Handling String Operators
Comp 412

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.

Chapter 7 in EaC2e
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 (for strings)

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?
Representing & Manipulating Strings

Strings are fundamentally different from scalars, arrays & records

- Fundamental unit is a character
  - Typical character sizes are one or two bytes (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, & (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

---

1 Translation in the sense of a character by character substitution based on some fixed translation table. 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.

See § 7.6 in EaC2e
Representing & Manipulating Strings

Two common representations for a string $b$

- Explicit length field
  
  $8\ a\ b\ s\ t\ r\ i\ n\ g$

- Null termination
  
  $a\ b\ s\ t\ r\ i\ n\ g\ \backslash 0$

  Length field may take more space than a terminator

- This issue is a language design issue
  - Are strings fixed length, or varying length?
  - How do we talk about storage for a string versus contents of a string?
As often happens, the implementation is slightly more complex

• The compiler and runtime system may need to know how much space is allocated to hold the string, as well as how long the actual string is

• Explicit length field

```
18 8 a b s t r i n g
```

16 is the length of the space allocated to hold the string b. Assume the length field is 2 bytes. Hence, allocated length is $16 + 2 = 18$.

• Null termination

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

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.”
Representing & Manipulating Strings

Each representation has advantages and disadvantages

<table>
<thead>
<tr>
<th>Operation</th>
<th>Explicit Length</th>
<th>Null Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>Straightforward</td>
<td>Straightforward</td>
</tr>
<tr>
<td>Checked Assignment</td>
<td>Checking is $O(1)$</td>
<td>Must count length$^1$</td>
</tr>
<tr>
<td>Length</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Concatenation</td>
<td>Must copy data</td>
<td>Count length + copy data</td>
</tr>
</tbody>
</table>

Unluckily, null termination is almost considered normal

- Hangover from the design of C
- Embedded in various OS and API designs

$^1$Checked assignment requires both a current length for the string and an allocated length for the buffer.
Encoding String Operations

**Single character assignment**

- With character operations
  - Compute address of rhs, load character
  - Compute address of lhs, store character

- With only word operations (\(> 1\) char per word)
  - Compute address of word containing rhs & load it
  - Shift character to destination position within word
  - Compute address of word containing lhs & load it
  - Mask out (\textbf{and}) current character & mask in (\textbf{or}) new character
  - Store lhs word back into place
  - \textit{This style of manipulation gets messy and complicated very quickly}
Encoding String Operations

Multiple character assignment

Two strategies:
1. Wrap a loop around the single character code, or
2. Work up to a word-aligned case, repeat whole word moves, and handle any partial-word end case

(1) Character operations
1. Easy to generate; inefficient use of resources
2. Harder to generate; better use of resources

(2) Only word operations
1. Lots of complication to generate
2. Fold complications into end case; reasonable efficiency

Source & destination aligned differently
⇒ much harder cases for word operations

Either approach requires explicit code to check for buffer overflow (⇒ length)
Input Encoding

Length Computation

- Representation determines cost
  - Explicit length turns $length(b)$ into a memory reference
  - Null termination turns $length(b)$ into a loop of memory references and arithmetic operations or, as in C, a procedure call to such a loop

- Length computation arises in other contexts
  - Whole-string or substring assignment
  - Checked assignment (buffer overflow prevention)
  - Concatenation
  - Evaluating call-by-value actual parameter or concatenation as an actual parameter

And a function call is not cheap...
Encoding String Operations

**Concatenation**

- String concatenation touches every character
  - Need to copy each character to the result string
    - Unless the operation is destructive ($b \leftarrow b || c$)
  - Often, need to compute size of result to allocate space for it

- Exposes representation issues
  - Is string a descriptor that points to text?
  - Is string a buffer that holds the text?
  - Consider $a \leftarrow b || c$
    - Compute $b || c$ and assign descriptor to $a$?
    - Compute $b || c$ into a temporary & copy it into $a$?
    - Compute $b || c$ directly into $a$?

- What about a call to `fee(b || c)`?  
  - Where does $b || c$ go?

Some of you observed that excessive string concatenation was a source of slowdown in your local allocators. That is a common problem. Program with abstractions. Measure performance. Replace the abstractions that cause performance problems.
Encoding String Operations

As a matter of safety, string operations should be checked

• String assignments and references are potentially dangerous
  – Reading beyond the end of a string can provide access to critical data
  – For example, the return address in the current AR is at a (negative?) offset from a string in the local data area.
  – Buffer overflow attack might write code, then invoke it with the return

• Checking assignments, references, & concatenations is tedious
  – Requires knowledge of the string’s allocated size
  – Adds conditional logic plus potential of an exception

• Checking assignments makes string operations even slower
  – But, compilers and runtimes need to do it

Eliminating such checks is a potential source for significant improvement

All the same statements hold true for array & structure operations

1 “Exception” might be orderly, as in Java, or disorderly, as in C (“segmentation fault”)