CSc 520

Principles of Programming Languages

4: Memory Management — Introduction

Christian Collberg

collberg+520@gmail.com

Department of Computer Science
University of Arizona

Copyright © 2008 Christian Collberg
Memory Management

In a language such as C or Pascal, there are three ways to allocate memory:

1. **Static allocation.** Global variables are allocated at compile time, by reserving.
2. **Stack allocation.** The stack is used to store activation records, which holds procedure call chains and local variables.
3. **Dynamic allocation.** The user can create new memory at will, by calling a `new` or (in unix) `malloc` procedure.

The compiler and run-time system divide the available address space (memory) into three sections, one for each type of allocation:
1. The static section is generated by the compiler and cannot be extended at run-time. Called the uninitialized data section in unix’s a.out.

2. The stack. The stack grows and shrinks during execution, according to the depth of the call chain. Infinite recursion often leads to stack overflow. Large parameters can also result in the program running out of stack space.

3. The heap. When the program makes a request for more dynamic memory (by calling malloc, for example), a suitable chunk of memory is allocated on the heap.
Memory Management...

- **Static allocation** – Global variables
- **Stack allocation** – Procedure call chains, Local variables.
- **Dynamic allocation** – `NEW`, `malloc`, On the heap.

<table>
<thead>
<tr>
<th>Program Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialized Data</td>
</tr>
<tr>
<td>(strings, reals...)</td>
</tr>
<tr>
<td>Uninitialized Data</td>
</tr>
<tr>
<td>(Global Variables)</td>
</tr>
</tbody>
</table>

![Memory Management Diagram](image)
Dynamic Memory Management

The run-time system linked in with the generated code should contain routines for allocation/deallocation of dynamic memory.

**Pascal, C, C++, Modula-2** Explicit deallocation of dynamic memory only. I.e. the programmer is required to keep track of all allocated memory and when it’s safe to free it.

**Eiffel** Implicit deallocation only. Dynamic memory which is no longer used is recycled by the garbage collector.

**Ada** Implicit or explicit deallocation (implementation defined).

**Modula-3** Implicit and explicit deallocation (programmer’s choice).
### Run-Time Memory Organization

<table>
<thead>
<tr>
<th>Low Addresses</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>⇓</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>↑</td>
</tr>
<tr>
<td>Stack</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Static Data</td>
</tr>
<tr>
<td>High Addresses</td>
<td>Initialized Data</td>
</tr>
<tr>
<td></td>
<td>Text Segment</td>
</tr>
</tbody>
</table>
Run-Time Memory Organization...

This is a common organization of memory on Unix systems.

The Text Segment holds the code (instructions) of the program. The Initialized Data segment holds strings, etc, that don’t change. Static Data holds global variables. The Stack holds procedure activation records and the Heap dynamic data.
Storage Allocation

Global Variables are stored in the Static Data area.

Strings (such as "Bart!") are stored in the Initialized Data section.

Dynamic Variables are stored on the Heap:

```pascal
PROCEDURE P ();
    VAR X : POINTER TO CHAR;
BEGIN
    NEW(X);
END P
```
Own Variables are stored in the Static Data area. An Own variable can only be referenced from within the procedure in which it is declared. It retains its value between procedure calls.

PROCEDURE P (X : INTEGER);
   OWN W : INTEGER;
   VAR L : INTEGER;
BEGIN W := W + X; END P
Global Variables – MIPS

How do we allocate space for and access global variables? We’ll examine three ways.

**Running Example:**

```plaintext
PROGRAM P;
    VAR X : INTEGER; (* 4 bytes. *)
    VAR C : CHAR; (* 1 byte. *)
    VAR R : REAL; (* 4 bytes. *)
END.
```
Allocate each global variable individually in the data section. Prepend an underscore to each variable to avoid conflict with reserved words.

Remember that every variable has to be aligned on an address that is a multiple of its size.

```assembly
.data
_X: .space 4
_C: .space 1
.align 2 # 4 byte boundary.
_R: .space 4
.text
main: lw $2, _X
```
 Allocate one block of static data (called _Data, for example), holding all global variables. Refer to individual variables by offsets from _Data.

```assembly
.data
_Data:  .space 48
.text
main:  lw  $2, _Data+0  # X
       lb  $3, _Data+4  # C
       l.s  $f4, _Data+8  # R
```
Allocate global variables on the bottom of the stack. Refer to variables through the **Global Pointer** $gp$, which is set to point to the beginning of the stack.

```assembly
main:    subu $sp,$sp,48
         move $gp,$sp
         lw   $2, 0($gp)  # X
         lb   $3, 4($gp)  # C
         l.s  $f4, 8($gp) # R

_X: .space 4 Each access lw $2, _X takes 2 cycles.
_Data: .space 48 Each access lw $2, _Data+32 takes 2 cycles.

subu $sp,$sp,48 1 cycle to access the first 64K global variables.
```
Storage Allocation...

Local Variables: stored on the run-time stack.

Actual parameters: stored on the stack or in special argument registers.

Languages that allow recursion cannot store local variables in the Static Data section. The reason is that every Procedure Activation needs its own set of local variables.

For every new procedure activation, a new set of local variables is created on the run-time stack. The data stored for a procedure activation is called an Activation Record.

Each Activation Record (or (Procedure) Call Frame) holds the local variables and actual parameters of a particular procedure activation.
Readings and References

- Read Scott, pp. 103–113.