Perl Basics

Structure, Style, and Documentation
Easy to read programs

Your job as a programmer is to create programs that are:
- easy to read
- easy to understand,
- easy to modify, and
- safe and robust,

They must of course be correct solutions to the problems they are supposed to solve.

You must believe in the importance of these objectives, otherwise you will be creating throw-away code of no use to anyone, not even yourself, not long after you write it.
What makes code more readable?

- The following is a correct but unreadable Perl program:

```
$s=0;$i=1;while($i<=10){$s=$s+$i;$i=$i+1;
}print "The sum of 1 through 10 is $s\n";
```

- It is a program that no one will want to use because it is difficult to read and understand. Why?
- For one thing, there is no use of space to separate the parts.
White space for readability

- The same program using "white space" (blanks, tabs, newlines) to make it readable:

  ```perl
  $s = 0;
  $i = 1;
  while ( $i <= 10 ) {
    $s = $s + $i;
    $i = $i + 1;
  }
  print "The sum of 1 through 10 is $s\n";
  ```

- Perl ignores all white space, so you can use it to make the program easier to read without affecting the meaning of the program.
White space wisdom

These are good rules for using white space well:

- **Always** put at most one simple statement per line.
- **Always** put spaces between operators (such as `+` `<=` `=`) and their operands (variables like `$s` and `$i`, and literals like `1` and `10`).
- **Almost never** put multiple newlines between statements that are part of a block. A **block** is a sequence of statements between curly braces `{}`.
- **Always** use indentation to indicate logical structure, as explained next.
The `while` keyword introduces a loop. We'll learn about loops soon. Loops are just one of many program structuring concepts in general, and in Perl too.

```perl
while ( $i <= 10 ) {
    $s = $s + $i;
    $i = $i + 1;
}
```

It is easier to identify and understand loops if the loop body, i.e., the code between the curly braces, is indented from the start of the `while` keyword, usually by 4 spaces.
Indentation to indicate structure

- The **if** keyword introduces a *conditional branch*, sometimes called a *decision structure*. We'll learn about these very soon. They also require indentation, as illustrated:

```perl
if ( $i < 0 ) {
    statement;
    statement;
} else {
    statement;
    statement;
}
```

**true branch**

**false branch**
Naming things

- In all programming languages, the programmer gets to choose the names of various things used in a program. What "things" you can name depend on the language. We begin by focusing on variables. These “things” are technically called symbols.

- A variable is a storage cell for your program. The program can store data in it and retrieve data from it. It exists somewhere in memory. Variables have names.

- In the example program, $s and $i are variables. (In Perl, a variable name must begin with a '$', or one of another set of funny characters like '@' or '%').
Variable names

Naming a variable \$s is like naming a child, "child1". (Yes, George Foreman basically did that.) Pity the poor person who has to read this program and try to discern what \$s is for.

- Variables should be given descriptive names. In this program, \$s is being used to accumulate a sum that will be printed in the end. It is much better to name it \$sum.

- \$i stores the successive values to add to the sum. It is 1, then 2, then 3, and so on. It is like a counter. Naming it \$counter is so much clearer.

- Now look at the program.
The same program, improved again

```perl
$sum = 0;
$counter = 1;
while ( $counter <= 10 ) {
    $sum = $sum + $counter;
    $counter = $counter + 1;
}
print "The sum of 1 through 10 is $sum\n";
```

Even without knowing Perl, you can probably guess what this program is doing now.
Choosing good names

- Perl lets names be up to 251 characters long, so never blame Perl for being cheap with your names.

- *Names must begin with a letter or underscore, and after that it can contain letters, upper or lowercase, digits, and underscores.*

- Underscores can be used like spaces in names that you want to be several words (called *underscore_case*):

  ```
  $file_name   $money_spent_so_far    $user_reply
  $line_length  $prompt_string  $word
  $num_amino_acids
  ```
More about names

- Some people do not use underscores. Instead they make each word after the first start with an Uppercase letter (called *camelCase*):
  - `$fileName`  `$moneySpent`  `$userReply`  `$numAminoAcids`
- This is a matter of choice. In any one program though, you should adhere to one convention or the other.
- Whatever you do, do not use obscure names like `$p`, `$tt`, and `$xyz` when writing programs. There are certain limited conditions under which it is acceptable to name a variable `$i` or `$j`, but until you know what these are, they should not appear in your programs.
Every programming language provides a means to include text that will be ignored by the compiler. In Perl, the pound sign `#` tells the compiler that the rest of the line, including the `#`, should be ignored,

This is your chance to stick stuff into the program that exists solely for you and the thousands of people who will someday read your program.

So what stuff do you put there? **Comments**.

*A comment is explanatory text that helps the reader to understand some part of the program.*
Uses of comments

- Comments can be used in several ways:
  - At the beginning of every program file, to provide information about the file such as author, purpose, list of changes, etc. This is called a *prologue*.
  - To explain hard-to-follow algorithms;
  - To introduce new pieces of a large program visually;
  - To provide short descriptions of data structures used in the program;
  - To temporarily remove statements from a program while testing and debugging it.
An example program prologue

#!/usr/bin/perl -w
# sums     -- prompts user to enter a number N and
#             prints the sum of all numbers <= N
# Written by Stewart Weiss
#             October 14, 2006
# Usage:
#        sums

#-----------------------------
#           This is a title block
#-----------------------------

# .... code goes here
Good and bad comment styles

- You should look at the various “demo” programs for examples of good comments. The textbook has examples of bad commenting too.
- In general, if you have named your variables very well, you will not need to explain as much as if you have not.
- You should always have a comment block at the top of every file, with program name, author, creation date, purpose, usage information, and list of modifications.
- As we go along, I will fill in more about what needs to be in a good program.
Being strict

- Perl lets you get away with many dangerous and sloppy habits. It also provides the means to prevent you from doing this.
- In this class, you will be required to follow the second path, the path of programming righteousness.
- Perl has two tools to accomplish this: *warnings* and the "*use strict*" *pragma*. You will have to use both of these in each program you write.
- A *pragma* is a command telling Perl what to do about something.
Warnings

- Perl, like most compilers, gives you the option to turn on warning messages when it checks the syntax of your program.
- A warning can be issued when it sees something that might look like a mistake. Perl cannot be completely sure when something is a mistake, so it displays a warning and lets you decide.
- For example, if you store a value into a variable but never use the value from that variable, Perl will issue a warning, because there is no point in putting a value in a variable and never using it; this usually means that either you forgot a statement or you mistyped a variable name. By pointing it out to you, it helps you to find potential mistakes.
Warnings

- There are a few ways to turn on warnings.
- The `-w` flag in the first line turns them on, but only if you use the form
  ```bash
  #!/usr/bin/perl -w
  ```
- You can instead write
  ```bash
  #!/usr/bin/en v perl
  $WARNING = 1;
  ```
- You can use the shorter form of `$WARNING: $^w` if you want.
- When `$^w` has the value 1, it is the same thing as using the `-w` flag.
Example

To illustrate, suppose you made a mistake and wrote this program instead of the one we created earlier:

```perl
$sum = 0;
$i = 1;
while ( $i <= 10 ) {
    $sum = $s + $i;  # $s should be $sum
    $i = $i + 1;
}
```

The `-w` switch would catch the mistake where you forgot to change the `$s` to `$sum`. 
Strictness

- The "use strict;" \textit{pragma} tells the compiler to detect when you are using certain types of constructs that are error-prone.

- A common kind of sloppiness is using variables without first declaring them, as in

  ```perl
  #!/usr/bin/env perl
  $fastfood = "SloppyJoe";
  print "This program is like $fastfood\n"
  ```

- $\texttt{fastfood}$ was supposed to be "\texttt{declared}" first, meaning that perl was told that $\texttt{fastfood}$ is a certain type of variable.
Strictness

- You should declare all variables that your program uses with "my" statements:
  
  ```perl
  my $fastfood;
  my ($counter, $sum, $name);
  ```

- If you put a strictness pragma in the program, it will catch when you are not strict:

- You can think of “my” as meaning that these names are names you created. This is not really accurate, but for now the goal is an easy way to remember to do this.
A sloppy program with strictness checking

If you try to run this program

```perl
#!/usr/bin/env perl
# prologue here but no room in slide
use strict;        # turn on strictness
$WARNINGS = 1;     # turn on warnings
$fastfood = "SloppyJoe";
print "This program is like $fastfood\n";
```

you will get the error message

```
Global symbol "$fastfood" requires explicit package name at line 5 in sloppy.pl
```
A sloppy program with strictness checking

The correct version of the preceding program:

```perl
#!/usr/bin/env perl
# prologue here but no room in slide
use strict;        # turn on strictness
$WARNINGS = 1;     # turn on warnings
my $fastfood = "SloppyJoe";
print "This program is like $fastfood\n";
```
Summary of good style

- Use white space to indicate structure.
- Choose meaningful names.
- Use comments to make your code understandable.
- Declare all variables with "my" and comment them if necessary.
- Use the -w switch or the $^W variable and the use strict pragma.
- Be consistent in the style choices you make.