COS 301

Programming Languages

Data Types Continued

Topics

- Pointer and Reference Types
- Type Equivalence
- Functions as Types
- Heap Management

Pointer and Reference Types

- A *pointer* type variable has a range of values that consists of memory addresses and a special value, *nil* or *null*
  - Null is a unique value that is always recognized as an invalid address
  - Usually implemented numerically as 0
  - Modern hardware has been designed to treat addresses with value 0 as invalid
- Pointers are used to:
  - Simulate indirect addressing
  - A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)
  - Provide anonymous variables

Design Issues for Pointers

- What are the scope of and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

Pointers and Reference Variables

- Pointers are variables that contain memory addresses, and thus point to data or objects
- Pointers can be explicitly dereferenced

```
x = *y;
```

- Some languages such as C allow pointer arithmetic

```
s += *a++;
```

  - The ++ increments the pointer
- Reference variables are also pointers, and contain memory addresses, but are never manipulated as such

```
Name = New String("myname");
```

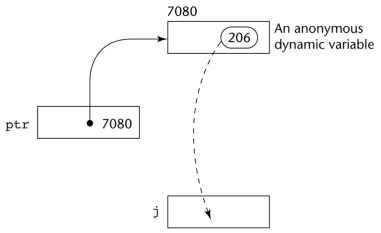
Pointer Operations

- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable's value to some useful address
- Dereferencing yields the value stored at the location represented by the pointer's value
  - Dereferencing can be explicit or implicit
  - C++ uses an explicit operation via *

```
j = *ptr
```

sets j to the value located at `ptr`

Pointer Assignment Illustrated



The assignment operation j = *ptr

Problems with Pointers

- Dangling pointers
  - A pointer that points to a heap-dynamic variable that has been deallocated
  - Memory pointed to could be another object or could be part of the control structures of the heap itself
- Lost heap-dynamic variable (garbage)
  - An allocated heap-dynamic variable that is no longer accessible to the user program
    - Pointer p1 is set to point to a newly created heap-dynamic variable
    - Pointer p1 is later set to point to another newly created heap-dynamic variable
    - We now have a memory leak

Pointers and Arrays in C

- In C an array variable passed to a function behaves like a pointer

```
float sum(float a[], int n) {
  int i;
  float s = 0.0;
  for (i =0; i < n; i++)
    s += a[i];
  return s;
}

float sum(float *a, int n) {
  int i;
  float s = 0.0;
  for (i =0; i < n; i++)
    s += *a++;
  return s;
}
```

Compiler generated code: array version

```
; Line 3
   mov  DWORD PTR _s$[ebp], 0
; Line 4
   mov  DWORD PTR _i$[ebp], 0
   jmp  SHORT $L472
$L473:
   mov  eax, DWORD PTR _i$[ebp]
   add  eax, 1
   mov  DWORD PTR _i$[ebp], eax
$L472:
   mov  ecx, DWORD PTR _i$[ebp]
   cmp  ecx, DWORD PTR _n$[ebp]
   jge  SHORT $L474
; Line 5
   mov  edx, DWORD PTR _i$[ebp]
   mov  eax, DWORD PTR _a$[ebp]
   fld  DWORD PTR _s$[ebp]
   fadd DWORD PTR [eax+edx*4]
   fstp DWORD PTR _s$[ebp]
   jmp  SHORT $L473
; Line 6
   fld  DWORD PTR _s$[ebp]
```

Compiler generated code: pointer version

```
   mov  DWORD PTR _s$[ebp], 0
; Line 12
   mov  DWORD PTR _i$[ebp], 0
   jmp  SHORT $L483
$L484:
   mov  eax, DWORD PTR _i$[ebp]
   add  eax, 1
   mov  DWORD PTR _i$[ebp], eax
$L483:
   mov  ecx, DWORD PTR _i$[ebp]
   cmp  ecx, DWORD PTR _n$[ebp]
   jge  SHORT $L485
; Line 13
   mov  edx, DWORD PTR _a$[ebp]
   fld  DWORD PTR _s$[ebp]
   fadd DWORD PTR [edx]
   fstp DWORD PTR _s$[ebp]
   mov  eax, DWORD PTR _a$[ebp]
   add  eax, 4
   mov  DWORD PTR _a$[ebp], eax
   jmp  SHORT $L484
$L485:
   fld  DWORD PTR _s$[ebp]
```

Optimized code (same for both versions)

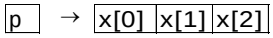
```
      mov    ecx, DWORD PTR _n$[esp-4]
   fld  DWORD PTR __real@4@00000000000000000000
   test ecx, ecx
   jle  SHORT $L531
; Line 3
   mov  eax, DWORD PTR _a$[esp-4]
$L529:
; Line 5
   fadd DWORD PTR [eax]
   add  eax, 4
   dec  ecx
   jne  SHORT $L529
$L531:
; Line 7
   ret  0
```

Pointers and Arrays in C

- A common misconception is that pointers and arrays are equivalent in C for example because this code returns 1

```
int x[3] = {1, 2, 3};
int *p = &x[0];     /* p points to the first element of x */
if (p[1] == x[1])
  return 1
else
 return 0;
```

But x is the actual address of the array, while p is a memory address that holds the address of x



C Pointer Arithmetic

```
void strcopy (char *s, char *t){
  /* a Kernighan and Ritchie classic */
  while (*s++ = *t++)
  ;
}

Ex push and pop a stack

  *p++ = val;  /* push */
  val = *--p;  /* pop  */
```

More Pointer Arithmetic in C and C++

```
float stuff[100];
float *p;
p = stuff;
```

***(p+5)** is equivalent to **stuff[5]**
***(p+i)** is equivalent to **stuff[i]**

Void Pointers

- C and C++ allow pointers of type **void ***
- These are in effect generic pointers
  - But they cannot be explicitly dereferenced
- Void pointers can be used to get around the type system

```
void * p;
char ch;
float num = 123.345;
p = &num;
ch = * (char *) p;
```

Reference Types

- C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters
  - A constant pointer that is always implicitly dereferenced
  - Primarily used to both pass paramters by reference rather than by value

```
void square(int x, int& result) {
      result = x * x;
  }
int myint = 123;
int z;
square(myint, &z);
```

Reference Types

- Java extends C++'s reference variables and allows them to replace pointers entirely
  - References are references to objects, rather than being addresses
- C# includes both the references of Java and the pointers of C++

Representation of Pointers

- In the wild-west days before ANSI C, programmers often treated pointers and ints as interchangeable types
  - Most machines had addresses that were the same size as the integer registers
- With Intel x86 microprocessors pointers became more complex (segment and offset in some cases)
- Since ANSI C, implementation has ceased to an issue of concerns to most programmers

Type Checking

- We can generalize the concept of operands and operators to include subprograms and assignments
  - Subprograms are operators whose operands are the parameters
  - Assignment operator is a binary operator with lhs and rhs being the two operands
- Type checking is the activity of ensuring that the operands of an operator are of compatible types
  - A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type
  - This automatic conversion is called a *coercion*.
- A *type error* is the application of an operator to an operand of an inappropriate type

2

- As a Character 00110010
- C Char 00000010
- Short 0000000000000010
- Int 00000000000000000000000000000010
- Float 01000000000000000000000000000000

Type Conversions

- A type conversion is a *narrowing* conversion if the result type permits fewer bits, thus potentially losing information.
- Otherwise it is termed a *widening* conversion.
- Note that any numeric type can be converted accurately to string (widening conversion)
- Double to integer is considered a narrowing conversion because the range of doubles is much larger
- Integer to double is usually considered a widening conversion
  - Any 32-bit int can be converted to a 64-bit double without loss of precision.
  - But a 32-bit int converted to a 32-bit float or a 64-bit int converted to a 64-bit float can lose precision

Type Casts and Coercions

- Type conversions can be explicit (usually called a *type cast*) or implicit (usually called a *coercion*)
- Language rules for implicit conversions are complex and a fruitful source of error
- Rules are especially complex with languages such as C that have signed and unsigned integer types

C Type Coercion Rules

IF	Then Convert
either operand is long double	the other to long double
either operand is double	the other to double
either operand is float	the other to float
either operand is unisgned long int	the other to unsigned long int
the operands are long int and unsigned int and long int can represent unsigned int	the unsigned int to long int
the operands are long int and unsigned int and long int cannot represent unsigned int	both operands to unsigned long int
one operand is long int	the other to long int
one operand is unsigned int	the other to unsigned int

The same paragraph from K&R also cautions “Unexpected results may occur when an unsigned expression is compared to a signed expression of the same size.”

Type Checking

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is *strongly typed* if type errors are always detected
  - Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Strong Typing

- Some examples:
  - Fortran 95 is not strongly type. EQUIVALENCE statements simply map one block of memory over another
  - C and C++: parameter type checking can be avoided; unions are not type checked
  - Ada is except when the generic function Unchecked_Conversion is used
  - Java and C# are strongly typed, but casts and coercions can introduce errors
  - Most scripting languages are weakly typed because free use of coercions can lead to type errors

Strong Typing

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada
  - And even though Java disallows narrowing coercions loss of precision can still occur

Type Equivalence

- Type compatibility

Type Equivalence

- From the Pascal report (Jensen and Wirth):

  The assignment statement serves to replace the current value of a variable with a new value specified as an expression. … The variable (or the function) and the expression must be of identical type.
- Unfortunately *identical type* was not defined
  - e.g., can an integer value be assigned to an enum variable?
- This problem was rectified with the ANSI/ISO Pascal standard

C example

```
struct complex {
     float re, im;
};
struct polar {
     float x, y;
};
struct {
     float re, im;
} a, b;
struct complex c, d;
struct polar e;
int f[5], g[10];
```

- Which of these are equivalent types?

Type Equivalence

- Name Equivalence: two types are equivalent if they have the same name
  - 3 type-equivalent groups of variables a,b - c,d - e
- Structural Equivalence: two types are equivalent if they have the same structure
- C uses structural equivalence for all types except unions and structs where member names are significant
  - 2 type-equivalent groups of variables a,b,c,d - e
  - Note that types complex and polar have different member names

Type Equivalence and Pointers in C

- A pointer to a float is structurally equivalent to a pointer to an int, but the object pointed to determines type equivalence
  - Note that void * is a generic "pointer to anything"
- In the array declarations `int f[5], g[10];` we have two different types (K&R ANSI C: "Array sizes and function parameter types are significant")

Type Equivalence in Ada and Java

- Ada uses name equivalence for all types, including arrays and pointers and forbids most anonymous types
- Java uses name equivalence for classes; method signatures must match for implementations of interfaces

Name Type Equivalence

- *Name type equivalence* means the two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
  - Subranges of integer types are not equivalent with integer types
  - Formal parameters must be the same type as their corresponding actual parameters

Structure Type Equivalence

- *Structure type equivalence* means that two variables have equivalent types if their types have identical structures
- More flexible, but harder to implement

Type Equivalence (continued)

- Consider the problem of two structured types:
  - Are two record types equivalent if they are structurally the same but use different field names?
  - Are two array types equivalent if they are the same except that the subscripts are different?
    (e.g. [1..10] and [0..9])
  - Are two enumeration types equivalent if their components are spelled differently?
  - With structural type equivalence, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Functions as Types

- The inability to assign a function as the value of a variable in many languages is usually expressed as "functions are not first-class citizens" of the language
- It's easy to see where functions could be useful when passed as parameters
  - A graphing routine that graphs y = f(x)
  - Root solver for y = f(x)
  - Quicksort routine for arbitrary types need a comparator
- Same concept is present in operating systems APIs where "callbacks" are used
  - E.g., create a timer that "calls back" after specified clock ticks have elapsed

Function Parameters

- Primary need is to pass functions as parameters
- Languages have varying levels of support
- In general functional languages have the best support (discussed further in Ch 14)
- Fortran allowed function pointers, but could not do type checking
- Pascal-like languages have function prototype in the parameter list

```
Function Newton (A,B : real; function f(x: real):real): real;
```

Function Pointers in C

- K&R 2nd Ed (ANSI C):
- "In C a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on."

Java Interface Classes

- An abstract type that is used to specify an interface that classes must implement.
- Contains method signatures only; no body

```
public interface RootSolvable {
        double valueAt(double x);
}
```

- Any class method that implements this interface can be passed as an argument

```
public double Newton(double a, double b, RootSolvable f);
```

First-Class Citizenship

- Generally the criterion is the ability of a function to construct a function at runtime and return it as a value
- This ability is characteristic of and generally confined to the functional languages
- But one very widely used language – Javascript – treats functions as first-class citizens.

Javascript Example – iQuery Extension

- This example is a menu animation

```
(function($){
  $.fn.extend({
      //plugin name - animatemenu
      animateMenu: function(options) {
            var defaults = {
                animatePadding: 60,
                  defaultPadding: 10,
                  evenColor: '#ccc',
                  oddColor: '#eee',
            };
      var options = $.extend(defaults, options);
```

Javascript Example – iQuery Extension

```
    return this.each(function() {
      var o =options;
      var obj = $(this);
      var items = $("li", obj);

      $("li:even", obj).css('background-color', o.evenColor);

      $("li:odd", obj).css('background-color', o.oddColor);


      items.mouseover(function() {
            $(this).animate({paddingLeft: o.animatePadding}, 300);

      }).mouseout(function() {
            $(this).animate({paddingLeft: o.defaultPadding}, 300);

      });
  });
}
});
})(jQuery);
```

jQuery Extension

- Source:
  http://www.queness.com/post/112/a-really-simple-jquery-plugin-tutorial
- The program is invoked as follows:

```
$(document).ready(function() {
    $('#menu').animateMenu({animatePadding:
        30, defaultPadding:10});
});
```

Javascript Example – iQuery Extension

```
  function debug($obj) {
    if (window.console && window.console.log)
      window.console.log('hilight selection count: ' +
   $obj.size());
  };
  $.fn.hilight.format = function(txt) {
    return '<strong>' + txt + '</strong>';
  };
  $.fn.hilight.defaults = {
    foreground: 'red',
    background: 'yellow'
  };
})(jQuery);
```

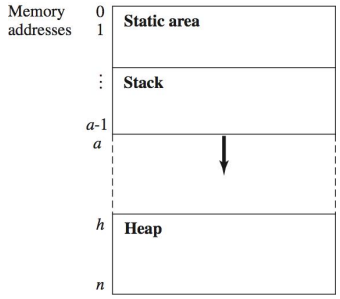
Memory and Heap Management

- Bjarne Stroustrop (designer of C++)
  "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."

The Heap

- The major areas of memory:
- *Static area*: fixed size, fixed content, allocated at compile time
- *Run-time stack*: variable size, variable content, center of control for function call and return
- *Heap*: fixed size, variable content, dynamically allocated objects and data structures

Structure of Run-Time Memory



Solving the Dangling Pointer Problem

- 1. Tombstone: an extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable de-allocated, tombstone remains but set to null
  - This prevents dangling pointers but is expensive in time and space
    - Extra space for the tombstone
    - Every heap reference requires one more indirect memory access

.

Solving the Dangling Pointer Problem

- 2. Locks-and-keys: Pointer values are represented as (key, address) pairs
  - Heap-dynamic variables are represented as variable plus cell for integer lock value
  - When heap-dynamic variable allocated, lock value is created and placed in lock cell of heap variable and in the key cell of pointer
  - Any copies of the pointer also copy the key value
  - With every heap access lock and key are compared
    - If they match the memory access is legal
  - When the heap variable is deallocated, a different value is placed in the lock cell

Heap Management

- Heap management (garbage collection) is a very complex run-time process
- Approaches to reclaim garbage
  - Reference counters (*eager approach*): reclamation is gradual
  - Mark-sweep (*lazy approach*): reclamation occurs when the list of variable space becomes empty
  - Copy collector: like mark and sweep, but heap is divided into two blocks and space is compacted by copying into the other block on garbage collection

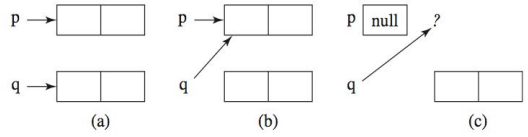
Garbage Collection

- *Garbage* is any block of heap memory that cannot be accessed by the program – no pointer in the program references that block
- Garbage can occur when either:

  1. An allocated block of heap memory has no reference to it (an "orphan" or a "memory leak"), or
  2. A reference exists to a block of memory that is no longer allocated (a "widow").

Garbage Example

```
class node {
  int value;
  node next;
}
node p, q;
p = new node();
q = new node();
q = p;
delete p;
```



Programmer or Runtime?

- If heap allocation is exclusively the responsibility of the programmer:
  - Poor programming can result in memory leaks
  - Programming is more complex
  - Bugs can be difficult to find
- But
  - For a well-designed, error free program, performance will be better than runtime garbage collection
- Garbage collection algorithms are complex and costly in time

Garbage Collection Algorithms

- Garbage collection algorithms were first designed and used in the 1960's
- Emergence of OOP in the 1990's renewed interest in the field
- Inactive objects or areas of the heap that are no longer in use are "garbage"
- Garbage collection has two functions:

1. Reclaim garbage and return it to the free space list;
2. Compact free space into the largest possible blocks.

Criteria for Evaluating Garbage Collection

- Pause time and predictability. Some GC's "stop the world" while they operate
- Memory usage. The actual heap may be double or more the size of memory available to the program
- Virtual memory. Does the GC cause page faults in normal operation? Cache misses?
- Can the GC improve locality of reference for the program?
- How much runtime bookkeeping is needed? How does this impact program speed?

Garbage Collection Algorithms

- Three classical garbage collection strategies:
  - Reference Counting - occurs whenever a heap block is allocated, but doesn't detect all garbage.
  - Mark-Sweep - Occurs only on heap overflow, detects all garbage, but makes two passes on the heap. Also known as a "tracing collector" because it traces memory references from the program and runtime stack.
  - Copy Collection - Faster than mark-sweep, but reduces the size of the heap space.

Reference Counting

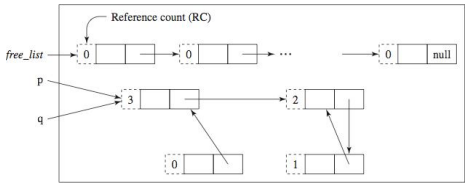
- With reference counting an object allocated in the heap has an additional field (the reference count) that tracks the number of pointers pointing to that object
- When the reference count becomes 0 the object is no longer in use and memory can be returned to the heap
- Not a very effective algorithm
  - Increases the cost of an assignment operation
  - Linked orphans are never detected
  - Requires additional storage space for reference counts
- Has some advantages:
  - Objects reclaimed as soon as possible
  - No long pauses while garbage collector inspects heap

Reference Counting

- The heap is a chain of nodes (the *free_list).*
  - *Each node has a reference count (RC) field that contains a count of the number of pointers referencing the object*
  - *As the program runs nodes are taken from free list and connected to each with pointers, forming chains*
- Algorithm is activated dynamically with calls to new and `delete`
  - *With new, a heap node is allocated and reference count set to 1*
  - *With `delete`, reference count is decrementd for the node and all linked nodes. Nodes with RC=0 are returned to the free list.*

Pointer Assignments

- Assignment of a pointer variable incurs a fair amount of bookkeeping work. Assuming q = p:
  - RC for p (and all linked nodes) is incremented
  - RC for q (and all linked nodes) is decremented
  - Any nodes where RC = 0 are returned to the free list
  - q is then assigned the value of p



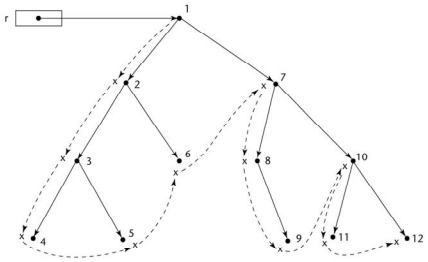
Reference Counting

- *Disadvantages*: space required, execution time required, complications for cells connected circularly
- *Advantage*: it is intrinsically incremental, so significant delays in the application execution are avoided

Mark-Sweep

- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins
  - Every heap cell has an extra bit used by collection algorithm
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All garbage cells returned to list of available cells

Marking Algorithm



Dashed lines show the order of node_marking

Mark-Sweep

- Mark-Sweep only takes place when the heap is full
  - Heap operations have little overhead cost
  - But mark and sweep is very time consuming when invoked
- Semantics of heap allocation `q = new node()` :

```
if (free_list == null)
  mark_sweep();
if (free_list != null) {
  q = free_list;
  free_list = free_list.next;
}
else abort('Heap full')
```

The Root Set

- If we're going to inspect all objects we have to be able to find ones with active references, sometimes called the "root set"
- This contains
  - Pointers in global memory
  - Pointers on the stack
- This is a fairly complex topic in itself. Java defines six classes of reachability:
  - Strongly reachable, weakly reachable, softly reachable, finalizable, phantom reachable, unreachable

Mark-Sweep

- Each node in the *free_list* has a mark bit (MB) initially 0.
- Requires two passes through the heap
- When heap overflow occurs:

  Pass I: Mark all nodes that are (directly or indirectly) accessible from the root set by setting their MB=1.

  Pass II: Sweep through the entire heap and return all unmarked (MB=0) nodes to the free list.
- *Note: all orphans are detected and returned to the free list.*

Problems with Mark-Sweep

- Because all memory referenced by the program must be inspected and every allocated object must be visited, GC pauses can be significant
- Problems with page faults when visiting old, inactive objects
- Leaves heap space fragmented; may run out of heap when allocating a large objects because all of the blocks are small
- Need to maintain complex structures such as linked lists of free blocks

Copy Collection

- A time-space compromise compared to mark-and -sweep: more space, less time
  - The heap is partitioned into two halves (from_space and to_space); only one is active.
  - No additional space for reference counts or mark bits
  - Does not require a free list; just a free pointer to the end of the allocated area
- Heap initialization:

```
from_space = h;
top_of_space = h + (n - h)/2;
to_space = top_of_space + 1;
free = from_space;
```

Copy Collection

- When a new node is allocated the next available block referenced by *free* is allocated and value of *free* is updated
- Semantics of heap allocation `q = new node()` :

```
if (free + 1 > top_of_space)
  flip();
if (free + 1 > top_of_space)
 abort('Heap full');
q = free;
free = free + 1;
```

Advantages and disadvantages

- Biggest advantage is heap compaction: improves locality of reference; free heap is one big block always; allocation is cheap and easy (move heap pointer n bytes)
- Disadvantages are increased memory usage, page faults when copying; copying overhead
- Does well when most objects are short-lived; degrades with long-lived objects

Generational Garbage Collection

- Empirical studies show that most objects created and used in OOPs tend to “die young.”
- Pure copying collectors do not visit dead objects; they simply copy live objects to another heap area
- If an object survives one garbage collection, then there is a good chance it will become long lived or permanent
  - Most source state that over 90% of garbage-collected objects have been created since the last GC pass

Generational Garbage Collection

- Generational GCs divide the heap into two or more generations
  - Objects that meet some promotion criteria are promoted to a longer lived generation
  - Different algorithms may be used for different generations
- Minor Collections
  - When the heap manager is unable to satisfy a request, it can trigger a ‘minor’ collection that only operates on the youngest generation
  - Proceed to older generations if necessary
  - Expand heap or return out-of-memory error only when all generations have been collected.

Intergenerational References

- Tracing collectors start from the root set and trace all memory references. A generational GC also does this but only visits objects in the youngest generation
- What if an object in an older generation references an object in the youngest generation that is not otherwise reachable?
- Solution is to explicitly track intergenerational references
  - Easy to do when an object is promoted
  - More difficult when pointer references are changed

Tracking intergenerational references

- Naïve algorithm would check every pointer assignment to see if it is an intergenerational reference
- The most common algorithm is called a "card table" or "card marking"
  - A structure called a card map contains one bit per block of memory (usually smaller than a virtual memory page).
  - A set bit means that the memory is dirty (written to)
  - When a pointer is assigned or modified the corresponding bit in the card map is set
  - During GC the cards for the older generations are scanned for objects with references to a younger generation and set bits are cleared

Garbage Collection in Java and .NET

- See

Java theory and practice: A brief history of garbage collection

http://www.ibm.com/developerworks/library/j-jtp10283/

Java theory and practice: Garbage collection in the HotSpot JVM
http://www.ibm.com/developerworks/java/library/j-jtp11253/index.html

.NET Garbage collection - MSDN documentation
http://msdn.microsoft.com/en-us/library/0xy59wtx.aspx

Garbage collection in .NET
http://www.csharphelp.com/archives2/archive297.html