

Lecture 17 — Garbage collection, and miscellaneous

- V-tables (carried over from lecture 16)
- The structure of compilers; code optimization
- Automatic memory management
 - Non-reachable heap nodes
 - Reference-counting
 - Garbage collection via mark-and-sweep
 - Garbage collection via stop-and-copy

V-tables

- Draw a table for each class, listing all the methods belonging to that class (including inherited ones). The order should be from top of the hierarchy to the bottom.

```
class B {  
    void f() {}  
    void g() {}  
}
```

```
class C1 extends B {  
    void h() {}  
}
```

```
class C2 extends B {  
    void g() {}  
}
```

```
class D extends C1 {  
    void i() {}  
    void g() {}  
}
```

Compiler structure

- Compilers don't usually translate directly from AST to native machine code.
- IR = “intermediate representation” (often a simplified machine language)
- MI = “machine-independent”
- Code gen. incorporates machine-dependent optimization

Machine-independent optimizations

- Reducing total number of operations
- Reducing amount of work in loops

⇒ Optimizations that are likely to increase speed on *any* machine.

```
int A[100][100];  
  
while (j<n) {  
    x = x + A[i][j];  
    j++;  
}
```

```
t1 = &A  
t2 = i*100  
t3 = t2+j  
t4 = t3*4  
t5 = t1+t4  
t6 = LOADIND t5  
x = x+t6  
j = j+1
```

Machine-dependent optimization

- Machine-dependent optimization = optimizations that exploit features of target machine such as registers, pipeline, special instructions
 - Register allocation
 - Instruction selection
 - Instruction scheduling

Compilation targets: Real vs. abstract machines

- ***Traditional:*** Compile to target machine
 - Produces most efficient code
- ***Java/C#:*** Compile to virtual (i.e. fake) machine code (JVM or CLR/CIL)
 - Execute by interpreting that machine, *or*
 - Translate to native machine code *at load time or run time*
 - Advantages: portability, security
 - Disadvantage: Non-optimal performance
- **LLVM:** optimizable abstract machine

Dynamic languages vs. static languages

- **Dynamic:** Python, Perl, Ruby, JavaScript
- **Static:** C, C++, Fortran
- **Intermediate:** Java, C#, OCaml
- **Two primary distinguishing features:**
 - **Dynamic type-checking; tagged values**
 - **Automatic memory management**

Automatic memory management

- Objects allocated in the heap can become unreachable.

```
let rec range m n = if m>n then [] else m :: range (m+1) m
let x = range 1 100 @ range 1 100
```

- Unreachable nodes are called *garbage*.
- Depending on programming style, programs can create a lot of garbage or none; modern programming styles, as used in scripting languages, generate a lot of garbage.
- These languages would be unusable without *automatic memory management*.

Reachability (aka accessibility) of heap cells

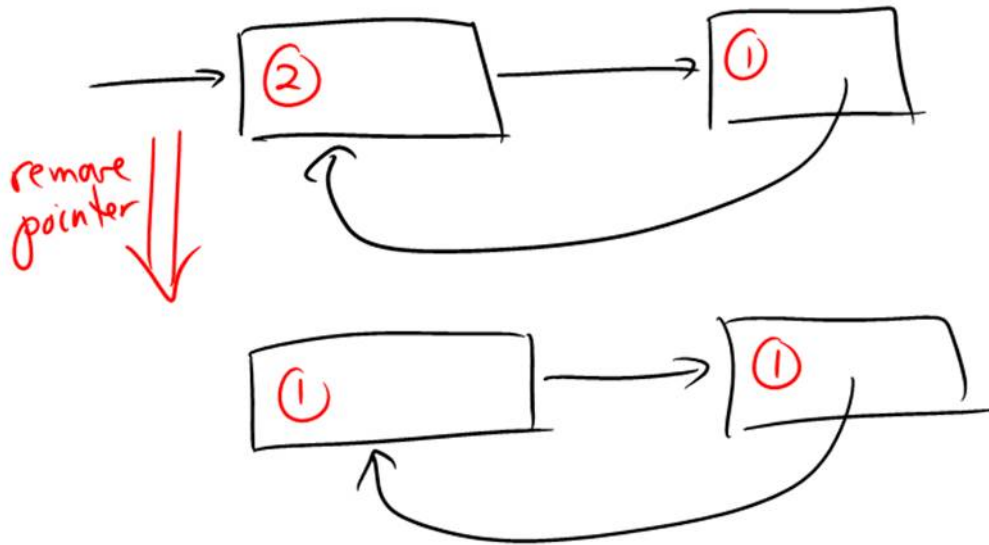
- Key concept in automatic memory management.
- data in a program accessible only through variables on stack.
- Heap objects accessible only through pointers
- Traversal of cells from all pointers on stack can reach some nodes, not others. Those nodes are called reachable, or accessible. Others are called garbage.
- Garbage nodes can never affect computation, because they can never be seen by the program.
- *Automatic memory management attempts to make garbage cells available for allocation.*

Reference-counting

- Keep free memory areas on a list
- Track number of pointers to every object — every object has an additional field giving count of in-pointers.
- Adjust count each time a pointer is copied/assigned
 - $p = q$:
 - Increment `refcnt(*q)`
 - Decrement `refcnt(*p)`
 - if `refcnt(*p)=0` then return memory of p to free list *and* decrement `refcnt` of all objects that $*p$ points to

Reference-counting (cont.)

- **Advantage:** Recover memory of an object as soon as the object becomes non-reachable
- **Big disadvantage:** Cannot handle cycles in the heap:



Garbage collection

- Don't keep track of reachability continually — instead, wait until memory runs out, then run a program to find all the non-reachable objects and recover them - a *garbage collector*.
- Two basic methods (with many variations and combinations):
 - Mark-and-sweep
 - Stop-and-copy
- Advantage: Handles cyclic structures easily
- Disadvantage: Creates a pause in the calculation while g.c. algorithm runs

Mark-and-sweep

- Reserve one bit in each object header, called the *reachable bit*
- Start with reachable bit zero in every header
- Traverse reachable data (depth-first search), setting reachable bit
- Sweep over entire heap. For each object, if reachable bit is 1, reset it; if it is zero, place that memory chunk on free list.
- Observations:
 - Reachable data is not moved
 - Reachable data remains spread across memory
 - Cost is linear in total size of heap

Stop-and-copy

- Only half of memory available for the heap is used at any time; that half is called *half-in-use*, the other half *reserved*.
- Half-in-use is divided into used area and free area.
- Allocate memory from top of used area (bottom of free area). When free area is exhausted, do g.c.
- Garbage collection method:
 - Traverse *reachable* objects, moving each object encountered to reserved area, allocating sequentially from bottom.
 - Complicated part is adjusting pointers.
 - Reserved area now becomes half-in-use.
- Advantage: Cost proportion to amount of reachable data.