Handling Assignment

Comp 412

Copyright 2018, 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.

Chapter 7 in EaC2e
What is Left in Translation?

We have talked about the mechanism of translation. Next, we need to focus on the content, or product, of translation.

Expressions
- More operators in expressions
  - Boolean expressions, relational expressions, string operations
- Type-related issues in expressions
  - Mixed-type operations and value conversions.
- Assignment (*including addressability*)

Control Flow
- Code shape and SDT strategies

Procedure and Function Calls
- Functionality: what, precisely, does a procedure call do?
- Implementation: how does the compiler emit code to do that?

Today
Handling Assignment

Some languages treat assignment as a distinct statement.
Some languages treat assignment as an expression operator.

\[ \text{lhs} \leftarrow \text{rhs} \]

- Evaluate \( \text{rhs} \) to a **value**
- Evaluate \( \text{lhs} \) to a **location**
  - \( \text{lvalue} \) is a register \( \Rightarrow \) move \( \text{rhs} \) into the register
  - \( \text{lvalue} \) is an address \( \Rightarrow \) store \( \text{rhs} \) into the memory location

**Some values go into registers, others go into memory**
- Storage allocation makes that decision, based on available knowledge
- Lifetime, storage class, knowledge all factor into that decision
Handling Assignment

The compiler must emit code to compute an address for any \textit{lhs} that is not kept in a register.

Many interesting values cannot be kept in a register

- Strings, arrays, structures, objects
- Global and static values
- Known values that are too large

Aggregate Value (arrays, structures/records, strings, objects)

- Compiler needs to know a starting address
- Compiler needs to compute an internal layout to derive an offset
- Compiler needs to understand size & type

\textbf{From this information, it emits code to compute the runtime address}

<table>
<thead>
<tr>
<th>\textit{Factor}</th>
<th>( \textit{Expr} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>\textit{Reference}</td>
</tr>
</tbody>
</table>

\textit{All those cases lumped together as “Reference”}
Scheme to Generate Code For A[i,j]?

Compiler must generate the runtime address of the element A[i,j]

- The compiler needs to know where A begins
  - Tie between compile-time knowledge and runtime behavior
  - We will defer this discussion until we discuss procedure calls ...
    - assume that @A is the address of A's first element

- The compiler must have a plan for the internal layout of A
  - Programming language usually dictates array element layout
  - Three common choices
    - Row-major order
    - Column-major order
    - Indirection vectors

- And a formula for calculating the address of an array element
  - General scheme: compute address, then issue load (rvalue) or store (lvalue)

The Concept

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1</td>
</tr>
<tr>
<td>2,1</td>
</tr>
</tbody>
</table>
Array Layout

Row-Major Order
- Lay out as a sequence of consecutive rows
- Rightmost subscript varies fastest
- A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]

Storage Layout

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
<th>2,1</th>
<th>2,2</th>
<th>2,3</th>
<th>2,4</th>
</tr>
</thead>
</table>

Stride One Access

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        A[i][j] = 0;
```

Declared arrays in C (and most languages)

Stride one access: successive references in the loop are to successive locations in the level one (or L1) data cache. Stride one access maximizes spatial reuse & the effectiveness of hardware prefetch units.

The Concept

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
</tr>
</tbody>
</table>
Array Layout

Column-Major Order
• Lay out as a sequence of columns
• Leftmost subscript varies fastest
• A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

Storage Layout

A

\begin{array}{cccccccc}
1,1 & 2,1 & 1,2 & 2,2 & 1,3 & 2,3 & 1,4 & 2,4 \\
\end{array}

Stride One Access

\begin{verbatim}
for ( j = 0; j < n; j++)
for ( i = 0; i < n; i++)
  A[i][j] = 0;
\end{verbatim}

The Concept

A

\begin{array}{cccc}
1,1 & 1,2 & 1,3 & 1,4 \\
2,1 & 2,2 & 2,3 & 2,4 \\
\end{array}

All arrays in FORTRAN
Array Layout

Indirection Vectors
• Vector of pointers to pointers to ... to values
• Takes much more space, trades indirection for arithmetic
• Not amenable to analysis

Storage Layout

Stride One Access
No reference pattern guarantees stride one access in cache, unless rows are contiguous

Arrays in Java

int **array; in C

COMP 412, Fall 2017
Computing an Address for an Array Element

The General Scheme for an Array Reference

• Compute an address
• Issue a load

A[i]

• @A + (i – low) x sizeof(A[1])
• In general: base(A) + (i – low) x sizeof(A[1])

@A is the base address of A.
Depending on how A is declared, @A may be
• an offset from the pointer to the procedure’s data area (its ARP),
• an offset from some global label, or
• an arbitrary address.
The first two are compile time constants.

low
Lower bound on index

ARP ≡ Activation Record Pointer (see next lecture)
Computing an Address for an Array Element

A[ i ]

• \( @A + ( i - \text{low} ) \times \text{sizeof}(A[1]) \)

• In general: base(A) + ( i – low ) x sizeof(A[1])

Note, for the record, that a naïve vector reference turns into a subtract, a shift, and an address-offset load (e.g., loadAO) — three operations before optimization. This sequence is amazingly cheap.

Almost always a power of 2, known at compile-time ⇒ use a shift for speed

low

Lower bound on index
Computing an Address for an Array Element

A[i]

- \( @A + (i - \text{low}) \times \text{sizeof(elt)} \)
- In general: base(A) + (i - low) \times \text{sizeof(A[1])}

What about \( A[i_1, i_2] \)?

Row-major order, two dimensions

\[ @A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times \text{sizeof(elt)} \]

\[ \begin{array}{cccc}
1,1 & 1,2 & 1,3 & 1,4 \\
2,1 & 2,2 & 2,3 & 2,4 \\
3,1 & 3,2 & 3,3 & 3,4 \\
4,1 & 4,2 & 4,3 & 4,4 \\
5,1 & 5,2 & 5,3 & 5,4 \\
\end{array} \]

(i-low) is 3
(high\(_2\) - low\(_2\) + 1) is 4
3 x 4 is 12
i\(_2\) - low\(_2\) is 2

Combined, the 2 terms take us to the start of \( A[4,3] \)

Address computation took 3 adds, 3 subtracts, 1 multiply and 1 shift
Cheap in comparison to the register-save costs of a function call.
Array Layout

Row-Major Order
• Lay out as a sequence of consecutive rows
• Rightmost subscript varies fastest
• A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]

Storage Layout

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
<th>2,1</th>
<th>2,2</th>
<th>2,3</th>
<th>2,4</th>
</tr>
</thead>
</table>

Address Polynomial for A[i,j]

\[ @A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times \text{sizeof(elt)} \]

Why does C start arrays indices at zero?
To avoid those subtractions

This polynomial follows Horner’s rule for evaluation.
Array Layout

Column-Major Order
• Lay out as a sequence of columns
• Leftmost subscript varies fastest
• A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

Storage Layout

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>2,1</th>
<th>1,2</th>
<th>2,2</th>
<th>1,3</th>
<th>2,3</th>
<th>1,4</th>
<th>2,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address Polynomial for A[i,j]

\[ @A + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times \text{sizeof(elt)} \]

This polynomial follows Horner’s rule for evaluation.

Change in layout order swaps the subscripts ...
Array Layout

Indirection Vectors
• Vector of pointers to pointers to ... to values
• Takes much more space, trades indirection for arithmetic
• Not amenable to analysis

Storage Layout

Address Polynomial for A[i,j]
*(A[i_1])[i_2] — where each of the [i]’s is, itself, a 1-d array reference

Back when systems supported 1 or 2 cycle “indirect load” operations, this scheme was efficient. It replaces a multiply & an add with an indirect load.

COMP 412, Fall 2017
Generating Better Code for $A[i,j]$ 

**In row-major order**

$@A + ((i - low_1) \times (high_2 - low_2 + 1) + j - low_2) \times w$

Can be re-distributed to

$@A + (i - low_1) \times (high_2 - low_2 + 1) \times w + (j - low_2) \times w$

Which can be re-distributed to

$@A + i \times (high_2 - low_2 + 1) \times w + j \times w$

$- (low_1 \times (high_2 - low_2 + 1) \times w) - (low_2 \times w)$

If $low_i$, $high_i$, and $w$ are known, the last term is a constant

Define $@A_0$ as

$@A - (low_1 \times (high_2 - low_2 + 1) \times w - low_2 \times w)$

And $len_2$ as $(high_2 - low_2 + 1)$

Then, the address expression becomes

$@A_0 + (i \times len_2 + j) \times w$

where $w = \text{sizeof}(A[1,1])$

If $@A$ is known, $@A_0$ is a known constant.

Just a couple of operations: 2 adds, 1 multiply, and 1 shift

Why does C start arrays indices at zero?

Compile-time constants

COMP 412, Fall 2017
How do we know what to do?

Accumulated wisdom

Certain simple optimizations are performed as the quadruples are constructed. For example, an integer multiplication by a constant power of two (e.g., $I \times 4$ or $I \times 16$) is replaced by a left-shift operation. An exponentiation involving an integer constant power (e.g., $A^{**9}$) is replaced by a series of in-line multiplications. Some operations involving minus signs are converted to simpler forms; for instance, $-(B-C)$ becomes $C - B$.

Finally, constants employed in a subscript expression [for example, each number 7 in $A(7,1-7,1+7)$] are often extracted from the subscript, evaluated as constant offsets from the start of the subscripted array, and combined into an aggregate constant offset which does not require computation during execution.

(p. 662; description of the “old” compiler, e.g., Medlock and Lowry)

The standard FORTRAN compiler evaluates subscripted array references in six steps. First, numerical constants embedded in the subscripts are extracted, evaluated, and combined into an aggregate constant subscript when the program is translated into quadruples. Second, the subscript expression remaining in each dimension is evaluated and converted to integer. Third, each of these evaluated subscripts is multiplied by the span in bytes represented by a unit subscript in the subscripted dimension. Fourth, these products are added together to produce an aggregate computed subscript. Fifth, the constants extracted from the subscript and combined to form the aggregate constant subscript are added to the aggregate computed subscript to produce the aggregate effective subscript. For an aggregate constant subscript in the range 0-4095, this addition is accomplished implicitly by encoding the constant in the displacement field of an indexed machine instruction. Sixth, the address of the array itself is added to the aggregate effective subscript to produce the address of the subscripted array element. This addition is always accomplished implicitly by using the base and index registers of an indexed machine instruction.

(p. 665; describing the new compiler)

How do we know what to do?

Accumulated wisdom

Certain simple optimizations are performed as the quadruples are constructed. For example, an integer multiplication by a constant power of two (e.g., I*4 or I*16) is replaced by a left-shift operation. An exponentiation involving an integer constant power (e.g., A**9) is replaced by a series of in-line multiplications. Some operations involving minus signs are converted to simpler forms; for instance, -(B - C) becomes C - B. Finally, constants employed in a subscript expression (for example, each number 7 in A(7,1+7)] are often extracted from the subscript, evaluated as constant offsets from the subscripted array, and combined into an aggregate constant offset during execution.

The standard FORTRAN compiler evaluates subscripted array references in six steps. First, numerical constants embedded in the subscript when the program is translated into quadruples. Second, the subscript expression remaining in each dimension is evaluated and converted to integer. Third, each of these evaluated subscripts is multiplied by the span in bytes represented by a unit subscript in the subscripted dimension. Fourth, these products are added together to produce an aggregate computed subscript. Fifth, the constants extracted from the subscript and combined to form the aggregate constant subscript are added to the aggregate computed subscript to produce the aggregate effective subscript. For an aggregate constant subscript in the range 0-4095, this addition is accomplished implicitly by encoding the constant in the displacement field of an indexed machine instruction. Sixth, the address of the array itself is added to the aggregate effective subscript to produce the address of the subscripted array element. This addition is always accomplished implicitly by using the base and index registers of an indexed machine instruction.

They didn’t solve the general problem of profitably distributing multiplication over addition. Instead, they encoded a good answer into the code shape.

Scheme to Address a Structure Element

A structure aggregates a collection of fields into one named object

• Structures are a convenient way to organize data
• Structures are often used to specify a layout for some interface
  → e.g., an io control block for a file or a tuple for a return value

To address a structure element, compiler needs a base and an offset

• Base address depends on declaration, scope, & lifetime
  – If it is a declared instance, then it is in a data area for the declaring scope
    → Local data area, static data area, object’s object record, some global area
  – If it is an allocated instance, then it’s base address is in some reference that
    the programmer designed and created
    → A named variable, the next or previous link in some list node, ...

• Offset is derived from the structure’s layout
Scheme to Address a Structure Element

Structure Layout

• A structure’s layout is usually dictated by its declaration (PL decision)
  – Programmer control of layout is critical for consistency in interfaces
• Compiler computes an offset from the declaration

```c
struct OPERATION {
    int Opcode;
    int Line;
    int Ordinal;
    int LiveNow;
    struct ARG Operands[3];
    struct OPERATION*Next;
};
```

Name | Type       | Offset |
-----|------------|--------|
Opcode | int        | 0      |
Line   | int        | 4      |
Ordinal| int        | 8      |
LiveNow| int        | 12     |
Operands| struct ARG * | 16     |
Operation| struct OPERATION * | 24     |

Assumes int is 4 bytes (word-aligned) and * is 8 bytes (double-word aligned)

Each structure needs a small symbol table (YAST)
Scheme to Address a Structure Element

But wait. That leads to wasted space!

• Yes. It does. Values must be padded to their alignment restrictions
• Wasted space is the cost of providing the programmer with control over structure layout
• What if alignment does not match my needs? (hardware control block?)
  – If the language does not support your needs, change languages or cheat

```
struct FEE {
    int a;
    long int b;
    char c;
    int *d;;
};
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>long int</td>
<td>8</td>
</tr>
<tr>
<td>c</td>
<td>char</td>
<td>16</td>
</tr>
<tr>
<td>d</td>
<td>long int</td>
<td>24</td>
</tr>
</tbody>
</table>

Assumes char is 1 byte (byte-aligned), int is 4 bytes (word-aligned) and * is 8 bytes (double-word aligned)
Scheme to Address an Object

An object is just a structure\(^{(to \text{ a } 1\text{st approximation})}\)

- Base address depends on declaration, scope, & lifetime
  - If it is a declared instance, then it is in a data area for the declaring scope
  - If it is an allocated instance, then it’s base address is in some reference that the programmer designed and created

- Offsets are computed from the class definition and hierarchy
  - Compiler is, typically, not bound to lay out object in declaration order
  - Compiler will use a consistent order for interoperability

\[\rightarrow \text{Adopt rules regarding object layout from PL or from convention (more later)}\]

Class Point {
    public int x, y;
    public void draw();
}
Class ColorPoint extends Point {
    Color c;
    public void draw() {...}
    public void test() { y = x; draw(); }
}

COMP 412, Fall 2018
What About an Array of Structures?

An array of structures is just an array with a non-standard element size

• Language must specify array layout
  – Row major, column major, indirection vectors
• Base address of array instance is found with the address polynomial
  – Many of the optimizations still apply \(\text{but, likely, not shift for multiply}\)

• Structure alignment restrictions may dictate internal padding between elements
  – The compiler cannot always win \(\text{see thermodynamics}\)
Representing & Manipulating Strings

Strings are fundamentally different from scalars, arrays & records

• Fundamental unit is a character
  – Typical character sizes are one or two bytes \( (\text{subword data}) \)
  – Target ISA may (or may not) support character-size operations

• Set of PL supported operations on character & string data is limited
  – Assignment, length, concatenation, & \( (\text{sometimes}) \) translation

• Efficient string operations are hard to implement from basic operations
  – Particularly difficult on RISC processors
  – Implications for the IR, the procedure linkage convention, & source language design

The IBM 370 had a machine instruction that did a byte-by-byte translation of a string through a 256-byte table, at high speed.
Representing & Manipulating Strings

Two common representations

• Explicit length field

```
8  a  b  s  t  r  i  n  g
```

Length field may take more space than a terminator

• Null termination

```
a  b  s  t  r  i  n  g  \0
```

• This issue is a language design issue
  – Are strings fixed length, or varying length?

String representation is a great case study in the way that one design decision (C, Unix) can have a long term impact on computing (security, ISA, buffer overflow). See the article on the COMP 412 Lectures page, “The most expensive one-byte mistake.”