Chapter 8

Memory Use

- In this chapter we will consider memory use during compilation and run time.
- During compilation, the principal memory-use problem is the management of the symbol table.
- But the compiler must also lay the groundwork for how data will be stored and accessed during program execution.

1. The Symbol Table

- The symbol table is consulted at almost every point in the compilation process.
- The main issue in symbol table design is organization
- We will consider the gross organization of the table and then questions of how the structure of the source program influences the structure of the symbol table.

1.1 Organization

- A symbol table can be organized as a simple array, a linked list, a binary search tree, or an array accessed by hashing.
  - The choice of structure is a trade-off between memory requirements and access speed.
  - Since the symbol table will typically be consulted thousands of times in the course of compiling a program, speed of access is generally given first priority.

- The simplest organization is the array.
  - The advantages of using an array are:
    - simplicity
    - and economy of storage.
  - The disadvantages are:
    - the limited size of the table
    - and the time it takes to search the table.

- A linked list offers only two advantages over an array.
  - First, it is expandable
  - Second, using self-organizing storage can speed search considerably.
    - Self-organizing storage is described in Sedgwick's texts
    - Basically, you move accessed members to the front of the list, since their access will probably come again soon.

- A binary search tree offers quicker access to data items.
  - It offers the minor subsidiary advantage that if the table is to be listed at the end of compilation, it can be displayed in alphabetical order by a simple in-order traversal.
  - Its main drawback is increased memory consumption (2 pointers), and the fact that deleting items tends to unbalance it and increase search time.

- Most compilers use hashed storage.
  - Usually hashing with chaining.
  - The array size is limited, but the linked-lists can be any length.
  - The search time is O(k), where k is the length of the list
  - k is also the number of items stored in the table divided by the size of the array, so you can set a bound on it.

1.2 Storing Identifiers

- Storing variable names can present special difficulties.
  - In FORTRAN, variable names are generally limited to eight characters, and it is no great problem to provide an eight character field in the symbol-table record.
  - BUT, many modern languages support very long variable names (31 is a frequent limit), and that can waste space in an array implementation if you have i, j, and k used.

- One compromise is to allow very long identifiers but recognize only the first six or eight characters.
- Or, if you must store the entire name,
  - Use dynamic memory
  - or have a big array for names, and have the symbol table point to the start of the name, with some marker at the end.
    - This represents a cheap and optimal way of storing them.

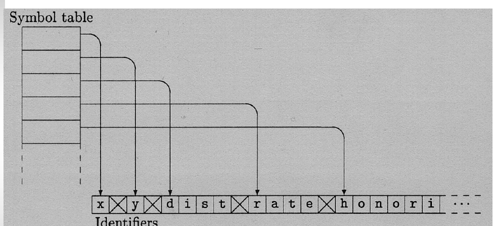

Figure 8.1

1.3 Record Formats

- Symbol-table records must include either the identifier name or a pointer to the name, and a code indicating the data type of the object to which the entry refers. Beyond that, the contents of the record may vary widely.
- It follows that it is probably not a good idea to use a uniform format for all symbol-table entries.

1.4 Special Problems

- In a language like FORTRAN, this is virtually all there is to symbol-table management.
  - Find a suitable organization,
  - Decide on record formats
  - Perhaps concoct a hashing function
  - -- and you are done
- But other languages present additional problems.

- Block-structured languages with static scope rules, like Algol, Pascal, PL/I, and C, have to model the scope rules in the symbol table.
- In addition, these languages permit records, and again the symbol table must be designed to facilitate the handling of these structures.

- Scope Rules:
  - There are two basic approaches to implementing scope rules in the symbol table.
    - You can have a separate table for each scope
    - You can have a global table in which each identifier is marked with the scope to which it belongs.
  - It is probably easier to have a separate table for each scope. Then you treat it like a stack.

- Problems with using a stack
  - If you have a multi-pass compiler
  - If you need to generate cross-reference tables
    - where a variable was defined
    - every place it was used
  - Such tables are typically generated at the end of compilation, but at that time all the stack entries have been deleted. So it may be necessary to save these symbol tables for later use when they are “popped” off the stack.

- Records:
  - A new record is in many ways like a new scope.
    - Identifier names in a record may be duplicates of other identifier names used outside the record, just as they may in scopes.
    -

```
var
    a,
    b: integer;
    c: record
        a,
        b: integer
       end;
```


- Therefore, you have the choice
  - creating a subtable for the record, just as we would for a new scope and hang it off the record’s node
  - add the record’s fields to the symbol-table with a flag/pointer to the record to whom they belong.
  -

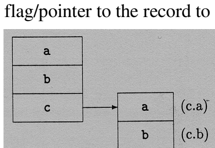

Figure 8.3(a)

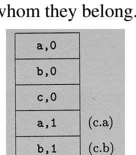

Figure 8.3(b)

2. Run-Time Memory Management

- We can classify the use of memory management during execution as static or dynamic.
- In static memory management, storage for all variables is allocated at compile time.
  - FORTRAN, COBOL
  - allows no recursion, but is easy to do.

- In dynamic memory management, storage is found for the variables at run time as it is needed.
- The main issues in dynamic memory management are
  - using a stack for variable storage
  - and supporting dynamic memory allocation for statements like new and delete

2.1 Static Memory Management

- This is the oldest and simplest scheme.
  - every variable is assigned an address when the program is translated.
  - Advantage: It is simple, and requires minimal overhead.
  - Disadvantage: No recursion.

- Addresses for variables may be specified as
  - absolute memory locations,
  - or they may be given as offsets from some reference address.
- If you specify the addresses relative to a reference address, the program can go anywhere in memory -- it is *relocatable*.
  - Architectures with segmented memory usually require relative addresses.
- Many programming languages allow a mix of static and dynamic allocation.

2.2 Dynamic Memory Management: Recursion

- In dynamic memory management, memory addresses are not fixed (bound) until run time.
- We use this approach to support recursive subprograms and to permit dynamic allocation of memory.

- Recursive Subprograms:
  - A recursive subprogram is one that calls itself, either directly or indirectly.
  - When recursive subprograms are used, we refer to each call as an instantiation, or *activation*.
  - If the subprogram has local variables, the old values of these variables must be saved before the recursive call so that the new activation won’t wipe them out.

- Saving these variables means that memory must be allocated for each activation.
- And when each activation terminates, its local variables must be jettisoned so that the variables for the previous activation will be in effect again.
- We manage this by means of a stack. The thing pushed on the stack is called an activation record
  - it provides storage for all the local variables
  - it usually includes other things as well, such as:
    - parameters being passed.
    - the return address of the subprogram that called it.

- Any time a subprogram needs to access a variable, it looks in the topmost activation record on the stack. If it is a local variable it will be there.
- If it needs a non-local variable, one that is "scoped in" from some other procedure, then it must find that in the activation record for the block that owns the variable.
- Let's look at example of this with a recursive function to compute factorial values.


```
Program FACTORIAL;
var
   f, n, lim: word;

function FACT(n: word): word;
      begin
      if n > lim then
         fact := 0
      else if n = 0 then
         fact := 1
      else
         fact := n * fact(n - 1);
      end;  { Fact }

   begin  { Main }
      lim := 9;
      repeat
         write ('n: ');
         readln (n);
         f := fact(n);
         writeln (f);
      until n = 0;
   end.
```

- Suppose we call fact(2). The stack of activation records looks like this before the call.

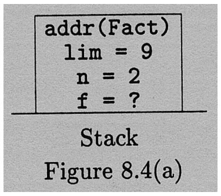

Figure 8.4(a)

- When we call fact, an activation record for that call goes on the stack:

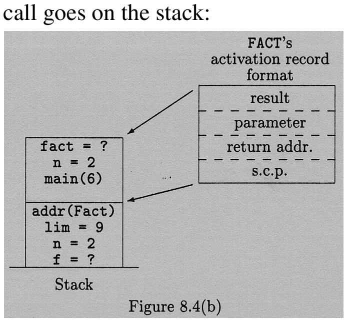

Figure 8.4(b)

- In order to access lim, Fact uses the s.c.p or Static Chain Pointer to know where the enclosing scope is, no matter how deep the direct recursion has gone.
- It is called static, because it is used to implement the static scope rules of a language like Pascal, or C.

- So after the second call to Fact, the stack (with the s.c.p's drawn in would look like this:

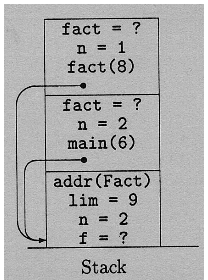

Figure 8.4(d)

- Now to access a variable one level down in the Symbol Table Stack, we go down one level of the s.c.p.
- So, in general, accessing a nonlocal variable is simple. We know the nesting level at compile time; that gives us the number of links in the chain.
- From this we can see that each variable can be identified as two numbers:
  - how many links in the static chain are needed to take us to the correct activation record,
  - and how far up from the start of that activation the desired variable is.

- There are a couple of ways the static chain pointer can be set.
  - When a function calls another, it can copy the base of its own activation record into the s.c.p of the function it calls.
    - This happens if a sub-scope is entered.
  - If the function called has the same scope as the one being called, the s.c.p is just copied.
  - In our language, where you can have functions inside of functions, you can have function A call function B, where B is at a lower level of scope.
    - In this case, you have to go find the right s.c.p. by going back through the s.c.p. chain and copying the right one in.
    - You do this by checking how many ST stack levels you went through to find the name of the function.

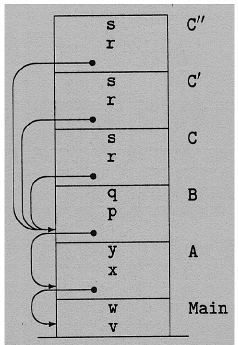

Figure 8.5

- The Display.
  - Most programs have only a modest number of levels of nesting, and traversing the chain of pointers is not excessively time consuming.
  - But there is a faster way to access nonlocal variables.
  - We can have an array of pointer to activation records as shown on the next slide.
  - Such an array is called a *display*.

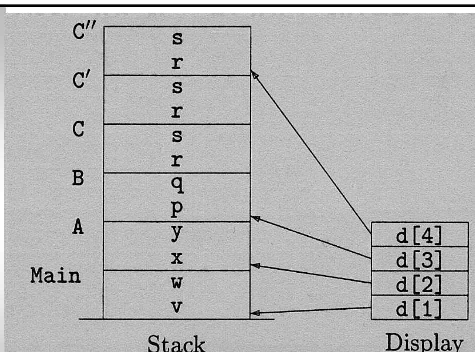

Figure 8.6

- The display offers quicker access to activation records because we require only a change of subscript to find the right activation pointer.
- We have shown the display as a separate array here, but it could be included as a part of the activation record.
- Notice that we need as many pointers as levels of nesting, and each pointer is aimed at the most recent activation of that level
  - The question is, how many levels of nesting are there?
  - When do you know that number?

- So, when there is a call to someone at the same level (recursion for example) the old pointer needs to be saved, and the new pointer placed in the display.

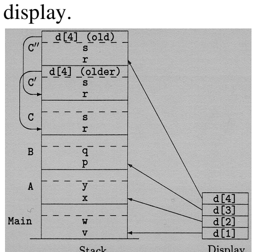

Figure 8.7(c)

- The creation of a new activation record is a shared responsibility.
  - The calling program will push the things it knows about onto the stack
    - parameters,
    - chain pointer,
    - return address
  - The subprogram, knowing how many local variables it requires, will take care of the rest.

- When translating the called subprogram,
  - the compiler generates code for accessing local variables via references to the stack,
  - and it must generate special code for each access to a nonlocal variable,
    - either by traversing the chain pointers,
    - or by consulting the display.

2.3 Dynamic Memory Management: Run-Time Allocation

- Procedures like `new` or `malloc` allocate a block of memory and return a pointer to the block.
- We must now consider where this memory comes from and how the allocation (and the de-allocation) are managed.

- The blocks come from an area of unused memory known as the *heap*.
- The size of the heap depends upon the amount of available memory, the size of the program and its static data.
- In many cases, all unused memory is shared between the heap and the stack.
  - The stack grows downward from the top,
  - and the heap grows upward from the bottom.
- If memory demands are so great that we run out of space, we get a *heap-stack collision* and the program crashes.

- The run-time support software has access to the heap and has information about what parts of it are in use and what parts are available.
- When a request for memory comes, it must somehow specify the amount of memory required.
- There are a number of ways of managing the heap, and over the next few slides we will discuss one of them.

- Initially, the entire heap is free
- As blocks are requested, they are allocated sequentially starting from the bottom of the heap; a heap pointer marks the boundary between the blocks that are in use (active blocks) and the free area.

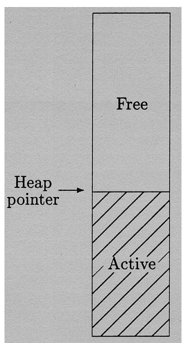

Figure 8.8(a)

- Eventually, the heap pointer may reach the top of the heap (or the bottom of the stack).
- If no blocks have been freed by this time, we are dead; but this rarely happens. Instead, we have a picture like this:

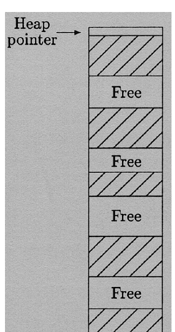

Figure 8.8(b)

- There is now free space scattered in little chunks through the heap; this situation is called *fragmentation*.
- If we are to make use of this free space that is available, we must know where it is.
- The most common way to keep track of free blocks is to form them into a linked list known as the *free-space list*.

- A small amount of space is borrowed from each free block and used to hold its size and a pointer to the next free block.

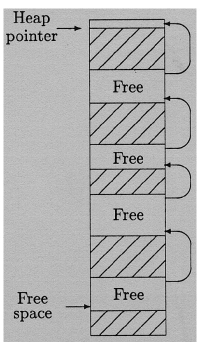

Figure 8.8(c)

- We continue allocating blocks from the free-space list.
- Searching for a big enough block takes a number of forms; two of the more common are
  - *first fit* -- select the first block it finds that is big enough.
  - *best fit* -- traverse the entire free list and pick the smallest block that is just big enough.
    - This saves large blocks for large requests
    - But it can lead to lots of slivers of free memory and therefore to extreme fragmentation.

- The general solution to the fragmentation problem is a procedure known as *compaction*.
- This is a laborious undertaking that slides all the blocks that are in use down to the bottom of the heap so that the free space moves up to the top and forms one big block again.

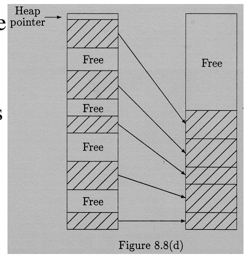

Figure 8.8(d)

- There are four basic steps for this algorithm:
  - Finding the pointers
    - This is the hard job.
    - make a list of all the pointers in the user's program.
  - Dress rehearsal
    - don't actually move the blocks, but figure out where to move them
  - Pointer Updating
    - Use the list you found in step 1, and update pointers with their new value.
  - Moving.
    - Now that pointers are correct, move the blocks.

- Garbage and Dangling References.
  - If a programmer loses the only pointer to an allocated block of memory, that block is termed *garbage*.
  - There are a few ways in which the system can help the programmer avoid the creation of garbage.
    - One method is to provide each block with a reference count (the number of pointers pointing to the block)
      - It is invisible to the programmer
      - It is managed by code inserted into pointer assignments by the compiler.

- If the count ever goes to zero, the block is returned to the free-space list, whether the user disposed of it or not.
- This works most of the time, but not always.

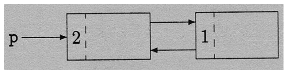

Figure 8.9(a)

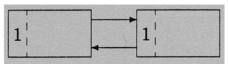

Figure 8.9(b)

- Garbage Collection.
  - Some languages, most notably LISP, provide no explicit disposal mechanism, and hence no opportunity to create dangling references.
  - You are free to create garbage with wild abandon, and at intervals the support system cleans up your mess for you.
  - This process is known as *garbage collection*.
  - Garbage collection achieves the seemingly impossible: it locates garbage (which we though was by definition impossible to locate) and returns it to the free-space list.

- The trick is that we don't locate the set of all garbage; we locate its complement.
- Here is the basic procedure:
  - Go sequentially through the heap and unmark every block.
  - Starting with all the pointers in the user's code, follow the chains of pointers and mark all the blocks encountered. (At this point we have marked everything that *isn't* garbage)
  - Now go sequentially through the heap again and identify all unmarked blocks and return them to the free-space list.

- It should be obvious that while garbage collection (or compaction) is going on, the execution of the user's program must be suspended.
- Hence you may wish to provide for some notification that garbage collection is going on.
- Otherwise when the program stops running, the user may suspect a bug, and panic.

3. Summary

- The topics discussed in this chapter are the basics of memory management at compile time and at run time.
- Some languages may create special problems:
  - unlabeled and labeled COMMON blocks in FORTRAN require some special handling.
  - Languages like PL/I, in which subprograms are separately compiled, require that the calling program have enough information to create activation records properly.

- Garbage collection and compaction are not always available.
  - Garbage collection is normally associated with languages in which there is no explicit de-allocation of variables.
- Many languages do the best they can with the heap, and if you run out of space because of fragmentation, you're dead.