CS3157: Advanced Programming

Lecture #10

June 26

Shlomo Hershkop
shlomo@cs.columbia.edu

1

Outline

- wrap up CPP
  - templates
  - file handling
- Reading:
  - chapters 20
- Advanced Topics
- Review for final

2

Announcements

- wrapping up cpp today
- Advanced Topics
- Please fill out the course evaluations on course works….consider this part of your final

3

development question

- is there a different if you code in visual cpp vs gcc in cygwin if you are working on windows ??
- when would it matter ?

4

virtual functions

- in C++ virtual functions allow you to define a specific function in the base class, which is undefined, and each of the subclasses need to override (implement a definition)

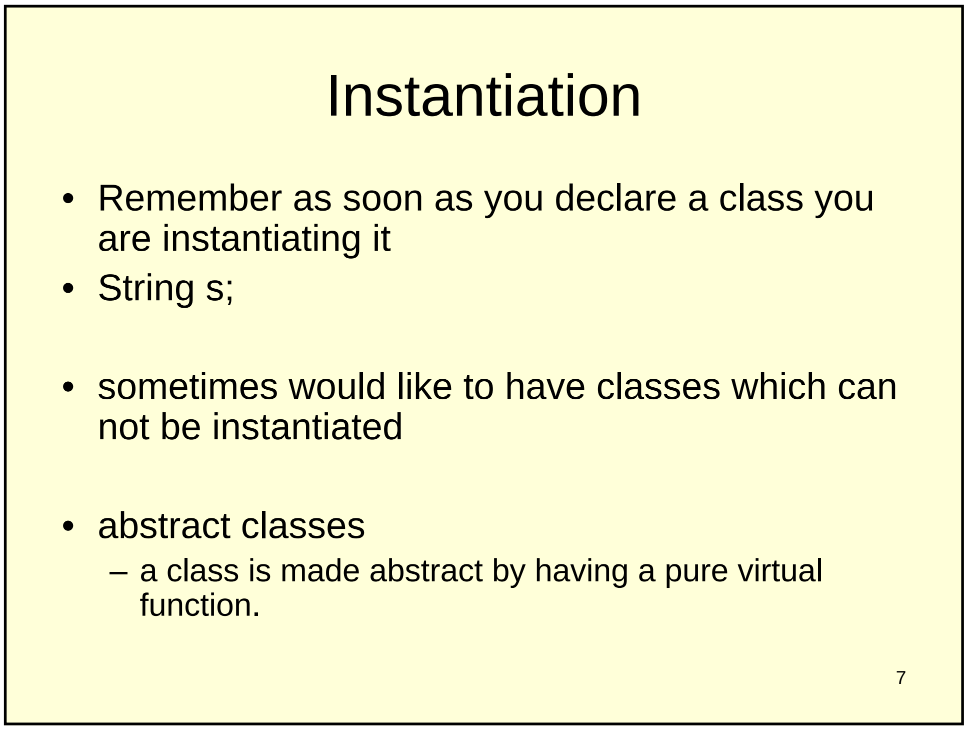
- virtual char * md5sum();

5

- so if we use a base class pointer at a derived class object, calling md5sum will call the correct one

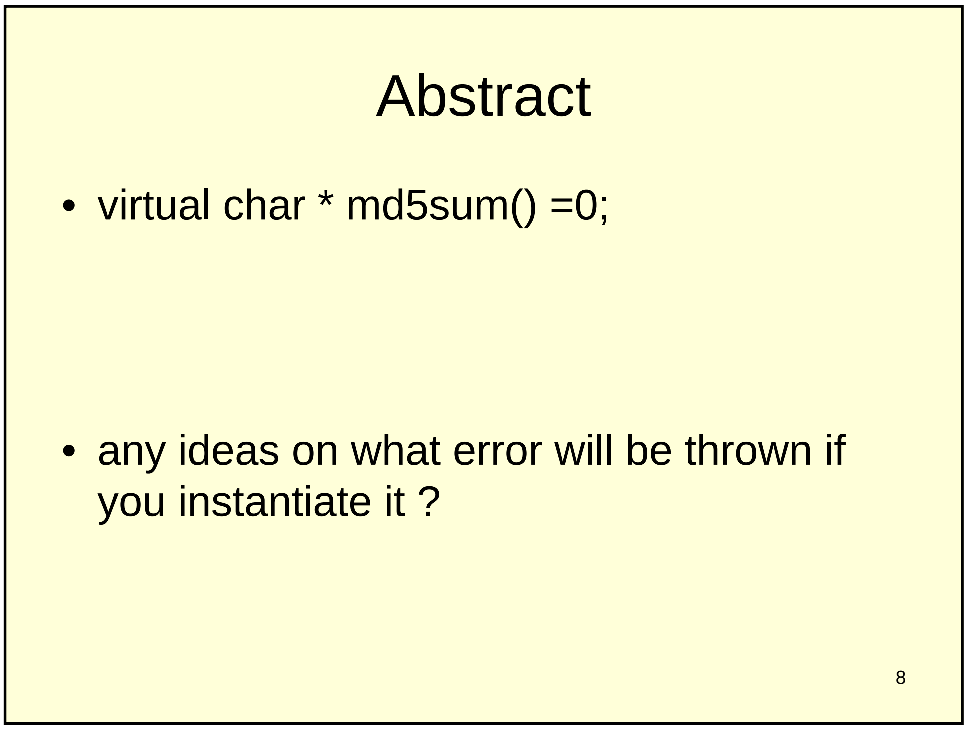
- compile time resolution
  – static binding

6



Instantiation

- Remember as soon as you declare a class you are instantiating it
- String s;
- sometimes would like to have classes which can not be instantiated
- abstract classes
  - a class is made abstract by having a pure virtual function.



Abstract

- virtual char * md5sum() =0;
- any ideas on what error will be thrown if you instantiate it ?

non virtual base functions

- if you have a parent class A.foo()
- derived class B defines B.foo()

- A *a_ptr = B_object

- a_ptr.foo()
  - which foo will be triggered?
  - why ?

9

abstract classes II

- remember that making a pointer doesn’t instantiate anything
- can create pointers of type abstract classes
- used to enable polymorphic behavior

- Example: Operating system device
  - read/write behvaior

10

destructors

- when creating and manipulating objects in a polymorphic context, destructors will only be called on base class

11

solution

- define a virtual base class destructor
- will correct destructor will be called

12

- Example 20_1

13

- here is another set of examples

14

Abstraction with member functions

- example #1: array1.cpp
- example #2: array2.cpp
  - array1.cpp with interface functions
- example #3: array3.cpp
  - array2.cpp with member functions
- class definition
- public vs private
- declaring member functions inside/outside class definition
- scope operator (::)
- this pointer

15

array1.cpp

```
struct IntArray {
  int *elems;
  size_t numElems;
};
main() {
  IntArray powersOf2 = { 0, 0 };
  powersOf2.numElems = 8;
  powersOf2.elems = (int *)malloc( powersOf2.numElems *
   sizeof( int ));
  powersOf2.elems[0] = 1;
  for ( int i=1; i<powersOf2.numElems; i++ ) {
    powersOf2.elems[i] = 2 * powersOf2.elems[i-1];
  }
  cout << "here are the elements:\n";
  for ( int i=0; i<powersOf2.numElems; i++ ) {
    cout << "i=" << i << " powerOf2=" <<
   powersOf2.elems[i] << "\n";
  }
  free( powersOf2.elems );
}
```

16

array2

```
void IA_init( IntArray *object ) {
  object->numElems = 0;
  object->elems = 0;
} // end of IA_init()

void IA_cleanup( IntArray *object ) {
  free( object->elems );
  object->numElems = 0;
} // end of IA_cleanup()

void IA_setSize( IntArray *object, size_t value ) {
  if ( object->elems != 0 ) {
    free( object->elems );
  }
  object->numElems = value;
  object->elems = (int *)malloc( value * sizeof( int ));
} // end of IA_setSize()

size_t IA_getSize( IntArray *object ) {
  return( object->numElems );
} // end of IA_getSize()
```

17

hierarchy

- composition:
  - creating objects with other objects as members
  - example: array4.cpp
- derivation:
  - defining classes by expanding other classes
  - like “extends” in java
  - example:

```
class SortIntArray : public IntArray {
public:
void sort();
private:
int *sortBuf;
}; // end of class SortIntArray
```

- “base class” (IntArray) and “derived class” (SortIntArray)
- derived class can only access public members of base class

18

- complete example: array5.cpp
  - public vs private derivation:

- public derivation means that users of the derived class can access the public portions of the base class

- private derivation means that all of the base class is inaccessible to anything outside the derived class

- private is the default

19

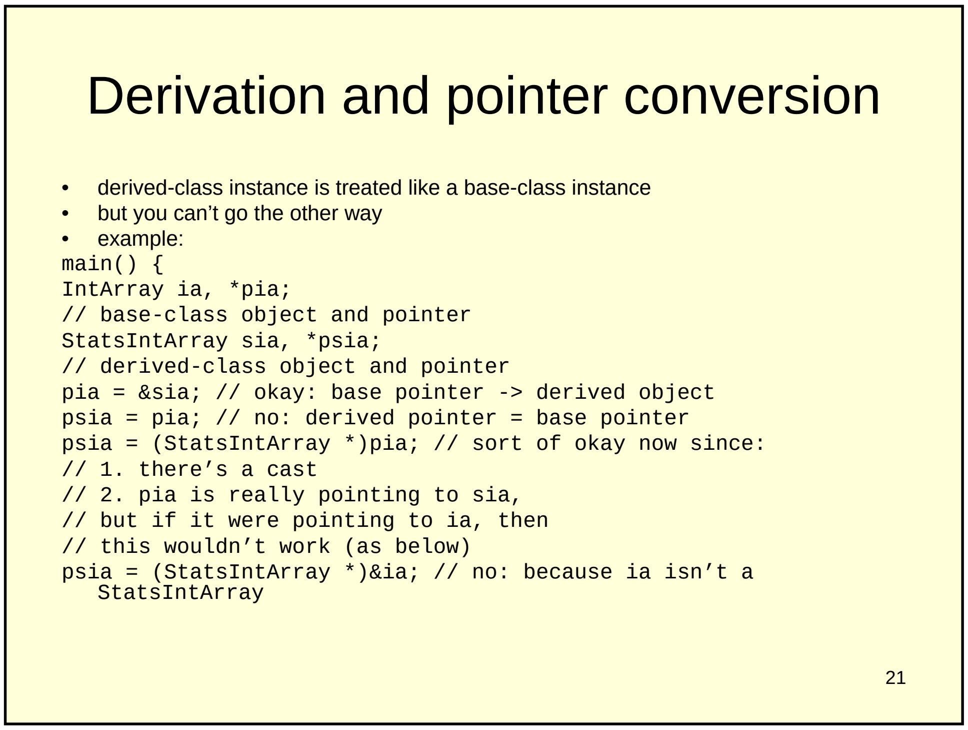
Class derivation

- encapsulation
  - derivation maintains encapsulation
  - i.e., it is better to expand IntArray and add sort() than to modify your own version of IntArray

- friendship
  - not the same as derivation!!
  - example:

- is a friend of
- B2 is a friend of B1
- D1 is derived from B1
- D2 is derived from B2
- B2 has special access to private members of B1 as a friend
- But D2 does not inherit this special access
- nor does B2 get special access to D1 (derived from friend B1)

20

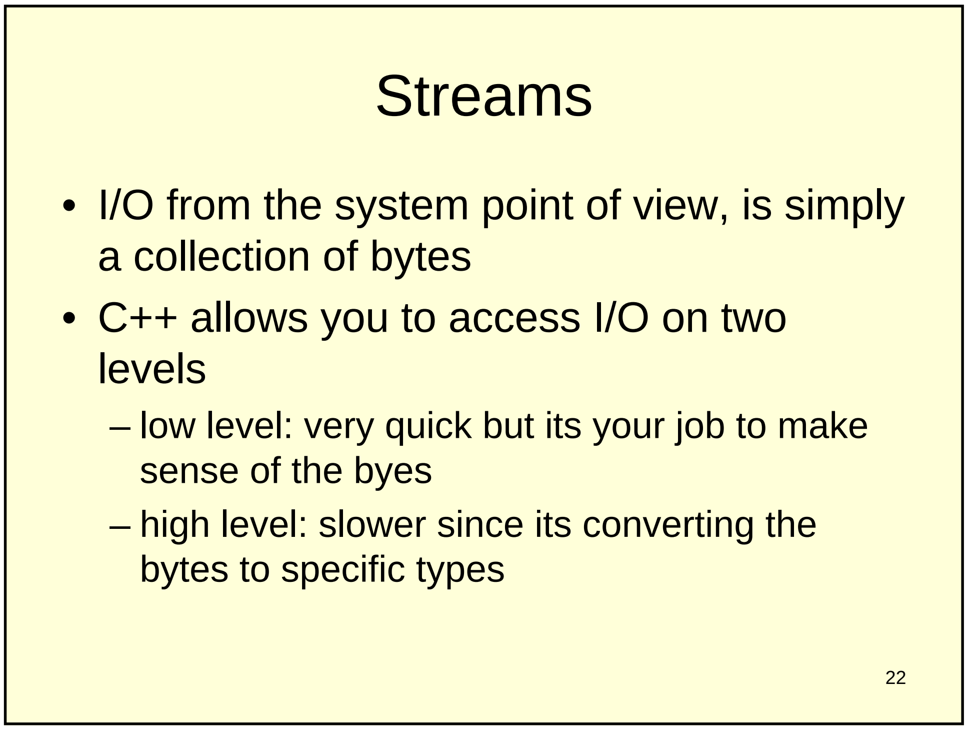


Derivation and pointer conversion

- derived-class instance is treated like a base-class instance
- but you can’t go the other way
- example:

```
main() {
IntArray ia, *pia;
// base-class object and pointer
StatsIntArray sia, *psia;
// derived-class object and pointer
pia = &sia; // okay: base pointer -> derived object
psia = pia; // no: derived pointer = base pointer
psia = (StatsIntArray *)pia; // sort of okay now since:
// 1. there’s a cast
// 2. pia is really pointing to sia,
// but if it were pointing to ia, then
// this wouldn’t work (as below)
psia = (StatsIntArray *)&ia; // no: because ia isn’t a
   StatsIntArray
```

21



Streams

- I/O from the system point of view, is simply a collection of bytes
- C++ allows you to access I/O on two levels
  - low level: very quick but its your job to make sense of the byes
  - high level: slower since its converting the bytes to specific types

22

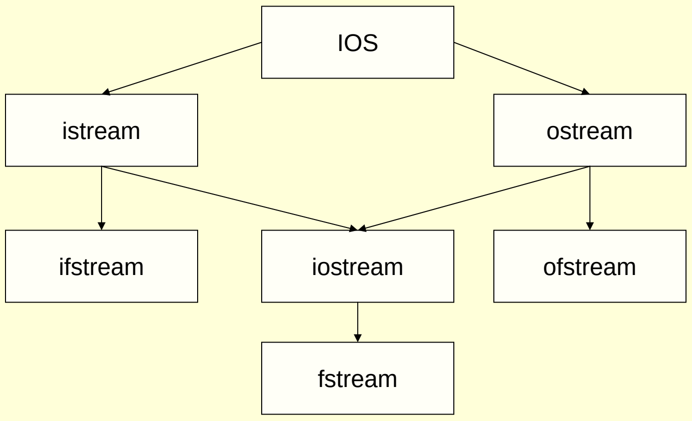
libraries

- <iostream>
  - cin
  - cout
  - cerr (unbuf)
  - clog (buf)
- <istream>
- <ostream>
- <iomanip>

23

fstream

- for file processing C++ uses ifstream and ofstream objects to read/write from files


$$\begin{tikzcd}
 & \text{IOS} \arrow[dl] \arrow[dr] & \\
\text{istream} \arrow[d] \arrow[dr] & & \text{ostream} \arrow[d] \arrow[dl] \\
\text{ifstream} & \text{iostream} \arrow[d] & \text{ofstream} \\
 & \text{fstream} &
\end{tikzcd}$$

24

- since formatting is automatically figured out for you, how do you print out the pointer address (i.e. %p) using cout ??

25

casting

- casting is your friend
- << static_cast<void *>(str_ptr) << endl;

26

- cout.put('a');
  - will print single character to the output stream
- can cascade

- cout.put('a').put('b').put('c');

27

Examples

- 21_10

- 21_11

28

get

- one character at a time
- even whitespaces
- EOF

- different version of get

- 21_12
- 21_13

29

getline

- read till delim
- throws it out
- replaces it with \0

- 21_14

30

low level

- low level read/write
- 21_15

31

stream formatting

- in c it's a headache and half to format the output
- c++ has lots of help in steam formatters
- 21_16
- 21_17

32

- any ideas when formatting can make/break your program ???

33

- example: telnet checkers program ☺

34

Templates

- the idea is to generalize your code as much as possible to deal with any class you will be handling
- can allow your classes to store any type correctly (think void pointer)
- allows a function to adopt to specific constraints

35

Templates

```
template<typename X>
void foo(X &first, X second){
  first += second;
  }
```

36

- 22_1

37

templates

- can also take non general arguments
- template<class T, int x> …
- integrates into inheritance
- allows friendly relationships

38

STL

- standard template library
- tons of useful stuff here
- if you do any serious programming you should consider STL
  - they've worked out all the bugs ☺
  - very efficient
  - make sure you understand what you are doing

39

static members

- `array<float> X;`
- `array<int> Y;`
- if there are static variable members each gets separate copy while
- `array<float> x2`
  - will share with above X

40

responding to errors

- part of good object oriented programming is anticipating problems

- what can we do if something goes wrong ??

41

- try
- catch

- similar to java
  - except won't escalate if not caught unless specifically enclose in try->catch blocks
  - general exception class (<exception> and std::exception)

42

- 23_2

43

wrap up

44

More than just reacting

- We have been working with perl/c/cpp in a static context
- Some information is presented to the user
- React to user input

- Is this how google maps works ?

45

Ajax

- **A**synchronous **J**avaScript **A**nd **X**ML

- technique for developing interactive applications over the web

- Style
- Platform
- Format
- XMLHttpRequest
- Objects

46

Basic HTML

- Specific set of tags (depends on version)
- Up to user to set things correctly
- Most browsers will attempt to figure out what you want
  - Example not putting end body end html, will still work

47

Advanced HTML

- CSS
  - Cascading style sheets
    - Define format of the WebPages
    - Central location of style
    - With a few clicks can completely change thousands of WebPages.
- DOM
  - Document object model
    - Formal data structure to represent web based documents/information
- Client side scripting

48

DOM problems

- Different browsers supported things differently

```
if (document.getElementById &&
   document.getElementsByTagName) {
 // as the key methods getElementById and getElementsByTagName
 // are available it is relatively safe to assume W3CDOM support.

   obj = document.getElementById("navigation")
   // other code which uses the W3CDOM.
   // .....
}
```

49

Examples

- http://www.dynamicdrive.com/dynamicindex12/pong/pong.htm
- http://www.dynamicdrive.com/dynamicindex4/butterfly.htm

50

javascript

- Client side
  - PHP & CGI were both server side
- Developed under Netscape as LiveScript
  - Currently 1.5
- Developed under IE as Jscript
- Object oriented approach
- Syntax like c
  - No input/output support native
  - Keywords
  - DOM for interfacing between server and client
- Can evaluate reg expressions (eval)

51

Javascript

- Heavy use of defined functions
  - Example: MouseOver
- Need to adopt to specific browser if doing anything fancy
- Adobe
  - Support javascript in pdf
- MAC
  - Dashboard widgets

52

Programming

- You need to learn on your own
- Many good books/websites
- Most of the time .js file if not in html
- Powerful example:
  - Thunderbird/firefox
- Get good debugger

53

How to do research?

- Practical research
  - Know a programming language
  - Have an inquisitive mind
  - Keep an open mind to new ideas
  - Try to solve an open research problem ☺
- Theory research
  - Learn some math
  - Learn some theory
  - Relearn the math
  - Solve something ☺

54

Where to start?

1. Need an idea
2. See if anyone’s done it or tried it in your way
   1. Citeseer (citeseer.ist.psu.edu)
   2. Google
   3. Appropriate Faculty/Researcher
   4. Google groups

55

Sketch out the idea on small scale

- Design a small experiment which can validate your idea
- Data, data, data, and Data
  - Make or break you
  - Will help your research
    - Make sure it isn’t a circular relationship
- Evaluate results
  - Don’t fake them
  - Even bad results are results
  - Can learn of what not to do
- Write up results

56

Write up options

- Word vs Latex
- gnuplot
- cvs
- Element of Style

57

In the real world

1. Keep it simple
   1. Don’t re-invent the wheel
   2. Design first
   3. Even with fancy blinking lights, a bad idea is still a bad idea (but with bad taste)
2. Incremental testing
   1. Recognize when the bug is your fault
   2. See if others have faced it too
3. Make sure version 1 works on most popular browsers

58

Question

- What is this designed with?
- Can you do a better job?

- Theyrule.net

59

Bottom line

- We've covered a lot this semester
  - Some of it was fun
  - Some of it was hard work (ok most)
  - Some of it was frustrating.
- BUT
  - You have lots of tools
  - Have an idea of where to start when dealing with programming projects

60

Important lessons for learning new languages

- CS is not meant to be a trade school
- Language isn't important…things change
- Ideas and design are more important

- Lessons:
  - Choose correct environment
  - Choose correct tools
  - Make sure to test out ideas…might be someone else's fault (program think)
  - Enjoy what you are doing

61

Important

- To get the most out of a language find comfortable programming environment

- Emacs – color files
- Eclipse
- Others , see
  - www.freebyte.com/programming/cpp/

62

Review time

- Perl
- C
- CPP
- Shell programming stuff
- misc stuff

- Review the labs/hw

63

Perl related stuff

- basics on types
- regular expressions
- perl file handling
- perl modules
- perl classes
- cpan.org

64

Word list

- Compiling
- Linking
- Reference parameter
- Variable scope
- Stdio.h
- Stdlib.h
- cout
- cast
- Inline
- Linked list
- Preprocessor
- Typedef
- Struct
- Pointer
- Void pointer
- . Vs ->
- Function pointer
- Reference
- const
- malloc

65

Word list II

- Huffman
- getopt
- constructor
- destructor
- iostream
- overloading
- extern
- private
- Public
- GDB
- Cgi
- GET/POST
- overload
- overriding
- Template
- This
- Friend class
- New/delete
- virtual

66

C

- Basic constructs
- Basic type
- Advanced types
- (review labs and class examples)
- Memory stuff – understand what is happening
- Arrays
- Functions
- Pointers
- Debuggers

67

C

- Working with CGI
- Working on different platforms
- Makefiles
- How we built libraries

68

C++

- Basic language
- Difference to c
- Classes
- Permissions
- new/free memory allocations
- Inheritance and polymorphism
- Keywords
- Working with files….

69

Sample exam

- You’ve done most of the work for the course, the exam is just to make sure you remember the important concepts
- posted online

- Couple Definitions
- 2 code checking question
- Shell code question
- C++ class manip question
- Small CGI question

70

Thinking question

- Say you are writing code which uses a random number generator….
- What is important to know about it ?
- How can your code be affected ?
- If you crash, how to reconstruct events, since based on random numbers ??

71

Closing Remarks

- If you like this…..just the beginning
- If you didn't ….. You now know how complicated it is….never trust a program ☺
- Hope you had a fun semester..

72

study tips

- go through class notes
- go through lab assignments
- make sure you understand it all, email/aim questions….
  - please don't save it for 10 minutes before the exam

73

- Good Luck!
- will be open books
- see you Wednesday!
- reminder: please go on coursework and fill in the course evaluation
- if you need more time for assignments…contact me

74

- time permitting

75

- switch back to perl…….

76

Outline for next section

- Code review
- Optimization
- Caching
- Memorization
- Profiling for optimization
- HTML parsers

77

Benchmarking

- In many cases during development, you will have different options for choosing how to code your ideas
- would like to know which choice would run faster

78

Simple idea

```
#!/usr/bin/perl

# declare array
my @data;

# start timer
$start = time();

# perform a math operation 200000 times
for ($x=0; $x<=200000; $x++)
{
      $data[$x] = $x/($x+2);
}

# end timer
$end = time();

# report
print "Time taken was ", ($end - $start), " seconds"
```

```
#!/usr/bin/perl

use Benchmark;

# declare array
my @data;

# start timer
$start = new Benchmark;

# perform a math operation 200000 times
for ($x=0; $x<=200000; $x++)
{
      $data[$x] = $x/($x+2);
}

# end timer
$end = new Benchmark;

# calculate difference
$diff = timediff($end, $start);

# report
print "Time taken was ", timestr($diff, 'all'), " seconds"; [/code]
```

```
#!/usr/bin/perl

use Benchmark;

# run code 100000 times and display result
timethis(100000, '
      for ($x=0; $x<=200; $x++)
      {
            sin($x/($x+2));
      }
');
```

81

- so timethis takes anything an eval would take
- tells you exactly how much time it took to compute x iterations

- how about the other way, say I have 5 minutes to calculate an answer and I want to see how many iterations I can do ?

82

```
#!/usr/bin/perl

use Benchmark;

# run code for 10 seconds and display result
timethis(-10, '
      for ($x=0; $x<=200; $x++)
      {
            sin($x/($x+2));
      }
');
```

83

- so how would you turn this into an interactive script ?

84

```
#!/usr/bin/perl

# use Benchmark module
use Benchmark;

# ask for count
print "Enter number of iterations:\n";
$count = <STDIN>;
chomp ($code);

# alter the input record separator
# so as to allow multi-line code blocks
$/ = "END";

# ask for code
print "Enter your Perl code (end with END):\n";
$code = <STDIN>;

print "\nProcessing...\n";

# run code and display result
timethis($count, $code);
```

85

multiple tests

- usually if you want to compare want to be able to run a bunch of different tests and compare their results

86

```
#!/usr/bin/perl

# use Benchmark module
use Benchmark;

# time 3 different versions of the same code
timethese (1000, {
     'huey' => '$x=1;
               while ($x <= 5000)
                    {
                         sin ($x/($x+2));
                         $x++;
                    }',
     'dewey' => 'for ($x=1; $x<=5000; $x++)
                    {
                         sin ($x/($x+2));
                    }',
     'louie' => 'foreach $x (1...5000)
                    {
                         sin($x/($x+2));
                    }'
});
```

```
•   Benchmark: timing 1000 iterations of dewey, huey, louie...
•         dewey: 92 wallclock secs (91.72 usr +  0.00 sys = 91.72 CPU) @
    10.90/s (n=1000)
•          huey: 160 wallclock secs (159.56 usr +  0.00 sys = 159.56 CPU) @
    6.27/s (n=1000)
•         louie: 45 wallclock secs (44.98 usr +  0.00 sys = 44.98 CPU) @
    22.23/s (n=1000)
```

- sometimes want to get percentage comparisons

89

```
#!/usr/bin/perl

# use Benchmark module
use Benchmark qw (:all);

# time 3 different versions of the same code cmpthese (100, {
      'huey' => '$x=1;
                  while ($x <= 5000)
                  {
                        sin ($x/($x+2));
                        $x++;
                  }',
      'dewey' => '      for ($x=1; $x<=5000; $x++)
                  {
                        sin ($x/($x+2));
                  }',
      'louie' => '      foreach $x (1...5000)
                  {
                        sin($x/($x+2));
                  }'
});
```

90

- ok, lets run a quick test to compare 2 ways of doing the same thing in perl:

91

Version 1

```
my $string =
  'abcdefghijklmnopqrstuvwxyz';
my $concat = '';

foreach my $count (1..999999)
{
    $concat .= $string;
}
```

92

Version 2

```
my $string = 'abcdefghijklmnopqrstuvwxyz';
my @concat;

foreach my $count (1..999999)
{
    push @concat,$string;
}
my $concat = join('',@concat);
```

93

Optimization tips

- Most optimization can be done by simply paying attention to how you decided to code your ideas
- knowing some basic background information ☺

94

- any ideas why this is very slow ?

```
foreach my $item (keys %{$values})
{
    $values->{$item}->{result} = calculate($values-
   >{$item});
}

sub calculate
{
    my ($item) = @_;
    return ($item->{adda}+$item->{addb});
}
```

95

inlining it

```
calculate_list($values);

sub calculate_list
{
    my ($list) = @_;
    foreach my $item (keys %{$values})
    {
        $values->{$item}->{result} = ($item->{adda}+$item-
   >{addb});
    }
}
```

96

Loops

- try not to do the same work twice
- try to avoid looping more than once
- keep loop operations within the current variable range
- keep code local, avoid sub jumps

97

hashes

- if you will be looking through all values of the hash, its faster to call values
- but this is a constant ref, so you wont be able to delete

98

sort

- can tell the sort routine how to compare data structures but should be careful of the following :

99

```
my @marksorted = sort {
   sprintf('%s%s%s',
       $marked_items->{$b}->{'upddate'},
       $marked_items->{$b}->{'updtime'},
       $marked_items->{$a}->{itemid})
<=>
    sprintf('%s%s%s',
       $marked_items->{$a}->{'upddate'},
       $marked_items->{$a}->{'updtime'},
       $marked_items->{$a}->{itemid}) } keys
   %{$marked_items};
```

100

- anyone have an idea how to improve this ?

101

```
map { $marked_items->{$_}->{sort} =
   sprintf('%s%s%s',
      $marked_items->{$_}->{'upddate'},
      $marked_items->{$_}->{'updtime'},
      $marked_items->{$_}->{itemid}) }
   keys %{$marked_items};

my @marksorted = sort {
   $marked_items->{$b}->{sort}
   <=>
   $marked_items->{$a}->{sort}
}keys %{$marked_items};
```

102

multiple choices

```
if ($userchoice > 0)
{
    $realchoice = $userchoice;
}
elsif ($systemchoice > 0)
{
    $realchoice = $systemchoice;
}
else
{
    $realchoice = $defaultchoice;
}
```

103

```
$realchoice = $userchoice || $systemchoice ||
   $defaultchoice;
```

104

caches

- what are they
- some background
- how are they used in operating systems

105

Memoization

- cache swaps time for space
  - want a speedup, so willing to use up memory to achieve it
  - wrap our function to do lookup first before actually calling it
  - not always appropriate:
    - time
    - rand

106

pure function

- any function whose return is based on its input

107

bench marking

- lets take the regular fib vs memoized fib

- use Memoize;
- memoize fib2;

108

another version of fib3

```
{ my @cache;
   BEGIN { @cache = (0,1);
   sub fib3 {
      my $n = shift;
      return $cache[$n] if defined $cache[$n];

   return $cache[$n] = fib($n-1) + fib($n-2);

   }

}
```

109

actual code

- http://perl.plover.com/MiniMemoize/numberedlines.html

110

Code tips

- Here are some general code writing tips which will make your programs easier to work with
- some of them are taken from “Perl Best Practices” from Oreilly

111

Coding Advice

- There is no one size fits all solution
- but here are some guidelines for better code generation
- choose and pick what fits best your style

112

Code Readability

- Many factors go into making your code legible
- variable naming convention
- layout of code
- comment style

113

Variables

- best to be clear exactly what each variable is
- good idea to name reference variables in some way
  - $temp_ref
  - $r_something
  - $command_r

114

Layout

- sub foo {

  }

- sub foo
  {

  }

115

default

- Try to minimize reliance on default behavior, spell out variables

```
for my $person_ref (@waitingList) {
          print $person_ref->name;
  }
```

116

parenthesis your subs

- you aren't paying by the word:
  - add max parenthesis next to your sub and args as needed
  - remove spaces from around parenthesis when possible

```
my @peopleList = getElevatorPeople($n);
```

117

Common Perl commands

- common perl commands
  - print
  - chomp
  - split
- not strictly necessary to have parenthesis, will make code look neater sometimes
- keep same style as your own subs if you want to use it to be safe

118

spaces

- use spaces when referencing complex hash key combinations

```
$candidates[$i] =
$incumbent{ $candidates[$i]{ get_region( ) } };

my $displacement
= $ini_velocity * $time  +  0.5 * $acceleration *
   $time**2;
```

119

semicolons

- use semicolons everywhere
- even when optional

- easy way to indicate section end

120

commas

- in a list/hash when listing items place comma after each one (even last)

- no overhead
- allows copy paste between items

121

indent

- indent loop/condition blocks to make it easier to read them

122

commands

- Never place two statements on the same line
- foo(); $name = $house + foo2();

123

code breaks

- break code into sections
- i.e. start coding by sketching system through comment flows
- expand each comment to a few instructions

124

if/else

- make sure else can be seen

-
  ```
  if {
  …
  } else {
  …
  }
  ```

125

vertical

- when possible aline items vertical

```
my %expansion_of = (
        q{it's}    => q{it is},
        q{we're}   => q{we are},
        q{didn't}  => q{did not},
        q{must've} => q{must have},
        q{I'll}    => q{I will},
    );
```

126

operators

- use operators to break long lines
- use equals to break long lines

127

neatness counts

```
my @months = qw(
        January  February   March
        April       May          June
        July         August     September
        October  November December
    );
```

128

boolean subs

- Part of naming conventions:
  - if a sub returns a boolean should follow a similar pattern of naming the sub like
  - is_Full
  - has_Item

129

variables

- use case to separate types of variables
- for things which hold sets, use plural
- try to name them for their usage
- when shortening, try to make sure that it is clear
  - $dev   #device or developer ?
  - $last #is it the final or the last one used?

130

Privacy

- should use the underscore in the beginning of a variable/sub to show its private

131

String operations

- Should try to use single quotes when no interpolation
- replace empty string with q{} to make it easier to separate double quote from two single quotes
- when saying tab always be clear by saying \t explicitly
- for comma character its easier to say q{,} and easier to read
  - **`my $printable_list = '(' . join(q{,}, @list) . ')';`**
  - `my $printable_list = '(' . join(',', @list) . ')';`

132

escape characters

- saying $delete_key = \127;
  - delete is the 127th value on the ascii table
  - problem: escape numbers are specified by base 8 numbers ☺ should be \177
  - Solution:

```
use charnames qw( :full );

    $escape_seq =
   "\N{DELETE}\N{ACKNOWLEDGE}\N{CANCEL}Z";
```

133

constants

- use constants is really creating a compile time sub
- can't be interpolated
- can't be used directly in hash key

- For values that are going to be unchanged in the program can use a readonly pm

```
use Readonly;
Readonly my $MOLYBDENUM_ATOMIC_NUMBER => 42;


# and later...

print $count * $MOLYBDENUM_ATOMIC_NUMBER;
```

134

formatting

- leading zeros on a number says its octal
- be clear about it if you need octal say octal(number)
- use Readonly;

```
Readonly my %PERMISSIONS_FOR => (
      USER_ONLY     => oct(600),
      NORMAL_ACCESS => oct(644),
      ALL_ACCESS    => oct(666),
  );
```

135

formatting II

- use underscores to separate long numbers
- # In the US they use thousands, millions, billions, trillions, etc...
- can also do the same on floats

```
$US_GDP              = 10_990_000_000_000;
$US_govt_revenue     =  1_782_000_000_000;
$US_govt_expenditure =  2_156_000_000_000;

use bignum;
$PI =
3.141592_653589_793238_462643_383279_502884_197169_399375;
$subnet_mask= 0xFF_FF_FF_80;
```

136

formatting III

- for long strings break it up and connect using .

```
$usage = "Usage: $0 <file> [-full]\n"
        . "(Use -full option for full dump)\n"
        ;
```

137

here doc

- can also use a here doc to break up lines

```
$usage = <<"END_USAGE";
Usage: $0 <file> [-full] [-o] [-beans]
Options:
     -full  : produce a full dump
     -o     : dump in octal
     -beans : source is Java
END_USAGE
```

138

Better

```
use Readonly;
Readonly my $USAGE => <<'END_USAGE';
   Usage: qdump file [-full] [-o] [-beans]
   Options:
        -full  : produce a full dump
        -o     : dump in octal
        -beans : source is Java
   END_USAGE


# and later...


    if ($usage_error) {
        warn $USAGE;
    }
```

139

Avoid barewords

- when perl sees bare words it thinks they are strings (different under strict)

```
$greeting = Hello . World;
print $greeting, "\n";

my @sqrt = map {sqrt $_} 0..100;
for my $N (2,3,5,8,13,21,34,55) {
     print $sqrt[N], "\n";
}
```

140

beware operator precedence

- `next CLIENT if not $finished;`
  - `# Much nicer than: if !$finished`
- `next CLIENT if (not $finished) || $result < $MIN_ACCEPTABLE;`
  - `what is this do ?`
- `next CLIENT if not( $finished || $result < $MIN_ACCEPTABLE );`

141

localization

- if you need to set a package’s variable use the local keyword
- you need to initialize it, since its reset to undef

142

Random question

```
#!C:\perl\bin

print 'this is a small test\n';

@list = (1,2,3);

print join q{,}, @list;
print '\n';

$list[-4] = 3;

print join q{,} ,@list;
print ‘\n’;
```

143

block

- use blocks rather than modifiers in most cases

```
  if (defined $measurement) {
       $sum += $measurement;
   }


$sum += $measurement if defined $measurement;
```

144

avoid unclear loops

```
RANGE_CHECK:
   until ($measurement > $ACCEPTANCE_THRESHOLD) {
       $measurement = get_next_measurement(  );
       redo RANGE_CHECK unless defined $measurement;
       # etc.
   }

RANGE_CHECK:
   while ($measurement <= $ACCEPTANCE_THRESHOLD) {
       $measurement = get_next_measurement(  );
       redo RANGE_CHECK if !defined $measurement;
```

145

Outline for next section

- More code tips
- Memory leaking
- HTML Parsing
- Parsing flat files
- Database Theory
- DBI
- Perl and databases
- Code Review II

146

More code tips

- Lets pick up where we left yesterday

147

map

- as mentioned yesterday, map can be used when we want to replace all values in a list
- but: it will need enough memory for all the copies….think about it before using it

```
@temperature_measurements = map { F_to_K($_) }
   @temperature_measurements;
```

148

Better version

- we can reuse the array location if we don't need the old version

```
for my $measurement (@temperature_measurements) {
        $measurement = F_to_K($measurement);
    }
```

149

What do you think of this ?

```
use List::Util qw( max );

Readonly my $JITTER_FACTOR => 0.01;   # Jitter by a maximum of 1%

my @jittered_points
        = map {
                 my $x = $_->{x};
                 my $y = $_->{y};

                 my $max_jitter = max($x, $y) / $JITTER_FACTOR;
               {
x => $x + gaussian_rand({mean=>0, dev=>0.25, scale=>$max_jitter}),
y => $y + gaussian_rand({mean=>0, dev=>0.25, scale=>$max_jitter}),
               }
         } @points;
```

150

usually replace with for loop

```
my @jittered_points;
    for my $point (@points) {
        my $x = $point->{x};
        my $y = $point->{y};

        my $max_jitter = max($x, $y) / $JITTER_FACTOR;

        my $jittered_point = {
            x => $x + gaussian_rand({ mean=>0, dev=>0.25,
    scale=>$max_jitter }),
            y => $y + gaussian_rand({ mean=>0, dev=>0.25,
    scale=>$max_jitter }),
        };

        push @jittered_points, $jittered_point;
    }
```

151

Better, separate the two

```
my @jittered_points = map { jitter($_) } @points;


# Add a random Gaussian perturbation to a point...
sub jitter {
        my ($point) = @_;
        my $x = $point->{x};
        my $y = $point->{y};

        my $max_jitter = max($x, $y) / $JITTER_FACTOR;

        return {
            x => $x + gaussian_rand({ mean=>0, dev=>0.25,
    scale=>$max_jitter }),
            y => $y + gaussian_rand({ mean=>0, dev=>0.25,
    scale=>$max_jitter }),
        };
    }
```

152

some observations

- The $_ references a touched scalar
- it is not a copy
- when you execute a foreach you are walking across a list
- so if combine commands which touch $_ be careful

153

what is the idea ?

```
#########################
# Select .pm files for which no corresponding .pl
# file exists...
#########################
@pm_files_without_pl_files
        = grep { s/.pm\z/.pl/xms && !-e } @pm_files;
```

154

Thinking

- $_ successively holds a copy of each of the filenames in @pm_files.
- replace the .pm suffix of that copy with .pl
- see if the resulting file exists
- If it does, then the original (.pm) filename will be passed through the grep to be collected in @pm_files_without_pl_files

- Any issues ?

155

- $_ only holds aliases
- substitution in the grep block replaces the .pm suffix of each original filename with .pl;
- then the -e checks whether the resulting file exists.
- If the file doesn't exist, then the filename (now ending in .pl) will be passed through to @pm_files_without_pl_files.
- we will have modified the original element in @pm_files.
- Oops!

- unintentionally mess up the contents of @pm_files and did not even do the job it was supposed to do.

156

visual

- besides neat code
- if possible have code fit within single view window, so can keep track of what is happening
  - more for loop iterations
- not always possible

157

```
sub words_to_num {
        my ($words) = @_;

        # Treat each sequence of non-whitespace as a word...
        my @words = split /\s+/, $words;

        # Translate each word to the appropriate number...
        my $num = $EMPTY_STR;
        for my $word (@words) {
            if ($word =~ m/ zero | zéro /ixms) {
                $num .= '0';
            }
            elsif ($word =~ m/ one | un | une /ixms) {
                $num .= '1';
            }
            elsif ($word =~ m/ two | deux /ixms) {
                $num .= '2';
            }
            elsif ($word =~ m/ three | trois /ixms) {
                $num .= '3';
            }
            # etc. etc. until...
            elsif ($word =~ m/ nine | neuf /ixms) {
                $num .= '9';
            }
            else {
                # Ignore unrecognized words
            }
        }

        return $num;
    }

    # and later...

    print words_to_num('one zero eight neuf');    # prints: 1089
```

158

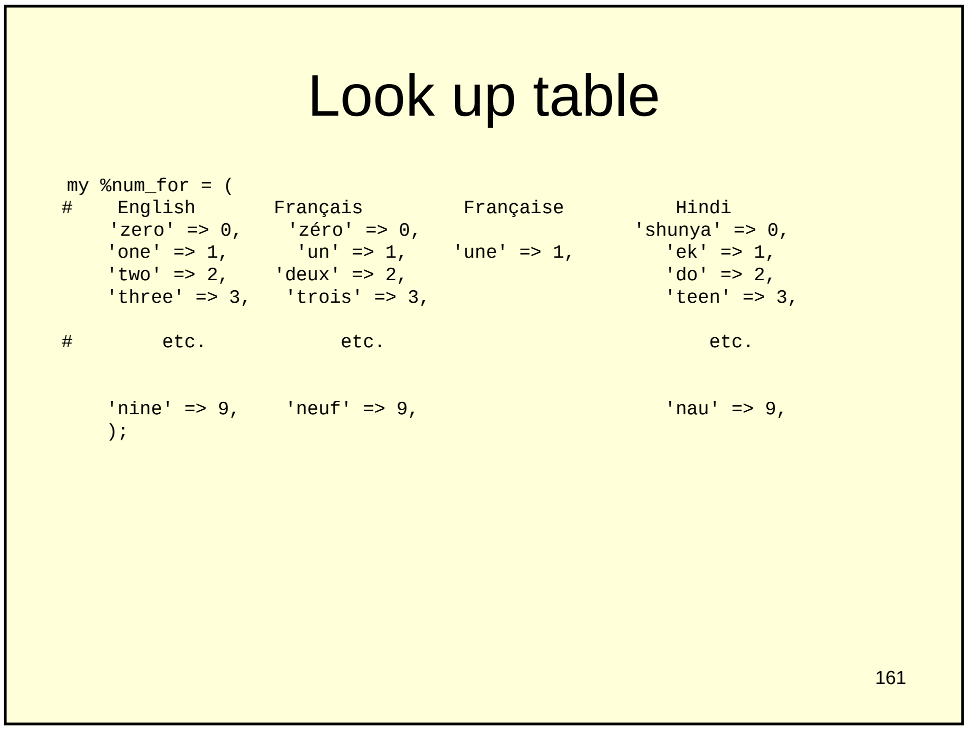
```
 my @words = split /\s+/, $words;


# Translate each word to the appropriate number...


        my $num = $EMPTY_STR;
        for my $word (@words) {
            my $digit = $num_for{lc $word};
            if (defined $digit) {
                 $num .= $digit;
            }
        }

        return $num;
    }
```

159

difference

- adding more info

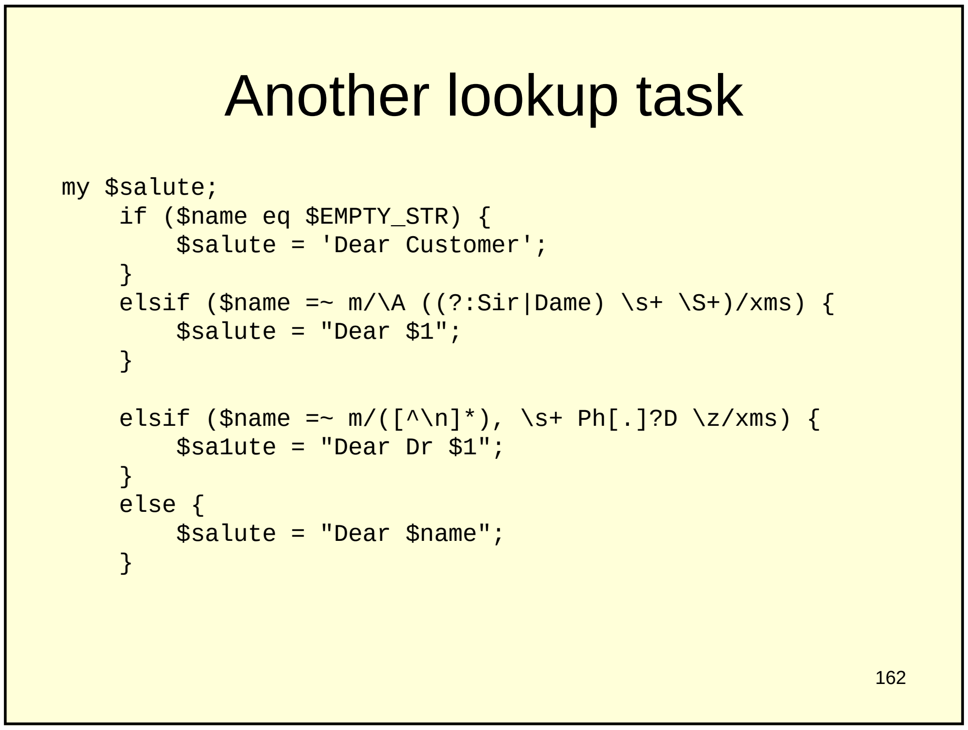
160



Look up table

```
 my %num_for = (
#    English       Français         Française         Hindi
    'zero' => 0,    'zéro' => 0,                    'shunya' => 0,
    'one' => 1,      'un' => 1,    'une' => 1,         'ek' => 1,
    'two' => 2,    'deux' => 2,                        'do' => 2,
    'three' => 3,   'trois' => 3,                      'teen' => 3,

#        etc.            etc.                              etc.


    'nine' => 9,    'neuf' => 9,                       'nau' => 9,
    );
```

161



Another lookup task

```
my $salute;
    if ($name eq $EMPTY_STR) {
        $salute = 'Dear Customer';
    }
    elsif ($name =~ m/\A ((?:Sir|Dame) \s+ \S+)/xms) {
        $salute = "Dear $1";
    }

    elsif ($name =~ m/([^\n]*), \s+ Ph[.]?D \z/xms) {
        $sa1ute = "Dear Dr $1";
    }
    else {
        $salute = "Dear $name";
    }
```

162

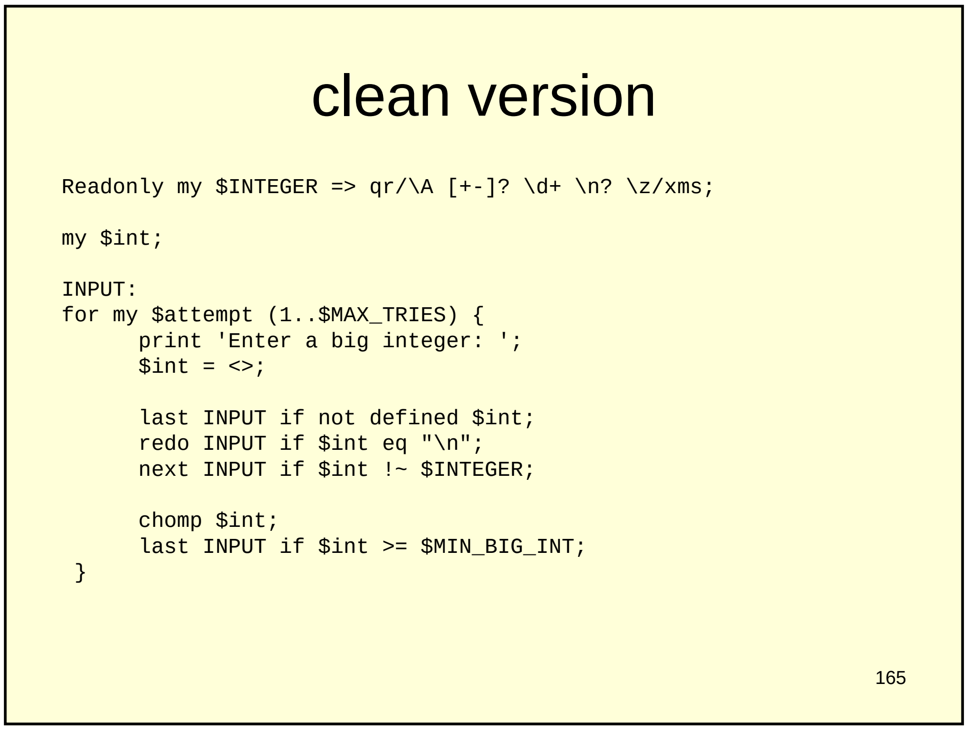
```
my $salute = $name eq $EMPTY_STR                       ? 'Dear Customer'
              : $name =~ m/ \A((?:Sir|Dame) \s+ \S+) /xms ? "Dear $1"
              : $name =~ m/ (.*), \s+ Ph[.]?D \z     /xms ? "Dear Dr $1"
              :                                             "Dear $name"
              ;
```

163

Loops

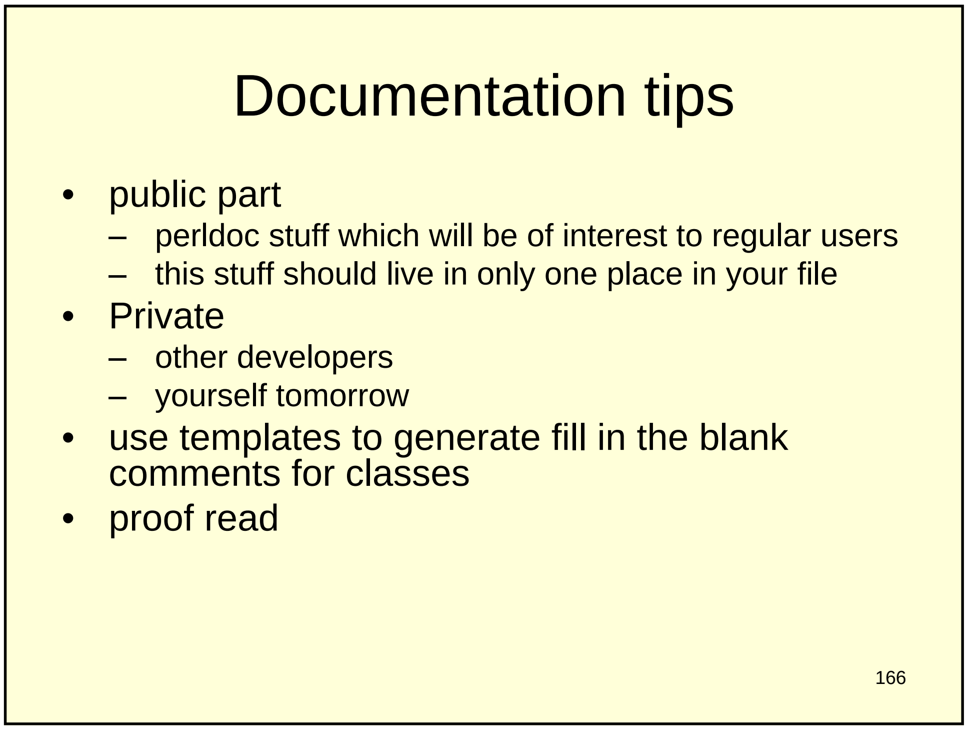
- try to avoid do..while loops
  - can’t use next, last
  - logic is at the end
- Reject as early as possible
  - use lots of next to avoid computation per loop
- Add labels to loops to make it clear we might exit early

164



clean version

```
Readonly my $INTEGER => qr/\A [+-]? \d+ \n? \z/xms;

my $int;

INPUT:
for my $attempt (1..$MAX_TRIES) {
      print 'Enter a big integer: ';
      $int = <>;

      last INPUT if not defined $int;
      redo INPUT if $int eq "\n";
      next INPUT if $int !~ $INTEGER;

      chomp $int;
      last INPUT if $int >= $MIN_BIG_INT;
 }
```



Documentation tips

- public part
  - perldoc stuff which will be of interest to regular users
  - this stuff should live in only one place in your file
- Private
  - other developers
  - yourself tomorrow
- use templates to generate fill in the blank comments for classes
- proof read

some ideas

- =head1 EXAMPLES
- =head1 FREQUENTLY ASKED QUESTIONS
- =head1 COMMON USAGE MISTAKES
- =head1 SEE ALSO

167

private comment templates

```
############################################
# Usage      : ????
# Purpose    : ????
# Returns    : ????
# Parameters : ????
# Throws     : no exceptions
# Comments   : none
# See Also   : n/a
```

168

where to sprinkle

- single line comments before/after
- anywhere you had a problem
- to clarify
  - if you are doing too much commenting, maybe a good idea to recode it

169

built in

- Try to use as many built ins before you go find other libraries
  - generally they have been optimized to run with perl
  - for specific things, you might want to find replacement
  - at the same time each built in is optimized for a specific scenario

170

sort – don't recompute

```
 # (optimized with an on-the-fly key cache)

@sorted_files
= do {
      my %md5_of;
      sort { ($md5_of{$a} ||= md5sum($a))
               cmp
               ($md5_of{$b} ||= md5sum($b))
            }
          @files;
        };
```

171

- if doing sort more than once
  - globalize the cache
  - take a slice at some point
  - memoize

172

reverse of a sort

- standard:
  - `@sorted_results = sort { $b cmp $a } @unsorted_results;`
- Optimized
  - `@sorted_results = reverse sort @unsorted_results;`

173

Reverse

- use `scalar reverse` when you want to reverse a scalar
- `my $visible_email_address = reverse $actual_email_address;`
- `my $visible_email_address = scalar reverse $actual_email_address;`
- reason:
  - `add_email_addr(reverse $email_address);`

174

split

- For data that is laid out in fields of varying width, with defined separators (such as tabs or commas) between the fields, the most efficient way to extract those fields is using a split.

175

```
 # Specify field separator
Readonly my $RECORD_SEPARATOR => q{,};
Readonly my $FIELD_COUNT      => 3;

# Grab each line/record
while (my $record = <$sales_data>) {
        chomp $record;
# Extract all fields
my ($ident, $sales, $price)
            = split $RECORD_SEPARATOR, $record, $FIELD_COUNT+1;
# Append each record, translating ID codes and
# normalizing sales (which are stored in 1000s)
push @sales, {
            ident => translate_ID($ident),
            sales => $sales * 1000,
            price => $price,
        };
    }
```

176

Reality check

```
my ($ident, $sales, $price, $unexpected_data)
 = split $RECORD_SEPARATOR, $record, $FIELD_COUNT+1;

if($unexpected_data){
carp
"Unexpected trailing garbage at end of record id
   '$ident':\n",
"\t$unexpected_data\n“;
}
```

177

sorting

- stable sort
  - keeps items which are equal (in a sort sense) in order as the sort progress
  - B A E` D G E`` F Q E```
  - A B D E` E`` E``` F G Q

178

optimization

- internally the sort routine will sometimes compute all keys and store them along with the items to sort efficiently

179

reuse sorting

```
use Sort::Maker;
# Create sort subroutines (ST flag enables
   Schwartzian transform)
...
make_sorter(name => 'sort_md5', code => sub{ md5sum($_)    }, ST => 1 );
make_sorter(name => 'sort_ids', code => sub{ /ID:(\d+)/xms }, ST => 1 );
make_sorter(name => 'sort_len', code => sub{ length        }, ST => 1 );
# and later
...
@names_shortest_first = sort_len(@names);
@names_digested_first = sort_md5(@names);
@names_identity_first = sort_ids(@names);
```

180

Any ideas ?

- my @stuff = <*.pl>;

181

equivalent

- `my @files = glob($FILE_PATTERN);`

182

sleep

- takes integer args
- `sleep 0.5;  #??`

- Solution:
  - `use Time::HiRes qw( sleep );`
  - `sleep 0.5;`

183

Beware

- before this package, programmers were taking advantage of another call
- `select undef, undef, undef, 0.5;`

- it is supposed to check if i/o streams are free
- can take second fractions

184

- even if doing it wrong, at least encapsulate

```
sub sleep_for {
 my ($duration) = @_;
 select undef, undef, undef, $duration;
 return;
 }

# and then
sleep_for(0.5);
```

185

- map BLOCK LIST
- map EXPR, LIST

- hard to tell when expression part ends

- `@args = map substr($_, 0, 1), @flags, @files, @options;`
- `@args = map {substr $_, 0, 1} @flags, @files, @options;`

186

Scalar::Util

- blessed $scalar
  - If $scalar contains a reference to an object, blessed( ) returns a true value (specifically, the name of the class).
  - Otherwise, it returns undef.
- refaddr $scalar
  - If $scalar contains a reference, refaddr( ) returns an integer representing the memory address that reference points to.
  - If $scalar doesn't contain a reference, the subroutine returns undef.
  - useful for generating unique identifiers for variables or objects
- reftype $scalar

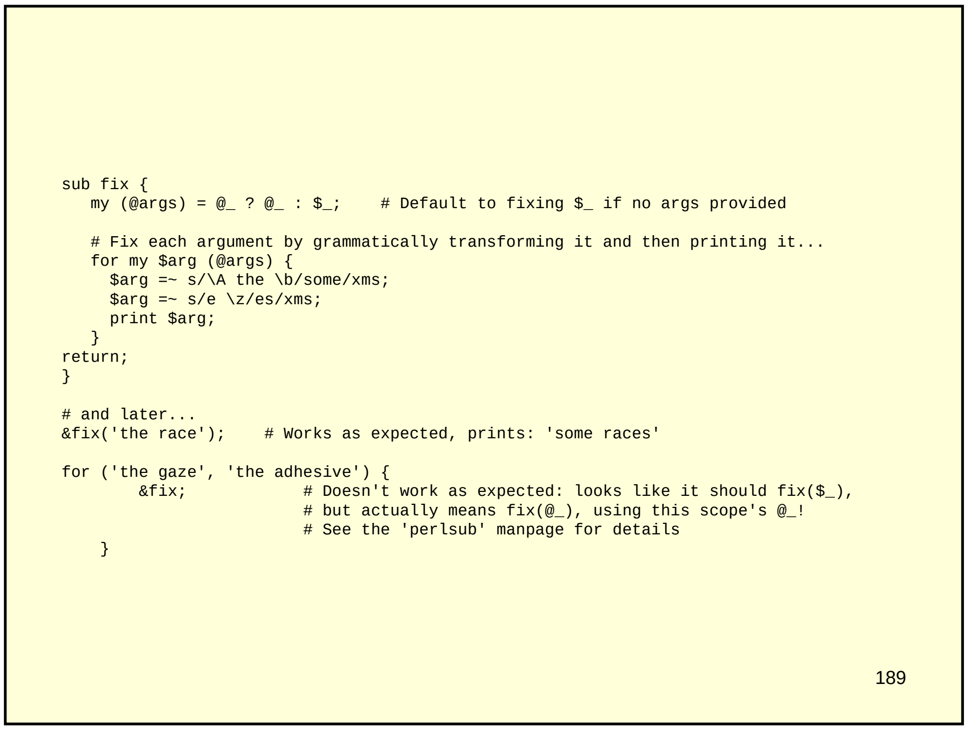
187

List::Util

- first {<condition>} @list
- shuffle @list
- max @list
- sum @list

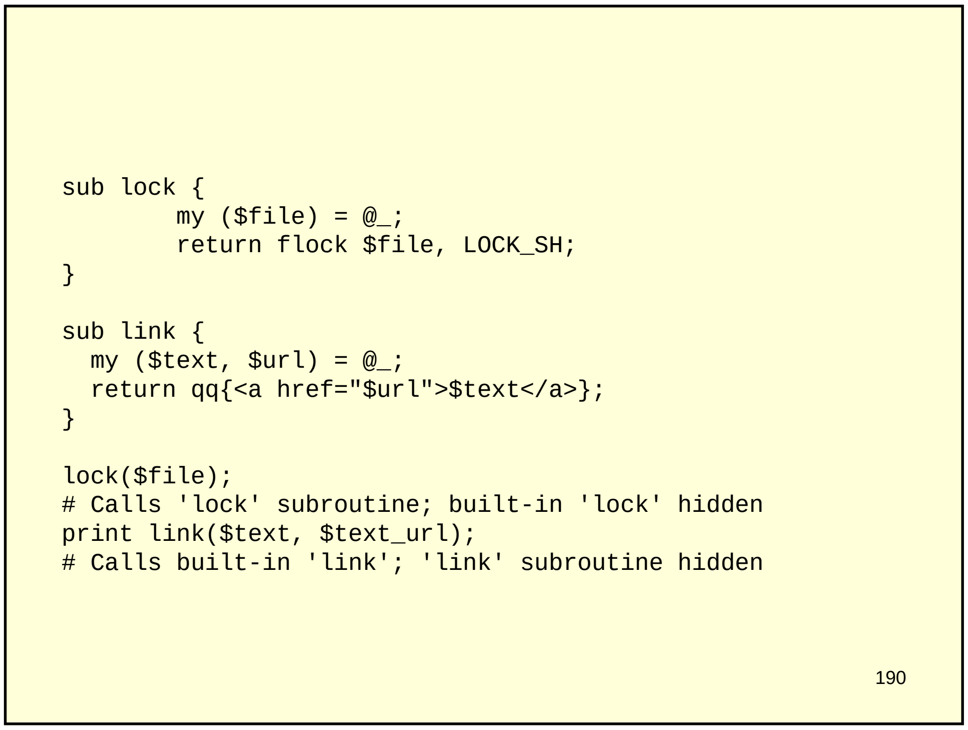
- List::MoreUtils
  - all {<condition>} @list

188



```
sub fix {
   my (@args) = @_ ? @_ : $_;    # Default to fixing $_ if no args provided

   # Fix each argument by grammatically transforming it and then printing it...
   for my $arg (@args) {
     $arg =~ s/\A the \b/some/xms;
     $arg =~ s/e \z/es/xms;
     print $arg;
   }
return;
}

# and later...
&fix('the race');    # Works as expected, prints: 'some races'

for ('the gaze', 'the adhesive') {
        &fix;            # Doesn't work as expected: looks like it should fix($_),
                         # but actually means fix(@_), using this scope's @_!
                         # See the 'perlsub' manpage for details
    }
```

189



```
sub lock {
        my ($file) = @_;
        return flock $file, LOCK_SH;
}

sub link {
  my ($text, $url) = @_;
  return qq{<a href="$url">$text</a>};
}

lock($file);
# Calls 'lock' subroutine; built-in 'lock' hidden
print link($text, $text_url);
# Calls built-in 'link'; 'link' subroutine hidden
```

190

subs

- name sub arg so that it makes your code easier to work with
  - as opposed to working with $_[0], $_[1] etc
  - can make mistakes with offsets

191

- for more than three args, pass in hash ref

192

```
sub padded {

my ($arg_ref) = @_;

my $gap   =
$arg_ref->{cols} - length $arg_ref->{text};
my $left  = $arg_ref->{centered} ? int($gap/2) : 0;
my $right = $gap - $left;

        return $arg_ref->{filler} x $left
              . $arg_ref->{text}
              . $arg_ref->{filler} x $right;
    }
```

193

use Contextual::Return;

```
return (
 LIST    { @server_data{ qw( name uptime load users ) };
   }
 BOOL    { $server_data{uptime} > 0;
   }
 NUM     { $server_data{load};
   }
 STR     { "$server_data{name}: $server_data{uptime},
   $server_data{load}"; }
 HASHREF { \%server_data;
   }
       );
```

194

sub prototypes

- this is good only if programmers can see the sub
  - that is good for private subs
- issues
  - can't specify how they will be used
  - can introduce bugs if adding it to code

195

returns

- always type out your returns
- covers your bases
- plain return for failure
  - returning undef can be misinterpreted in list context as non false return

196

Files

- pay attention how you use bareword filenames when creating them
- Never open, close, or print to a file without checking the outcome.

197

```
SAVE:
while (my $save_file = prompt 'Save to which file? ') {

# Open specified file and save results...

   open my $out, '>', $save_file  or next SAVE;
   print {$out} @results          or next SAVE;
   close $out                     or next SAVE;

# Save succeeded, so we're done...

    last SAVE;
}
```

198

filehandles

- if you don’t need them, close them asap
  - will free up memory and buffers much earlier
- Use while (<>), not for (<>)
  - for implemented very inefficiently
  - any ideas why ?

199

side point

- ranges are different, although files are slurped in all at once for list context in for loop
- the following is lazily evaluated

```
for my $n (2..1_000_000_000) {
        my @factors = factors_of($n);

        if (@factors == 2) {
            print "$n is prime\n";
        }
        else {
            print "$n is composite with factors:
   @factors\n";
        }
    }
```

200

fast way to slurp

- if you need to read in a file at once
- default a little inefficient as it looks for record seperators (\n)
- override the definition

```
my $text = do { local $/; <$in> };
```

201

- faster way involves system level calls
  - sysread $fh, $text, -s $fh;
- File::Slurp
  - read_file
    - wraps that system call

202

- use Perl6::Slurp;
- my $text = slurp $file_handle;

203

- Avoid using *STDIN, unless you really mean it.
- might not have a regular stdin
  - example: in a pipeline
- Always put filehandles in braces within any print statement.

204

interfacing with user

- Always prompt for interactive input
- can run a check:

```
use IO::Interactive qw( is_interactive );


# and later...
    if (is_interactive(  )) {
        print $PROMPT;
    }
```

205

- if you are doing a lot of interaction consider:

```
use IO::Prompt;

my $line = prompt 'Enter a line: ';
```

206

references

- Wherever possible, dereference with arrows
  - neater code
  - interpolates
- Where prefix dereferencing is unavoidable, put braces around the reference (next slide)
- Never use symbolic references
- for circular references, be sure to call weaken

207

```
push @{$list_ref}, @results;

    print substr(${$str_ref}, 0, $max_cols);

    my $first = ${$list_ref}[0];
    my @rest  = @{$list_ref}[1..$MAX];

    my $first_name = ${$name_ref}{$first};
    my ($initial, $last_name) =
   @{$name_ref}{$middle, $last};

    print @{${$ref_to_list_ref}}[1..$MAX];
```

208

Objects

- always use the base form
  - use base qw(Avian Agrarian Alien);
- bless class explicitly
- Pass constructor arguments as labeled values, using a hash reference
- Separate your construction, initialization, and destruction processes

209

Databases

- lets talk about databases

210

Database

- a collection of data stored on a computer with varying layers of abstraction sitting on top of it.
- Each layer of abstraction generally makes the data stored within easier to both organize and access, by separating the request for particular data from the mechanics of getting that data.

211

Relational Database

- relational database is a database that is perceived by the user as a collection of tables, where a table is an unordered collection of rows.
- (Loosely speaking, a relation is a just a mathematical term for such a table.)
- Each row has a fixed number of fields, and each field can store a predefined type of data value, such as an integer, date, or string.

212

API

- Databases use Application Programming Interfaces (APIs) to provide access to the data stored within the database.
- In the case of the simplest databases, the API is simply the file read/write calls provided by the operating system
- An API allows programmers to interact with a more complex piece of software through access paths defined by the original software creators.
- Example: Berkeley Database Manager API. In addition to simply accessing the data, the API allows you to alter the structure of the database and the data stored within the database.

213

Query Language

- Allows you to manipulate the underlying data in the database

- Fetch
- Store
- Update
- Delete

214

Simplest case

- We can use a flat file as the simplest database
- How to separate data
  - delimiter
  - fix length

- how would the operations work ?

215

- what is missing from this picture ?

216

- concurrency
- we need to protect the integrity of the database

217

SQL

- Structured Query Language, or SQL is a language designed for the purpose of manipulating data within databases.

218

RDBM

- relational database model revolves around data storage units called tables, which have a number of attributes associated with them, called columns.
- Example:
  - user
  - password
  - homedirectory
  - GUI
  - UID

219

schema

- A schema is a collection of logical data structures, or schema objects, such as tables and views.
- In some databases, a schema corresponds to a user created within the database.
- In others, it's a more general way of grouping related tables.

220

data

- data is stored in rows
- each column has a type associated with it
- types:
  - numeric
  - text
  - binary
  - type specific (time)
  - null

221

Select

- SELECT column, column, ..., column
- FROM table

- SELECT user,passwd from usertable

- SELECT user,passwd from usertable where user like ‘%sh%’;

222

- joining tables
- grouping data
- ordering data

223

- lets take a quick look at the background slides

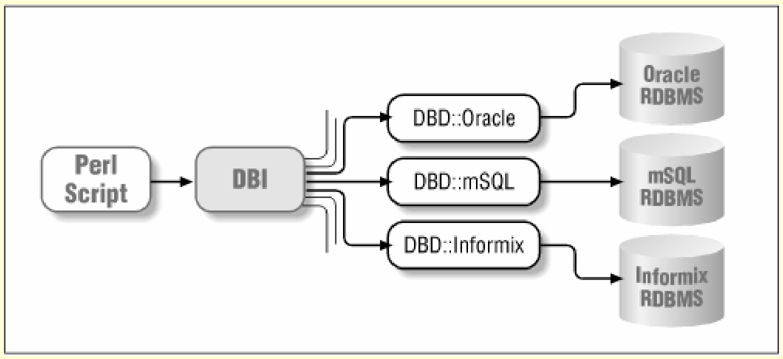
224

Perl implementation

- The DBI architecture is split into two main groups of software:

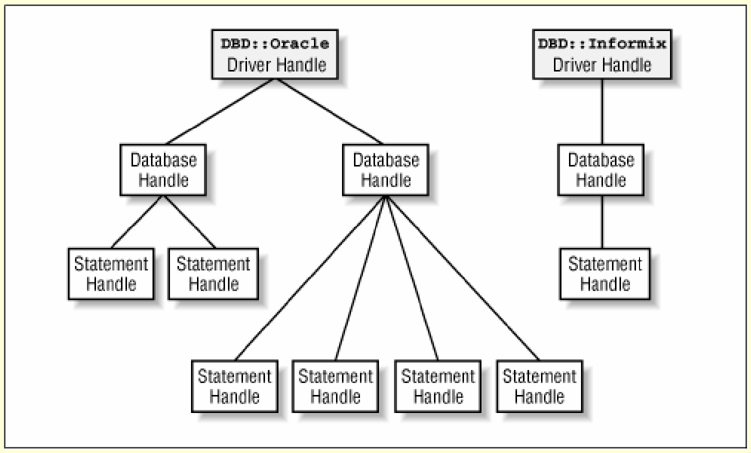
1. The DBI defines the actual DBI programming interface, routes method calls to the appropriate drivers, and provides various support services to them.
2. Specific drivers are implemented for each different type of database and actually perform the operations on the databases.

225



226

DBI handles



227

- can have mulitple driver handles open at the same time
  - great for data transfers
- $dbh = DBI->connect($data_source, ...);
- statement handles are for working with sql

228

- $dbh = DBI->connect($data_source, $username, $password, \%attr);

```
#!/usr/bin/perl -w
#
# ch04/connect/ex1: Connects to an Oracle database.

use DBI;            # Load the DBI module

### Perform the connection using the Oracle driver
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username",
    "password" )
    or die "Can't connect to Oracle database: $DBI::errstr\n";

exit;
```

229

```
#!/usr/bin/perl -w
#
# ch04/connect/ex3: Connects to two Oracle databases simultaneously.

use DBI;            # Load the DBI module

### Perform the connection using the Oracle driver
my $dbh1 = DBI->connect( "dbi:Oracle:archaeo", "username", "password" )
    or die "Can't connect to 1st Oracle database: $DBI::errstr\n";

my $dbh2 = DBI->connect( "dbi:Oracle:seconddb", "username", "password" )
    or die "Can't connect to 2nd Oracle database: $DBI::errstr\n";

exit;
```

230

```
#!/usr/bin/perl -w
#
# ch04/connect/ex4: Connects to two database, one Oracle, one mSQL
#                   simultaneously. The mSQL database handle has
#                   auto-error-reporting disabled.

use DBI;            # Load the DBI module

### Perform the connection using the Oracle driver
my $dbh1 = DBI->connect( "dbi:Oracle:archaeo", "username", "password" )
    or die "Can't connect to Oracle database: $DBI::errstr\n";

my $dbh2 = DBI->connect( "dbi:mSQL:seconddb", "username", "password" , {
            PrintError => 0
        } )
    or die "Can't connect to mSQL database: $DBI::errstr\n";

exit;
```

231

```
### Now, disconnect from the database
$dbh->disconnect
 or warn "Disconnection failed:
   $DBI::errstr\n";

exit;
```

232

For example:

```
### Attributes to pass to DBI->connect(  )
%attr = (
    PrintError => 0,
    RaiseError => 0
);

### Connect...
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" , \%attr );

### Re-enable warning-level automatic error reporting...
$dbh->{PrintError} = 1;
```

Most DBI methods will return a false status value, usually undef, when execution fails. This is easily tested by Perl in the following way:

```
### Try connecting to a database
my $dbh = DBI->connect( ... )
    or die "Can't connect to database: $DBI::errstr!\";
```

233

getting ready to talk

- every database has its own system of representing
  - empty strings
  - quotes
  - escapes

```
my $quotedString = $dbh->quote( $string );

### Use quoted string as a string literal in a SQL statement
my $sth = $dbh->prepare( "
    SELECT *
    FROM media
    WHERE description = $quotedString
  " );

$sth->execute();
```

234

trace

- 0
  - Disables tracing.
- 1
  - Traces DBI method execution showing returned values and errors.
- 2
  - As for 1, but also includes method entry with parameters.
- 3
  - As for 2, but also includes more internal driver trace information.
- 4
  - Levels 4, and above can include more detail than is helpful

235

```
use DBI;

### Remove any old trace files
unlink 'dbitrace.log' if -e 'dbitrace.log';

### Connect to a database
my $dbh = DBI->connect( "dbi:Oracle:archaeo", "username", "password" );

### Set the tracing level to 1 and prepare()
DBI->trace( 1 );
doPrepare();

### Set the trace output back to STDERR at level 2 and prepare()
DBI->trace( 2, undef );
doPrepare();

exit;

### prepare a statement (invalid to demonstrate tracing)
sub doPrepare {
    print "Preparing and executing statement\n";
    my $sth = $dbh->prepare( "
        SELECT * FROM megalith
    " );
    $sth->execute();
    return;
}

exit;
```

236

utilities

- neat
  - allows you to print out type and formatted output of any data
- looks_like_number

237

Life of sql query

1. The prepare stage parses an SQL statement, validates that statement, and returns a statement handle representing that statement within the database.
2. Providing the prepare stage has returned a valid statement handle, the next stage is to execute that statement within the database. This actually performs the query and begins to populate data structures within the database with the queried data. At this stage, however, your Perl program does not have access to the queried data.
3. Fetch stage, in which the actual data is fetched from the database using the statement handle. The fetch stage pulls the queried data, row by row, into Perl data structures, such as scalars or hashes, which can then be manipulated and post-processed by your program.

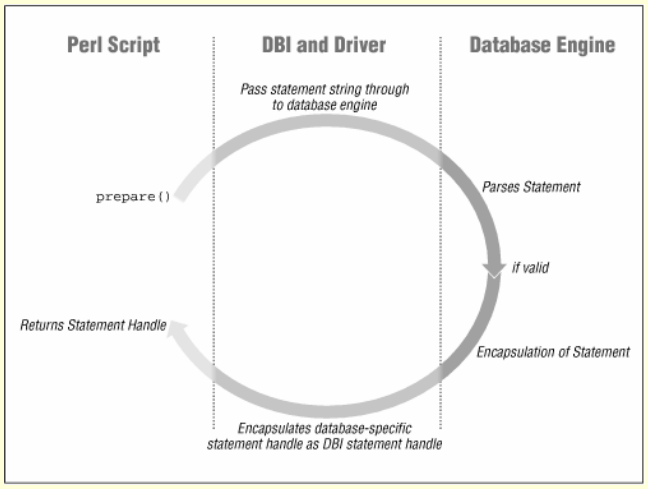
   The fetch stage ends once all the data has been fetched, or it can be terminated early using the finish() method.

   If you'll need to re-execute() your query later, possibly with different parameters, then you can just keep your statement handle, re-execute() it, and so jump back to stage 2.

4. deallocation stage. This is essentially an automatic internal cleanup exercise in which the DBI and driver deallocate the statement handle and associated information. For some drivers, that process may also involve talking to the database to tell it to deallocate any information it may hold related to the statement

238

prepare stage


Perl Script
DBI and Driver
Database Engine
Pass statement string through to database engine
prepare()
Parses Statement
if valid
Returns Statement Handle
Encapsulation of Statement
Encapsulates database-specific statement handle as DBI statement handle

239

stage 2:

```
### Create the statement handle
my $sth = $dbh->prepare( "SELECT id, name FROM
   megaliths" );

### Execute the statement handle
$sth->execute();
```

240

stage 3

- The data retrieved by your SQL query is known as a result set (so called because of the mathematical set theory on which relational databases are based).
- The result set is fetched into your Perl program by iterating through each record, or row, in the set and bringing the values for that row into your program. This form of fetching result set data on a row-by-row basis is generally termed a cursor.
- Cursors are used for sequential fetching operations: records are fetched in the order in which they are stored within the result set. Currently, records cannot be skipped over or randomly accessed. Furthermore, once a row addressed by a cursor has been fetched, it is ``forgotten'' by the cursor. That is, cursors cannot step backwards through a result set.
- We fetch data from the database's result set is to loop through the records returned via the statement handle, processing each row until no rows are left to fetch:

```
while ( records to fetch from $sth ) {
    ### Fetch the current row from the cursor
    @columns = get the column values;
    ### Print it out...
    print "Fetched Row: @columns\n";
}
```

241

```
### Prepare the SQL statement ( assuming $dbh exists )
$sth = $dbh->prepare( "
            SELECT meg.name, st.site_type
            FROM megaliths meg, site_types st
            WHERE meg.site_type_id = st.id
          " );

### Execute the SQL statement and generate a result set
$sth->execute();

### Fetch each row of result data from the database as a list
while ( ( $name, $type ) = $sth->fetchrow_array ) {
    ### Print out a wee message....
    print "Megalithic site $name is a $type\n";
}
```

242

- returned data is in the same order your sql statement was worded

- select user, passwd
- select passwd, user

243

- can use references instead of data copy of run through results

```
### Fetch the rows of result data from the database
### as an array ref....
while ( $array_ref = $sth->fetchrow_arrayref ) {
    ### Print out a wee message....
    print "Megalithic site $array_ref->[0] is a
   $array_ref->[1]\n";
}
die "Fetch failed due to $DBI::errstr" if $DBI::err;
```

244

what is wrong here

```
### The stash for rows...
my @stash;

### Fetch the row references and stash 'em!
while ( $array_ref = $sth->fetchrow_arrayref ) {
    push @stash, $array_ref;
}

### Dump the stash contents!
foreach $array_ref ( @stash ) {
    print "Row: @$array_ref\n";
}
```

245

```
### The stash for rows...
my @stash;

### Fetch the row references and stash 'em!
while ( $array_ref = $sth->fetchrow_arrayref ) {
    push @stash, [ @$array_ref ];  # Copy the array
   contents
}

### Dump the stash contents!
foreach $array_ref ( @stash ) {
    print "Row: @$array_ref\n";
}
```

246

- call finish if you don’t want all the results or are done processing
- will signal the db that it can “forget” your resultsets

- if you have any results pending, shutdown of db will give you warning messages

247

non select statements

```
### Assuming a valid database handle exists....
### Delete the rows for Stonehenge!
$rows = $dbh->do( "
            DELETE FROM megaliths
            WHERE name = 'Stonehenge'
        " );
```

- will return number of rows affected

248

binding

```
$sth = $dbh->prepare( "
            SELECT name, location
            FROM megaliths
            WHERE name = ?
        " );
$sth->bind_param( 1, $siteName );
```

249

type binding

```
use DBI qw(:sql_types);

$sth = $dbh->prepare( "
            SELECT meg.name, meg.location, st.site_type, meg.mapref
            FROM megaliths meg, site_types st
            WHERE name = ?
            AND id = ?
            AND mapref = ?
            AND meg.site_type_id = st.id
          " );
### No need for a datatype for this value. It's a string.
$sth->bind_param( 1, "Avebury" );

### This one is obviously a number, so no type again
$sth->bind_param( 2, 21 );

### However, this one is a string but looks like a number
$sth->bind_param( 3, 123500, { TYPE => SQL_VARCHAR } );

### Alternative shorthand form of the previous statement
$sth->bind_param( 3, 123500, SQL_VARCHAR );

### All placeholders now have values bound, so we can execute
$sth->execute(  );
```

250

binding results

```
### Perl variables to store the field data in
my ( $name, $location, $type );

### Prepare and execute the SQL statement
$sth = $dbh->prepare( "
            SELECT meg.name, meg.location, st.site_type
            FROM megaliths meg, site_types st
            WHERE meg.site_type_id = st.id
          " );
$sth->execute(  );

### Associate Perl variables with each output column
$sth->bind_columns( undef, \$name, \$location, \$type );

### Fetch the data from the result set
while ( $sth->fetch ) {
    print "$name is a $type located in $location\n";
}
```

251

Do vs Prepare

- do statement actually does a 4 step process in the background
- if you have multiple statements in a do, better call prepare+execute instead

252

memory usage

- There is a saying that to estimate memory usage of Perl, assume a reasonable algorithm for memory allocation, multiply that estimate by 10, and while you still may miss the mark, at least you won't be quite so astonished. This is not absolutely true, but may provide a good grasp of what happens.

- Assume that an integer cannot take less than 20 bytes of memory, a float cannot take less than 24 bytes, a string cannot take less than 32 bytes (all these examples assume 32-bit architectures, the result are quite a bit worse on 64-bit architectures).
- If a variable is accessed in two of three different ways (which require an integer, a float, or a string), the memory footprint may increase yet another 20 bytes. A sloppy malloc(3) implementation can inflate these numbers dramatically.

253

- sub foo;
  - may take up to 500 bytes of memory, depending on which release of Perl you're running.

- Anecdotal estimates of source-to-compiled code bloat suggest an eightfold increase. This means that the compiled form of reasonable (normally commented, properly indented etc.) code will take about eight times more space in memory than the code took on disk.

- There are two Perl-specific ways to analyze memory usage: $ENV{PERL_DEBUG_MSTATS} and -DL command-line switch. The first is available only if Perl is compiled with Perl's malloc(); the second only if Perl was built with -DDEBUGGING. See the instructions for how to do this in the INSTALL podpage at the top level of the Perl source tree.

254

- If your perl is using Perl's malloc() and was compiled with the necessary switches (this is the default), then it will print memory usage statistics after compiling your code when $ENV{PERL_DEBUG_MSTATS} > 1, and before termination of the program when $ENV{PERL_DEBUG_MSTATS} >= 1. The report format is similar to the following example:

255

```
$ PERL_DEBUG_MSTATS=2 perl -e "require Carp"
 Memory allocation statistics after compilation: (buckets 4(4)..8188(8192)
    14216 free:   130   117    28     7     9   0   2     2   1 0 0
            437    61    36     0     5
    60924 used:   125   137   161    55     7   8   6    16   2 0 1
             74   109   304    84    20
 Total sbrk(): 77824/21:119. Odd ends: pad+heads+chain+tail: 0+636+0+2048.
 Memory allocation statistics after execution:   (buckets 4(4)..8188(8192)
    30888 free:   245    78    85    13     6   2   1     3   2 0 1
            315   162    39    42    11
   175816 used:   265   176  1112   111    26  22  11    27   2 1 1
            196   178  1066   798    39
 Total sbrk(): 215040/47:145. Odd ends: pad+heads+chain+tail:
      0+2192+0+6144.
```

256