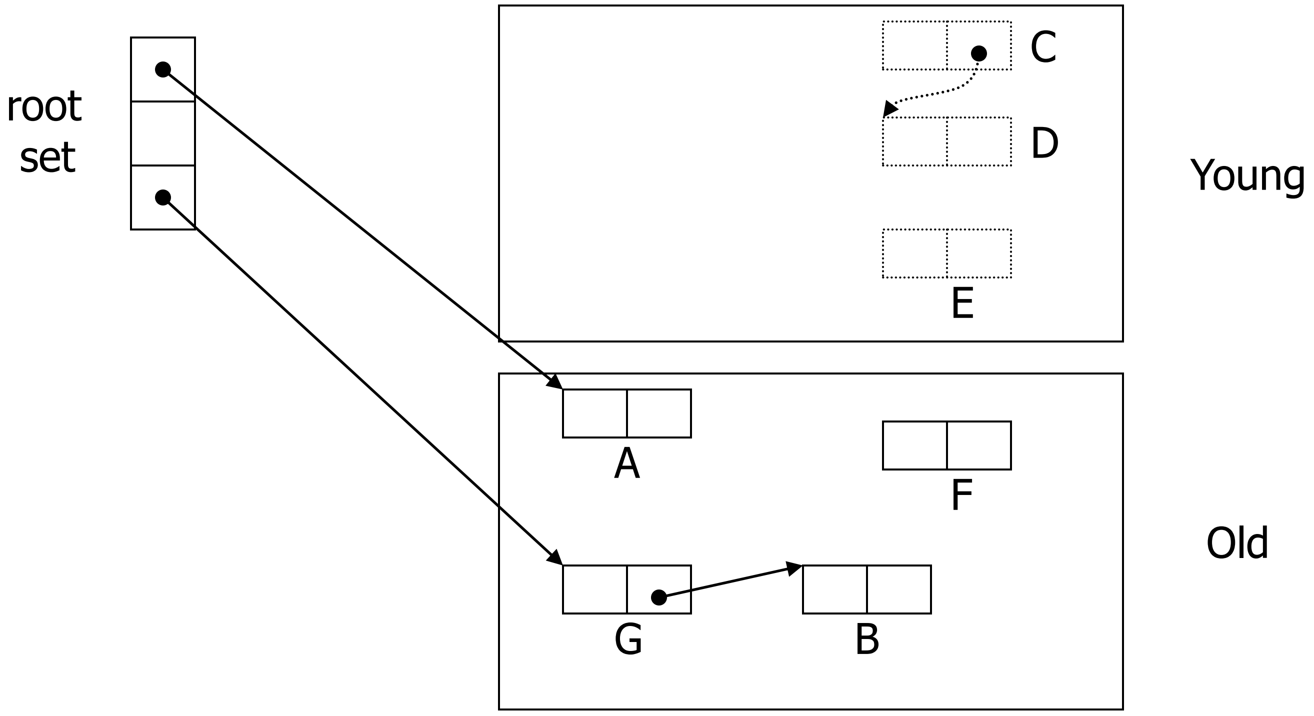
Generational Observations

- Can measure “youth” by time or by growth rate
- Common Lisp: 50-90% of objects die before they are 10KB old
- Glasgow Haskell: 75-95% die within 10KB
  - No more than 5% survive beyond 1MB
- Standard ML of NJ reclaims over 98% of objects of any given generation during a collection
- C: one study showed that over 1/2 of the heap was garbage within 10KB and less than 10% lived for longer than 32KB

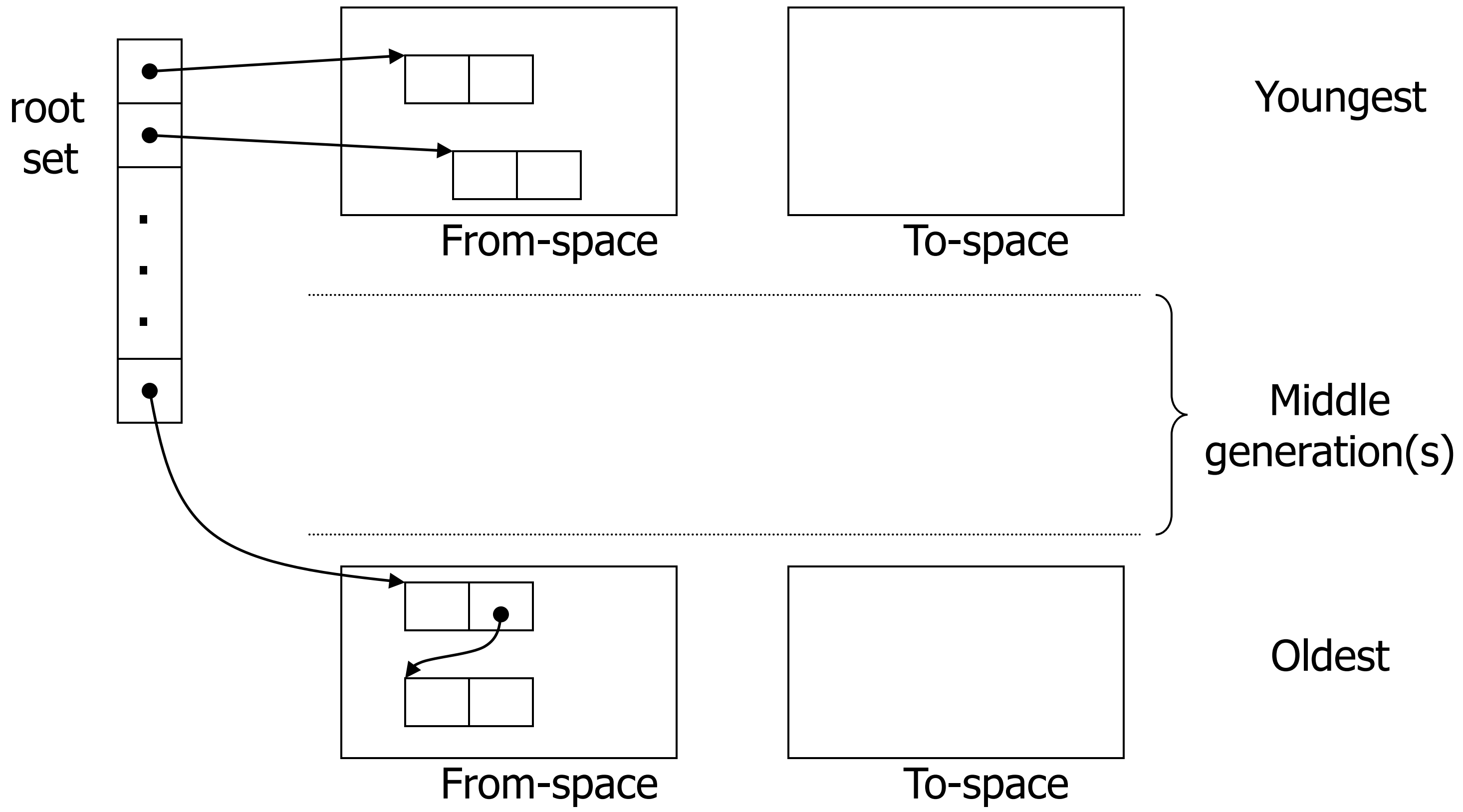
Example with Immediate "Aging" (1)



Example with Immediate "Aging" (2)



Generations with Semi-Spaces


root
set
From-space
To-space
Youngest
Middle
generation(s)
Oldest
From-space
To-space