

Automatic memory management

Amer Diwan

What is automatic memory management

- Frees up memory automatically when that memory needs to be freed
- May even do allocation “automatically”
 - e.g., in lisp dialects, “cons” is about the only way of allocating memory

We will focus on automatically freeing memory

Why free memory automatically?

- Avoids bugs
 - Memory leaks
 - Dangling pointers
 - ...
- Improves code quality
 - Don't need to worry about incorporating memory management issues in interfaces

How to free memory automatically

- Use garbage collection
 - In today's PC world, perhaps we should call it "automatic storage reclamation"

How do we find out when an object needs to be freed?

Bugs avoided with garbage collection

```
x = new int  
x = new int
```

The object allocated at the first statement is never freed

```
x = new int;  
delete x;  
*x = 10;
```

*x is accessing freed memory. If you are really unlucky, some other unrelated object has been already given that space

GC leads to “better” code

- Add a student/score pair to the linked list all_students
- LL = class { char *student; int score; LL *next; }
- void note_score(char *student, int score) {
 all_students = new LL(student, score, all_students);
}
- Is there any problem with this code?

Example continued

- ```
void note_score(char *student, int score) {
 all_students = new LL(student, score, all_students);
}
```
- ```
void client() {  
    char student_name[100]; int score;  
    for (i = 1; i <= num_students; ++i)  
        scanf("%99s %d\n", student_name, &score);  
    note_score(student_name, score);  
}
```

What's the problem here?

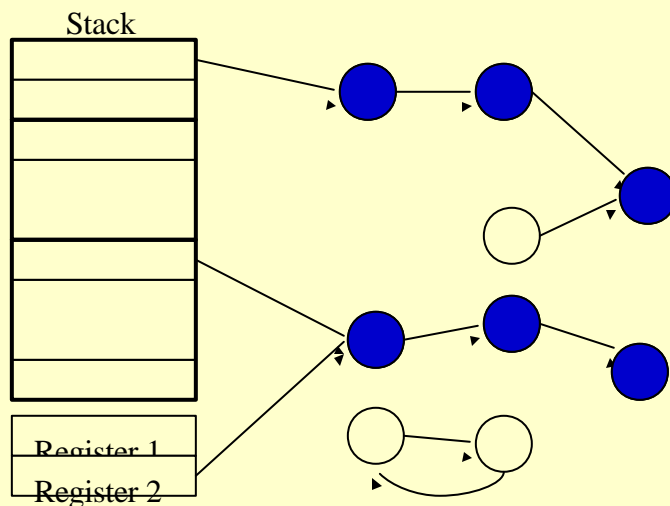
Observation from example

- For code written without GC, the memory management protocol must become a part of the interfaces
 - Should note_store copy its arguments into new memory?
 - Who is responsible for deleting this data?
 - Who is the owner of this data?
- Of course, for a program to determine when an object is freed is a non-trivial problem too!

How does garbage collection work?

- **Observation:** all memory is accessed by following pointers starting from some variable
- All objects **reachable** from the variables (registers, local and global variables) in a running program are not “garbage”
- **Unreachable** objects may be freed
- Different gc algorithms will find different supersets of reachable objects
- **Not guaranteed to deallocate all “dead” objects...**

How GC works



When GC doesn't work

- ```
main() {
 p = new T;
 Code that doesn't use p
}
```
- Simple GC will not free the object pointed to by 'p'
  - Could make GC smarter to only consider “live” variables
  - But no matter what you do, GC is (and had better be) conservative

## Conservatism of GC

- It will never free an object (barring bugs in GC...) that a program needs
- It may retain objects that the program does not need

Not too different in this way from what the programmers try to do

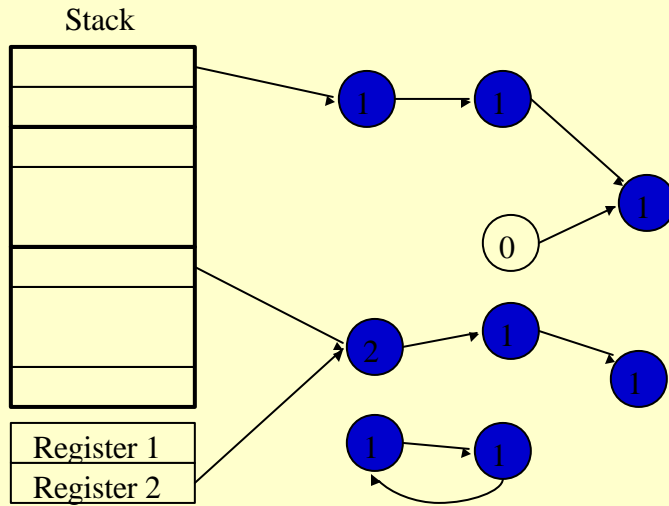
## Basic GC algorithms

- Reference counting
- Mark and sweep
- Copying

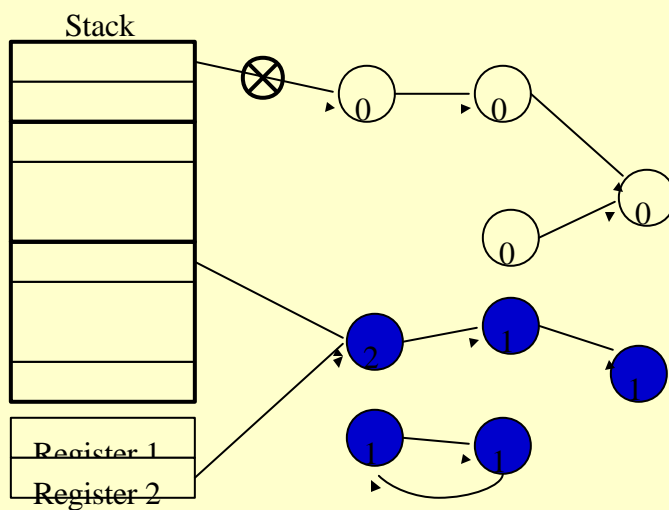
## Reference counting

- At every pointer operation, update counts
- Reclaim an objects when its count becomes 0
- When an object is reclaimed decrement counts of its referents
- Garbage collection is interleaved with program execution

## Example of reference counting



## Reference counting example (cont.)



## Strengths and weaknesses of reference counting

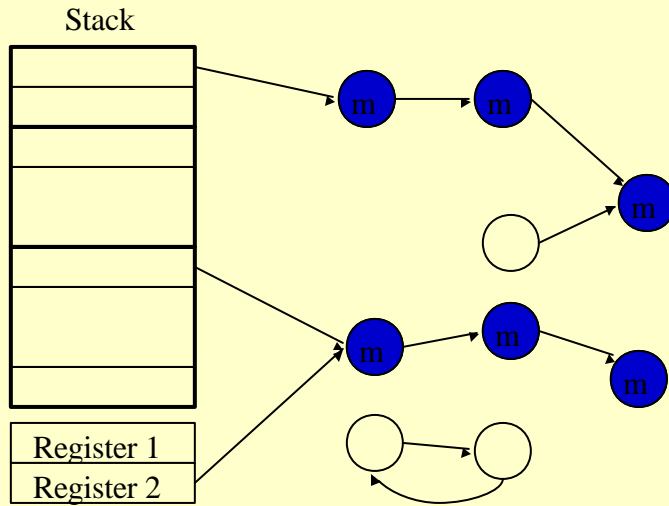
- **Strengths**
  - Incremental
  - Works in sync with the program: typically has good memory system behavior
- **Weaknesses**
  - Does not collect cycles
  - Allocation needs to search the free list
  - Work is proportional to the number of objects

Some of these weaknesses can be addressed by enhancing the basic algorithm or combining algorithms

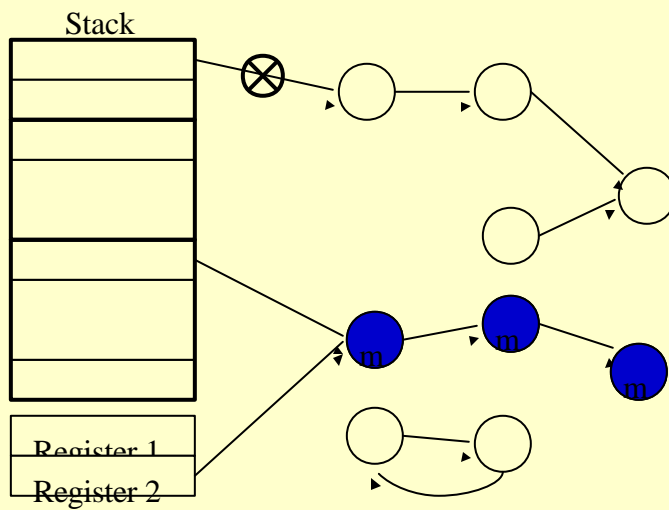
## Mark and sweep

- Mark objects reachable from the roots (locals, globals, and registers)
- Reclaim objects that are not marked
- Garbage collection is typically triggered when a “new” fails

## Example of mark and sweep



## Example of mark and sweep (cont.)



## Strengths and weaknesses of mark and sweep

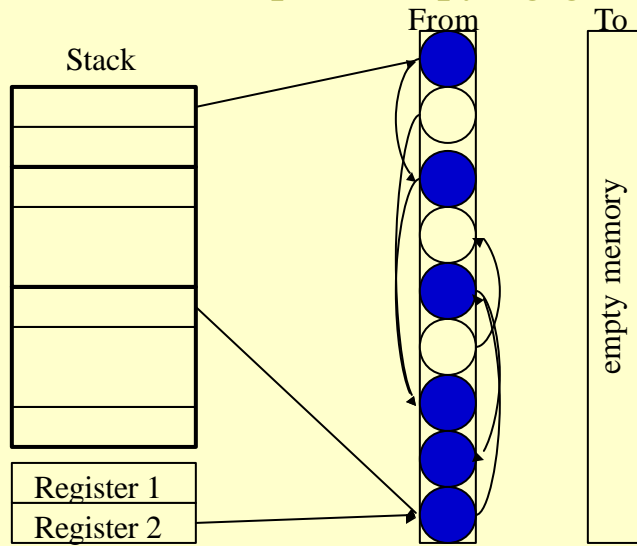
- **Strengths**
  - Collects cycles
- **Weaknesses**
  - Stop and collect
  - Does work proportional to the number of objects
  - Allocation needs to search the free lists
  - May have bad cache performance

Some of these weaknesses can be addressed by enhancing the basic algorithm or combining algorithms

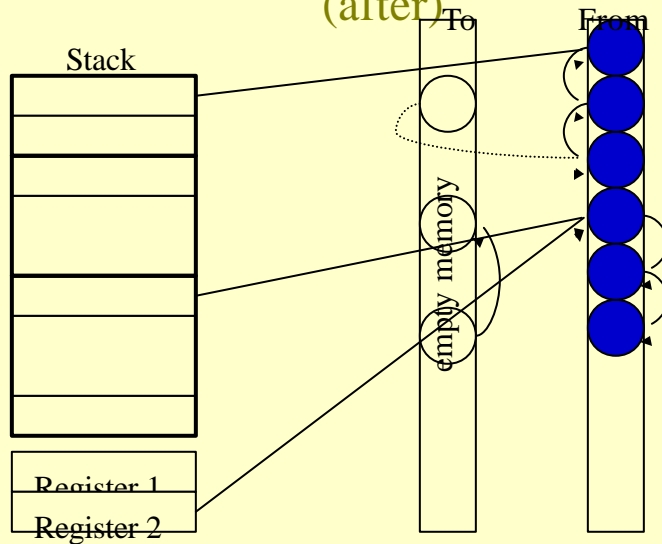
## Copying GC

- Divide memory into **from** and **to** spaces
- Copy objects reachable from program variables into the **to** space from the **from** space
- All objects not copied may be reclaimed
- Swap the roles of the **from** and **to** spaces

### Example of copying gc



### Example of copying gc (after)



## Strengths and weaknesses of copying collection

- **Strengths**
  - Compacts memory: allocation is fast
  - Work is proportional to the number of **live** objects
- **Weaknesses**
  - Stop and collect
  - May have bad cache performance

## Variants on the algorithms

- No one really implements the simple algorithms anymore
- Variations:
  - Incremental
  - Parallel
  - Generational (focus effort on recently allocated objects)
  - “Lazy” or deferred

## Summary of algorithms

- State-of-the art garbage collectors are very sophisticated
  - No one implements the basic algorithms anymore
  - There are techniques for addressing the weaknesses of each of the three basic techniques
  - The right algorithm may depend on the application and performance characteristics of the underlying system

## Language and garbage collector interactions

- Safe languages (e.g., Modula-3, Java, SML)
  - Must be able to tell which memory locations contain pointers
  - Can do any kind of garbage collection (particularly “accurate” collection)
- Unsafe languages (e.g., C++)
  - Cannot reliably tell which memory locations contain pointers
  - Can only do “conservative” collection

## Difficulty with unsafe languages

```
if (cond)
 i := 123;
else
 i := (long) malloc(sizeof(int))
```

Is *i* a pointer or an integer?

- If *i* is a pointer, then, gc must follow *i* when detecting reachable objects
- If *i* is not a pointer, then gc must not follow *i*

## Requirements for accurate collection

The compiler must generate enough information such that:

- the collector can identify all the pointers
- the collector can find what object each pointer points to

Specific garbage collectors, such as **generational** or **incremental**, may have other requirements too.

## Finding all the pointers

The stack, registers, and heap variables have different properties

- local variables, globals, and registers are similar
  - there are a finite number of each in any program
  - they are very frequently accessed
- heap locations are different
  - there are an infinite number of heap locations in a program
- Different strategies are applicable to different categories of objects

## Finding pointers in heap objects

Approach 1: Every heap object has a header that says where the pointers are located in the object (**object descriptor**)



Compiler support: construct and initialize the headers

Approach 2: Every location has a bit that says whether or not it is a pointer (**tagged pointers**)



Compiler support: generate the bits and manipulate them as needed

## Advantages and disadvantages of the two approaches

| Object descriptor         | Tagged pointers                       |
|---------------------------|---------------------------------------|
| More wasted space         | Less wasted space                     |
| Full sized integers       | 31 (or 63) bit integers               |
| No overhead on arithmetic | Tag manipulation needed on arithmetic |

## Finding pointers in the stack

- Can use same techniques as for heap but they are very wasteful: can do much better!
- Idea: compiler generates a table that identifies pointers in each stack frame
  - Problem: some stack locations may contain pointers at some point and non-pointers at another

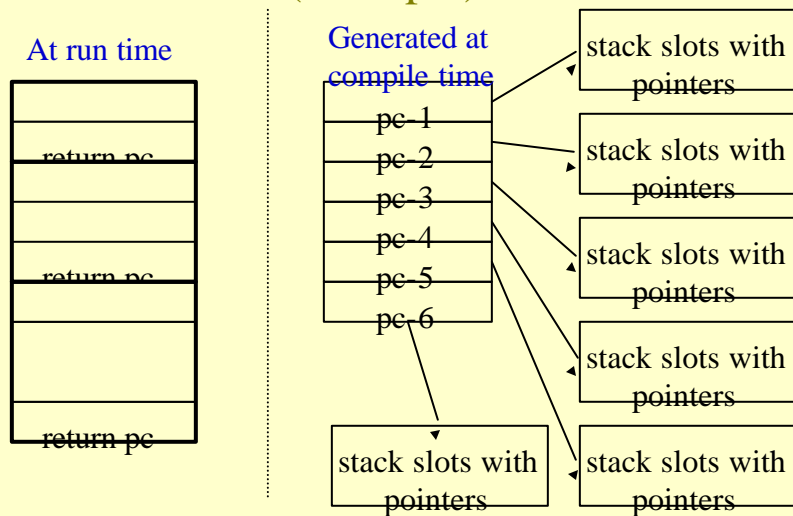
```
long i;
<use i>
long *p;
<use p>
```

What if i and p are assigned to the same location?

## Finding pointers in the stack (cont.)

- Generate a table for each program point where garbage collection can be triggered
  - allocation points
  - calls
  - for multithreaded systems also need thread switch points and/or backward branches

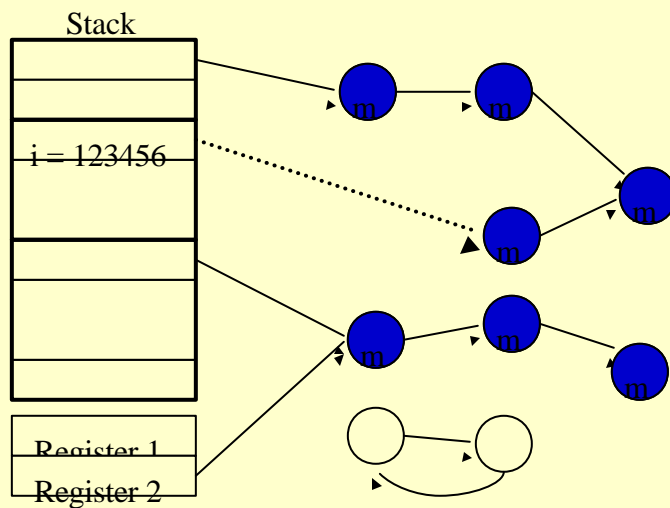
## Finding pointers in the stack (example)



## Conservative collection

- Works even when accurate collection is not possible
- Idea:
  - Assume that everything that looks like a pointer is a pointer

## Example of conservative collection



## Properties of conservative collection

- Advantages
  - Works for any language
  - Requires minimal compiler/language support
- Disadvantages
  - No compaction
  - Worse than accurate gc about reclaiming storage

## Summary

- Many different algorithms for garbage collection
  - The language type system affects what kinds of garbage collection are possible
  - An unfortunate decision in a language can make certain kinds of gc difficult or impossible

## Next topic: Accurate gc in Java

- Reading: Agesen, Detlefs, and Moss, *Garbage collection and local variable type-precision and liveness in Java virtual machines*