Michael Brudno[*]
CS 61B Lecture Notes: Memory Allocation and Garbage Collection
7/31/00

1. Classification of storage

In most modern languages like C, C++ and Java there are three different types of storage: s**tatic storage, local storage,** and **dynamic storage.**

Static storage refers to those variables whose lifetime[1] encompasses the execution of the entire program. In Java, those variables are the static variables of each class. You can ask for them from any method of any class (if you have set the proper modifiers), and they are available from the time you execute the first instruction and until you finish executing the last. The size of static storage required by any program is known when the program is starting.

As the name might imply, local storage is the storage used within each method or block. This storage is only available in the method or block which declared it. The size of local storage which every procedure or block may need should be known at compile time.

Dynamic Storage is probably the most complicated (and therefore the most interesting) of the three. The lifetime of some piece of dynamic storage is started by allocation (using the **new** command in Java, **malloc** in C) , and it lasts until it is de-allocated, either through direct intervention of the programmer (in C, C++, and Pascal) or automatic garbage collection (in Java and Scheme). Both of these will be discussed later.

2. Stack: the implementation of local storage

In order to help us manage all of the local variables we have the *run-time* stack. This stack consists of a series of frames, each of which contains all of the storage which a certain method needs. When we make a procedure call **foo()** we add another frame to the stack to keep all of **foo**'s local variables. As soon as **foo** exits, the frame which it used is destroyed. Along with it die all of **foo**'s variables. Earlier I mentioned that the size of each procedure's frame should be known at compile time. The reasons for this is that the program should be capable of easily finding it's variables within the frame, and if the location of variables could change from call to call it would be very hard to locate them. However is it not true that Java allows the creation of arrays as local variables? Can't I write **int[] arr = new int[size];** in the middle of any method? Well, yes you can, but if you remember what we said at the very beginning of the class you will realize why: Arrays (and Objects) are reference types: arr is going to be just a pointer, the size of which is known at compile time. The actual array of **size** elements is going to be allocated in the place where dynamic storage goes.

3. Implementation of storage classes

Different architectures use different places of memory to keep the three types of storage mentioned above. One common method (demonstrated in Fig. 1) is to use a Stack which goes down from the larger addresses to the smaller, and a heap which goes from the smaller to the larger. All dynamic storage is allocated "on the heap." The static storage and the executable code go below the heap, since the size of both is known at compile time. The use of the name heap for the place where dynamic storage is allocated is confusing. This heap has nothing in common with the data structure used to implement priority queues.

---

[*] Copyright © Michael Brudno, 1998

[1] Lifetime is different from scoping. The lifetime of a variable is how long it stays in memory, regardless of who is allowed to refer to it. Scope refers to where in your code you can actually refer to the variable. Here we will only be concerned with the lifetime of the variables.

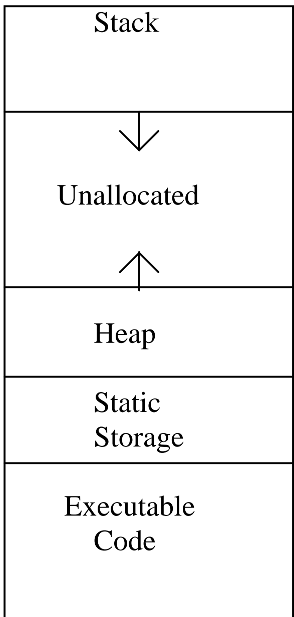

Fig. 1: An example of run-time storage layout.

4. Dynamic storage management

4.1 Allocation

You may have heard during the course of this class that **new** is an expensive operation. Now you find out why. The following section deals with what your computer goes through every time you use **new**. This section will deal with what is known as the boundary tag method, while section 4.3 will present a possible improvement: the buddy system.

In the boundary tag method, for every block of storage there will be an administrative word of storage, containing the information about the **size** of the block, whether it is free or used, and whether the block immediately preceding this one is free. The need for the first two is obvious. The third will come in handy later. The unallocated blocks of memory will be joined using a circular, doubly-linked list. Whenever we need to allocate a new piece of storage we will go through our list and find a block which is large enough to hold the required storage. We will split that block into two parts, declare the first one allocated and return its address to the procedure which asked for the memory. For the second one we will create the necessary word of storage and put it into the linked list of free objects. If there is not enough space for the new object we can ask the operating system to give us more. If the operating system has no more then we are out of luck, and should probably terminate the program with an out of memory error.

The following is a pseudo-code implementation of **malloc** (or **new**).

```
static Address FREE_LIST;
Address malloc (int size) {
   Address FREE0, result;
    if (FREE_LIST == null)
      askOSforStorage();
    FREE0 = FREE_LIST;
    while (true) {
      FREE_LIST = FREE_LIST.next();
      if (FREE_LIST = FREE0)
        askOSforStorage();
      if (FREE_LIST.size >= size)
        break;
    }
    if (FREE_LIST.size == size){
       result = FREE_LIST;
       change the isFree boolean,
       update the previous next block's
prevIsFree
       remove FREE_LIST from the list of free
storage
       and return result.
    }
    else {
       split up the block into two parts
       return the leftovers to the free list
       return the good part;
    }
}
```

The algorithm described above is known as *first-fit*: we return the very first block which is large enough to hold the size requested. At each new request to allocate memory we start at the spot where the previous allocation attempt left off, rather then at the beginning. If we were to do it the latter way, the memory early on in the linked list will be chopped up into many small, and therefore unusable, blocks. These blocks will increase the time it takes to find a good block, since we will first have to look at all of them. The rotating free list overcomes this problem.

Another possible strategy is *best-fit*, where you search for the closest match to the size you are trying to allocate, but this has been shown to be inferior: it takes longer to find the block (since we must look at all free blocks before we determine which one fits best), and it also has the tendency to chop up memory into small blocks.

4.2 Freeing

The size of the heap has limits: the calls to the Operating System for more memory can fail if not enough physical memory is available. So at some point you may want to free the memory allocated by your program earlier. In Java this de-allocation is done automatically. In C manually. In either case they use the same algorithms to actually free the block. The difference comes in who asks to free it the programmer with an explicit call to **free** or the program itself when it runs out of space. Freeing a used block of memory is pretty trivial: we reset the isFree flag to true and add it to the free list. However we can make one improvement. Since we are interested in keeping the blocks of unallocated memory as large as possible, we can coalesce the block we are freeing with the block immediately before or after it (if that block is already free). This is where it is useful to know whether the block right before you is free. We can find the next block quite trivially by adding the size to the address of the current block and getting a pointer to the next one, but to find the previous block takes a bit of work. Thus by having previous is free flag we will only have to search for it if it is free (and hence on the FREE_LIST).

4.3. The buddy system

By sacrificing a little bit of memory, we can improve the performance of the allocation and de-allocation algorithms. In stead of keeping blocks of all possible sizes around, we can keep only blocks of 2^n bytes, where n can range from k0 (where 2^k0 bytes is just large enough to hold the administrative word) to m (where 2^m is the total size of the memory available). The FREE_LIST is now an array[k0...m] of doubly linked lists, each of which includes all of the free blocks of size 2^n, where n is the index of the array. An additional rule instituted to help in the coalescing of free objects is that a block of memory of size 2^n can only start at an address which is divisible at 2^n. Thus a 16bit object can start at address 32 or 48, but not 40. If we have just de-allocated a 16 bit object at address 48 we can coalesce it with another 16bit object at address 32 (since the resulting block of size 32 will start at address 32), but not with a free block at address 64, since a block of 32 bits cannot start at address 48. It turns out really easy to calculate whether an adjacent block is your buddy (whether you can coalesce with it) if this scheme is used. To allocate memory using the buddy system we round the size we are asking for up to the next power of 2, and look in FREE_LIST whether there are blocks of that size available. If yes we return it, if not we get a larger block and divide it into smaller parts. to de-allocate you just return the memory to the free list of the appropriate size and see if its buddy is also free, whether you could coalesce with it. The down side of the buddy system is that when you ask for 33 bytes of memory it gives you a block of size 64, a small waste but waste never the less. With the modern computers having insane amounts of RAM there seems to be little reason not to use the buddy system.

5. Automatic memory de-allocation

Having gone through a whole semester of Java without using a single time a call to **free** or **delete** you may be now wondering what is that little demon sitting in your programs which does all of this de-allocation for you. This demon is actually doing you a very big favor. Perhaps the most common bug in C and C++ programs is forgetting to de-allocate memory. This leads to problems known as memory leeks. Another,

though less common (but more disastrous) problem is de-allocating memory which is actually still in use. Java eliminated this problem by using an automatic device known as a garbage collection. If you ever want to run it explicitly, you can use the **System.gc()** call. Otherwise it runs when the program does not have enough memory to allocate an object (though this is implementation specific).

There are several algorithms for garbage collection, all of them require certain assumptions. Perhaps the main one is that I have to be able to tell a pointer from some other type of value. In Java this is easy: the variables are neatly divided into Reference Types and Primitives, and casting between the two is not allowed. In C, however, you can cast between the two, and as a result garbage collection is almost impossible. The second assumption is that I should be able to find all of the roots of the dynamic data structures. A root refers to any variable (static or local) to which the program can possibly mention. this may require some help from the compiler to leave the relevant information around[2].

5.1 Reference counting garbage collection

This method of garbage collection keeps a counter for each object of how many pointers are pointing to it. As a result, any sort of pointer assignment becomes somewhat complicated: the simple Java expression

```
x = y;
```

generates the following code:

```
if (y != null)
    y.incrementCount()
if (x != null) {
   x.decrementCount()
   if (x.count == 0)
      freeStructure(x);
}
x=y;
```

[2] In Lisp, for instance, it is possible to have a single static variable which is a pointer to a hashtable of all possible variables.

where **freeStructure(x)** assigns null to all of the pointers contained in **x** and decrements counts appropriately.

This assignment procedure must be used not only for explicit assignments but also whenever a function exits: null must be assigned to all of the local variables. This is expensive. An even larger problem with reference counting is that it does not support circular structures like doubly linked lists. Even when no root object can access a doubly linked list, there are still pointers to each object, and as a result they will never be de-allocated.

Reference counting is used on the UNIX system, where the files that you see are actually pointers (hard links) to the actual file (inodes). The inode is not erased until all of the hard links pointing to it are gone[3]. When it comes to computer languages, however, the large overhead of all pointer assignments is way too much for a lot of the simple programs which will never use up all of the memory available to them. Because of this most of the automatic schemes used do periodic garbage collection: they are called in when there is not enough memory and free whatever is unreachable at that point.

5.2 Mark-and-sweep garbage collection

Mark and sweep garbage collection is actually a simple exercise in graph traversals. Starting at the roots, we trace all the pointers we can find and mark all objects as we see them. All reachable objects are exactly the ones we can still access, while all unreachable ones are garbage. When we are done, we make a sweep through memory and de-allocate any unmarked objects. Sweeping through the memory is trivial if we store the size of every object as the first element in it, as we did in the memory allocation/de-allocation algorithms. The time required by this method is proportional to the number of Objects in memory.

[3] Unix directory structures are doubly-linked, and because of this you have to delete the lowest-level directories before you can delete the parents: when you delete the leafs you break the double linkages one level up.

5.3 Copying garbage collection

Mark and sweep collection never moves objects: this has both advantages and disadvantages. The advantages is that this saves time: copying is some-what expensive. The disadvantage is that this leaves the memory fragmented. In copying garbage collection, instead of de-allocating the unmarked objects we will move the marked ones into a different portion of the memory (called the to_space). When I write these into memory, I will write them consecutively, eliminating any free blocks in the middle of the memory. Whatever is left is declared garbage when I am done.

The way copying works is quite simple, with one twist: After copying object A into the two space, I will copy all objects to which it contains pointers, in the process updating these pointers to point to the new location of the objects. The problem arises when we encounter an object pointing to A after we have already copied A. This is resolved by leaving a forwarding pointer where A used to be pointing to A's new location.

After we are done the to_space contains one large unallocated block, instead of several ones. Afterwards memory allocation is very cheap since you can just grab the next chunk of the free block. At first glance copying garbage collection is wasteful: half of the allocatable storage, from_space, is unused between garbage collections. While this is a disadvantage, we do not have to actually use half of the computer's memory for the from_space. Using virtual memory we can decrease the amount used for this purpose quite significantly. A further improvement is possible using generational garbage collection.

5.4 Generational garbage collection

A fact of computer life (a sad one, but a fact never the less) is that objects die young. If an object has lived through several stages of a program it is unlikely to die anytime soon, while we go around creating temporary strings left and right which we will not use after a couple operations on each. In order to take advantage of this fact, it is possible to garbage-collect just among the younger elements. In order to do this, we divide all our objects into several generations. When the space for the youngest generation fills up, we garbage collect it, but ignore almost all pointers going up to any generation older then it. Any object that survives the garbage collection is tenured - copied into the next oldest generation. Once that generation fills up it is garbage collected.

The reason that we can ignore most, but not all pointers into the older generations is that in some rare cases you will have an older object pointing to a younger one. In this case we have to traverse the older element in order to update the pointers to the younger ones. Such cases are rare, and it suffices to keep an array (called a "remembered list") of all such older objects which point to younger ones. The pointers from these objects to younger ones can be listed as roots when we do our traversal.

Generational garbage collection has proven to be very effective. In one Smalltalk system developed in Berkeley it accounted for only 3% of the total execution time.