An Introduction to Perl
A Review, Sorting Part II and A New Problem

IT Computing Services
October 2002
Version 1.0
Where We’re Going Today...

• It’s been a while --> A Short Review

• Perl Case Study: A Non-Trivial Sort

• A New Problem --> A Cliffhanger
Previously on “Intro to Perl”...

- Scalar Variables
- Tests and Loops Part I
- Files
- Loops Part II, Arrays, Lists, Hashes and Sorts Part I
- Perl Modules and Database Access w/ DBI
Scalar Variables

• Numbers and strings (depending on context)
  Eg:  $offset=7;  $names=“Barry”;  

• Operations
  Eg:  Addition, increment by 1, string concatenation
  $position=1+ $offset;
  $i++;  $i++;
  $names= $names . “, Chris”;
  $names= $names . “, Ben”;
Arrays (Lists) and Hashes

• **Arrays** ... store scalars in consecutive “pigeonholes”. Integers are keys. *Note arrays always start at zero.*

  ```
  $apples[0]=“Apple II”;
  $apples[1]=“Lisa”;  
  ```

  When referring to an array as a whole, prefix with `@` . Eg: `@apple=();`

• **Associative Arrays** or **Hashes** ... store scalars with keys of your choosing! *(like strings)*

  ```
  $addresses{“Barry”}=“2020 Ridge”;  
  $addresses{“Frank”}=“1603 Orrington”;  
  $months{“January”}=1;  
  $months{“February”}=2;  
  ...  
  ```

  When referring to a hash as a whole, prefix with `%` . Eg: `%months=();`

  The function `keys()` returns a list of keys of a hash. For `%addresses`, it would be `Barry, Frank, ...`
Tests

- **Numerical**
  Eg: equality == , inequality != , > , >= , < , ...
  
  $\text{record\_type} == 1$
  $\text{record\_type} != 1$

- **String**
  Eg: equality eq , inequality ne, gt, ge, lt, ...
  
  $\text{string eq ""}$
Program Control Statements

• IMNSHO, there are two fundamental control statements:

  • if ... used to divert or branch the flow of action
    
    ```
    if ($username eq "Barry") {
        printf ("Welcome!\n");
    } else {
        printf ("Intruder alert!\n");
    }
    ```

  • while ... used to repeat actions
    
    ```
    $i=1;
    while ($i<=10) {
        print ("$i\n");
        $i++;
    }
    ```

• Remember to indent your programs for readability!
Other Popular Loops

- There are two other loop constructs that you’ll see often:
  - `for` ... usually used to iterate over numbers
    ```php
    for ($i=1; $i<=10; $i++) {
        print ("$i\n");
    } # this is much more compact than the while example
    ```
    ```php
    for ($i=1; $i<=10; $i++) {
        print ($apples[$i]."\n");
    }
    ```
  - `foreach` ... used to iterate over strings
    ```php
    foreach $apple (@apples) {
        print ($apple."\n");
    }
    ```
    # this is more concise than the prev fruit example
File I/O

• Reading Files
  ```php
  open (INFILE, " < infile.txt") || die ("can’t read");
  # open file for writing or quit with error
  while (<INFILE>) {  # loop to read file
    $input=$_;
    # save temporary variable
    chop $input;
    # remove newline
    # ...  
  }
  close INFILE;  # close file
  ```

• Writing Files
  ```php
  open (OUTFILE, " > outfile.txt") || die ("can’t write");
  print (OUTFILE “blah blah blah!
”);
  close OUTFILE;
  ```

• Always close your open files!

file handle
note: no comma!
Other Things We’ve Covered

• Modules and
  Database Access with DBI

*If you’re interested, be sure to review those slides on the website or ...*

*feel free to ask me about it.*

*Thanks.*
Simple Sorts

- We can do a simple sort of an array:
  ```perl
  @sorted_apples = sort (@apples);
  foreach $apple (@sorted_apples) {
    print ("$apple\n");
  }
  ```

- We can sort the keys of a hash:
  ```perl
  @keys = keys (%addresses);
  @sorted_keys = sort (@keys);
  foreach $person (@sorted_keys) {
    print ($person . "":"" . $addresses{$person} . "\n");
  }
  ```
  or simply
  ```perl
  foreach $person (sort (keys (%addresses))) {
    print ($person . "":"" . $addresses{$person} . "\n");
  }
  ```

You can eliminate the temporary arrays for the first example too...
The Mission, Should you ...

- Student Invoices need to be printed!

- There a huge file with millions of records that all need to be sorted for printing the subrecords in the proper order and for mailing.

- This is not a typical sort of a flat text data file that most UNIX people encounter. This file has parent records and children records that follow the parent around as the file gets sorted.
The Record Definitions

- See *Attachment 1* for the definitions.
- Note that the data is organized into parent and children records.
- Since the purpose of this is to print invoices to mail to students, extra fields (like zip codes) have been prepended to the beginning of each record. But the original record definition doesn’t show this. It’s likely that the extra prepended fields will be removed later.

Anyway, this creates a funny offset situation.
The Parent Record Sorting Key

- Parent records are of type 1.
- The Parent Records were explicitly stated to be sorted on Group, ZipCode and Addresses.
- There is no mention about how addresses are to be sorted. Street and then number? Due to the inconsistencies in data entry, the specifics were never worked out, but addresses stayed in the parent sorting key.
- Anyway, the address is still useful in helping to keep the records unique.

The SSN was thrown in for uniqueness. Eg: Two siblings with the same address!
The Child Record Sorting Key

- Sort children records by record type first: 2, 3, 4
- Record type 2 sorts on Term and Post Date ---> Eg: FA0102 05242002
- But they are in wrong arrangement to be sorting key, must rearrange.
- Seasons don’t sort right. Must fix.
- What to do about Term & Post Date for Non-Type-2 Records?
  Insert a dummy sorting key.
- Padded line number thrown in for uniqueness
  Eg: 000000000001 otherwise, records with same Term and Post date get lost.
Assumptions

• Assumed that SSNs are unique.
• Child records are always follow the parent in a contiguous fashion.
• Data is Y2K OK. No years earlier than 2000 like 1999 in which the strings 99 and 00 sort funny (unexpectedly).
• Assume have no more then $10^{12}$ parent and child records total. That’s 12 digits!
• Double underscore characters __ are never found in the data. They’ll be used as data separators.
Before We Begin...

• There is an alternative sorting solution; and it’s actually a “shorter” solution.

• You will find attached it as Attachment 3.

• All the data is thrown into one huge array and sorted as one huge array whereas the solution I’m talking about today uses two arrays, one for parent records and one for children records.

• Remember, there is no one “correct” answer.

• We can discuss tradeoffs at the end if you want...
Diversion: The Finite State Machine

- A “fictitious machine” that computer scientists create to gobble up and process input data.

- They’re kinda like flowchart diagrams. Each circle is a specific state or action in a program and the arcs are decisions that determine the next. All arcs are deterministic!

This example prints \( a \) records and discards \( b \) records.
The View From 10,000 Feet

• A “finite state machine” for my sort:

```
start
  init
  rec type = 1
  read input
  rectype != 1
  is first
    rectype1
    is first
      rectype1
      eof
      read input
      rec type = 1
      extract parent key fields from parent record and insert into %parent_records hash
      extract child key fields of child record & insert into %temp_child_records hash
      end
end
```
The View From 10,000 Feet - Part 2

- A “finite state machine” for my sort:

And now on to the details ...
The Data Structures I Use

• There are three hashes that I use:
  
  • %parent_records
    This record stores the parent record under the parent’s record sorting key.
  
  • %child_records
    This record stores all the child records under their parent’s record sorting key. Note that the child records are stored in sorted order separated by double underscores __.
  
  • %temp_child_records
    Before all the child records are read in, they can’t be sorted yet. Child records that haven’t been sorted yet are stored here under the child’s key.
My Favorite String Extraction Function

- The function `substr()` extracts a substring from another string.
- Eg:

  ```php
  $a = substr("abcdefg", 0, 5);  // $a is abcde
  $b = substr("abcdefg", 3, 3);  // $b is def
  ```

Strings start at character position zero for `substr()`!
Who likes “Magic numbers”?  

- I hate reading a program that has the number 42 hardcoded into it and wonder, “Where the heck did that number come from?”  
- And if record definitions change, I also find changing the variables in one place so much easier!

```php
$group_code_pos=0; $group_code_len= 2;
$zip_code_pos=2; $zip_code_len= 5;
$offset= $group_code_len+ $zip_code_len;

$ssn_pos= 0+ $offset; $ssn_len= 9;
$record_type_pos= 9+ $offset; $record_type_len= 1;
$addr_line1_pos= 52+ $offset; $addr_line1_len= 30;
$addr_line2_pos= 82+ $offset; $addr_line2_len= 30;

$term_pos= 16+ $offset; $term_len= 6;
$post_date_pos= 10+ $offset; $post_date_len= 8;

$term_ss_pos= 10+ $offset; $term_ss_len= 2;
$term_y1y2_pos= 16+ 2+ $offset; $term_y1y2_len= 4;
$post_date_mm_pos= 16+ $offset; $post_date_mm_len= 2;
$post_date_dd_pos= 16+ 2+ $offset; $post_date_dd_len= 2;
$post_date_yyyy_pos= 16+ 4+ $offset; $post_date_yyyy_len= 4;
```

I also take into account that funny little offset.
Collecting Parent Records

- We just extract the key fields for the parent out of its record and insert the entire record with its key into the %parent_records hash.

And we mark having seen the first parent record ...

```php
if ($record_type == 1) {
    $parent_sort_key =
        substr ($input, $group_code_pos,
                $group_code_len + $zip_code_len).

    substr ($input, $addr_line1_pos,
            $addr_line1_len + $addr_line2_len).

    substr ($input, $ssn_pos, $ssn_len);

    $parent_records[$parent_sort_key] = $input;
}

$first_parent_record = 0;
```
Dummy Keys for RecTypes 3&4

- Only child records of type 2 pay attention to the Term and Post date.
- Other record types don’t pay attention to them for sorting, but I have to stick something in their place to unify the children in their sort.
- So at the beginning of the program, I create a dummy key to be used later for those records that don’t involve these fields.

```php
$dummy_record2_sorting_key_len= $term_y1y2_len+ $term_ss_len+ $post_date_yyyy_len+ $post_date_mm_len+ $post_date_dd_len;
for ($i=1; $i<=$dummy_record2_sorting_key_len; $i++) {
    $dummy_record2_sorting_key= $dummy_record2_sorting_key . " ";
}
```
Hashes for Dealing with Seasons

• Remember, seasons don’t sort right? We fix it here.
• Part of the Term field is sorted differently from the traditional ASCII sort of their strings.
• That is, the traditional sort order would be FA, SP, SU and WI. But, the intended sort is: FA, WI, SP and SU which stand for the seasons.
• I chose a hash to store the “arbitrary” sort order:

```javascript
$term_season{"FA"}=1;
$term_season{"WI"}=2;
$term_season{"SP"}=3;
$term_season{"SU"}=4;
```

which I use later on in the sort.
Numbers & Padding Them

• We’ve dealt with `print()` ing before and I’ve said that `print()` and `printf()` are the same. Not true.
• You can print formatted strings with `printf()`

```php
$a=1;
printf("%0.3d\n", $a);
```

this is a number ... decimal
pad this many characters ... 3

would print 001 and a new line.
Numbers & Padding Them 2

• You can print formatted text into strings with sprintf().

        $a= \text{sprintf(“%0.3d\n”, 5)}; 

and now $a$ will have the value of “005\n” in a string context.

• We will use this padding of numbers to pad line numbers in the sort.

• Man printf() and sprintf() on any UNIX system for more specifics.
Collecting Child Records

- We just extract the key fields for the child out of its record for type 2 records, otherwise create a dummy key record.

```php
if ($record_type!=1) {
    if ($record_type==2) {
        $record_type2_sorting=
        substr ( $input, $term_y1y2_pos, $term_y1y2_len ) .

        $term_season{substr ( $input, $term_ss_pos, $term_ss_len )} .
        substr ( $input, $post_date_yyyy_pos, $post_date_yyyy_len ) .
        substr ( $input, $post_date_mm_pos, $post_date_mm_len ) .
        substr ( $input, $post_date_dd_pos, $post_date_dd_len ) ;
    } else {
        $record_type2_sorting=$dummy_record2_sorting_key;
    }
    ...
}
```

- Field notations:
  - term
  - post date
  - yyyy
  - mm
  - dd
  - y1
  - y2
  - s

- Record type:
  - type 2
  - non-type 2

- Dummy record key:
  - $dummy_record2_sorting_key
Collecting Child Records

• We insert the record into the %temp_child_records hash with the key we just created.

• if ($record_type!=1) {
  # create child record key... blah blah blah
  ...

  $child_sort_key= $record_type . $record_type2_sorting . $line_number_padded;

  $temp_child_records{$child_sort_key}= $input;

}
Final Destination of Child Records,
\[ \% \text{child\_records} \]

- All the child data records, like the parent data records, are stored in a hash with the parent’s key.
- However, since there is potentially more than one child record, all the child data records are concatenated together in sorted order. They are separated by double underscores \( __ \).

\[
\begin{array}{c}
\text{key} \\
\rightarrow \\
\text{a6}\_\text{a9}\_\text{d5}\_\text{d6}\_\text{d7}\_\text{e6}
\end{array}
\]

And they’ll be “unrolled” later when printing out the sorted list.
Assembling the Child Records

```plaintext
    if ($record_type==1 && $first_parent_record==0) {

        foreach $child_record_key (sort(keys(%temp_child_records))) {

            if ($child_records{$parent_sort_key} eq "") {
                $child_records{$parent_sort_key} = $temp_child_records{$child_record_key};
            } else {
                $child_records{$parent_sort_key} = $child_records{$parent_sort_key} . "__" . $temp_child_records{$child_record_key};
            }

        }

        %temp_child_records=();
    }
```

- sort all `%temp_child_records` keys and loop through them to ... 
- ... insert those records
- case of first record
- case of next record; use "__" separator
- remove all temporary `%temp_child_records` entries
When to Assemble Child Recs

- You know that you’ve collected all the child records when you come across another parent record.

- So, every time you come across a new parent record, assemble all the previous child records.

- Every time... except the first time! There’s nothing the sort from beforehand on the first parent record!
Printing Out Collected Records

• Sort the parent records and for each of the parent records, ...

• Pull up the sorted child records and ...

• Expand the child records with `split()` and print them in order.

  ``` perl
  foreach $parent_key (sort(keys(%parent_records))) {

    print (OUTFILE subst($parent_records{$parent_key}, 16, 1) . " " . $parent_records{$parent_key} . "\n");
    foreach $child_record (split(/__//, $child_records{$parent_key})) {
      print (OUTFILE subst($child_record, 16, 1) . " " . $child_record . "\n");
    }
  }
  ```

• I assume that I’m writing to a file using a file handle `OUTFILE`. 
Analysis

• The completed program is Attachment 2.

• **Pros of this Design**
  - Since the child records depend on and follow the parent records, the sort is really a smaller problem.
  - Minimizes the hash and hopefully memory req.
  - Minimizes the number of keys to sort.
  - Many of the records are “presorted” which may not be important depending on your sorting algorithm.
  - May actually run faster than a program that sorts all the records all at once.

• **Cons of this Design**
  - More data structures
  - Program is more complicated.
  - ...
Problems with Child Record Assembly

- Notice that child records are sorted from `%temp_child_records` and inserted into `%child_records` only when the program comes across a parent record.

- What happens to the last line? Its children records won’t get inserted.
Duplicated Code?

• We have to duplicate that code outside of the `while` loop. This is do-able, but ...

• That creates opportunities for bugs if someone changes one part of the program and forgets to change the other part of the program.

• This is a problem for subroutines, `next time` ...
Questions?

Thank you!