CMPSC 160
Translation of Programming Languages

Lecture 12: Address Translation and Memory Allocation

Address of Variables

How does the compiler represent memory location for a specific instance of variable *x* for a procedure?

- Name is translated into a ***static coordinate**: < level, offset >*
  - *"level"* is lexical scoping level
  - *"offset"* is *unique* within that scope
  - *"offset"* is assigned at compile time and it is used to generate code that executes at run-time
- Static distance coordinate is used to generate addresses
  - For each lexical scope *level* we have to generate a *base address*
  - *offset* gives the location of a variable relative to that base address

Memory Allocation

$P \rightarrow D$
$D \rightarrow D; D$
$D \rightarrow id : T$
$T \rightarrow char \mid int \mid float \mid array[num]\ of\ T \mid pointer\ T$

Attributes:	T.type, T.width
Basic types:	char width 4, integer width 4, float width 8
Type constructors:	array(size,type) width is size * (width of type)
	pointer(type) width is 4

- Enter the variables to the symbol table with their type and memory location:
  enter(name, type, location)
- Set the type attribute T.type and calculate the width (T.width) for each type
- Layout the storage for variables
  - Calculate the offset for each local variable and enter it to the symbol table
  - Offset can be offset from a static data area or from the beginning of the local data area in the activation record

Translation Scheme for Memory Allocation

$P \rightarrow$ {offset $\leftarrow$ 0;} D
$D \rightarrow$ D; D
$D \rightarrow$ id : T {enter(id.name, T.type, offset); offset $\leftarrow$ offset + T.width; }
$T \rightarrow$ char { T.type $\leftarrow$ char; T.width $\leftarrow$ 4; }
| int { T.type $\leftarrow$ integer; T.width $\leftarrow$ 4; }
| float { T.type $\leftarrow$ float; T.width $\leftarrow$ 8; }
| array[num] of T_1 { T.type $\leftarrow$ array(num.val, T_1.type);
T.width $\leftarrow$ num.val * T_1.width; }
| pointer T { T.type $\leftarrow$ pointer(T_1.type); T.width $\leftarrow$ 4; }

- Note that if the size of the array is not a constant we cannot compute its width at compile time
- In that case, allocate the memory for the array in the heap at runtime, and allocate the memory for the pointer to the heap at compile time

Question Time ☺

```c
// structure A
typedef struct structa_tag
{
    char        c;
    short int   s;
} structa_t;
```

```c
// structure B
typedef struct structb_tag
{
    short int   s;
    char        c;
    int         i;
} structb_t;
```

```c
// structure C
typedef struct structc_tag
{
    char        c;
    double      d;
    int         s;
} structc_t;
```

```c
// structure D
typedef struct structd_tag
{
    double      d;
    int         s;
    char        c;
} structd_t;
```

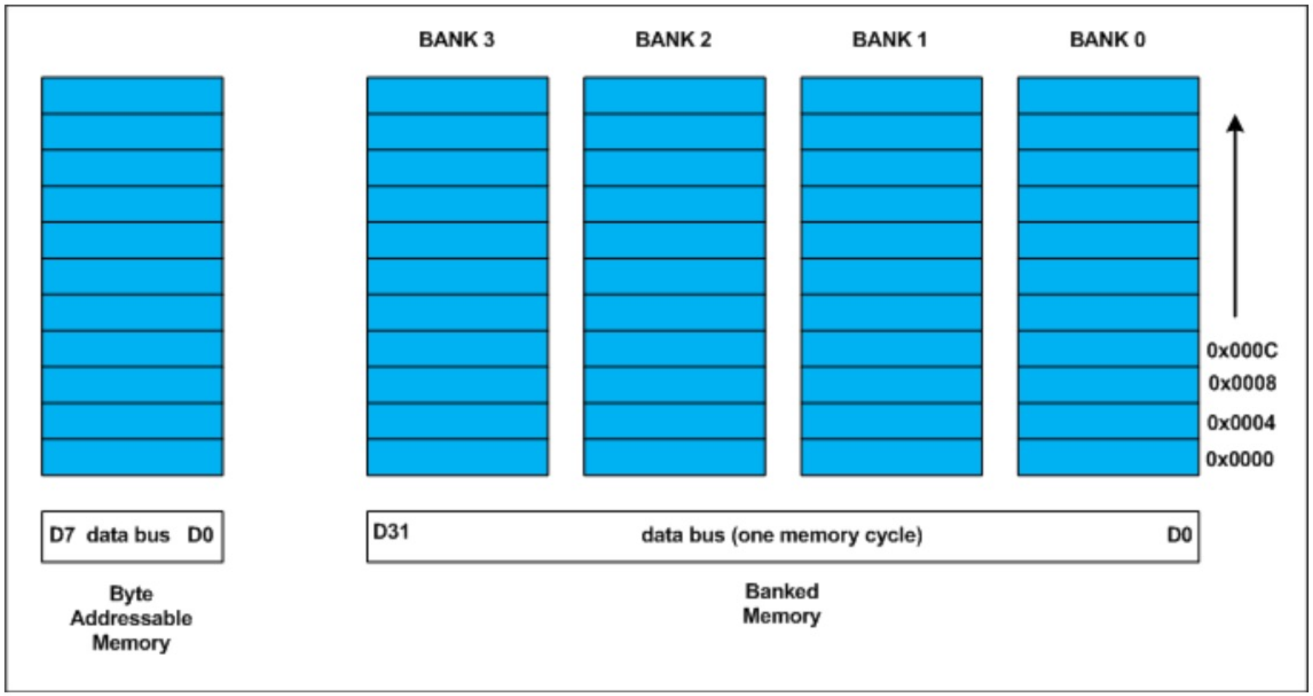
```c
int main()
{
    printf("sizeof(structa_t) = %lu\n", sizeof(structa_t));
    printf("sizeof(structb_t) = %lu\n", sizeof(structb_t));
    printf("sizeof(structc_t) = %lu\n", sizeof(structc_t));
    printf("sizeof(structd_t) = %lu\n", sizeof(structd_t));

    return 0;
}
```

What are the results?

Memory Alignment and Padding

- The storage layout for data objects is strongly influenced by 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.

Question Time ☺

```c
// structure A
typedef struct structa_tag
{
    char        c;
    short int   s;
} structa_t;
```

```c
// structure B
typedef struct structb_tag
{
    short int   s;
    char        c;
    int         i;
} structb_t;
```

```c
// structure C
typedef struct structc_tag
{
    char        c;
    double      d;
    int         s;
} structc_t;
```

```c
// structure D
typedef struct structd_tag
{
    double      d;
    int         s;
    char        c;
} structd_t;
```

```c
int main()
{
    printf("sizeof(structa_t) = %lu\n", sizeof(structa_t));
    printf("sizeof(structb_t) = %lu\n", sizeof(structb_t));
    printf("sizeof(structc_t) = %lu\n", sizeof(structc_t));
    printf("sizeof(structd_t) = %lu\n", sizeof(structd_t));

    return 0;
}
```

What are the results?

```
sizeof(structa_t) = 4
sizeof(structb_t) = 8
sizeof(structc_t) = 24
sizeof(structd_t) = 16
```

Another Example

```
struct data_
{
    char a;       // 1 byte
    int b;        // 4 bytes
    short c;      // 2 bytes
    char d;       // 1 byte
} point[3];
```

- sizeof(struct data_) gives you 12 bytes.

```
// the actual memory layout
struct data_
{
    char a;           // 1 byte
    char _pad0[3];    // padding to put 'b' on 4-byte boundary
    int b;           // 4 bytes
    short c;         // 2 bytes
    char d;          // 1 byte
    char _pad1[1];   // padding to make sizeof(data_) multiple of 4
} point[3];
```

Questions/Challenges

- Beyond memory alignment, how to include lexical scoping information for “blocks”, “functions” into our translation scheme?
  - Shall we treat “blocks” and “functions” in the same way?

- How to treat more complicated cases?
  - function called by multiple different places with different #parameters?
  - the number times that a function being called is only known at runtime.
  - malloc() to allocate some space of memory but the size is only known at runtime.

Motivating Example

- Consider the following program to compute the factorial function:

```
int fact(int n) {
    if (n == 0)
        return 1;
    else
        return n * fact(n-1);
}
```

In the evaluation of fact(10)

- How many different variables are used?
- How many times do fact function is being called under different input parameters?

On hardware: just code and data.

- How many copies of **fact** and **n** shall we pre-allocate when generating the program? What is the address for **n** ?

No **abstraction** of procedure at al from the hardware side.

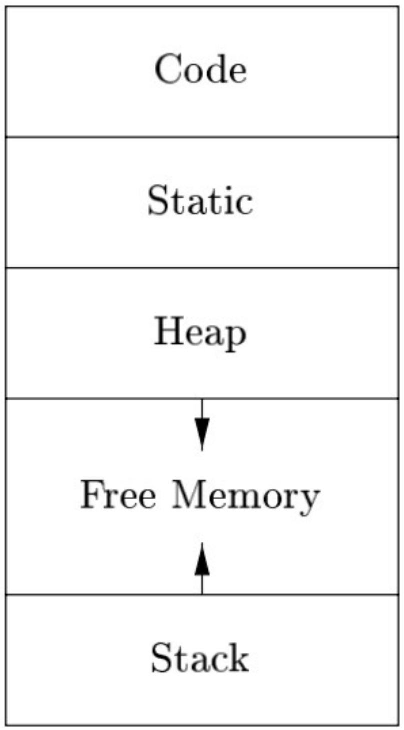
Key Topics

- Several key issues:
  - the layout and allocation of **storage locations** for the objects named in the source program
    - memory management: stack allocation, heap management, and garbage collection.
  - the mechanisms used by the target program to **access variables and data**

- Others
  - The linkage between procedures
  - the interface to the operating system, input/out device

Overall: Runtime Memory

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



- Compiler writer: the executing target program runs in its own continuous logical address space in which each program value has a location

- The Operating System then maps the logical addresses into physical addresses, which are usually spread throughout memory.

e.g., a C++ compiler on an operating system like Linux might subdivide memory in this way.

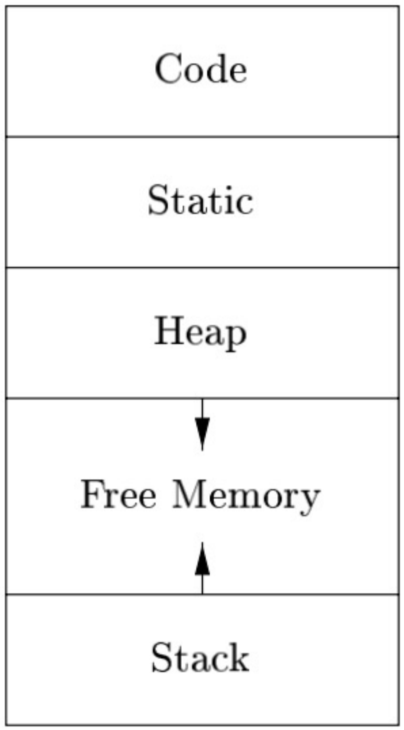
Static Storage Allocation

We say that a storage allocation decision is **static** if it be made by the compiler looking only at the text of the program.

Static allocation

- Code: 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

- Static Data: such as global constants. These data objects can be placed in another statically determined area called **Static.**
  - Benefits: the addresses of these objects can be compiled into the target code.



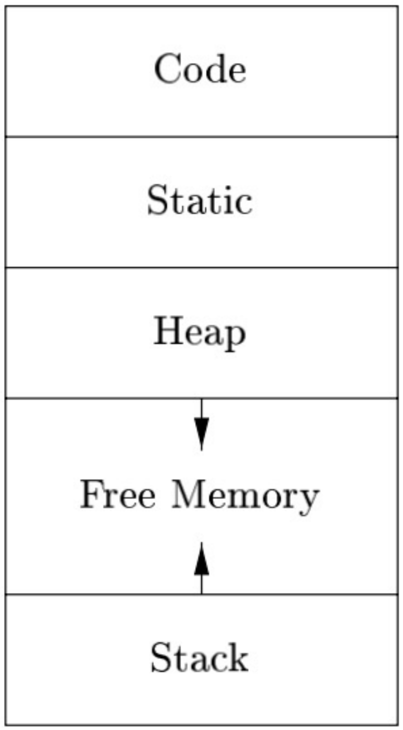
Dynamic Storage Allocation

Conversely, a decision is dynamic if it can be decided only while the program is running

Why do we need dynamic area?

- Some space we do not know the size at compile time: think of local variables for recursive function.

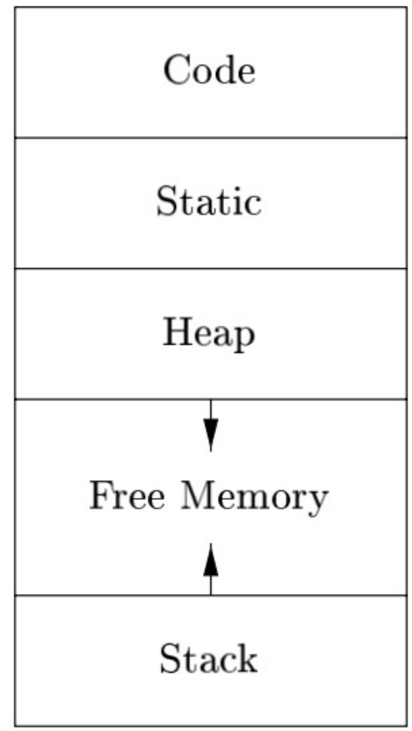
- To maximize the utilization of space at run time: space for local variables can be reclaimed for other usage.



Dynamic Storage Allocation

To dynamic space whose size can change as the program executes

- Stack
  - centering around procedures (same for functions, methods or any units of user-defined actions)
  - Dynamic (#number of copies + size) + Local
- Heap
  - Dynamic but not Local: data that may outlive the call to the procedure that created it is usually allocated on a “heap” of reusable storage
  - Garbage collection.



Examples: Stack and Heap Memory (Both at Runtime)

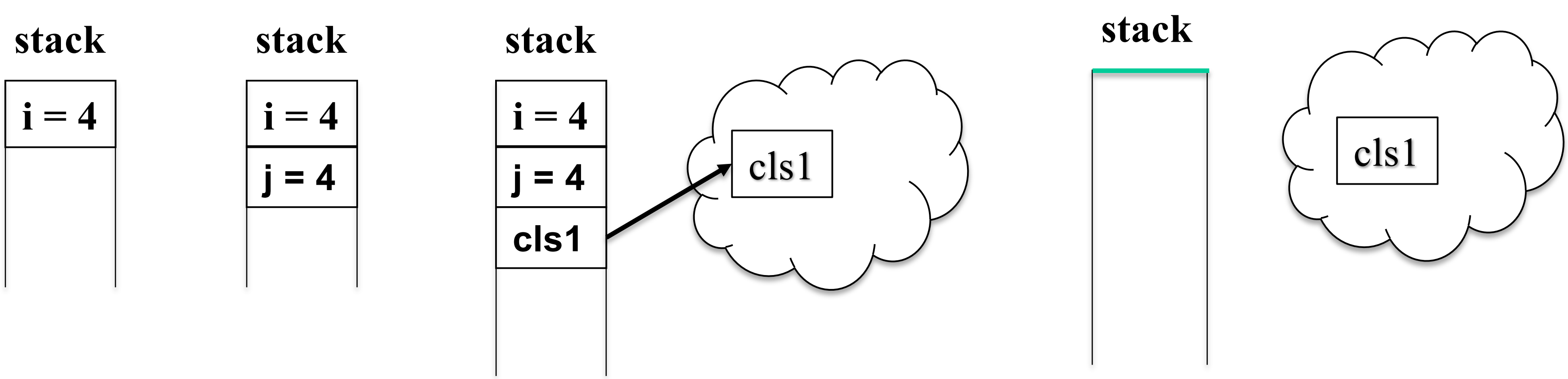
```
int main()
{
   // All these variables get memory
   // allocated on stack
   int a;
   int b[10];
   int n = 20;
   int c[n];
}
```

```
int main()
{
   // This memory for 10 integers
   // is allocated on heap.
   int *ptr  = new int[10];
}
```

Examples: Stack and Heap Memory

```
public void Method1()
{
    int i = 4;
    int j = 2;
    class cls1 = new class();
}
```

Intermixed example of both kind of memory allocation Heap and Stack in *java*.



Stack Memory for Procedures

- When a procedure is called, a block is reserved on the top of the **stack** for local variables and some bookkeeping data.
- When that procedure returns, the block becomes unused and can be used the next time a function is called.
- The stack is always reserved in a **LIFO (last in first out)** order; the most recently reserved block is always the next block to be freed.
- This makes it really simple to keep track of the stack; **freeing a block from the stack is nothing more than adjusting one pointer**.

Other Advantages of Stack Memory

- Memory: Super efficient memory reuse as this arrangement allows space to be reused by procedure calls whose durations do not overlap in time.

- Computation: It allows us to compile code for a procedure in such a way that the relative addresses of its local variables are always the same regardless of the sequence of procedure calls (use of relative addressing).

Stack Memory Management

Stack allocation would not be feasible if procedure calls or activations of procedures did not nest in time.

- Calling Sequences
- Activation Tree
- Activation Record
- Compiler-generated code for control and stack management
  - prologue + epilogue + pre-call + post-return

Nesting of Procedure Calls: Quicksort

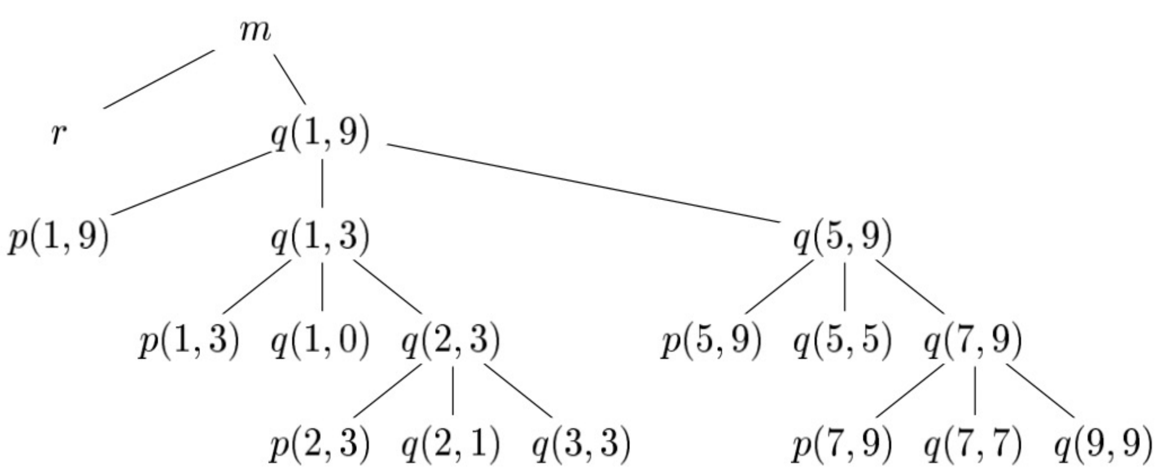
```
int a[11];
void readArray(){
//Reads 9 integers into a[1]-a[10];
...
}

int partition(int m, int n) {
// Choose a pivot value "v", and
// reorder sub-array a[m..n] such that
// a[m..p-1] are all < a[p] = v,
// and a[p+1...n] are all >= v;
    return 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] = -99999;
    a[10]= 99999;
    quicksort(1,9);
}
```

Activation Tree: Quicksort



A possible activation tree

```
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()
```

A possible execution sequences

- The sequence of procedure calls ≡ pre-order traversal of activation tree.
- The sequence of returns ≡ post-order traversal of activation tree.

Activation Tree

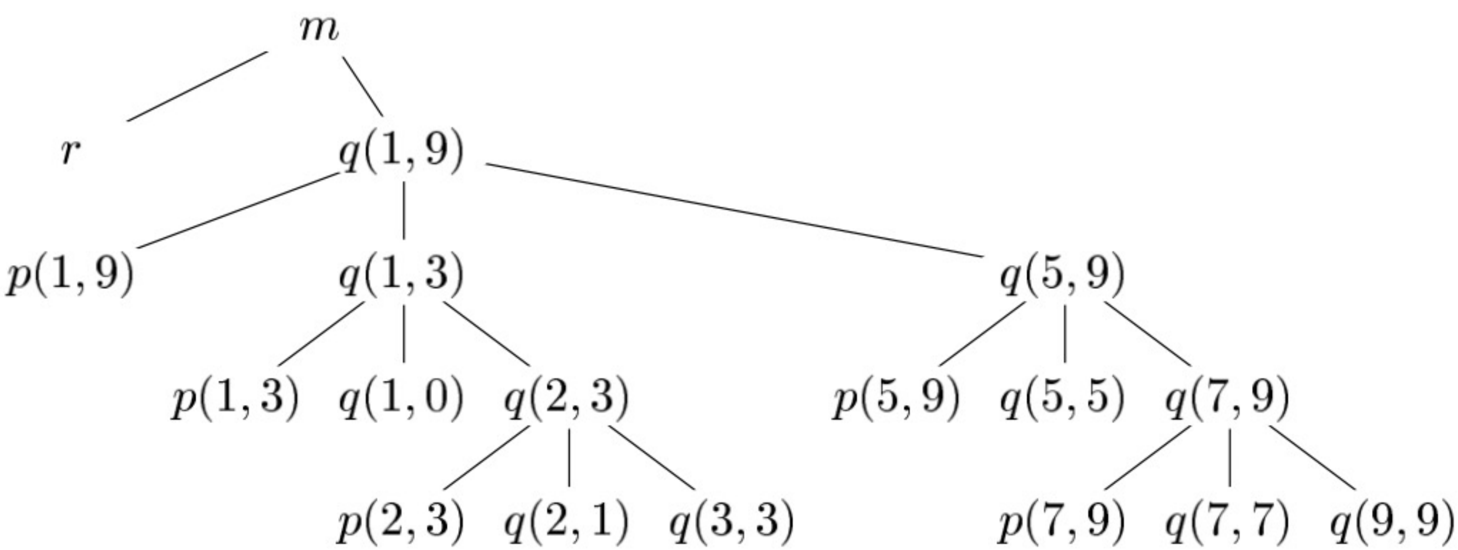
Activation Tree:

- Nodes of the tree are all procedure calls done in one execution of a program
- Root of the tree is the call to **main**
- q is a descendant of p if a call to p results in a call to q.

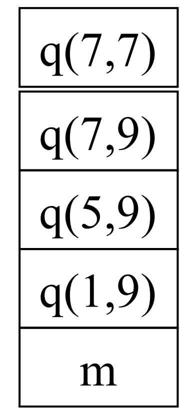
Useful relationships between the activation tree and the behavior of the program

- The sequence of procedure calls ≡ **pre-order** traversal of activation tree.
- The sequence of returns ≡ **post-order** traversal of activation tree

Activation Stack



A snapshot of the control stack at a time



Stack of current activations:

- If a procedure p has been called, but not yet returned, then ***p*** is "live" on the stack.
- The information regarding the live activations are kept on a stack, with the most recent call on top of the stack.
- If control is at a procedure ***p***, then all activations on the path from root to ***p*** of the activation tree are "live".

A Single Activation Records: What to include?

- Activation Record stores the key information that needed for a procedure.

Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

- The actual parameters used by the calling procedure.
- Space for the return value of the called function if any.
- Control link: the activation record of the caller.
- Access link: next-level of lexical scope (still remember name analysis?)
- Saved machine status: for example, the return address (the program counter) to which the called procedure must return and the contents of registers of the calling procedure.
- Temporary values such as those arising from the evaluation of expressions