UNIT–IV (Part I)
Runtime Environments

4.1 Storage Organization

From the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location. The management and organization of this logical address space is shared between the compiler, operating system, and target machine. The operating system maps the logical addresses into physical addresses, which are usually spread throughout memory.

The run-time representation of an object program in the logical addressspace consists of data and program areas as shown in Fig. 4.1.

The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area ***Code,*** usually in the low end of memory.

Similarly, the size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area called ***Static.*** One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code.

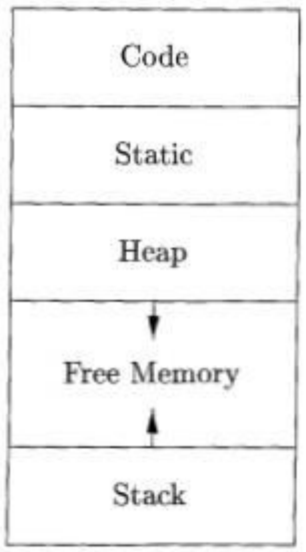


Figure 4.1: Typical subdivision of run-time memory into code and data areas

To maximize the utilization of space at run time, the other two areas, ***Stack* and *Heap*,** are at the opposite ends of the remainder of the address space. These areas are dynamic; their size can change as the program executes. These areas grow towards each other as needed. The stack is used to store data structures called activation records that get generated during procedure calls. In practice, the stack grows towards lower addresses, the heap towards higher.

4.1.1 Static versus Dynamic Storage Allocation

The layout and allocation of data to memory locations in the run-time environment are key issues in storage management. We say that a storage-allocation decision is ***static***, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes. Conversely, a decision is ***dynamic*** if it can be decided only while the program is running. Many compilers use some combination of the following two strategies for dynamic storage allocation:

1. ***Stack storage.*** Names local to a procedure are allocated space on a stack.The stack supports the normal call/return policy for procedures.
2. ***Heap storage.*** Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage.

4.2 Stack Allocation of Space

Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack. Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack. This arrangement not only allows space to be shared by procedure calls whose durations do not overlap in time, but it allowsus to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same, regardless of the sequence of procedure calls.

4.2.1 Activation Trees

Stack allocation would not be feasible if procedure calls, or *activations* of procedures,did not nest in time. The following example illustrates nesting ofprocedure calls.

Example 4.1: Figure 4.2 contains a sketch of a program that reads nine integersinto an array *a* and sorts them using the recursive quicksort algorithm.

```
int a[11];
void readArray() { /* Reads 9 integers into a[1],...,a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Figure 4.2: Sketch of a quicksort program

The main function has three tasks. It calls *readArray,* sets the sentinels, and then calls *quicksort* on the entire data array.

Figure 5.3 suggests a sequence ofcalls that might result from an execution of the program. In this execution, the call to *partition(1,9)* returns 4, so a[1] through a[3] hold elements less than its chosen separator value *v,* while the larger elements are in a[5] through a[9].

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 4.3: Possible activations for the program of Fig.5.2

In this example, procedure activations are nested intime. If an activation of procedure *p* calls procedure *q,* then that activation of*q* must end before the activation of *p* can end. There are three common cases:

1. The activation of *q* terminates normally. Then in essentially any language, control resumes just after the point of *p* at which the call to *q* was made.
2. The activation of *q,* or some procedure *q* called, either directly or indirectly, aborts; i.e., it becomes impossible for execution to continue. In that case, *p* ends simultaneously with *q.*
3. The activation of q terminates because of an exception that *q* cannot handle. Procedure p may handle the exception, in which case the activation of *q* has terminated while the activation of *p* continues, although not necessarily from the point at which the call to *q*

was made. If *p* cannot handle the exception, then this activation of *p* terminates at the same time as the activation of *q,* and presumably the exception will be handled by some other open activation of a procedure.

We therefore can represent the activations of procedures during the running of an entire program by a tree, called an ***activation tree.*** Each node corresponds to one activation, and the root is the activation of the "main" procedure that initiates execution of the program. At a node for an activation of procedure *p,* the children correspond to activations of the procedures called by this activationof *p.* We show these activations in the order that they are called, from left to right. Notice that one child must finish before the activation to its right can begin.

One possible activation tree that completes the sequence of calls and returns suggested in Fig. 4.3 is shown in Fig. 4.4. Functions are represented by the first letters of their names. Remember that this tree is only one possibility, since the arguments of subsequent calls, and also the number of calls along any branch is influenced by the values returned by ***partition.***

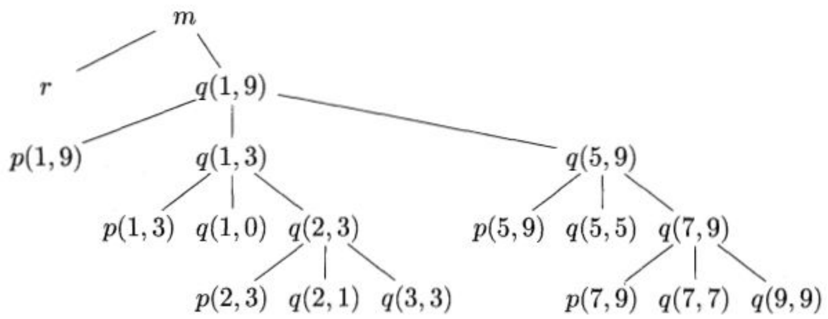


Figure 4.4: Activation tree representing calls during an execution of quicksort

The use of a run-time stack is enabled by several useful relationships betweenthe activation tree and the behavior of the program:

1. The sequence of procedure calls corresponds to a preorder traversal of the activation tree.
2. The sequence of returns corresponds to a postorder traversal of the activation tree.
3. Suppose that control lies within a particular activation of some procedure, corresponding to a node *N* of the activation tree. Then the activations that are currently open *(live)* are those that correspond to node *N* and its ancestors. The order in which these activations were called is the order in which they appear along the path to *N,* starting at the root, and they will return in the reverse of that order.

4.2.2 Activation Records

Procedure calls and returns are usually managed by a run-time stack called the ***control stack.*** Each live activation has an ***activation record*** (sometimes called a ***frame)*** on the control stack, with the root of the activation tree at the bottom,and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.The latter activation has its record at the top of the stack.
If control is currently in the activation *q(2,*3) of the tree of Fig. 4.4, then the activation record for *q(2,*3) is at the top of the control stack.Just below is the activation record for q(1,3), the parent of q(2,3) in the tree. Below that is the activation record q(1,9), and at the bottom is the activation record for m, the main function and root of the activation tree.
We shall conventionally draw control stacks with the bottom of the stack higher than the top, so the elements in an activation record that appear lowest on the page are actually closest to the top of the stack.
The contents of activation records vary with the language being implemented. The fields of activation record are shown in Fig.4.5:

1. ***Temporary values***, Contains compiler generated temporaries.
2. ***Local data*** belonging to the procedure whose activation record this is.
3. A ***saved machine status***, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.
4. An "***access link***" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.
5. ***A control link,*** pointing to the activation record of the caller.

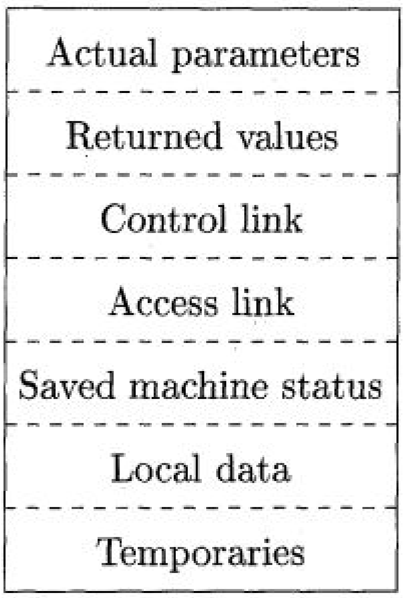


Figure 4.5: A general activation record

6. Space for the ***return value*** of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
7. The ***actual parameters*** used by the calling procedure. Commonly, thesevalues are not placed in the activation record but rather in registers, whenpossible, for greater efficiency. However, we show a space for them to becompletely general.

Figure 4.6 shows snapshots of the run-time stack as controlflows through the activation tree of Fig. 4.4. Dashed lines in the partial treesgo to activations that have ended. Since array *a* is global, space is allocated forit before execution begins with an activation of procedure *main,* as shown inFig. 4.6(a).
When control reaches the first call in the body of *main,* procedure r is activated, and its activation record is pushed onto the stack (Fig. 5.6(b)). The activation record for r contains space for local variable *i.* Recall that the top of stack is at the bottom of diagrams. When control returns from this activation,its record is popped, leaving just the record for *main* on the stack.

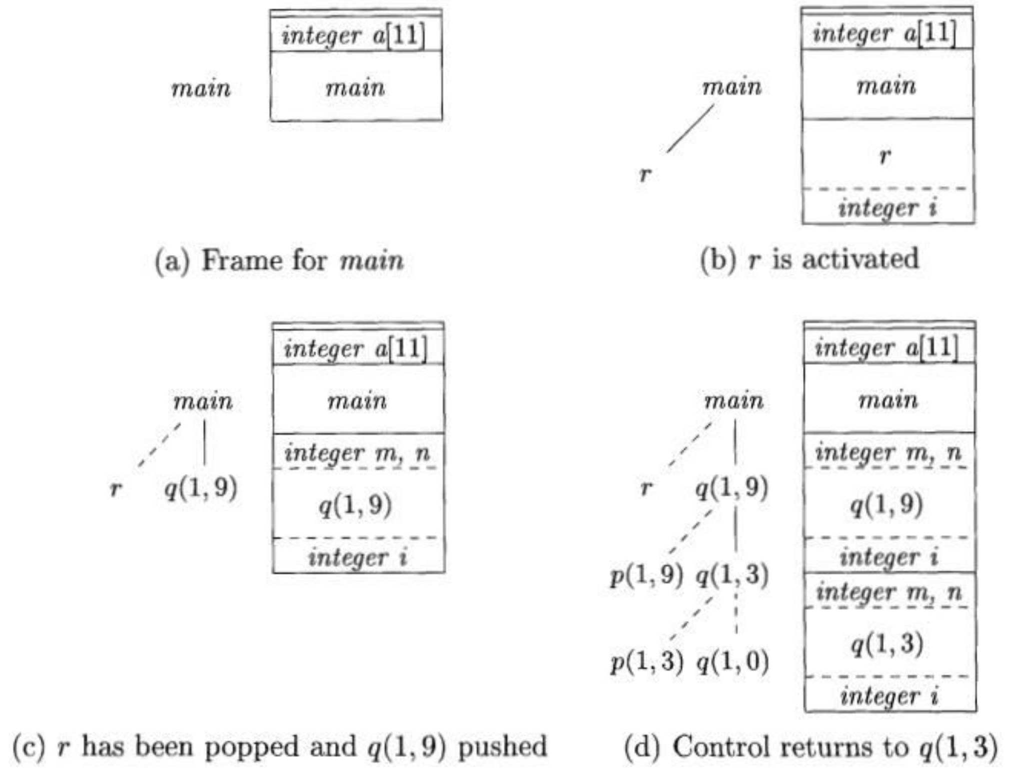


Figure 4.6: Downward-growing stack of activation records

Control then reaches the call to *q* (quicksort) with actual parameters 1 and 9, and an activation record for this call is placed on the top of the stack, as inFig. 4.6(c). The activation record for *q* contains space for the parameters *m* and *n* and the local variable *i,* following the general layout in Fig. 4.5. Notice that space once used by the call of r is reused on the stack. No trace of datalocal to r will be available to *q(1,* 9). When *q(1,* 9) returns, the stack again has only the activation record for *main.*

Several activations occur between the last two snapshots in Fig. 4.6. A recursive call to q(1,3) was made. Activations p(1,3) and *q(1,0)* have begun and ended during the lifetime of *q(1,*3), leaving the activation record for *q(1,*3) on top (Fig.4.6(d)). Notice that when a procedure is recursive, it is normal to have several of its activation records on the stack at the same time.

4.2.3 Calling Sequences

Procedure calls are implemented by what are known as ***calling sequences,*** which consists of code that allocates an activation record on the stack and enters information into its fields. A ***return sequence*** is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.

The code in a calling sequence is often divided between the calling procedure (the "caller") and the procedure it calls (the "callee").

An example of how caller and callee might cooperate in managing the stack is suggested by Fig. 4.7. A register ***top_sp*** points to the end of the machine status field in the current top activation record. This position within the callee's activation record is known to the caller, so the caller can be made responsible for setting *top_sp* before control is passed to the callee.

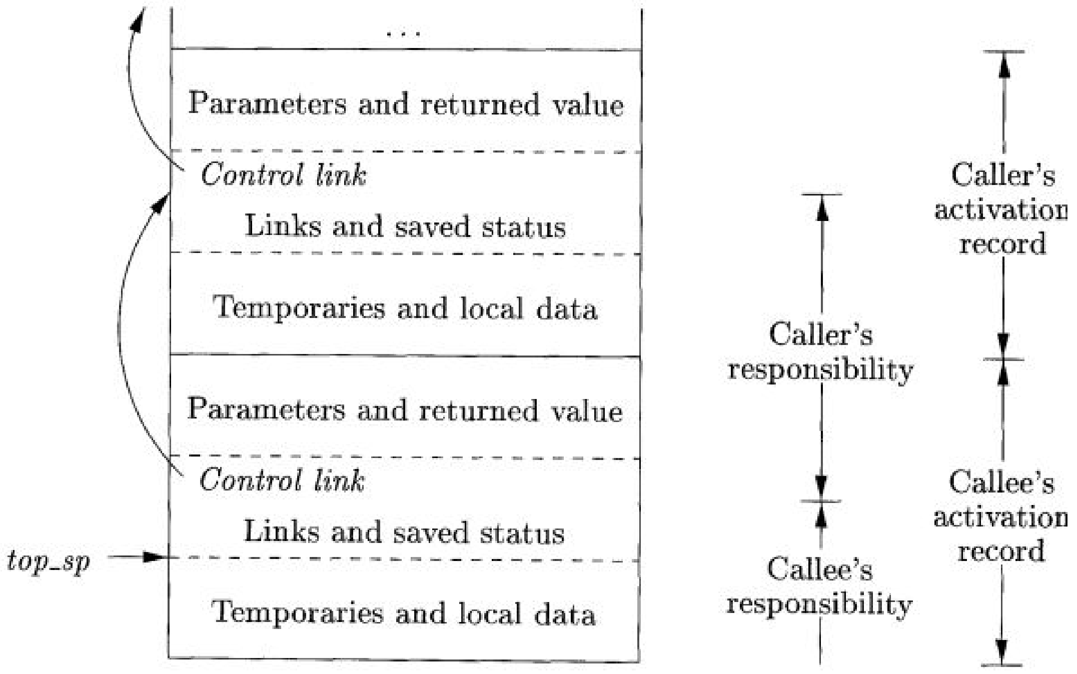

...
Parameters and returned value
Control link
Links and saved status
Temporaries and local data
Parameters and returned value
Control link
Links and saved status
Temporaries and local data
top_sp
Caller's responsibility
Callee's responsibility
Caller's activation record
Callee's activation record

Figure 4.7: Division of tasks between caller and callee

The **calling sequence**and its division between caller and callee is as follows:

1. The caller evaluates the actual parameters.
2. The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments *to_sp* to the position. That is, *top_sp* is moved past the caller's local data and temporaries and the callee's parameters and status fields.
3. The callee saves the register values and other status information.
4. The callee initializes its local data and begins execution.

A suitable, corresponding **return sequence** is:

1. The callee places the return value next to the parameters, as in Fig. 4.5.
2. Using information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.

3. Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp;* the caller therefore may use that value.

The above calling and return sequences allow the number of arguments ofthe called procedure to vary from call to call.

4.2.4 Variable-Length Data on the Stack

The run-time memory-management system must deal frequently with the allocation of space for objects the sizes of which are not known at compile time,but which are local to a procedure and thus may be allocated on the stack.

A common strategy for allocating variable-length arrays is shown in Fig. 4.8. The same scheme works for objects of any type if they arelocal to the procedure called and have a size that depends on the parametersof the call.

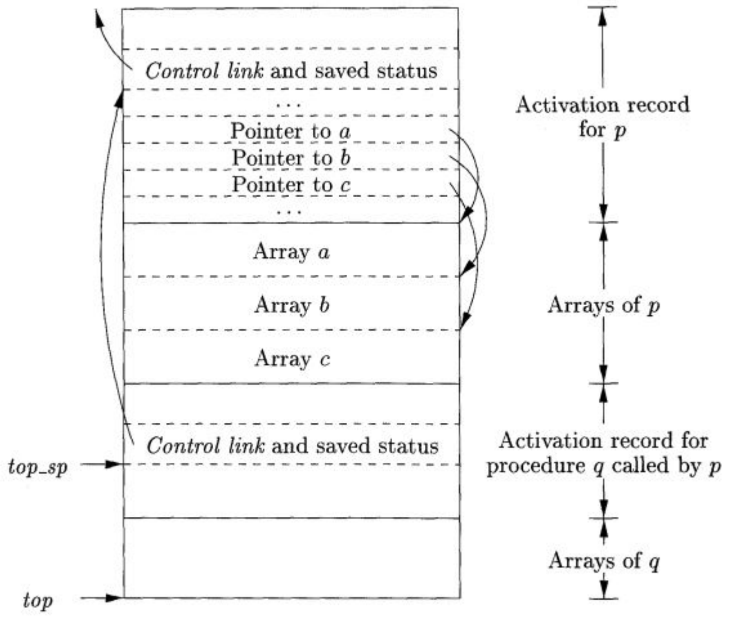


Figure 4.8: Access to dynamically allocated arrays

Also shown in Fig. 4.8 is the activation record for a procedure *q,* called by *p.*The activation record for *q* begins after the arrays of p, and any variable-length arrays of *q* are located beyond that.

Access to the data on the stack is through two pointers, ***top*** and ***top_sp***.

- ***Top*** marks the actual top of stack; it points to the position at which the next activation record will begin.
- ***top_sp*** is used to find local, fixed-length fields of the top activation record.

In Fig. 4.8, *top_sp* points to the end of this field in the activation record for *q.*From there, we can find the control-link field for *q,* which leads us to the place in the activation record for *p* where *top_sp* pointed when *p* was on top.The code to reposition *top* and *top_sp* can be generated at compile time, in terms of sizes that will become known at run time. When *q* returns, *top_sp* can be restored from the saved control link in the activation record for *q.* Thenew value of *top* is (the old unrestored value of) *top_sp* minus the length of the machine-status, control and access link, return-value, and parameter fields (as in Fig. 4.5) in q's activation record. This length is known at compile time to the caller, although it may depend on the caller, if the number of parameters can vary across calls to *q.*

4.3 Access to Nonlocal Data on the Stack

4.3.1 Data Access without Nested Procedures

In the C family of languages, all variables are defined either within a single function or outside any function ("globally"). Variables declared within a function have a scope consisting of that function only, or part of it, if the function has nested blocks.

For languages that do not allow nested procedure declarations, allocation of storage for variables and access to those variables is simple:

1. Global variables are allocated static storage. The locations of these variables remain fixed and are known at compile time. So to access any variable that is not local to the currently executing procedure, we simply use the statically determined address.
2. Any other name must be local to the activation at the top of the stack.

We may access these variables through the *top_sp* pointer of the stack.

4.3.2 Issues with Nested Procedures

Access becomes far more complicated when a language allows procedure declarations to be nested and also uses the normal static scoping rule; that is, a procedure can access variables of the procedures whose declarations surround its own declaration, following the nested scoping rule.

Finding the declaration that applies to a nonlocal name x in a nested procedure p is a **static decision**; it can be done by an extension of the static-scope rule for blocks. Suppose x is declared in the enclosing procedure q. Finding the relevant activation of q from an activation of p is a **dynamic decision**; it requires additional run-time information about activations. One possible solution to this problem is to use "**access links**."

4.3.3 Access Links

A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the ***access link*** to each activation record. If procedure *p* is nested immediately within procedure *q* in the source code, then the access link in any activation of *p* points to the most recent activation of *q* .**Note that the nesting depth of *q* must be exactly one less than the nesting depth of p.**

Suppose that the procedure *p* at the top of the stack is at nesting depth n_p, and *p* needs to access *x,* which is an element defined within some procedure *q* that surrounds *p* and has nesting depth n_q. Note that $n_q < n_p$, with equality only if *p* and *q* are the same procedure. To find *x,* we start at the activation record for *p* at the top of the stack and follow the access link $n_p — n_q$ times, from activation record to activation record. Finally, we wind up at an activation record for *q,* and it will always be the most recent (highest) activation record for *q* that currently appears on the stack. This activation record contains the element *x* that we want. Since the compiler knows the layout of activation records, *x* will be found at some fixed offset from the position in g's activation record that we can reach by following the last access link.

Figure 4.9 shows a sequence of stacks that might result from execution of the function *sort.* As before, we represent function names by their first letters, and we show some of the data that might appear in the various activation records, as well as the access link for each activation. In Fig. 4.9(a), we see the situation after *sort* has called *readArray* to load input into the array *a* and then called *quicksort(1,*9) to sort the array. The access link from *quicksort(1,*9) points to the activation record for *sort,* not because *sort* called *quicksort* but because *sort* is the most closely nested function.

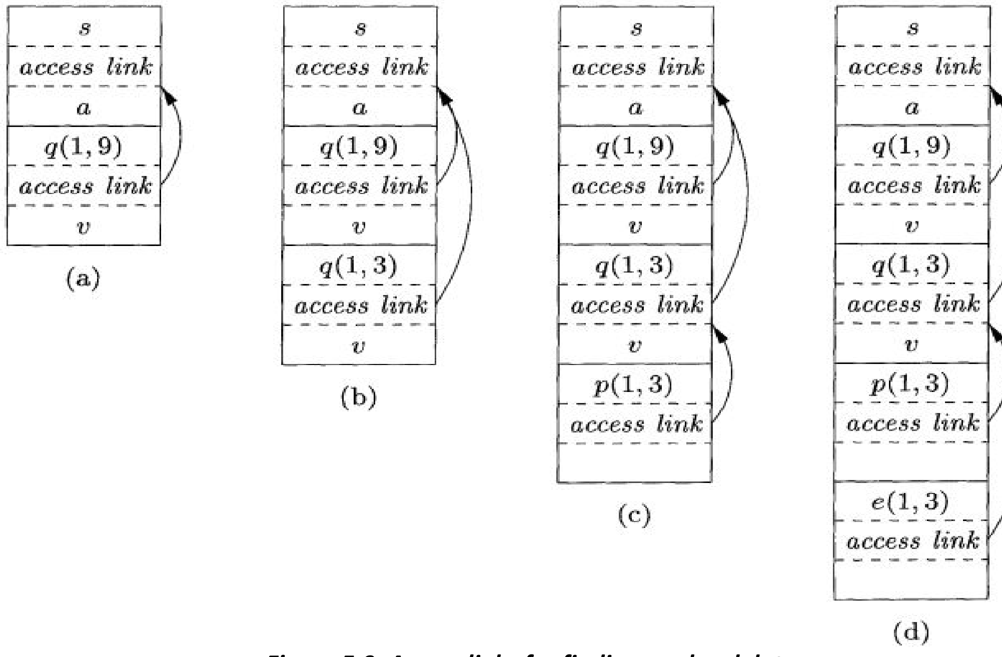


Figure 5.9: Access links for finding nonlocal data

In successive steps of Fig. 4.9 we see a recursive call to *quicksort(1*3),followed by a call to *partition,* which calls *exchange.* Notice that *quicksort(1,* 3)'s access link points to *sort,* for the same reason that *quicksort(1,* 9)'s does.

In Fig. 4.9(d), the access link for *exchange* bypasses the activation records for *quicksort* and *partition,* since *exchange* is nested immediately within *sort.*

4.4 Heap Management

The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it.

4.4.1 The Memory Manager

The memory manager keeps track of all the free space in heap storage at all times. It performs two basic functions:

- ***Allocation**.* When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size. If possible, it satisfies an allocation request using free space in the heap; if no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system. If space is exhausted, the memory manager passes that information back to the application program.
- ***Deallocation.***The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests.

Properties of memory manager:

- ***Space Efficiency**.* A memory manager should minimize the total heap space needed by a program. Doing so allows larger programs to run in a fixed virtual address space. Space efficiency is achieved by minimizing "fragmentation."
- ***Program Efficiency**.* A memory manager should make good use of the memory subsystem to allow programs to run faster.
- ***Low Overhead**.* Because memory allocations and deallocations are frequent operations in many programs, it is important that these operations be as efficient as possible.

4.4.2 The Memory Hierarchy of a Computer

Memory management and compiler optimization must be done with an awareness of how memory behaves. The large variance in memory access times is due to the fundamental limitation in hardware technology; we can build small and fast storage, or large and slow storage, but not storage that is both large and fast. All modern computers arrange their storage as a ***memory hierarchy**.* A memory hierarchy, as shown in Fig. 4.10, consists of a series of storage elements, with the smaller faster ones "closer" to the processor, and the larger slower ones further away.

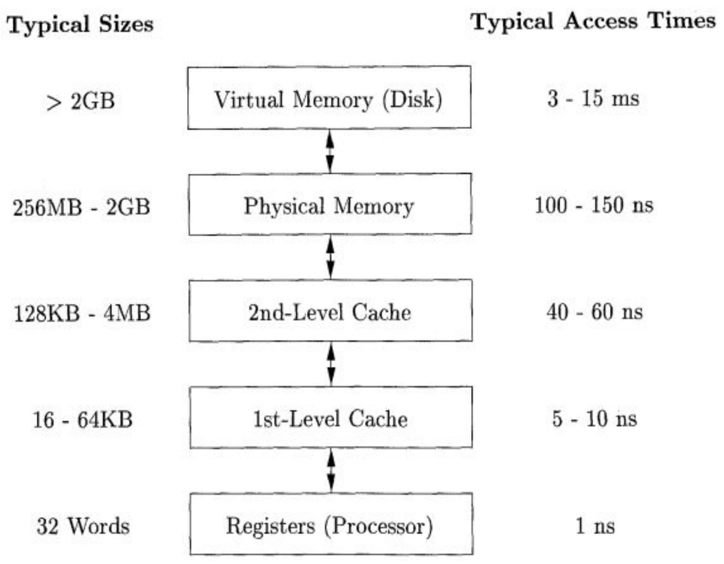


Figure 4.10: Typical Memory Hierarchy Configurations

Typically, a processor has a small number of registers, whose contents are under software control. Next, it has one or more levels of cache, usually made out of static RAM, that are kilobytes to several megabytes in size. The next level of the hierarchy is the physical (main) memory, made out of hundreds of megabytes or gigabytes of dynamic RAM. The physical memory is then backed up by virtual memory, which is implemented by gigabytes of disks. Upon a memory access, the machine first looks for the data in the closest (lowest-level)storage and, if the data is not there, looks in the next higher level, and so on.

4.4.3 Locality in Programs

Most programs exhibit a high degree of ***locality;*** that is, they spend most of their time executing a relatively small fraction of the code and touching only a small fraction of the data.

- We say that a program has ***temporal locality,*** *if* at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.
- We say that a program has ***spatial locality,*** if a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.

The conventional wisdom is that programs spend 90% of their time executing 10% of the code. Here is why:

- Programs often contain many instructions that are never executed.
- Only a small fraction of the code that could be invoked is actually executed in a typical run of the program.
- The typical program spends most of its time executing innermost loops and tight recursive cycles in a program.

Locality allows us to take advantage of the memory hierarchy of a modern computer. By placing the most common instructions and data in the fast-but-small storage, while leaving the rest in the slow-but-large storage, we can lower the average memory-access time of a program significantly.

Optimization Using the Memory Hierarchy

The policy of keeping the most recently used instructions in the cache tends to work well. When a new instruction is executed, there is a high probability that the next instruction also will be executed. This phenomenon is an example of spatial locality. One effective technique to improve the spatial locality of instructions is to have the compiler place basic blocks (sequences of instructions that are always executed sequentially) that are likely to follow each other contiguously — on the same page, or even the same cache line, if possible. Instructions belonging to the same loop or same function also have a high probability of being executed together.

We can also improve the temporal and spatial locality of data accesses in a program by changing the data layout or the order of the computation. For example, programs that visit large amounts of data repeatedly, each time performinga small amount of computation, do not perform well. It is better if we can bring some data from a slow level of the memory hierarchy to a faster level (e.g., disk to main memory) once, and perform all the necessary computations on this data while it resides at the faster level. This concept can be applied recursively to reuse data in physical memory, in the caches and in the registers.

4.4.4 Reducing Fragmentation

At the beginning of program execution, the heap is one contiguous unit of freespace. As the program allocates and deallocates memory, this space is brokenup into free and used chunks of memory, and the free chunks need not reside in a contiguous area of the heap. We refer to the free chunks of memory as ***holes.*** With each allocation request, the memory manager must *place* the requested chunk of memory into a large-enough hole. Unless a hole of exactly the rightsize is found, we need to *split* some hole, creating a yet smaller hole.

With each deallocation request, the freed chunks of memory are added back to the pool of free space. We combine contiguous holes into larger holes, as the holes can only get smaller otherwise. If we are not careful, the memory may end up getting ***fragmented,*** consisting of large

numbers of small, noncontiguous holes. It is then possible that no hole is large enough to satisfy a future request,even though there may be sufficient aggregate free space.
We reduce fragmentation by controlling how the memory manager places new objects in the heap. It has been found empirically that a good strategy for minimizing fragmentation for real-life programs is to allocate the requested memory in the smallest available hole that is large enough. This ***best-fit*** algorithm tends to spare the large holes to satisfy subsequent, larger requests. An alternative, called ***first-fit,*** where an object is placed in the first (lowest-address) hole in which it fits, takes less time to place objects, but has been found inferior to best-fit in overall performance.

Coalescing Free Space
When an object is deallocated manually, the memory manager must make its chunk free, so it can be allocated again. In some circumstance, it may also be possible to combine (coalesce) that chunk with adjacent chunks of the heap, to form a larger chunk. The advantage is that a larger chunk can hold a larger object whereas smaller chunk cannot.

Problems with Manual Deallocation
Manual memory management is error-prone. The common mistakes take two forms: failing ever to delete data that cannot be referenced is called a ***memory leak*** error, and referencing deleted data is a ***dangling-pointer-dereference*** error.

4.5. Garbage Collection
Data that cannot be referenced is generally known as *garbage.* Many high-level programming languages remove the burden of manual memory management from the programmer by offering automatic garbage collection, which deallocates unreachable data. Garbage collection is the reclamation of chunks of storage holding objects that can no longer be accessed by a program. We need to assume that objects have a type that can be determined by the garbage collector at run time. From the type information, we can tell how large the object is and which components of the object contain references (pointers) to other objects. We also assume that references to objects are always to the address of the beginning of the object, never pointers to places within the object. Thus, all references to an object have the same value and can be identified easily. A user program, which we shall refer to as the *mutator,* modifies the collection of objects in the heap. The mutator creates objects by acquiring space from the memory manager, and the mutator may introduce and drop references to existing objects. Objects become garbage when the mutator program cannot" reach" them.
Garbage Collection works as follows:

1. When an application needs some free space to allocate to the objects and if there is no free space available to allocate the memory for these objects then a system routine called ***garbage collector*** is invoked.
2. Garbage collector then searches the system for the objects that are no longer used. The memory allocated for these nodes is deallocated and added to the heap. Then the system can use the available free space from the heap.
   Reference count is a special counter used during implicit memory allocation. If any block is referred by any other block then its reference count is incremented by one. When a reference count of a block is zero it is not referred by any other block. Hence garbage collector searches the system for such unreferred blocks and the memory allocated for these blocks is deallocated and added to the heap.

Advantages:

- **i)** The manual memory management is done by the programmer and is time consuming and error prone. Hence automatic memory management is done.
- **ii)** Reusability of memory can be achieved with the help of garbage collector.

Disadvantages:

- **i)** The execution of the program is stopped for some time when the garbage collector is invoked automatically.
- **ii)** Sometimes a situation like **Thrashing** may occur due to garbage collector. Let us assume that garbage collector is called for getting some free space but only small amount of free space is available. Now garbage collector executes and returns only a

small amount of space. Again the system invokes garbage collector for getting some more free space. Once again garbage collector executes and returns very small amount of space. This happens repeatedly and garbage collector is executing almost all the time. This process is called **Thrashing**. Thrashing must be avoided for better system performance.

4.6. Trace-Based Collection

Instead of collecting garbage as it is created, trace-based collectors run periodically to find unreachable objects and reclaim their space. Typically, we run the trace-based collector whenever the free space is exhausted or its amount drops below some threshold.

4.6.1. A Basic Mark and Sweep Collector

Mark-and-sweep garbage-collection algorithms are straightforward, stop-the-world algorithms that find all the unreachable objects, and put them on the list of free space

Any garbage collection algorithm must perform 2 basic operations. One, it should be able to detect all the unreachable objects and secondly, it must reclaim the heap space used by the garbage objects and make the space available again to the program. The above operations are performed by Mark and Sweep Algorithm in two phases as listed below:

i) Mark phase
ii) Sweep phase

i) Mark Phase:

When an object is created, its mark bit is set to 0(false). In the Mark phase, we set the marked bit for all the reachable objects (or the objects which a user can refer to) to 1(true). Now to perform this operation we simply need to do a graph traversal, a depth-first search approach would work for us. Here we can consider every object as a node and then all the nodes (objects) that are reachable from this node (object) are visited and it goes on till we have visited all the reachable nodes.

- The root is a variable that refers to an object and is directly accessible by a local variable. We will assume that we have one root only.
- We can access the mark bit for an object by *'markedBit(obj)'*.

Algorithm:

```
mark(root)
If markedBit(root) = false then
    markedBit(root) = true
      For each v referenced by root
          mark(v)
```

ii) Sweep Phase

As the name suggests it "sweeps" the unreachable objects i.e. it clears the heap memory for all the unreachable objects. All those objects whose marked value is set to false are cleared from the heap memory, for all other objects (reachable objects) the marked bit is set to true. Now the mark value for all the reachable objects is set to false since we will run the algorithm (if required) and again we will go through the mark phase to mark all the reachable objects.

Algorithm:

```
Sweep()
For each object p in heap
If markedBit(p) = true then
            markedBit(p) = false
                    else
                      heap.release(p)
```

The mark-and-sweep algorithm is called a tracing garbage collector because it traces out the entire collection of objects that are directly or indirectly accessible by the program.

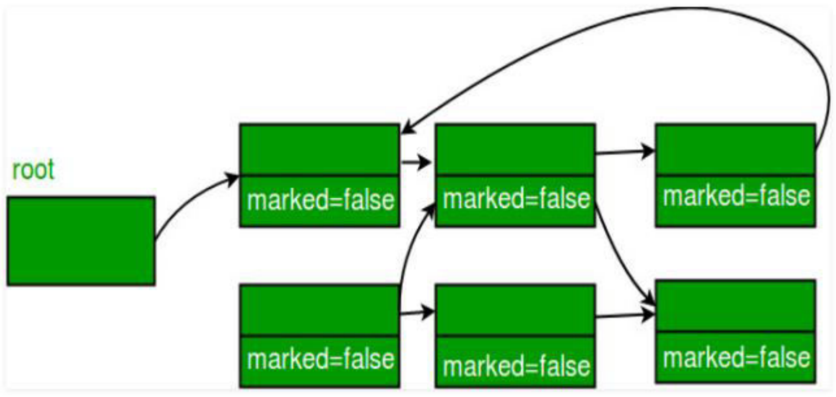


Fig. All the objects have their marked bits set to

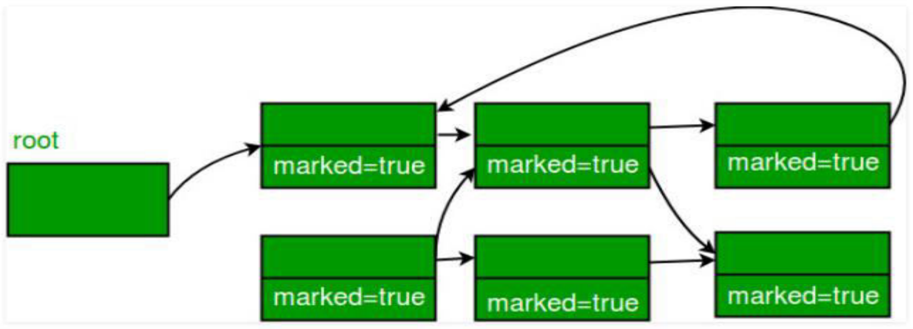


Fig. Reachable objects are marked true

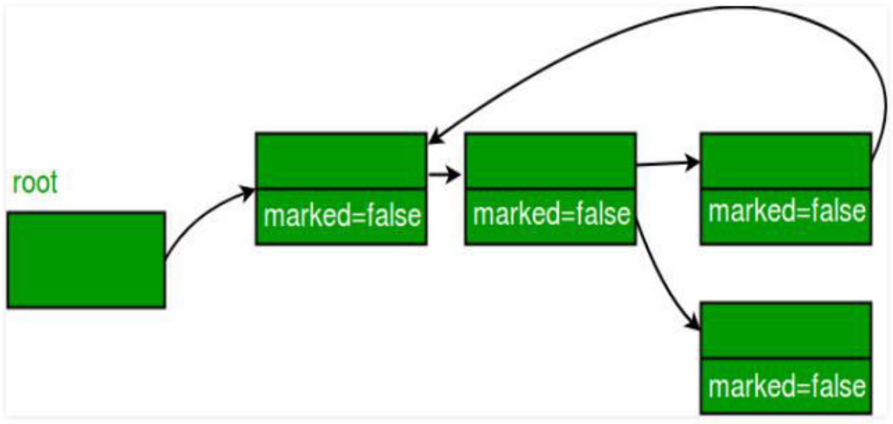


Fig.Nonreachable objects are cleared from the heap.

Advantages of Mark and Sweep Algorithm:

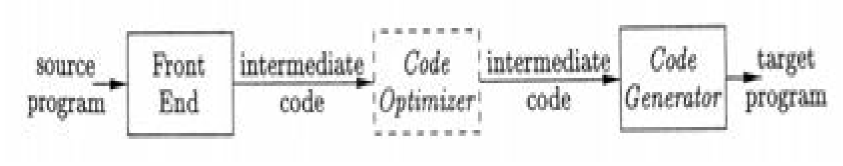
- It handles the case with cyclic references, even in the case of a cycle, this algorithm never ends up in an infinite loop.
- There are no additional overheads incurred during the execution of the algorithm.

Disadvantages of the Mark and Sweep Algorithm:

- The main disadvantage of the mark-and-sweep approach is the fact that that normal program execution is suspended while the garbage collection algorithm runs.

UNIT–IV (Part II)
Code Generation

Code Generation phase is the final phase of compilation. It takes as input the intermediate representation (IR) produced by the front end of the compiler,along with relevant symbol table information, and produces as output a semantically equivalent target program, as shown in Fig.



The requirements imposed on a code generator are severe. The target program must preserve the semantic meaning of the source program and be of high quality; that is, it must make effective use of the available resources of the target machine. Moreover, the code generator itself must run efficiently.

A code generator has three primary tasks:

1. **Instruction selection** involves choosing appropriate target-machine instructions to implement the IR statements.
2. **Register allocation and assignment** involves deciding what values to keep in which registers.
3. **Instruction ordering** involves deciding in what order to schedule the execution of instructions.

4.7. Issues in the Design of a Code Generator

The most important criterion for a code generator is that it produces correct code. Given the premium on correctness,designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.
The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code. The most common target-machine architecture sare

- **RISC** (reduced instruction set computer): A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
- **CISC** (complex instruction set computer): In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.
- **Stack based**: In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers.

4.7.1 Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by factors such as

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct. Forexample, every three-address statement of the form x = y + z, where x, y, and z are statically allocated, can be translated into the code sequence

```
LD R0, y            // R0 = y (load y into register R0)
ADD R0, R0, z       // R0 = R0 + z (add z t o R0)
ST x, R0            // x = R0 (store R0 into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

```
a = b + c
d = a + e
```

would be translated into

```
LD R0, b            / / R0 = b
ADD R0, R0, c       / / R0 = R0 + c
ST a, R0            // a = R0
LD R0, a            / / R0= a
ADD R0, R0, e        / / R0 = R0 + e
ST d, R0            // d = R0
```

Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if a is not subsequently used.

The quality of the generated code is usually determined by its speed and size. A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.

For example, if the target machine has an "increment" instruction **(INC),**then the three-address statement **a = a +** 1 may be implemented more efficiently by the single instruction **INC a,** rather than by a more obvious sequence that loads **a** into a register, adds one to the register, and then stores the result backinto **a:**

```
LD R0, a                    // R0 = a
ADD R0, R0, #1             // R0 = R0 + 1
ST a, RO                    // a = R0
```

We need to know instruction costs in order to design good code sequences but, unfortunately, accurate cost information is often difficult to obtain. Deciding which machine code sequence is best for a given three-address construct may also require knowledge about the context in which that construct appears.

4.7.2 Register Allocation

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine,but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

The use of registers is often sub divided into two sub-problems:

1. ***Register allocation,*** during which we select the set of variables that will reside in registers at each point in the program.
2. ***Register assignment,*** during which we pick the specific register that a variable will reside in.

4.7.3 Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem. Initially, we shall avoid the problem by generating code for the three-address statements in the orderin which they have been produced by the intermediate code generator.

4.8 The Target Language

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.

4.8.1 A Simple Target Machine Model

Our target computer models a three-address machine with load and store operations,computation operations, jump operations, and conditional jumps. The underlying computer is a byte-addressable machine with *n* general-purpose registers,R0,R1,... ,Rn - 1.

We assume the following **kinds of instructions** are available:

- ***Load operations***: The instruction **LD *dst, addr*** loads the value in location *addr* into location *dst.* The most common form of this instruction is **LD** *r, x* which loads the value in location *x* into register r. An instruction of the form **LD** *r1, r2* is a ***register-to-register copy*** in which the contents of register *r2* are copied into register *r1.*
- ***Store* operations**: The instruction **ST *x, r***stores the value in register *r* into the location *x.* This instruction denotes the assignment *x = r.*
- ***Computation* operations** of the form ***OP dst, src1, src2,*** where *OP* is an operator like **ADD** or **SUB,** and *dst, src1,* and *src2* are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by *OP to* the values in locations *src1* and *src2,* and place the result of this operation in location *dst.* For example, **SUB** *n, r2, r3* computes *n = r2 - r3.* Any value formerly stored in *n* is lost, but if *r1*is r2 or r3, the old value is read first. Unary operators that take only one operand do not have a *src2.*
- ***Unconditional jumps****:* The instruction **BR *L*** causes control to branch to the machine instruction with label *L.* (BR stands for *branch.)*
- ***Conditional jumps*** of the form *Bcond*r, *L,* where *r* is a register, *L* is a label, and *cond* stands for any of the common tests on values in the register r. For example, BLTZ r, *L* causes a jump to label *L* if the value in register r is less than zero, and allows control to pass to the next machine instruction if not.

We assume our target machine has a variety of **addressing modes**:

- In instructions, a location can be a **variable name *x* referring to the memory location** that is reserved for *x* (that is, the l-value of *x).*
- A location can also be an **indexed address** of the form ***a(r),*** where *a* is a variable and r is a register. The memory location denoted by *a(r)* is computed by taking the l-value of *a* and adding to it the value in register r. For example, the instruction ***LD R1, a(R2)*** has the effect of setting **R1 = *contents* (a + *contents* (R2)),** where *contents (x)* denotes the contents of the register or memory location represented by *x.* This addressing mode is useful for accessing arrays, where *a* is the base address of the array (that is, the address of the first element), and r holds the number of bytes past that address we wish to go to reach one of the elements of array *a.*
- A memory location can be an integer **indexed by a register**. For example, ***LD R1, 100(R2)*** has the effect of setting ***R1 = contents(100 + contents(R2)),***that is, of loading into R1 the value in the memory location obtained by adding **100** to the contents of register **R2.**
- We also allow **two indirect addressing modes**: **r* means the memory location found in the location represented by the contents of register r and **100(r)* means the memory location found in the location obtained by adding **100** to the contents of r. For example, LD R1, ***100(R2)** has the effect of setting R1 = *contents(contents(10Q +*

*content*s**(R2))),** that is, of loading into R1 the value in the memory location stored in the memory location obtained by adding **100** to the contents of register **R2.**

- Finally, we allow an **immediate constant addressing mode**. The constant is prefixed by #. The instruction ***LD Rl, #100***loads the integer **100** into register R1, and ***ADD R1, R1, #100***adds the integer **100** into register R1. '

Comments at the end of instructions are preceded by / /.

Example 1: The three-address statement x = y - z can be implemented bythe machine instructions:

```
LD R1, y            // R1 = y
LD R2, z            // R2 = z
SUB R1, R1, R2      // R1 = R1 - R2
ST x, R1            / / x = R1
```

One of the goals of a good code-generation algorithmis to avoid using all four of these instructions, whenever possible. For example,y and/or zmay have been computed in a register, and if so we can avoid the **LD** step(s). Likewise, we might be able to avoid ever storing x if its value is usedwithin the register set and is not subsequently needed.

Example 2: Suppose 'a' is an array whose elements are 8-byte values, perhaps real numbers. Also assume elements of a are indexed starting at 0. We may execute thethree-address instruction b = a [i] by the machine instructions:

```
LD R1, I                    // R1 = i
MUL R1, R1, 8               // R1 = R1 * 8
LD R2, a(R1)                // R2 = contents(a + contents(R1))
ST b, R2                    // b = R2
```

That is, the second step computes **8_i**, and the third step places in register **R2** the value in the ith element of a — the one found in the location that is **8_i** bytes past the base address of the array a.

Example 3*:* Similarly, the assignment into the array a represented by three-address instruction*a [j] = c*is implemented by:

```
LD R1, c            // R1 = c
LD R2, j            // R2 = j
MUL R2, R2, 8       // R2 = R2 * 8
ST a(R2), R1        // contents(a + contents(R2)) = R1
```

Example 4*:* To implement a simple pointer indirection, such as the three-address statement *x = *p*, we can use machine instructions like:

```
LD R1, p            // R1= p
LD R2, 0(R1)        // R2 = contents(0 + contents(R1))
ST x, R2            // x = R2
```

Example 5*:*The assignment through a pointer **p = y* is similarly implemented in machine code by:

```
LD R1, p            // R1= p
LD R2, y            // R2 = y
ST 0(R1), R2        // contents(0 + contents(R1)) = R2
```

Example 6*:* Finally, consider a conditional-jump three-address instruction likeif x < y goto L
The machine-code equivalent would be something like:

```
LD R1, x            // R1 = x
LD R2, y            // R2 = y
SUB R1, R1, R2      // R1 = R1 - R2
BLTZ R1, M          // if R1< 0 jump to M
```

Here, M is the label that represents the first machine instruction generated from the three-address instruction that has label L. As for any three-address instruction,we hope that we can save some of these machine instructions because the needed operands are already in registers or because the result need never be stored.

4.8.2 Program and Instruction Costs

For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands. This cost corresponds to the length in words of the instruction. Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction.Some examples:

- The instruction ***LD R0, R1*** copies the contents of register R1 into register R0. This **instruction has a cost of one** because no additional memory words are required.
- The instruction ***LD R0, M*** loads the contents of memory location M into register R0. The **cost is two** since the address of memory location M is in the word following the instruction.
- The instruction ***LD R1, *100(R2)*** loads into register R1 the value given by *contents(contents(100 + contents(R2))).* The **cost is three** because the constant 100 is stored in the word following the instruction.

Good code-generation algorithms seek to ***minimize the sum of the costs of the instructions*** executed by the generated target programon typical inputs.

4.9 Addresses in the Target Code

Executing program runs in its own logical address space that was partitioned into four code and data areas:

1. A statically determined area ***Code*** that holds the executable target code. The size of the target code can be determined at compile time.
2. A statically determined data area ***Static*** for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.
3. A dynamically managed area ***Heap*** for holding data objects that are allocated and freed during program execution. The size of the *Heap* cannot be determined at compile time.
4. A dynamically managed area ***Stack*** for holding activation records as they are created and destroyed during procedure calls and returns. Like the *Heap,* the size of the *Stack* cannot be determined at compile time.

4.10. Basic Blocks and Flow Graphs

The representation is constructed as follows:

1. Partition the intermediate code into ***basic blocks,*** which are maximal sequences of consecutive three-address instructions with the properties that
   a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
   b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.
2. The basic blocks become the nodes of a ***flow graph,*** whose edges indicate which blocks can follow which other blocks.

4.10.1 Basic Blocks

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

The procedure for partition a sequence of three-address instructions into basic blocks is explained as follows. We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction. In the absence of jumps and labels, control proceeds sequentially from one instruction to the next.

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.
   c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

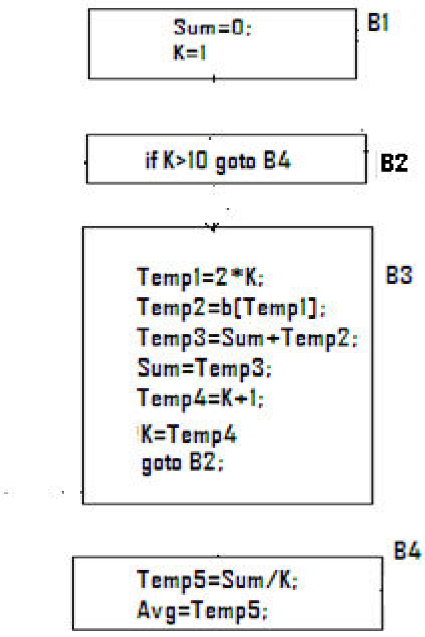
Example:

```
Sum=0;
K=1;
While K<=10 do
{
Sum=Sum+b[2*K];
K=K+1;
}
Avg=Sum/K;
```

The sequence of three address statements for the above C code is as follows:

1. Sum=0;
2. K=1;
3. If K>10 goto 11
4. Temp1=2*K;
5. Temp2=b[Temp1];
6. Temp3=Sum+Temp2;
7. Sum=Temp3;
8. Temp4=K+1;
9. K=Temp4;
10. goto 3;
11. Temp5=Sum/K;
12. Avg=Temp5;

The three address statement for the above code can be partitioned into four basic blocks B1, B2, B3 and B4. Block B1 consists of first two instructions.B2 consists of single statements. Block B3 consists of instructions numbered from 4 to 10 and Block B4 consists of instructions numbered from 11 to 12.



4.10.2. Flow Graph or Control Flow graph (CFG)

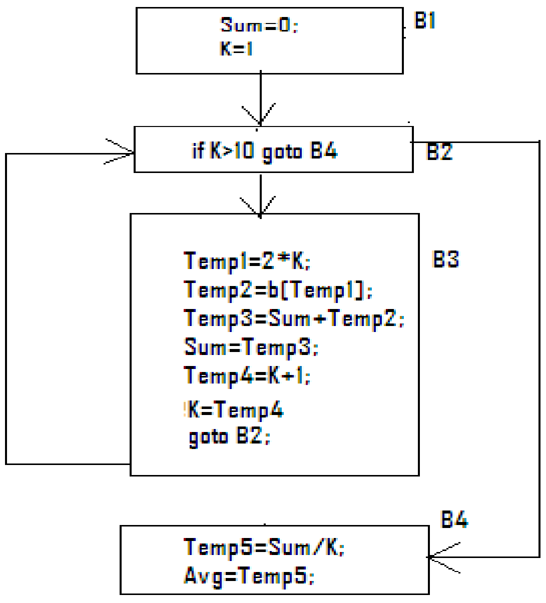
Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a control flow graph or flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B. There are two ways that such an edge could be justified:

1. There is a conditional or unconditional jump from the end of B to the beginning of C.
2. C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

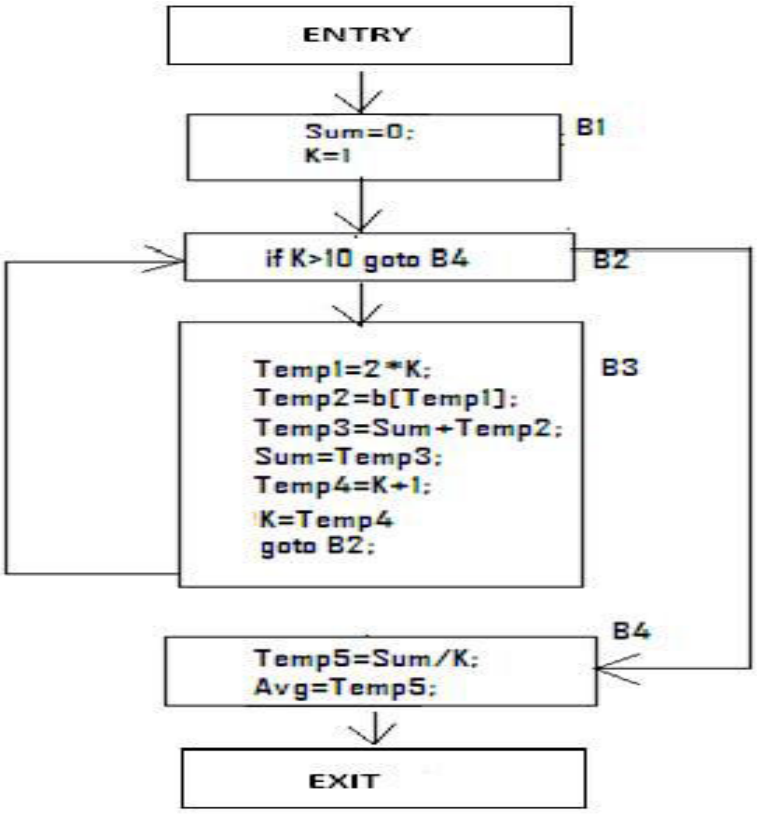
In any of the above cases B is a predecessor of C, and C is a successor of B.

Two nodes are added to the flow graph, called the entry (source) node and exit (sink) node. There is an edge from the entry to the first executable node of the flow graph, that is, to the basic block that comes from the first instruction of the intermediate code. There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program.

The Flow of control between Basic Blocks can be specified as follows.



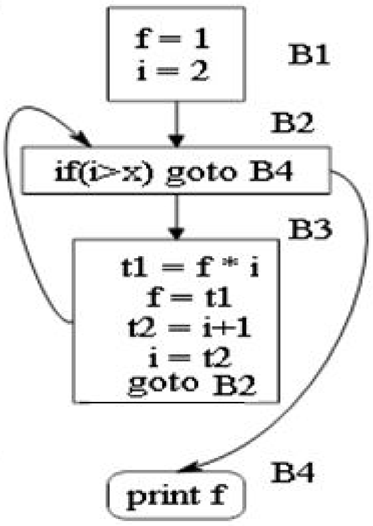
After adding Entry and Exit node to the above fig.



Example 2: Basic Block and control flow graph for a factorial Program is shown below.

```
int fact( x )
{
  int f = 1;
  i = 2
  while( i <= x )
  {
     f = f * i ;
     i = i + 1;
  }
  Print(f);
}
```

```
1)  f = 1
2)  i = 2
3)  if( i > x ) goto 9
4)  t1 = f * i
5)  f = t1
6)  t2 = i + 1
7)  i = t2
8)  goto 3
9)  print(f)
```



4.11. Register Allocation and Assignment

Instructions involving only register operands are faster than those involving memory operands. Therefore, efficient utilization of registers is vitally important in generating good code.

One approach to register allocation and assignment is to assign specific values in the target program to certain registers. For example, assign base addresses to one group of registers, arithmetic computations to another, the top of the stack to a fixed register, and so on. This approach has the advantage that it simplifies the design of a code generator. Its disadvantage is that, applied too strictly, it uses registers inefficiently; certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated into the other registers. Nevertheless, it is reasonable in most computing environments to reserve a few registers for base registers, stack pointers, and allow the remaining registers to be used by the code generator as it sees fit. The various techniques for register allocation are

4.11.1. Global Register Allocation: - The code generation algorithm used registers to hold values for the duration of a single basic block. However, all live variables were stored at the end of each block. To save some of these stores and corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally). Since programs spend most of their time in inner loops, a natural approach to global register assignment is to try to keep a frequently used value in a fixed register throughout a loop. One strategy for global register allocation is to assign some fixed number of registers to hold the most active values in each inner loop. The selected values may be different in different loops. Registers not already allocated may be used to hold values local to one block as in Section. This approach has the drawback that the fixed number of registers is not always the right number to make available for global register allocation.

Usage Counts: -In this technique we consider the savings obtained by keeping a variable x in a register for the duration of a loop L. However, if we use the approach to generate code for a block, there is a good chance that after x has been computed in a block it will remain in a register if there are subsequent uses of x in that block. Thus we count a savings of one for each use of x in loop L by using the

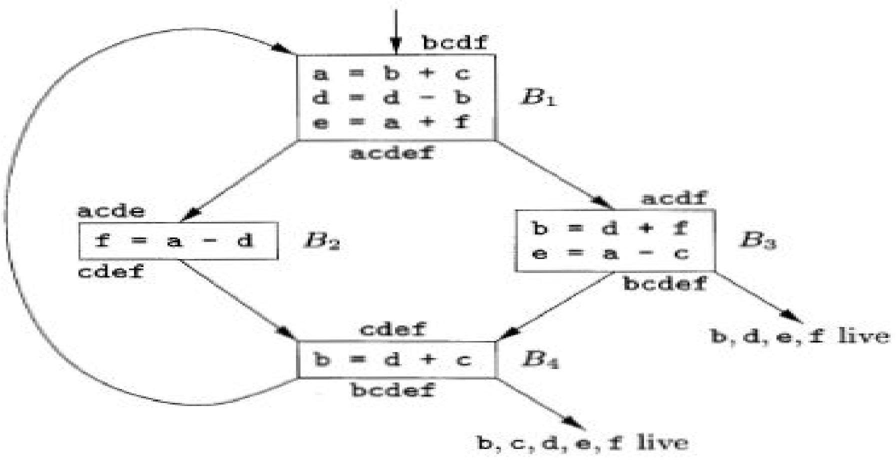
$$\sum_{B \text{ in } L} \text{use}(x, B) + 2 * \text{live}(x, B)$$

Where

- **use(x, B)** is the number of times x is used in B prior to definition of x in the same block.
- **live(x, B)** is 1 if x is live on exit from B and is assigned a value in B, and live(x, B) is 0 otherwise.

The variables whose usage count is more are stored in the global registers as they are frequently required during the processing of the inner loop.

Example: Consider the basic blocks in the inner loop as shown in figure and calculate the usage counts of each variable and show what variables are stored in global registers.



Assume registers R0, R1, and R2 are allocated to hold values throughout the loop. Variables live on entry into and on exit from each block are shown in Fig.

To evaluate usage count for x = a, we observe that a is live on exit from B1 and Used in B2& B3. Thus

Usage count for a = use in B2+ use in B3 + 2*live from B1

= 4

Usage count for b = use in B1 +2*live in B4+2*live in b3

= 6

Usage count for c = use in B1+ use in B3 + use in B4

= 3

Usage count for d = use in B1+use in B2+ use in B3 + use in B4 +2*live from B1

= 6

Usage count for e = 2*live from B1+2*live from B3

= 4

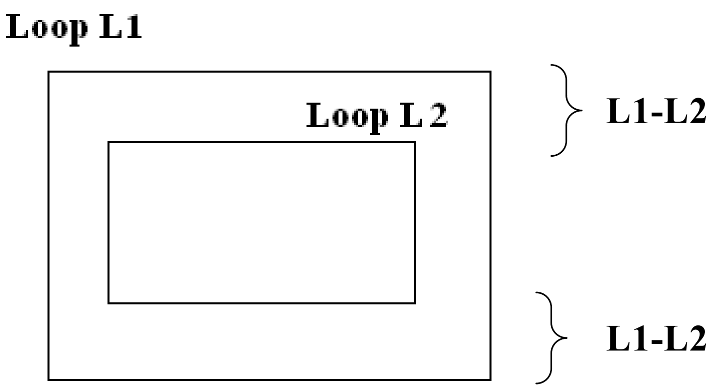
Usage count for f= use in B1 + use in B2 +2*live from B2

= 4

Thus, we may select a, b, and d for registers R0, R1, and R2, respectively. Using R0for e or f instead of ‘a’ would be another choice with the same apparent benefit.

4.11.2. Register Allocation for outer loops

Having assigned registers and generated code for inner loops, we may apply the same idea to progressively larger enclosing loops.



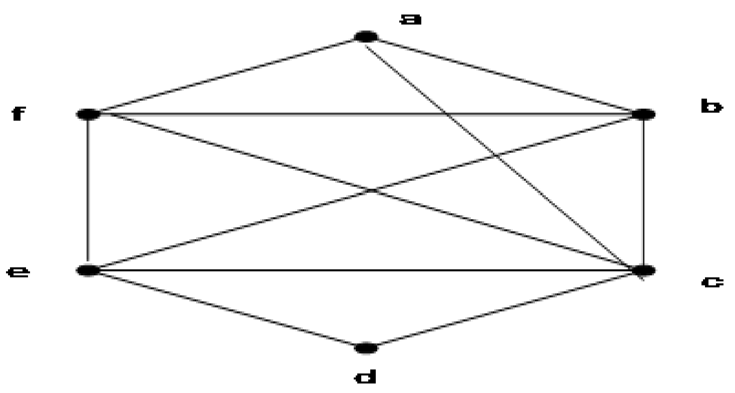
If an outer loop L1 contains an inner loop L2 the register allocation is as follows. If the variable x has allocated register in L2 need not be allocated registers in L1 - L2. If we allocate x a register in L2 but not L1, we must load x on entrance to L2 and store x on exit from L2.

4.11.3. Register Allocation by Graph Coloring

Graph coloring is a simple, systematic technique for allocating registers and managing register spills.

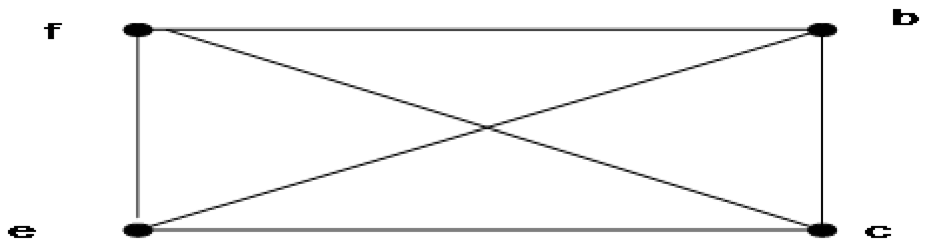
When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored (spilled) into a memory location in order to free up a register.

In this method, two passes are used. In the first, target-machine instructions are selected as though there are an infinite number of symbolic registers. Once the instructions have been selected, a second pass assigns physical registers to symbolic ones. The goal is to find an assignment that minimizes the cost of spills. In the second pass, a register-interference graph (RIG) is constructed. In RIG there is a node for each temporary and there is an edge between any two temporaries if they are live simultaneously at some point in the program. Two temporaries can be allocated to the same register if there is no edge connecting them b and c cannot be in the same register but b and d can be in the same register.

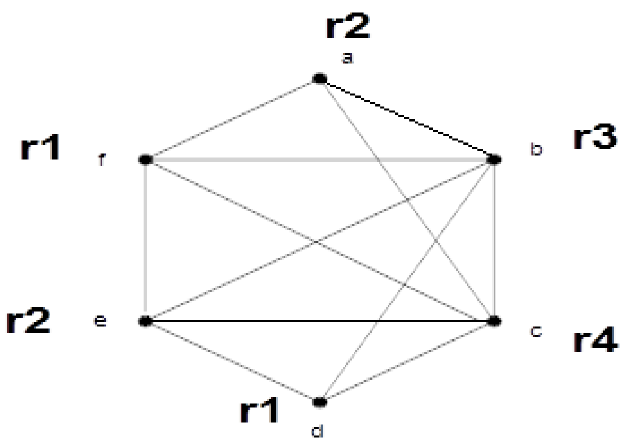


A coloring of a graph is an assignment of colors to nodes, such that nodes connected by an edge have different colors. Pick a node t with fewer than k neighbors in RIG. Eliminate t and its edges from RIG. If the resulting graph has a k-coloring then so does the original graph.

1. Start with the RIG and with k = 4. Initially stack is empty. S= { }
2. Remove 'a' and then 'd' as they have less than four neighbors.
3. Now the stack contains S= {d, a} and the modified RIG is as follows



4. Now all the nodes have less than four neighbors. So remove all of them and add to the stack. Thus **S= {f, e, c, b, d, a}.**
5. Start assigning colors to: f, e, b, c, d, a


r2
a
r1
f
b
r3
r2
e
c
r4
r1
d