Runtime Environment

Introduction

In the previous chapters the phases of the compilers to the scanning, parsing and intermediate code generation are studied. The study was completely independent of target language. This chapter deals with runtime environment which concentrates on target computers memory structure and maintenance of memory. There are manly three types of environments – Static Environment used in FORTRAN 77, Stack based environment used in languages like Pascal, C and C++. Fully dynamic environment used in languages like LISP. There can be hybrid environment also. This chapter explains about the relationship between language features and the environment. The environment takes care of properties like scoping, procedure calls and parameter passing mechanisms. The allocation and de-allocation of memory [data] objects is managed by the runtime support package which consists of routines loaded with the generated target code.

Memory organization during program execution

Memory for the program execution is broadly divided into two areas, one for storing user data called data area and other for storing program called program area. Normally the contents of program area do not change during the execution of the program. Data area stores the global or static constants or literals.

Example:

Printf (" The solution is = % d", 426);

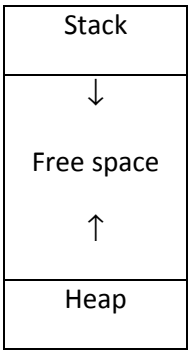
In the above example the value 426 is constant and the string "The solution is =" are to be stored in global area. Other than global variables, there will be local variables whose value changes during the execution, these are to be stored in area local of the particular function. For this purpose stacks are used. The runtime memory is divided into following parts.

a. Code area to store target code

b. Static data area – to store global variables or literals

c. Stack area – to store activation record during procedure calls and return. Stack Operates in LIFO fashion [Last In First Out]

d. Heap – This is used for dynamic memory allocation.



Stack and heap may have separate memory blocks or they may share the same memory area.

Activation Record

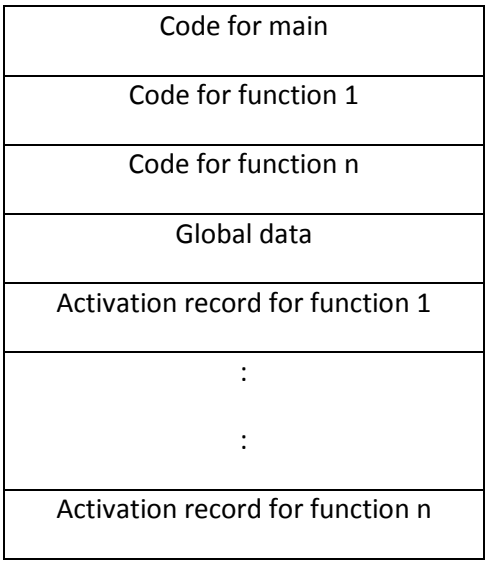
Important unit of memory allocation is for activation record for function/procedure call. The execution of function is referred to as activation of function. Function consists of function name, formal and actual parameter, body of function and return value. When the function is called, the activation record for that function is created and it is stored on stack. The components of activation record are

- **Return value** - Return value is used to store the value that the function returns to called function after its execution.
- **Actual parameters** - Actual parameters are those which are used for sending input to functions from caller function.
- **Optional control link** - This points to the activation record of the caller. This is very useful in case of recursion.
- **Optional access link** - This is used to access non-local data, it can point to caller data area to access global data.
- **Machine status** - Machine status consists of values of program counter, machine registers etc.,
- **Local data** - stores the local data of the called function
- **Temporaries** - used to store the intermediate results during execution of large expressions.

The runtime environment determines the sequence of operations that must be performed when a function is called. Some operations must be performed during returning, this is called as call sequence and return sequence respectively. Caller is responsible for computing arguments and placing them in location [activation record] during call sequence. Callee has to take care of control and access links along with temporaries and local data. Any additional book keeping information may be done either by callee or caller

Fully Static Runtime Environment

In static runtime environment, the data is stored in fixed memory location during the execution of program. The language that uses static environment will not have the concept of pointer variables and no dynamic memory allocation. They also can not have recursive functions. All the variables are allocated statically and the location of activation record is also fixed before execution. If does not require any runtime support as the names are bound to storage at compile time.



Memory organization for static environment is shown in above figure. This consists of code for main function followed by code for function 1 to function in code area. This is followed by global data and activation record for function 1 to function n.

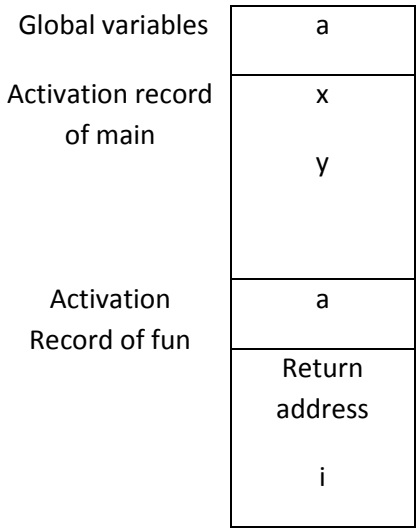
When a function is called, each argument is stored in activation record. And the return address is saved. Then a jump to first instruction of caller function is made. On return, a simple jump is made to the return address.

Example:

```
int a = 10,

main ( )

{    int x, y ;

     y = a;

     x = fun (g);

     print f (“x = %d”, x),

}
```

```
int fun (int a)

{

    int i;

    i = a + 10;

    return i;


}
```

The memory allocation for the above program is as follows



Stack based runtime environment

The machine that supports for stack allocation for runtime management stores the activation for procedure call in stack rather than static location. This kind of allocation is efficient with the languages that support recursion. Every procedure may have different activation records on the call stack at any time. New activation record is pushed (stored) on top of the stack for every function call and popped when function returns.

Stack based environment requires three pointer

a. Pointer to current activation record to access local variables. This pointer is called as current activation pointer (CAP)

b. Pointer to previous (caller) activation record (control link)

c. Stack pointer (SP) which pointes to top of the stack.

Example:

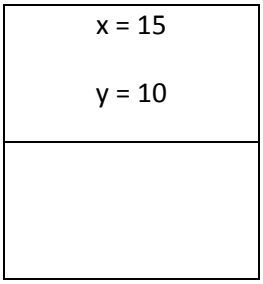
Consider simple recursive function to compute GCD of 2 positive integers

```
 int x, y

int gcd (int a, int b)

{

if (b = = 0)

      return a;

else

      return gcd (b, a% b) ;

}

main ( )

{     scanf (“% d % d”, & x, & y);

      printf (“ GCD of % d and % d = % d”, x, y, gcd(x,y));

      return 0;

}
```

If the input are 15 and 10, then x = 15 and y = 10, main calls gcd (15, 10)

The contents of stack are as in Fig 8.1.

Each time a gcd( ) is called activation record for gcd is pushed on to stack, The control link of ith activation record points to the return address of (i – 1)th activation record. Once the function completes its execution. The activation records are popped from stack.



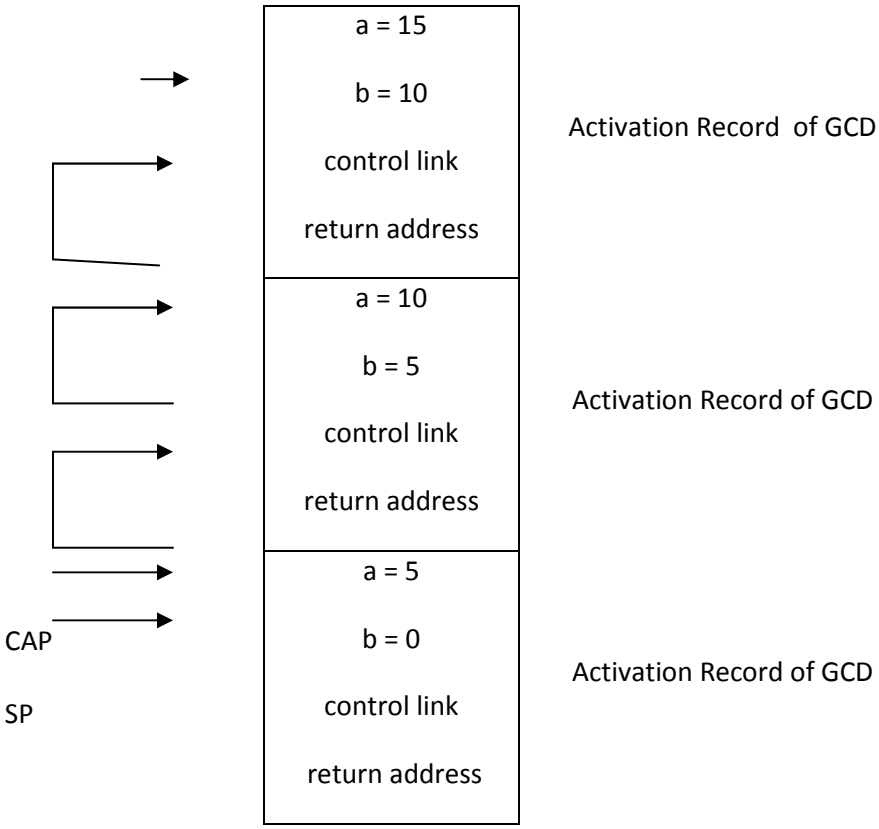


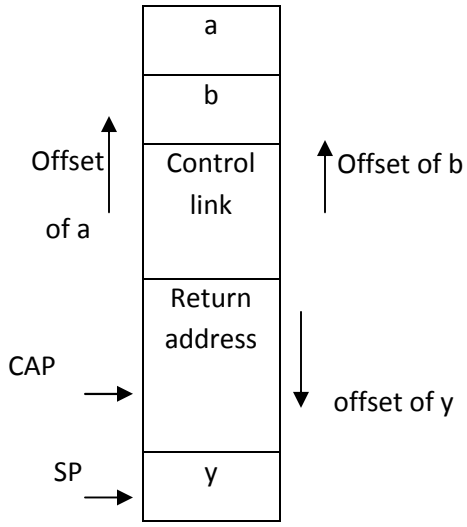
Fig 7.1 stack contents on recursive function call

Access to names

Parameters and local variables can be accessed by the offset from the starting point of activation record. As the declaration of a function is fixed at compile time and the memory size to be allocated for each declaration is fixed by its data type, the offset can be statically computed.

Example: Consider the C function

```
void fun (int a. char b)
{
    double y;
    …..
}
```



Assume two bytes for integer, one four character, eight for double precision floating point number and four byes for address. Assume that the stack grows from higher to lower address.

The following offset exists

Variable	Offset	
a	+ 5	Control link is address hence 4 bytes + 1byte for char b
b	+ 4	
y	– 4	

Non local and static names have fixed location and can be accessed directly. This mechanism is called as static scope.

Variable length data

Sometimes compilers are required to deal with variable length data. In cases where number of data objects for function call may vary and size of each object may also change.

Example: printf (" Hello");

This has only one argument

Printf ("%s%d%f", a,b,c); has three arguments.

The number of arguments to printf is defined by format string. Number of arguments may vary from call to call. C Compiler pushes the arguments in the reverse order onto stack. The first parameter is always located at a fixed offset from CAP.

Local Temporaries

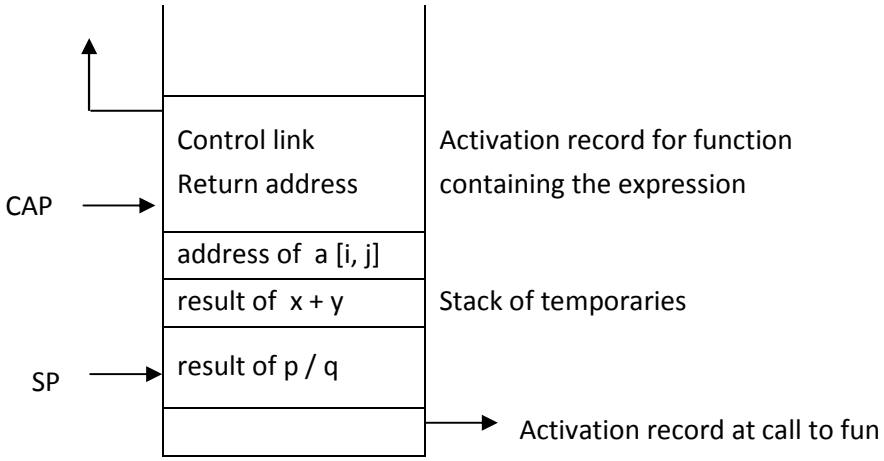
The stack based environment should take care of storing intermediate results during function calls. Consider the following example

a [i, j] = [x + y] * (p / q) * fun (z)

There will be three partial results

i) x + y ii) p / q iii) a [i, j]

These results could be stored in stack before function call fun or stored in registers. If stored on sack, then the stack will be as follows



Compiler can easily locate stack top from CAP.

Nested declaration

Consider the function declaration which has nested blocks.

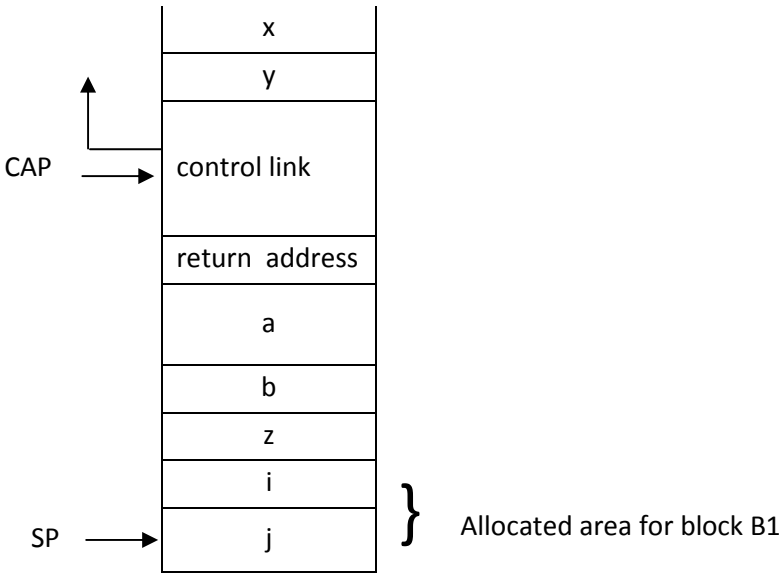
```
int fun (int x; int y)
{    char a;
     double b;
     int z;
  B1:   {      int  i ;
               float j;
               - - - -
```

```
        }
        - - - - -
B2:     {       int  p;
                double q;
                - - - -
        }
        - - - - - -
        return z ;
}
```

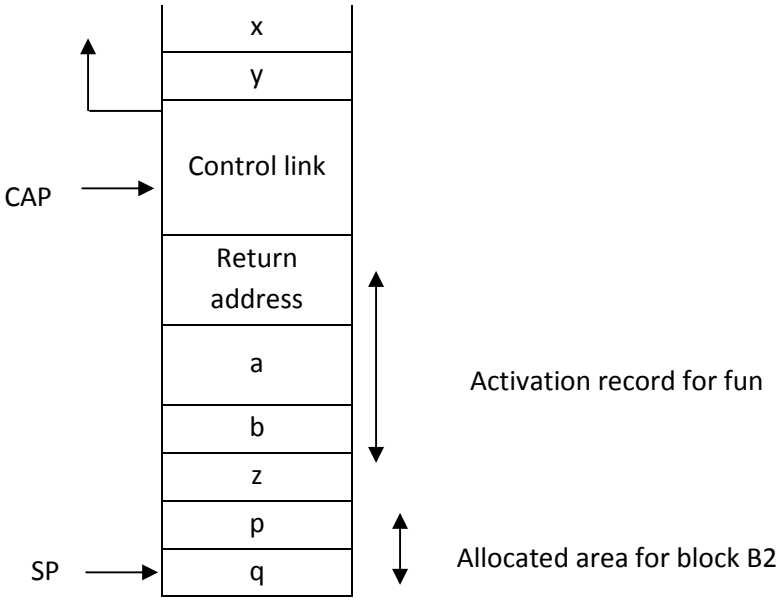
Function fun has two blocks B1 & B2 variables of each blocks are local, with in the blocks & their values do not exists outside the blocks.

One way of implementing this is treating each block as functions & creating Activation record, when entered the block and remove Activation record on exit. This is not efficient as blocks do not have parameters and return value. The simpler way of implementing this is allocating stack on entering and de-allocating on exit.

Example: The stack for above example is as follows.



As B1 and B2 do not exists at the same time the area in stack can be reused as follows.



Such implementation helps in computing location of variables with respect to offset from CAP during compile time.

Dynamic Memory

Though stack based runtime environment is efficient when compared to static environment it has the problem of dangling reference.

Example: Consider the following C code.

```
main ( )

{     int * a;

      P; dangle ( );

}

int. *dangle ( )

{     int. i = 20;

      return &i;

}
```

a is a dangling reference because i cannot be accessed after the activation record of dangle is freed. The stack cannot be used if the values of locals to be retained even after the activation ends. Hence stack based environment is not efficient for general environment. The next alternative environment is fully

dynamic environment. In this case the deallocation of activation record is done at arbitrary times during execution. The full of dynamic environment is very complex compared to stack based system as it has to take care of keeping references and deallocating unwanted areas of memory at arbitrary time. This concept is called as garbage collection. Basic structure of activation record remains same, i.e., allocating space during procedure calls for parameters, local variables, control link and access links, the only difference is deallocation of space at the later time.

Dynamic Memory in Object Oriented Language

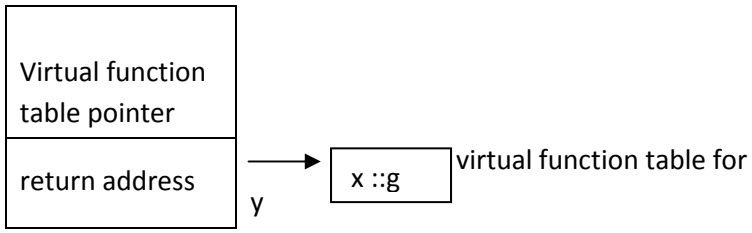
Object oriented language supports for objects, methods, inheritance and dynamic binding an object in memory is combination of record and an activation record with instance variables as fields of record. One way of implementing this is virtual function table. This table consists of list of pointers to methods of each class. Virtual table has advantage in computing offset, as each object points to virtual table than class structure.

Example:

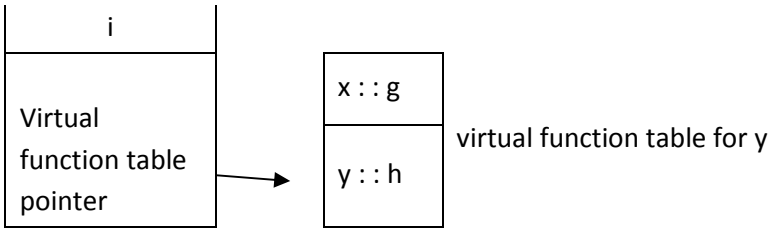
```
Class x

{     public

       Int. a , b;

       void f1 ( ) ;

        virtual void g ( ) ;

    };

Class y ; public x

{     public

      int. i ;

      void f1 ( );

      virtual void h ( );

};
```

Object of class x in memory is represented as follows

a
b



Similarly for Class y



Heap management

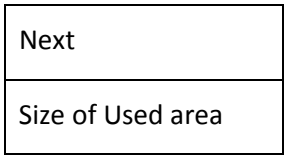
Heap is a linear block of memory which is used to handle pointer allocation and deallocation. Heap performs two operations, allocate and free. Allocate operation takes input as size in bytes and returns pointer to block of memory of defined size. If no memory exists, it returns null pointer. Free operation is used to free the allocated block. Pascal uses new and dispose, where as C+ uses new and delete for allocate and free operation respectively. C language uses malloc and free as a part of standard library stdlib.h for allocation and dellocating memory. The prototype of these functions are as follows

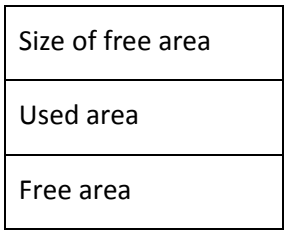
void *malloc (unsigned size);

void free (void *ptr);

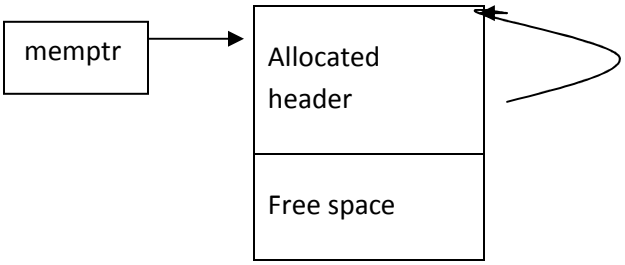
One way of implementing heap is maintaining a circular list of free blocks, from which memory can be drawn through malloc function and returned through free function. Though this is very simple to implement and maintain, it has few disadvantages. One of the disadvantage is that, the pointer to free block may not be the one given to malloc. Possibility of user giving invalid pointer corrupts the heap. Secondly there can be small fragments of free blocks. This has to be compacted so that large blocks of continuous memory are available for malloc.

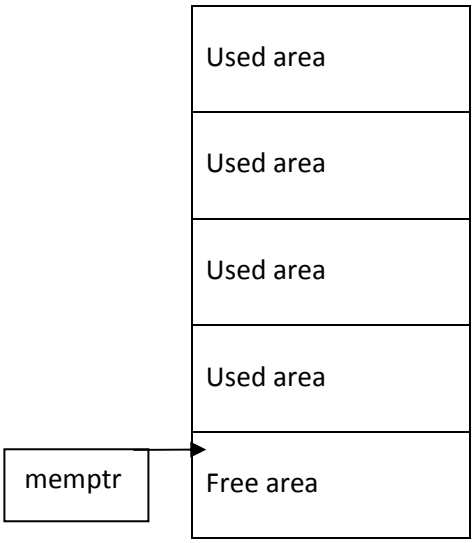
More efficient way of heap implementation is using circular linked list which keeps track of both allocated and free blocks. Heap consists of nodes (blocks) which has information of size of used area and size of free area followed by user space and free space as shown below.





It also has next pointer which points to next block in heap memory. Heap also uses one more pointer called memptr this points to a block that has some free space. This free space will always be initialized to null value.





Automated management of Heap

malloc and free are explicitly called in the program for dynamic management of memory. In case of run time stack the memory management should be automatically done by the calling sequence. Fully dynamic runtime environment automatically reclaim previous allocated blocks which are not used further without explicit free call. This process is called as garbage collection. Garbage collection can be achieved in any of the following methods

- mark and sweep

- stop and copy
- generational garbage collection

Mark and sweep: In this method no memory is freed until malloc fails for insufficient memory. At this point, the mark process marks the memory blocks whose values are not used any more. In the sweep process the marked memory blocks are cleared and put into free list. Some time memory compaction may be required in order to get large free block.

Stop and copy: In this method, the memory is divided into two halves and allocating storage only from one half at a time. During the marking process all the updated blocks (the blocks whose values are changed) are stored in second half. If performs memory compaction automatically. Once all blocks in the used area have copied, the used and unused halves of memory are interchanged and the processing continues.

Generational garbage collection: The aim of this method is to reduce the delay. In order to do this, the allocated objects that survive for long time are copied onto permanent space and are not deallocated during reclaimation. This reduces the search space for newer storage and hence reducing the time for searching.