Dynamic Memory Allocation: Optional Extra Info

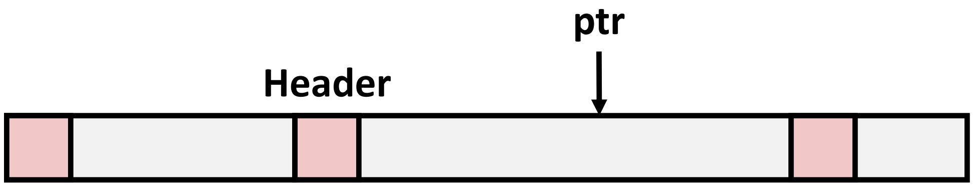
None of this will be on the test,
but we thought you might be curious.

Table of Contents

- **Garbage collection – the basics**
  - The mark-and-sweep algorithm
  - What to do if you don't know which memory words are pointers
- **How to read extra-gnarly C declarations**
  - like, "functions returning pointers to arrays of pointers to functions returning ints"

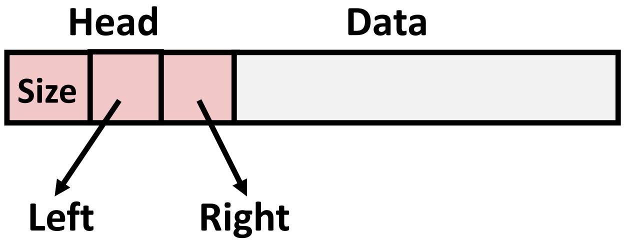
Conservative Mark & Sweep in C

- A "conservative garbage collector" for C programs
  - `is_ptr()` determines if a word is a pointer by checking if it points to an allocated block of memory
  - But, in C pointers can point to the middle of a block



Assumes ptr in middle can be used to reach anywhere in the block, but no other block

- To mark header, need to find the beginning of the block
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - Balanced-tree pointers can be stored in header (use two additional words)



Left: smaller addresses
Right: larger addresses

Assumptions For a Simple Implementation

- **Application**
  - **`new(n)`:** returns pointer to new block with all locations cleared
  - **`read(b,i)`:** read location **`i`** of block **`b`** into register
  - **`write(b,i,v)`:** write **`v`** into location **`i`** of block **`b`**

- **Each block will have a header word**
  - addressed as **`b[-1]`**, for a block **`b`**
  - Used for different purposes in different collectors

- **Instructions used by the Garbage Collector**
  - **`is_ptr(p)`:** determines whether **`p`** is a pointer
  - **`length(b)`:** returns the length of block **`b`**, not including the header
  - **`get_roots()`:** returns all the roots

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;
   if (markBitSet(p)) return;
   setMarkBit(p);
   for (i=0; i < length(p); i++)
     mark(p[i]);
   return;
}
```

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;
   setMarkBit(p);
   for (i=0; i < length(p); i++)
     mark(p[i]);
   return;
}
```

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;        // if not pointer -> do nothing
   if (markBitSet(p)) return;     // if already marked -> do nothing
   setMarkBit(p);
   for (i=0; i < length(p); i++)
     mark(p[i]);
   return;
}
```

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;        // if not pointer -> do nothing
   if (markBitSet(p)) return;     // if already marked -> do nothing
   setMarkBit(p);                 // set the mark bit
   for (i=0; i < length(p); i++)
     mark(p[i]);
   return;
}
```

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)   // for each word in p’s block
     mark(p[i]);
   return;
}
```

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;             // if not pointer -> do nothing
   if (markBitSet(p)) return;          // if already marked -> do nothing
   setMarkBit(p);                      // set the mark bit
   for (i=0; i < length(p); i++)       // for each word in p’s block
     mark(p[i]);                       //  make recursive call
   return;
}
```

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;        // if not pointer -> do nothing
   if (markBitSet(p)) return;     // if already marked -> do nothing
   setMarkBit(p);                 // set the mark bit
   for (i=0; i < length(p); i++)  // for each word in p’s block
     mark(p[i]);                  //  make recursive call
   return;
}
```

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {              // for entire heap
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p+1);
}
```

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;        // if not pointer -> do nothing
   if (markBitSet(p)) return;     // if already marked -> do nothing
   setMarkBit(p);                 // set the mark bit
   for (i=0; i < length(p); i++)  // for each word in p’s block
     mark(p[i]);                  //  make recursive call
   return;
}
```

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {              // for entire heap
      if markBitSet(p)            // did we reach this block?
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p+1);
}
```

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;           // if not pointer -> do nothing
   if (markBitSet(p)) return;        // if already marked -> do nothing
   setMarkBit(p);                    // set the mark bit
   for (i=0; i < length(p); i++)     // for each word in p’s block
     mark(p[i]);                     //  make recursive call
   return;
}
```

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                 // for entire heap
      if markBitSet(p)               // did we reach this block?
         clearMarkBit();             //  yes -> so just clear mark bit
      else if (allocateBitSet(p))
         free(p);
      p += length(p+1);
}
```

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;        // if not pointer -> do nothing
   if (markBitSet(p)) return;     // if already marked -> do nothing
   setMarkBit(p);                 // set the mark bit
   for (i=0; i < length(p); i++)  // for each word in p's block
     mark(p[i]);                  //  make recursive call
   return;
}
```

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {              // for entire heap
      if markBitSet(p)            // did we reach this block?
         clearMarkBit();          //  yes -> so just clear mark bit
      else if (allocateBitSet(p)) // never reached: is it allocated?
         free(p);
      p += length(p+1);
}
```

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p’s block
     mark(p[i]);                    //  make recursive call
   return;
}
```

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // for entire heap
      if markBitSet(p)              // did we reach this block?
         clearMarkBit();            //  yes -> so just clear mark bit
      else if (allocateBitSet(p))   // never reached: is it allocated?
         free(p);                   //  yes -> its garbage, free it
      p += length(p+1);
}
```

Mark and Sweep Pseudocode

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;        // if not pointer -> do nothing
   if (markBitSet(p)) return;     // if already marked -> do nothing
   setMarkBit(p);                 // set the mark bit
   for (i=0; i < length(p); i++)  // for each word in p’s block
     mark(p[i]);                  //  make recursive call
   return;
}
```

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {              // for entire heap
      if markBitSet(p)            // did we reach this block?
         clearMarkBit();          //  yes -> so just clear mark bit
      else if (allocateBitSet(p)) // never reached: is it allocated?
         free(p);                 //  yes -> its garbage, free it
      p += length(p+1);           // goto next block
}
```

C Pointer Declarations: Test Yourself!

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

Source: K&R Sec 5.12

C Pointer Declarations: Test Yourself!

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int

Source: K&R Sec 5.12

Parsing: `int (*(*f())[13])()`

`int (*(*f())[13])()`	`f`
`int (*(*f())[13])()`	`f is a function`
`int (*(*f())[13])()`	`f is a function that returns a ptr`
`int (*(*f())[13])()`	`f is a function that returns a ptr to an array of 13`
`int (*(*f())[13])()`	`f is a function that returns a ptr to an array of 13 ptrs`
`int (*(*f())[13])()`	`f is a function that returns a ptr to an array of 13 ptrs to functions returning an int`