Storage Layout
Comp 412

A very “meta” lecture.

Copyright 2019, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapters 4, 5, 6 & 7 in EaC2e
What About Storage Layout?

Where do all of the named entities go? And how do they get there?

The compiler must classify *all code & data* so that it can assign storage

- **Scopes**: local, global, subject to some set of lexical scoping constraints
- **Lifetimes**: entire execution, execution of a procedure, or indeterminate

Given these classifications & the state of the naming model, the compiler can assign data to specific **data areas**

- Each procedure has a **local data area**
  - *Local data from a procedure scope or sub-scope usually goes here*
- Declarations, procedures, files, & modules may have a **static data area**
  - *Depends on the specific declarations in the code*
- A program may have one or more **global data areas & constant pools**
- Each object has an object record
  - *An object record contains a private data area to hold an object’s data members*
What is a “Data Area”?

The compiler must set aside space for the various data areas

• Compiler reserves part of the virtual address space for the data area
  – Strategy set at design time; details worked out at compile time
• The compiler must plan how to use that data area

A data area is just a block of memory reserved for data by the compiler

The compiler needs, for each data area:

1. A list of what entities live in that data area
2. A plan for where in the virtual address space the DA lives
3. A plan for how to organize the DA internally
   – An offset for every entity in the DA
4. A plan for how to find the DA at runtime
   – “establish addressability” for the contents of the data area

Constant pool:
Some constants are either too big for a load, or are not known until link time. The compiler will create one or more data areas dedicated for these values. It can then use load on them.
Where Do Data Areas Live at Runtime?

Based on lifetime, visibility, and type, the compiler creates multiple data areas

- Each procedure has a local data area
  - Local data areas live on the heap or on a runtime stack
- Declarations, procedures, files, & modules may have a static data area
  - Static/global data areas are created once with an assembly pseudo-op
- A program may have one or more global data areas & constant pools
- Each object has an object record
  - Object records are typically allocated on the heap
When Does the Compiler Assign Storage?

The compiler must assign storage before it can use an address in the IR

1. Compiler may assign storage before generating initial IR
   - Implies a declaration before use rule or two passes (*which requires IR*)

2. Compiler may assign storage incrementally as it generates IR
   - Leads to some inefficiency in memory layout
     → *In practice, probably not too terribly wasteful*

3. Compiler may generate initial IR with abstracted addresses
   - See @x in any ILOC example with a loadAI
   - Lower the abstract addresses to concrete address in a later pass

Any of these schemes can work in the compiler

The PL designer makes decisions that can dictate the choice

- “*declare before use*” rule is fairly common
- No declarations generally force the second or third option
When Does the Compiler Assign Storage?

With declare before use: (an important special case)

- Compiler can make its decisions after parsing declarations
- Needs a time and place to perform storage layout

```
Procedure       →   Header { Declarations Statements }
Declarations    →   Declarations Decl
                  |       Decl
Statement       →   Statements Stmt
                  |       Stmt
```

Modified grammar:

```
Procedure       →   Header { AssignStorage Statement }
AssignStorage   →   Declarations
```

Typical Grammar for a Procedure

Modify the grammar to create a reduction to assign storage.
What is a “Data Area”? Or a global data area? Or a constant pool?

The compiler must set aside space for the various data areas

• Compiler reserves part of the virtual address space for the data area
  – Strategy set at design time; details worked out at compile time
• The compiler must plan how to use that data area

A data area is just a block of memory reserved for data by the compiler

The compiler needs, for each data area:

1. A list of what entities live in that data area
2. A plan for where in the virtual address space the DA lives
3. A plan for how to organize the DA internally
   – An offset for every entity in the DA
4. A plan for how to find the DA at runtime
   – “establish addressability” for the contents of the data area
Storage Layout

The compiler must assign a location to each named variable and each temporary value that it introduces.

- That location should depend on the value’s visibility & its lifetime

Lifetimes

Automatic variables
- Lifetime matches lifetime of declaring scope’s activation
- Implicit allocation & deallocation

Static variables
- Lifetime is the complete execution of the program
- Implicit allocation and deallocation

Values with irregular lifetimes
- Explicit allocation and deallocation (new, or malloc, or ...)

The compiler classifies all named entities by their visibility & lifetime to group them into data areas.
The compiler must assign a location to each named variable and each temporary value that it introduces.

- The location should depend on the value’s visibility & its lifetime

Visibility

Local Scopes
- Procedure scope: name is visible in defining procedure (& nested scopes)
- Block scope: name is visible in defining block (& nested scopes)

Global Scope
- Names are visible anywhere that they are declared, unless obscured by a local declaration

Unusual scopes
- C’s “file scope”; languages with explicit visibility control
Data Areas by Attributes

**Automatic & Local**
- Automatic local values live in the local data area, in the procedure’s AR*
  → *Or in a register, if it is an unambiguous scalar value*
- Compiler lays out a map for the local data area, including spill locations
- Local values from surrounding scopes live in the AR for their defining scope
  → *Runtime addressability mechanism establishes a way to reach them*

**Static**  
(implies lifetime of entire execution)
- Procedure static ⇒ procedure-specific data area  (e.g., &\.procname_da)
- File static ⇒ file-specific data area  (e.g., &\.filename_da)
- Necessary to ensure that they remain live across different activations

**Global**  
(implies lifetime of entire execution)
- Global values live in one or more global data area(s)
- One label per variable, or per file, or per program, ...
  → *With consistently mangled labels to connect them across compiles*
  → *Equivalent to an assembly language “defined storage” pseudo-operation*

Compilers obtain these labels by “mangling” the original name. The linker will report conflicting definitions of the same label.
What is this “activation record”? 

**Activation Record Basics**

In most systems, Activation Records have a similar layout:

- **Space for parameters**
- **Register save area**
- **Slot for return value**
- **Slot for return address**
- **Slot for addressability**
- **Slot for caller’s ARP**
- **Space for local variables**

"the local data area"

The Activation Record holds the state of one invocation (activation) of a procedure:

- Its local variables & parameters
- Control information to allow an orderly return to its caller
- Any values from the caller that it must preserve
- Any values that the code spills

The AR plus the code creates the incarnation of the procedure.

The *calling convention* creates & maintains the ARs

Classic chicken & egg problem defining an AR
What is this “activation record”? 

Activation Record Basics

In most systems, Activation Records have a similar layout

- **space for parameters**
- **register save area**
- **slot for return value**
- **slot for return address**
- **slot for addressability**
- **slot for caller’s ARP**
- **space for local variables**

ARP ≅ Activation Record Pointer

If ARs are stack allocated, the stack grows in this direction

Space for parameters to the current procedure
Contents of saved registers
If function, space for return value
Address to resume caller
Help with non-local access
To restore caller’s AR on return
Space for local variables, temporaries, & spills

One AR for each invocation of a procedure
One register dedicated to hold the current ARP
What is a “Data Area”?

The compiler must set aside space for the various data areas

• Compiler reserves part of the virtual address space for the data area
  – Strategy set at design time; details worked out at compile time
• The compiler must plan how to use that data area

A data area is just a block of memory reserved for data by the compiler

The compiler needs, for each data area:

1. A list of what entities live in that data area
2. A plan for where in the virtual address space the DA lives
3. A plan for how to organize the DA internally
   – An offset for every entity in the DA
4. A plan for how to find the DA at runtime
   – “establish addressability” for the contents of the data area
Where Do Data Areas Live?

The executable assumes that it has its own virtual address space

- Large, isolated, uninitialized tract of memory
- Compiled code manages the layout of that virtual address space
  - In collaboration with the operating system, which creates & maintains it

**Virtual Address Space**

```
| 0 | 1 | 2 | 3 | ... | n-2 | n-1 | n |
```

The compiler organizes that virtual address space into a collection of runtime areas and structures

- Collaborative effort with operating system & other language runtimes
Address Space Layout

Most language runtimes layout the address space in a similar way:

- Pieces (stack, heap, code, & globals) may move, but all will be there.
- Stack and heap grow toward each other (if heap grows).
- Arrays live on one of the stacks, in the global area, or in the heap.

The picture shows one virtual address space.

- The hardware supports one virtual address space per process.
- How does a virtual address space map into physical memory?
What is a “Data Area”?  

The compiler must set aside space for the various data areas

• Compiler reserves part of the virtual address space for the data area
  – Strategy set at design time; details worked out at compile time
• The compiler must plan how to use that data area

A data area is just a block of memory reserved for data by the compiler

The compiler needs, for each data area:

1. A list of what entities live in that data area
2. A plan for where in the virtual address space the DA lives
3. A plan for how to organize the DA internally
   – An offset for every entity in the DA
4. A plan for how to find the DA at runtime
   – “establish addressability” for the contents of the data area
Data Area Layout & Alignment

**Compiler must order the entities in each data area**

- Create a list of all entities by data area
- Order the list and assign offsets within the data area
- Record the offset in the symbol-table record for the entity

**Assume that:**

- a, b, c, & d reside in the same data area
- b & d are each 4 bytes long while a & c are 1 byte each

![](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>9</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
</tr>
</tbody>
</table>
Data Area Layout & Alignment

Having values of multiple types in the same data area creates potential alignment issues

• Each type (size) of value has its own alignment restriction
• Laying out a data area in arbitrary order can require padding

A Layout That Wastes Space

A Better Layout
To create a layout that minimizes space wasted in “padding”

- Sort variables by alignment restrictions
  - Most strict (e.g., quad word or double word) to least strict (e.g., byte)

- From most strict to least strict, lay out variables in contiguous memory
  - Alignment constraints generally decrease by ½ at a transition
  - This scheme avoids almost all padding for scalar variables.
  - Structures that have atypical alignment constraints may need padding

→ A seventeen-byte long structure followed by a quadword aligned double double

Can backfill holes with small entities (two-finger algorithm)
What is a “Data Area”? The compiler must set aside space for the various data areas

• Compiler reserves part of the virtual address space for the data area
  – Strategy set at design time; details worked out at compile time
• The compiler must plan how to use that data area

A data area is just a block of memory reserved for data by the compiler

The compiler needs, for each data area:

1. A list of what entities live in that data area
2. A plan for where in the virtual address space the DA lives
3. A plan for how to organize the DA internally
   – An offset for every entity in the DA
4. A plan for how to find the DA at runtime
   – “establish addressability” for the contents of the data area
How Can Compiled Code Find the Data Areas?

The compiler must establish addressability for each kind of data area

- **Local data area in the AR**
  - Compiler knows offset from the ARP & keeps ARP in a designated register

- **Static data area**
  - Creates name-mangled label and loads its value into a register

- **Constant pool**
  - Either a name-mangled label or a fixed offset from the code (PC-relative address?)

- **Each object has an object record (OR)**
  - Object’s “name” or “reference” is a pointer to its OR
  - A class, being an object, has an object record of class “class”

Objects are assumed to have irregular lifetimes, so they end up on the heap.

COMP 412, Fall 2019
Name Mangling

One of the most useful tools the compiler has is an assembly-code label

- Compiler can emit a label in the code
- Assembler will turn it into a relocatable symbol
- Linker will replace the label with the appropriate virtual address
- Hardware will translate virtual address to a physical address

Name mangling is the standard trick

- The compiler assumes that some source-language names are unique
  - Global variables, procedure names
  - Use fully-qualified names for nested objects
- The compiler builds strings based on the unique name
  - Code for procedure fee might be labelled _fee._ or _&fee or ...
  - Specific “mangles” for specific purposes
    → Static data area, entry point to the code, return address at a call, ...
  - Use character combinations that are not legal in the source language

Need to understand how assemblers & linkers work
One of the most useful tools the compiler has is an assembly-code label

- Compiler can emit a label in the code
- Assembler will turn it into a relocatable symbol
- Linker will replace the label with the appropriate virtual address
- Hardware will translate virtual address to a physical address

Given a “mangled” label, the compiler can use it to load the address

- Initialize a slot in the constant pool with the relocatable label
  - The linker will replace that label with the corresponding virtual address
- The compiler can get the virtual address using a load from that slot
  - At runtime, the load takes value from slot & moves it to a PR
- The same idea works with any relocatable assembly-level label
  - Data areas, code, return addresses, statically-allocated objects
Multiple Virtual Address Spaces?

The Big Picture

- **Compiler’s view**
  - virtual address spaces (one per process)

- **OS’ view**
  - TLB

- **1980 Hardware view**
  - Physical address space (big vector of memory)

TLB is an address cache used by the OS to speed virtual-to-physical address translation. A processor may have > 1 level of TLB.