11-0: Memory

- Three places in memory that a program can store variables
  - Call stack
  - Heap
  - Code segment

11-1: Memory

```
<table>
<thead>
<tr>
<th>Executable Code</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Storage</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

11-2: Memory

- Three places in memory that a program can store variables
  - Call stack
    - Local Variables
  - Heap
    - Dynamically allocated variables
    - (Most of the variables in Java)
  - Code segment
    - Static variables

11-3: Static Storage

- If a variable is declared static, there is only one instance of the variable
- Variable is typically stored in the code segment, not the stack or the heap
  - Why?

11-4: Static Storage

- If a variable is declared static, there is only one instance of the variable
• Variable is typically stored in the code segment, not the stack or the heap
  • Stack storage is too transient
  • Using the code segment guarantees a single instance of the variable

11-5: Static Storage

class StaticVars {
  int x;
  static int y;
}

void main() {
  StaticVars SV1 = new StaticVars();
  StaticVars SV2 = new StaticVars();
  SV1.x = 1;
  SV1.y = 2;
  SV2.x = 3;
  SV2.y = 4;
  print(SV1.x);
  print(SV1.y);
  print(SV2.x);
  print(SV2.y);
}

11-6: Static Storage

class StaticVars {
  int x;
  static int y;
}

void main() {
  StaticVars SV1 = new StaticVars();
  StaticVars SV2 = new StaticVars();
  SV1.x = 1;
  SV1.y = 2;
  SV2.x = 3;
  SV2.y = 4;
  print(SV1.x);
  print(SV1.y);
  print(SV2.x);
  print(SV2.y);
}

Output: 1 4 3 4

11-7: simpleJava Static Storage

• What do we need to do to implement static storage in simpleJava?

11-8: simpleJava Static Storage

• What do we need to do to implement static storage in simpleJava?
  • Looking at each portion of the compiler in turn:
    • Lexical Analysis – what needs to be done?

11-9: simpleJava Static Storage

• Lexical Analysis
  • Add a new keyword “static” to the language
    • – Add “static” token

11-10: simpleJava Static Storage

• Parsing & Building AST

11-11: simpleJava Static Storage
• Parsing & Building AST
  • Add a “static” tag to the AST for variable declarations (for both statements, and class instance variables)

11-12: **simpleJava Static Storage**
  • Semantic Analysis

11-13: **simpleJava Static Storage**
  • Semantic Analysis
    • No changes are necessary (apart from changes needed to implement building Abstract Assembly Tree)

11-14: **simpleJava Static Storage**
  • Abstract Assembly Tree Generation

11-15: **simpleJava Static Storage**
  • Abstract Assembly Tree Generation
    • Add a new field to variable entries – “static” bit
    • Generate code for static variables
      • Need to access variables in the code segment
      • Need to be able to access a “direct address”

11-16: **simpleJava Static Storage**
  • Need to access a “direct address”
    • Add an “AddressExp” node to our AAT
    • Single child – assembly language label
    • Represents the memory location at that address

11-17: **simpleJava Static Storage**

```java
static int x;
```

```
<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressExp</td>
</tr>
<tr>
<td>Label(&quot;x001&quot;)</td>
</tr>
</tbody>
</table>
```

11-18: **simpleJava Static Storage**

```java
class C1 {
    static int y;
    int x;
}
class C2 {
    int a;
    C1 class1;
}
```
AAT for \texttt{class2.class1.x}?

11-19: \texttt{simpleJava Static Storage}

\begin{center}
\begin{tikzpicture}
  \node {Memory}
    child {node {Operator(-)}
      child {node {Memory}
        child {node {Constant(x\_offset)}}
      }
    }
    child {node {Operator(-)}
      child {node {Memory}
        child {node {Constant(class1\_offset)}}
      }
    }
    child {node {Register(FP)}}
    child {node {Constant(class2\_offset)}};
\end{tikzpicture}
\end{center}

11-20: \texttt{simpleJava Static Storage}

class C1 {
  static int y;
  int x;
}
class C2 {
  int a;
  C1 class1;
}
...
C2 class2;

AAT for \texttt{class2.class1.y}?

11-21: \texttt{simpleJava Static Storage}

\begin{center}
\begin{tikzpicture}
  \node {Memory}
    child {node {AddressExp}
      child {node {Label("y001")}}
    };
\end{tikzpicture}
\end{center}

11-22: \texttt{simpleJava Static Storage}

- Code Generation

11-23: \texttt{simpleJava Static Storage}

- Code Generation
  - Add space to code segment to store static variables
  - Make sure the labels match!!
11-24: **Heap-based storage**

- There are 2 main memory-allocation dangers associated with heap-based storage
  - Dangling References
  - Memory leaks

11-25: **Dangling References**

```c
int main() {
    int *a;
    int *b;
    a = (int *) malloc(sizeof(int));
    (*a) = 4;
    b = a;
    free b;
    ...
}
```

- What happens if we change \((*a) \{(*a) = \ldots\}\)?

11-26: **Memory Leaks**

```c
int main() {
    int *intPtr;
    intPtr = (int *) malloc(sizeof(int));
    intPtr = NULL;
    ...
}
```

- Allocated memory that we can’t get to – *garbage*
- Eventually, use up heap memory

11-27: **Managing the Heap**

- Manage the heap to avoid memory leaks and dangling references
  - Give all decisions to the programmer
  - Automatic memory management

11-28: **Programmer Controlled**

- **Advantages**
  - Memory management system is less complicated
  - Lower run-time overhead for the memory manager
  - Can manage the memory needs for a specific program more efficiently than a general-purpose memory manager (at least in theory)

11-29: **Free List**

- List of all available blocks of memory
• When a request for a block of memory is made, it is removed from the free list
• Deallocated memory is returned to the free list

11-30: Free List

• Housekeeping
  • When a block is requested, allocated slightly more memory than requested.
  • Extra space is used to store header information (for now, just the size of the allocated block)
  • Return a pointer to just after the header information

<table>
<thead>
<tr>
<th>Size of allocated block</th>
<th>Pointer returned to program requesting memory</th>
</tr>
</thead>
</table>

11-31: Free List Example

class oneElem {
    int x;
}
class twoElem {
    int x;
    int y;
}

oneElem A = new oneElem();
oneElem B = new oneElem();
twoElem C = new twoElem();
twoElem D = new twoElem();
11-33: Free List Example

Free list

Allocated Memory

Size of block

984

Next block

11-34: Free List Example

class oneElem {
    int x;
}
class twoElem {
    int x;
    int y;
}

oneElem A = new oneElem();
oneElem B = new oneElem();
twoElem C = new twoElem();
twoElem D = new twoElem();
delete A;
delete C;

11-35: Free List Example
11-36: **Free List Example**

class oneElem {
    int x;
}
class twoElem {
    int x;
    int y;
}

oneElem A = new oneElem();
oneElem B = new oneElem();
twoElem C = new twoElem();
twoElem D = new twoElem();
delete A;
delete C;
delete D;

11-37: **Free List Example**
11-38: Free List Example

11-39: First Fit

- When there are several blocks to choose from on the free list, which do we use to fulfill a memory request?
  - First Fit
  - Return the first block that is large enough

11-40: First Fit

```java
class smallClass {
    int x;
}
void main() {
    int i;
    smallClass A[] = new smallClass[3000];
    smallClass B[] = new smallClass[3000];
    for (i=0; i<3000; i++)
        B[i] = new smallClass();
    for (i=0; i<3000; i = i + 2)
```
delete B[i];
delete A;

/* Point A */
for (i=0; i<3000; i = i + 2)
    B[i] = new smallClass();
/* Point B */
}

11-41: First Fit

- At Point A:

11-42: First Fit

- At Point B:

11-43: First Fit

- Plenty of space on the heap
- Divided into small blocks – can’t service a request for a large block of memory
- Memory fragmentation
11-44: Best Fit

- When there are several blocks to choose from on the free list, which do we use to fulfill a memory request?
  - First Fit
    - Return the first block that is large enough
  - Best Fit
    - Return the smallest block that is large enough

11-45: Best Fit

- At Point B (using Best Fit):

11-46: Best Fit

- Best Fit will usually lead to less memory fragmentation than first fit
  - Don’t “waste” large memory blocks on small requests
  - Large blocks should then be available when needed
  - Will Best Fit always lead to less memory fragmentation?

11-47: Best Fit vs First Fit

```c
for (i=0; i<100; i++)
    A[i] = malloc(4);
for (i=0; i<100; i++)
    B[i] = malloc(3);
for (i=0; i<100; i+=2)
    free(A[i]);
for (i=0; i<100; i+=2)
    free(B[i]);
for (i=0; i<100; i++)
    C[i] = malloc(2);
```
Segregated Free List

- Fragmentation problems caused by differing block sizes
- Remove the problem by having all blocks be the same size (like lisp)
  - Can’t make all blocks the same size
  - Can use a limited # of standard block sizes

Segregated Free List

- Memory can only be allocated in set block sizes
  - Typically powers of 2 – 2 words, 4 words, 8 words, etc
- Separate free list maintained for each block size
- When a request is made, the smallest block that can service the request is returned.

11-50: Segregated Free List

Free List Array

- Initially, all heap memory is placed in the largest block list
- If a request is made for a block of memory of size $2^k$, and list $k$ is empty:
  - Split a block from list $k + 1$ into two blocks of size $2^k$
  - Add these two blocks to list $k$
  - List $k$ is no longer empty – can service the request

11-52: Segregated Free List

Free List Array
A request for a block of size 16 is made

11-53: Segregated Free List

Free List Array

2 word blocks
4 word blocks
8 word blocks
16 word blocks
32 word blocks

11-54: Segregated Free List

Free List Array

2 word blocks
4 word blocks
8 word blocks
16 word blocks
32 word blocks

A request for a block of size 2 is made

11-55: Segregated Free List

Free List Array

2 word blocks
4 word blocks
8 word blocks
16 word blocks
32 word blocks

11-56: Garbage Collection

• Giving the user control of deallocation has problems:
  • Writing programs that properly deallocate memory is hard
  • Often, there are many pointers to the same block of memory (much like your current project!)
  • It can be difficult to determine when a block of memory should be freed
  • We don’t want to be too aggressive in freeing memory (why not?)

11-57: Garbage Collection

• Giving the user control of deallocation has problems:
  • Writing programs that properly deallocate memory is hard
  • Often, there are many pointers to the same block of memory (much like your current project!)
  • It can be difficult to determine when a block of memory should be freed
• We don’t want to be too aggressive in freeing memory (why not?)
• Solution – don’t let programmer control deallocation!

11-58: Garbage Collection

• Don’t allow programmer to dealloc any memory
• Garbage will collect
• Periodically collect the accumulated garbage, and return it to the free list

11-59: Mark & Sweep

• When Garbage Collection routine is invoked:
  • Mark all heap memory that is reachable by the program
  • Need to add a “mark” bit to each block of memory – can use the header
  • Sweep through the entire block of memory, moving unmarked blocks to the free list

11-60: Mark Phase

for each pointer P on the stack
mark(P)

mark(P) {
    if (P is not null and [mark bit of Mem[P] is not set])
        set mark bit of Mem[P]
        for each pointer Q in the block Mem[P]
            mark(Q)
}

11-61: Mark & Sweep

class Class1 {
    int x;
    int y;
}
class Class2 {
    Class1 C1;
}
class Class3 {
    Class1 C1;
    Class2 C2;
}
Class3 C3 = new Class3();
C3.C1 = new Class1();
C3.C2 = new Class2();
C3.C2.C1 = new Class1();

11-62: Mark & Sweep
11-63: Mark & Sweep

class Class1 {
    int x;
    int y;
}
class Class2 {
    Class1 C1
    int x;
}
class Class3 {
    Class1 C1;
    Class2 C2;
}
Class3 C3 = new Class3();
C3.C1 = new Class1();
C3.C2 = new Class2();
C3.C2.C1 = new Class1();
C3.C1 = new Class1();

11-64: Mark & Sweep

11-65: Mark & Sweep

11-66: Mark & Sweep
11-67: Whither Pointers?

- In order for the Mark phase to work correctly, we need to know which memory locations are pointers, and which are not
  
  - Tag pointers
  
  - Assume that any value that might be a pointer is a pointer (Conservative garbage collection)
  
  - Create tables that store memory locations of all pointers

11-68: Tagging Pointers

- If we wish to tag pointers themselves, we have two options:
  
  - Tag the pointer itself (high order bits)
  
  - Store tag in preceding word

11-69: Tagging Pointers

- Tag the pointer itself (high order bits)
  
  - If the high order bits are 11, then the memory location represents a pointer
  
  - If the high order bits are 00, 01, or 10, then the memory location represents an integer or boolean value

- Using 32-bit words, only 30 bits will be available for pointer values
  
  - Need to strip the tag before pointers can be dereferenced

- Using 32-bit words, slightly more than 31 bits are available for integer values (very large negative values prohibited)

11-70: Tagging Pointers

- Store tag in preceding word
  
  - Set aside a specific bit pattern as a sentinel value (something like -MAXINT)
  
  - Every pointer requires 2 words of storage – word for the sentinel, and a word for the pointer itself

11-71: Tagging Pointers
class Class1 {
    int x;
    Class2 C2;
    Class3 C3;
    boolean y;
}

11-72: Conservative GC

- Assume that every memory location that *could* be a pointer is a pointer
- The integer \( y \) will be considered a pointer if:
  - Heap addresses are in the range LOW .. HIGH, and LOW \( \neq \) \( y \) \( \neq \) HIGH
  - The memory location \( y \) is the beginning of an allocated block

11-73: Conservative GC

- Every memory block on the heap that is pointed to by something on the stack will be marked
  - No dangling references
- Some memory blocks on the heap that are *not* pointed to by something on the stack *may* be marked
  - May have some uncollected garbage
- Since no extra information (tagged pointers, etc.) is needed, Conservative Garbage Collectors can be run on languages not designed with garbage collection in mind (i.e., C)

11-74: Pointer Tables

- Create a table for each function & class, which keeps track of where the pointers are in that function or class
  - This can be done at compile time
- Each function & class will need a “kind” field, to store what kind of function or class it is (classes will need a “kind” field anyway, if we want instanceof to work)

11-75: Pointer Tables

ClassA C1;
int b;
ClassB C2;
C1 = new ClassA();
C2 = new ClassB();
C2.C1 = new ClassA();
C2.C2 = new ClassA();
/
/* Body of main */
)

11-76: Pointer Tables

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code Segment</th>
<th># of pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td></td>
</tr>
<tr>
<td>ClassA</td>
<td>16</td>
</tr>
<tr>
<td>ClassB</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pointer Table</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>24</td>
</tr>
<tr>
<td>C1</td>
<td>16</td>
</tr>
<tr>
<td>C2</td>
<td>8</td>
</tr>
<tr>
<td>z</td>
<td>12</td>
</tr>
</tbody>
</table>

11-77: Reference Counts

- Each block of allocated memory contains a count of how many pointers point to it
- Each time a pointer appears on the LHS of an assignment:
  - Count of what the pointer used to point to is decremented
  - Count of what the pointer now points to is incremented
- When a count hits zero, add block back to free list

11-78: Reference Count Problems

From the Jargon File: (aka Hacker's Dictionary)

One day a student came to Moon and said: “I understand how to make a better garbage collector. We must keep a reference count of the pointers to each cons (block of memory).”

Moon patiently told the student the following story:

“One day a student came to Moon and said: ‘I understand how to make a better garbage collector...’ ”