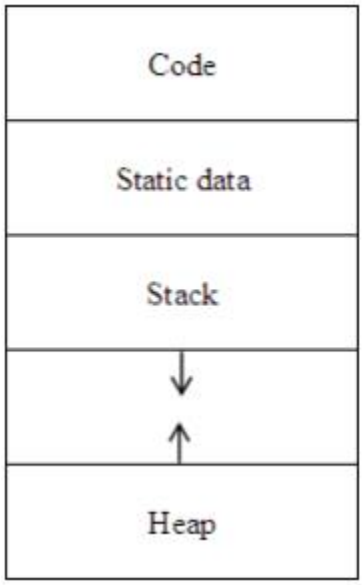
STORAGE ORGANIZATION

- 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 address space consists of data and program areas.



- The run-time storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory. Multi byte objects are stored in consecutive bytes and given the address of the first byte.
- The amount of storage needed for a name is determined from its data type, such as basic type character, integer, float or aggregate type like array, structure.
- The storage layout for data objects depends on the addressing constraints of the target machine. On many machines, instructions to add integers may expect integers to be aligned, that is, placed at an address divisible by 4.
- Although a character array (as in C) of length 10 needs only 10 bytes to hold ten characters, a compiler may allocate 12 bytes to get the proper alignment, leaving 2 bytes unused. Space left unused due to alignment considerations is referred to as padding.
- **CODE AREA**: The size of the generated target code is fixed at compile time, so the compiler places the executable target code in a statically determined area, the low end of memory.
- **STATIC AREA**: 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. In early versions of Fortran, all data objects could be allocated statically.
- **STACK AREA**: Used to store data structures called activation records that get generated during procedure calls.

- **HEAP AREA**: Many programming languages allow the programmer to allocate and deallocate data under program control. For example, C has the functions malloc and free that can be used to obtain and give back arbitrary chunks of storage. The heap is used to manage this kind of long-lived data.

Stack Allocation Space:

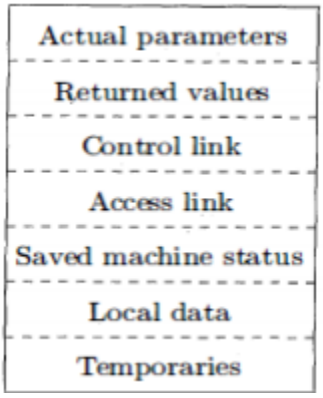
- Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space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

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 of Procedure calls.

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.



1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.
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.
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, these Values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

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").
- In general, if a procedure is called from n different points, then the portion of the calling sequence assigned to the caller is generated n times. However, the portion assigned to the callee is generated only once.