Topic 3-b
Run-Time Environment

Data Layout
Dynamic Memory Allocation and Heap
Data Layout

1. Static allocation
   • Storage allocation was fixed during the entire execution of a program

2. Stack allocation
   • Space is pushed and popped on a run-time stack during program execution such as procedure calls and returns.

3. Heap allocation
   • Allow space to be allocated and freed at any time.
Static Allocation

1. Bindings between names and storage are fixed.
2. The values of local names are retained across activations of a procedure.
3. Addressing is efficient
4. Limitations:
   - No recursive procedures
   - No dynamic data structures
The amount of storage needed for a name is determined from its type.

Basic data types: char, real, integer, .. etc

Aggregates: record (struct), array, .. etc.

Address for local data objects: fixed length data is laid out as declarations are processed. The compiler tracks the number of memory locations allocated as the relative address (or offset) for local data objects.

Alignment requirements: for example an array of 10 chars may end up allocating 12 bytes.
Stack Allocation

• Recursive procedure require stack allocation.
• Activation records (AR) are pushed and popped as activations begin and end.
• The offset of each local data relative to the beginning of AR is stored in the symbol table.

```c
float f(int k)
{
    float c[10], b;
    b = c[k] * 3.14;
    return b;
}
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter k</td>
<td>offset = 0</td>
</tr>
<tr>
<td>Return value</td>
<td>offset = 4</td>
</tr>
<tr>
<td>Local c[10]</td>
<td>offset = 8</td>
</tr>
<tr>
<td>Local b</td>
<td>offset = 48</td>
</tr>
</tbody>
</table>

3.14 goes to a literal pool, not to AR
Dynamic Memory Allocation

• **Who needs the dynamic memory allocation?**
  Objects whose size cannot be determined at compile time.

• **Where to allocate**
  Stack or Heap.

• **Objects of dynamic memory allocation**
  Variable-length objects, malloc() function, etc.

• **Allocation Strategy and trade-off**
  allocate on stack if possible.
  
  The stack can be allocated only for an object if it is local to a procedure and becomes inaccessible when the procedure returns.
Dynamic Arrays Allocation

• The bounds of dynamic arrays are determined at runtime rather than compile time. For example:

```c
f(int n) {
    float a[n], b[n*10];
    ...
}
```

• Dynamic arrays cannot be allocated within an AR, they can be allocated on the top of stack as soon as their associated declaration is evaluated.
Dynamic Arrays Allocation (Con’t)

f(int i,j,k)
{
    float A[i], B[j], C[k];
    ....
}

ARRAY

Pointer to A
Pointer to B
Pointer to C

fp

Pointer to A
Pointer to B
Pointer to C

sp

Array A
Array B
Array C
Dope vectors

Dope vector is a fixed size descriptor containing the array’s type declaration. It contains the dimension, size, and bounds info of the array.

f(int i,j,k)
{
    float A[i][j], B[j][k], C[i][j][k];
    ....
}

---

Array A
---

Array B
---

Array C
---

A Dope vector

pointer to A

B Dope vector

pointer to B

C Dope vector

pointer to C

sp
Heap Memory Allocation

Heap allocation is when an executing program requests that the operating system give it a block of main memory.

Features of heap allocation
• Programs may request memory and may also return previously dynamically allocated memory.
• Memory may be returned whenever it is no longer needed. Memory can be returned in any order without any relation to the order in which it was allocated.
• The heap may develop "holes" where previously allocated memory has been returned between blocks of memory still in use.
• A new dynamic request for memory might return a range of addresses out of one of the holes. But it might not use up all the hole, so further dynamic requests might be satisfied out of the original hole.
• Keeping track of allocated and deallocated memory is complicated. A modern operating system does all this.
Malloc() Allocation

The malloc() function requests a block of memory:

```c
void *malloc(size_t size)
```

The function allocates a block of memory of `size` number of bytes, and returns the address of the first byte of that block. (The data type of size is an unsigned integer. The return type of the function is void * which is the way ANSI C describes a memory address.)
Malloc() Allocation Example

```c
struct EMPLOYEE /* Declaration of layout of memory block */
{  
    int age;
    int pay;
    int class;};

main()
{  
    struct EMPLOYEE *empA;    /* declaration of the pointer variable empA */
    empA = (struct EMPLOYEE *)malloc( sizeof( struct EMPLOYEE ) );
    empA->age = 34;

    struct EMPLOYEE *empB;    /* declaration of a second pointer variable empB */
    empB = (struct EMPLOYEE *)malloc( sizeof( struct EMPLOYEE ) );
    empB->age = empA->age;

    PStruct( empA );  /* Write out the first struct */
    PStruct( empB );  /* Write out the second struct */
}

void PStruct( struct EMPLOYEE *emp )  
{  printf("age: %d ", emp->age ); }
struct EMPLOYEE
{int age;
 int pay;
 int class;};

main:
 . . . .
 # create the first struct
 li $v0,9    # allocate memory
 li $a0,12  # 12 bytes
 syscall    # $v0 <-- address
 move $s1,$v0 # $s1 first struct
 . . . .

main(){
 struct EMPLOYEE *empA;    /* declaration of the pointer variable empA */

 empA = (struct EMPLOYEE *)malloc( sizeof( struct EMPLOYEE ) );
}
How It is Compiled (Con’t)

```c
struct EMPLOYEE
{
    int age;
    int pay;
    int class;
};

main()
{
    struct EMPLOYEE *empA;   /* declaration of the pointer variable empA */
    empA = (struct EMPLOYEE *)malloc( sizeof( struct EMPLOYEE ) );
    empA->age = 34;
    struct EMPLOYEE *empB;   /* declaration of a second pointer variable empB */
    empB = (struct EMPLOYEE *)malloc( sizeof( struct EMPLOYEE ) );
    PStruct( empA );  /* Write out the first struct */
    PStruct( empB );  /* Write out the second struct */
}

void PStruct( struct EMPLOYEE *emp )
{
    printf("age: %d ", emp->age );
}
```

```
main:
    . . .
# create the first struct
li $v0,9    # allocate memory
li $a0,12   # 12 bytes
syscall     # $v0 <-- address
move $s1,$v0 # $s1 first struct

# initialize the first struct
li $t0,34    # store 34
sw $t0,0($s1) # in age field
```
How It is Compiled (Con’t)

```c
struct EMPLOYEE
{
    int age;
    int pay;
    int class;
};

main()
{
    struct EMPLOYEE *empA;  /* declaration of the pointer variable empA */
    empA = (struct EMPLOYEE *)malloc( sizeof( struct EMPLOYEE) );
    empA->age = 34;

    struct EMPLOYEE *empB;  /* declaration of a second pointer variable empB */
    empB = (struct EMPLOYEE *)malloc( sizeof( struct EMPLOYEE) );
    empB->age = empA->age;

    PStruct( empA );  /* Write out the first struct */
    PStruct( empB );  /* Write out the second struct */
}

void PStruct( struct EMPLOYEE *emp )
{
    printf("age: %d ", emp->age );
}
```

main:

```
    li $v0,9     # allocate memory
    li $a0,12    # 12 bytes
    syscall     # $v0 <-- address
    move $s2,$v0 # $s2 second struct

    # copy data from first struct to second
    lw $t0,0($s1) # copy age from first
    sw $t0,0($s2) # to second struct
```
How It is Compiled (Con’t)

```c
struct EMPLOYEE
{
  int age;
  int pay;
  int class;
};

main()
{
  struct EMPLOYEE *empA;  /* declaration of the pointer variable empA */
  move $a0,$s1
  jal       PStruct
  struct EMPLOYEE *empB;  /* declaration of a second pointer variable empB */
  move $a0,$s2
  jal       PStruct

  empA = (struct EMPLOYEE *)malloc( sizeof( struct EMPLOYEE) );
  empA->age = 34;
  struct EMPLOYEE *empB = (struct EMPLOYEE *)malloc( sizeof( struct EMPLOYEE) );
  empB->age = empA->age;

  PStruct( empA );  /* Write out the first struct */
  PStruct( empB );  /* Write out the second struct */
}

void PStruct( struct EMPLOYEE *emp )
{
  printf("age: %d ", emp->age);
}
```

# write out the first struct
move $a0,$s1
jal   PStruct

# write out the second struct
move $a0,$s2
jal   PStruct
struct EMPLOYEE
{
    int age;
    int pay;
    int class;
};

main()
{
    struct EMPLOYEE *empA;
    struct EMPLOYEE *empB;

    empA = (struct EMPLOYEE *)malloc( sizeof( struct EMPLOYEE));
    empB = (struct EMPLOYEE *)malloc( sizeof( struct EMPLOYEE));

    empA->age = 34;
    empB->age = empA->age;

    PStruct( empA );/* Write out the first struct */
    PStruct( empB );/* Write out the second struct */
}

void PStruct( struct EMPLOYEE *emp )
{
    printf("age: %d ", emp->age);
}
Heap Memory Manager

• It keeps track of the free space in heap storage at all times.
  ▪ Allocation
gives a chunk of contiguous heap memory of requested size
  ▪ Deallocation
return the deallocated space to the pool of free space
Reducing Fragmentation

• Initially, the heap is one contiguous unit of free space. As the program allocates and deallocates, this space is broken into free and used chunks.

• Allocation and deallocation must be careful in dealing with fragmentation issues.
**First-fit, Best-fit, Next-fit**

- **First-fit**
  
  To allocate the requested memory in the first hole in which it fits. *(fast, but lots of small holes)*

- **Best-fit**
  
  To allocate the requested memory in the smallest hole that is large enough. *(low locality)*

- **Next-fit**
  
  To allocate in the chunk that has last been split (like first-fit but remember the searching position)*
Heap Deallocation

- **No deallocation**
  - Stop when space run out
- **Explicit (manual) deallocation**
  - Free (C, PL/1), delete (C++), dispose (Pascal), deallocation (Ada)
  - May lead to memory leak and dangling reference
- **Implicit deallocation**
  - Reference count
  - Garbage collection
Non-local names

In a language with nested procedures (or blocks) and static scope (lexical scope), some names are neither local nor global, they are non-local names.
Non-local names in PASCAL

procedure A
real a;

procedure B
real b;
reference a; \( \leftarrow \) non-local name
end B
end A;
Example: Non-local names in C

```c
main () {
    int a = 0, b=0; {
        int b = 1; {
            int a = 2;
            print(a,b); }
        print(a,b); }
    print(a,b); }
print(a,b); }
```

Most closely nested rule

- 2, 1
- 0, 3
- 0, 1
- 0, 0
Another example

- int a = 0;
- int f1()
  - {
    return a;
  }

- int f2(int c)
  - {
    if(c>0) {
      int a = 5;
      return f1();
    }
    else {
      return f1();
    }
  }

Question: what value of a will Be returned by f1 ?
(1) assuming c > 0
(2) assuming c < 0

Please answer this under two Scenarios: (A) C scoping rule;
(B) Dynamic scoping rule;
Block-level and Procedure-level ARs

1. Each block can be considered as an in-line procedure without parameters. So we could create a new AR for each block. This is block-level AR and is very inefficient.

2. In procedure-level AR method, AR is only used for a true procedure. The relative location of variables in individual blocks within a procedure can be computed and fixed at compile time.

As above C example:

<table>
<thead>
<tr>
<th></th>
<th>Level 0</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>a, b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Static/Dynamic Chains

- **Static link**: each stack frame contains a reference to the frame of the lexically surrounding procedure. By following the static links (a static chain), the non-local object can be found. Static link is also called access link.

- **Dynamic link**: The saved value of the frame pointer, which points to the caller’s AR, is a dynamic link.
Displays

• A display is an embedding of the static chain into an array. The jth element of the display contains a reference to the frame of the most recently active procedure at lexical nesting level j.

• Display can be stored in a set of registers, or in memory. Non-local names are not references very frequently, so keeping the display in registers may not as important as it looks.
Example of Display
Summary

• Binding names to storage locations
• Activation Record (AR)
• Parameter passing
• Storage allocations
• Accessing non-local names