Introduction to Perl Programming

Liadh Kelly and Joe Carthy

What is Perl

Perl is an acronym, short for Practical Extraction and Report Language. It was designed by Larry Wall as a tool for writing programs in the UNIX environment. Perl has the power and flexibility of a high-level programming language such as C. In fact many of the features of the language are borrowed from C. Like shell script languages, Perl does not require a compiler - the Perl interpreter runs your programs. This means that Perl is ideal for producing quick solutions to small programming problems, or for creating prototypes to test potential solutions to larger problems.

Perl Programming

The first line of your Perl program **must** specify where the Perl interpreter on your system is. For many systems this is usually,

```
#!/usr/bin/perl
```

The # character is the Perl comment character. Except for the line above, all text following # is ignored in Perl.

Scalar Variables & Assignment

Unlike C, **variables do not have to be declared** in Perl. A variable in Perl is represented by the **$** symbol followed by the variable name e.g.

```
$nlines = 20 ;
$nc = 0 ;
```

In PERL a ***scalar variable*** can hold one piece of information i.e exactly one item, for example a line of input, a string, or a number. Items that can be stored in *scalar variables* are called *scalar* ***values***. Basically, a *scalar value* is one unit of data. This unit of data can be either a number or a chunk of text.

Note that Perl statements **must be terminated with a semicolon** as in the C language.

The PERL *assignment operator* is the = character, as in C.

For example,

```
$total = 23;
```

```
$name = “Joe Bloggs”;
```

(Here the string "Joe Bloggs", is assigned to the scalar variable `$name`.)

Output

To output text to the standard output, (typically the screen), the *print* function is used.

Example,

```
print(“Hello world”);

print(“Enter your name: \n”);
```

(The \n here is the newline character, which is an example of a *control character*. It causes the cursor to go on to a new line.)

Input

Reading from the standard input is carried out as follows:

```
$x = <STDIN>;
```

Here what you type on the keyboard is assigned to the scalar variable $x. This is an example of how keyboard input is carried out.

Because reading from the standard input is so commonly used, an abbreviated form for input is available in Perl so that the following two are equivalent:

```
$firstname =<STDIN> ;
$firstname = <> ;
```

A complete program

The following is a complete Perl program stored in a file called `hello`

```
#!usr/bin/perl

print(“Enter your name: ”);
$name = <STDIN>;
print(“Hello ”, $name);
```

It may be executed as follows

```
% perl hello
Enter your name: Sally
Hello Sally
```

Alternatively, you may change the permission on the program to make it executable (you only need to do this once) with the **`chmod`** command and then you may execute your program like any Unix command:

```
% chmod +x hello
% hello
Enter your name: Sally
Hello Sally
```

More Output

Perl also supports a *`printf`* function, like that of C.

The arguments passed to the `printf` function are as follows:

- string to be printed, which can contain one or more field specifiers.
- value for each field specifier appearing in the string to be printed.

For Example,

```
$stg = 20;
printf("The variable stg = %d pounds. \n", $stg);
```

will display:

```
The variable stg = 20 pounds.
```

The string to be printed contains one field specifier, *%d*, which represents an integer. The value stored in $stg is substituted for the field specifier and printed.

When printf processes a field specifier, it substitutes the corresponding value in the printf argument list. The representation of the substituted value in the string depends on the field specifier that is supplied.

Field specifiers consist of the % character followed by a single character that represents the format to use when printing. Table 1 lists the field-specifier formats and the field-specifier character that represents each.

Specifier	**Description**
%c	Single character
%d	Integer in decimal (base-10) format
%e	Floating-point number in scientific notation
%f	Floating-point number in "normal" (fixed-point) notation
%g	Floating-point number in compact format
%o	Integer in octal (base-8) format
%s	Character string
%u	Unsigned integer
%x	Integer in hexadecimal (base-16) format

Table 2. Field specifiers for printf.

Control Characters

The following table lists some of the commonly used control characters that are used in Perl.

Character	**Description**
\b	Backspace
\E	ends the effect of \L or \U
\l	Forces the next letter into lowercase
\L	All following letters are lowercase
\n	Newline
\r	Carriage return
\t	Tab
\u	Force next letter into uppercase
\U	All following letters are uppercase

Basic Arithmetic: Examples

`$i = 17 + 5;`	(This adds 17 and 5 and then assigns the result, 22, to the scalar variable $i.)
`$j = 20 + 3 - 6;`	(This adds 20 and 3 and subtracts 6 and then assigns the result, 17, to the scalar variable $j.)
`$j = 11;`	
`$k = $j * 6;`	(The second statement here takes the value currently stored in $j, 11, and multiplies it by 6. The result, 66, is assigned to $k.)
`$k = 6/2;`	(This divides 6 by 2 and then assigns the result, 3, to the scalar variable $k.)
`$i = $i + 1;`	(This adds 1 to the number currently stored in $i.)
`$i++;`	(This is another way of adding 1 to the number currently stored in $i.)
`$i = $i - 1;`	(This subtracts 1 from the number currently stored in $i.)
`$i--;`	(This is another way of subtracting 1 from the number currently stored in $i.)

Perl is a **case sensitive** language. This means it distinguishes between upper and lower case letters so that the following are three distinct variables

$NAME ≠ $name ≠ $Name

A common programming error in Perl is to misspell a variable name or use the wrong case. In a language like C, the compiler will detect the error. Because Perl does not require variable declarations, it **assumes the misspelled variable is a new variable**. When a Perl program misbehaves, you should check that you have spelled all variable consistently.

For example,

```
$Number = 20;
$number++;
```

Contrary to what you might expect, `$Number` now has the value 20 and `$number` now has the value 1.

Array Variables

Perl enables you to store lists in special variables designed for that purpose. These variables are called *array variables* (or *arrays* for short).

For Example,

```
@array = (1, 2, 3, 4, 5);
```

(Here, the list (1,2,3,4,5) is assigned to the array variable @array.)

OR we could write this as *@array = (1 .. 5);*

(**..** allows us specify a range of numbers, e.g. 20..24 is the same as 20, 21, 22,23, 24.)

Another example,

@people = ("John", "Pat", "Mary");

(Here, the list ("John", "Pat", "Mary") is assigned to the array variable @people.)

Note, the *array variable's* **name begins with @**.

This distinguishes *array variable* names from *scalar variable* names, which begin with *$*.

You refer to any element of an *array variable* as a *scalar variable*.

The first element of the list `@array` is	`$array[0]`
the second element of the list is	`$array[1]` ……….
the last element of the list is	`$array['length of list' - 1]`.

Examples

```
$x = $array[0];
```

(Here the first element of @array is assigned to the scalar variable $x)

```
$array[2] = 5;
```

(Here the **third** element of the array variable `@array` is assigned the number 5.)

Array Length

The length of a list assigned to an array variable can be determined by assigning the *array name* to a *scalar variable*.

Example:

```
@array = (1,2,3);
```

```
$len = @array;
```

(`$len`, now has the value 3, which is the length of the list currently assigned to `@array`.)

This is used very frequently. For example we may read a line of text from a file/standard input and break into a list of words which we then process one word at a time. Consequently, we need to compute the number of words in the list, as shown above.

Arrays & Strings

To parse a character string into separate items the library function **split** is used. (Note: The C library provides the **strtok()** function for similar purposes)

The syntax for *split* is

@words = split(string); (where string is the character string to split and *words* is the resulting array.)

Example:

$string = "words::separated::by::colons";

@words = split(/::/, $string);

The first argument passed to *split* tells it how to break the string into separate parts.

In this example, the first argument is `::` so *split* breaks the string into four separate parts in this case. The result is then the list

("words", "separated", "by", "colons")

which is assigned to the array variable *@words.* Now *$words[1]* is the string *"separated"*.

To give another example using *split*,

$string = "Hello wild windy world ";
@list = split(/ /, $string);

In this case *split* breaks the string into four separate parts. The result is then the array

("Hello", "wild", "windy", "world")

which is assigned to the array variable *@list*. Now
$list[0] is the string *"Hello"*.
$list[1] is the string *"wild"*.

..
$list[3] is the string *"world"*.

A string variable may be passed as an argument to split.

Example

```
#!usr/bin/perl

print("Enter    some    words    separated    by
commas");
$line = <STDIN>;

split( /,/, $line );

print("First word", $w[0]);
print("Second word", $w[1]);
```

The *chop* Library Function

This function assumes that a string is stored in the variable passed to it; *chop*'s job is to delete the last character in the string.

For Example,

> ***$line = "This is my line";***
> ***chop($line);***
>
> After chop is called, the value of $line becomes,
>
> ***This is my lin***

The *chop* function is used to remove the newline character from strings read from <STDIN>. When a string is read from <STDIN> the newline character is stored with the string. If for example, you enter 23 as a line of input, then *"23\n"* is read in *(i.e. 23 followed by the newline character).*

When *chop* is applied to this string (i.e. *"23\n"*) the newline character is removed from the string and we are left with *"23"*.

For Example,

> ***$age = <STDIN>;***
> ***chop($age);***

The above removes the newline character from the number entered from the standard input and stored in the scalar variable $age.

The same applies to strings:

$name = <STDIN>;

This reads a name, including the newline character into $name. We can remove it using chop:

chop($name);

It is a very common mistake for beginners to forget to remove the trailing newline and as a result get a comparison failure.

For example, if we read the string "bill" from <STDIN> what we actually read is "bill\n". If we then compare what we have read i.e. "bill\n" with the string "bill" we find they do not match.

The following example highlights the issue:

```
#!usr/bin/perl
print("Enter your name: ") ;

$name = <STDIN>;
print("hello ", $name, " nice to meet you!")
;

chop($name);
print("\n\nAfter the axeman!!\n") ;

print("hello", $name, " nice to meet you!\n")
;
```

Assume the user enters "bill"

Output

```
hello bill
 nice to meet you

After the axeman!!
hello bill nice to meet you
```

The first time we print $name, it still contains the newline character and so our output is split over two lines. After using the chop function, the newline character has been removed and the out is as expected on a single line.

String Operators

Comparison Operators		
String operator	**Comparison operator**	**Equivalent numeric operator**
lt	Less than	<
gt	Greater than	>
eq	Equal to	==
le	Less than or equal to	<=
ge	Greater than or equal to	>=
ne	Not equal to	!=
cmp	Compare, returning 1, 0, or –1	<=>

When characters (or strings of characters) are compared, it is their ASCII code that is used to determine the result. For example, the string *aaa* is less than the string *bbb*, because the ASCII code of *a* is less than the ASCII code of *b*.

Note that the ASCII codes of the *uppercase letters* are less than the ASCII codes of the *lower case letters*. This means for example that the character '*Z*' "is less than" the character '*a*'. The ASCII code for uppercase letters range from 65 ('A') to 90 ('Z') and for lowercase from 97 ('a') to 122 ('z').

String comparison examples:

```
$result = “ZZZ” lt “aaa”;          # result is true

$result = "aaa" lt "bbb";      # result is true

$result = "aaa" gt "bbb";    # result is false

$result = "aaa" eq "bbb";          # result is false

$result = "aaa" le "aaa";          # result is true

$result = "aaa" ge "bbb";          # result is false

$result = "aaa" ne "aaa";          # result is false

$result = "aaa" cmp "bbb";         # result is -1
```

The <=> and *cmp* operators are special cases. Unlike the other comparison operators, <=> and *cmp* return one of three values:

0, if the two values being compared are equal
1, if the first value is greater
-1, if the second value is greater

For example, consider the following statement:

$y = $x <=> 10;

The possible results are:

- If $x is greater than 10, the first value in the comparison is greater, and $y is assigned 1.
- If $x is less than 10, the second value in the comparison is greater, and $y is assigned -1.
- If $x is equal to 10, $y is assigned 0.

String Concatenation

The string-concatenation operator, **.** , joins two strings together.

For Example,

$newstring = "potato"."head"; (this assigns the string *potatohead* to $newstring.)

Conditional Statements

The if Statement

The syntax for the if statement is similar to that of C except that you MUST use {}:

```
if (expr)
{
        statement_block
}
```

For example,

```
if ($number >0 )
{
        print("The number is not zero.\n");
}
```

Another example,

```
if ($number == 21)
{
        print("The number is 21.\n");
}
```

A final example,

```
if ($word eq "hello")
{
        print("You entered hello.\n");
}
```

Note we use ***eq*** to test for **string equality**.

The if - else Statement

The syntax for this statement is:

```
if (expr)
{
        statement_block_1
}
else
{
        statement_block_2
}
```

The following sample program reads in two numbers and displays a message stating whether the two numbers are equal or not equal.

```
#!/usr/bin/Perl

print("enter a number:\n");
$number1 = <STDIN>;
chop($number1);
print("enter another number:\n");
$number2 = <STDIN>;
chop($number2);
if ($number1 == $number2)  l
{
        print("The two numbers are equal.\n");
}
else                       l
{
        print("The two numbers are not equal.\n");
}
print("This is the last line of the program.\n");
```

The if – elsif – else Statement

The syntax for this statement is:

```
if (expr_1)
{
        statement_block_1
}
elsif (expr_2)
{
        statement_block_2
}
elsif (expr_3)
{
        statement_block_3
}
……….
else
{
        default_statement_block
}
```

The while loop

A while loop allows you repeat one or more Perl statements.
The syntax for the while loop is:

```
while(expr)
{
        statement_block
}
```

For example, the following code fragment prints *"The number is still 5."* and reads in a new number until the number read in isn't 5.

```
$number = 5 ;
while ( $number == 5 )
{
        print("The number is still 5.\n");
        $number = <STDIN>;        #read a number
}
```

The following program uses the *while* statement. It continues looping, reading in two numbers and outputting the result of adding the numbers, until the first number read in is zero.

```
#!/usr/bin/Perl

$done = 0;    # $done is used to determine whether or not to continue looping
print("This line is printed before the loop starts.\n");
while($done == 0)     #continue looping until $done is not equal to 0
{
        print("Enter a number: \n");
        $number1 = <STDIN>;
        chop($number1);
        print("Enter another number: \n");
        $number2 = <STDIN>;
        chop($number2);

        if($number1 == 0)
        {
                $done = 1;
                print("Goodbye\n");
        }
        else
        {
                $result = $number1 + $number2;
                print("The result is $result \n");
        }

}
print("End of loop.\n");
```

The until Statement

There is a slight difference between the *while* and the *until* statement, namely,

- The *while* statement loops while its conditional expression is true.
- The *until* statement loops until its conditional expression is true (that is, it loops as long as its conditional expression is false).

The syntax for the *until* statement is:

```
until(expr)
{
        statement_block
}
```

For example,

```
until($count == 5)              #continue looping until $count is 5.
{
        print("still in the until loop.\n");
        $count ++;
}
print("$count is now 5.\n");
```

The following sample program reads an attempt at calculating '17+26'. The program then loops until the correct answer is read in, namely 43.

```
#!/usr/bin/Perl

print("What is 17 plus 26? \n");
```

```
$correct_answer = 43;                  #the correct answer
$input_answer = <STDIN>;
chop($input_answer);
until($input_answer == $correct_answer)
{
        print("Wrong.  Keep trying. \n");
        $input_answer = <STDIN>;
        chop($input_answer);
}
print("You got it. \n");
```

The for Statement

The syntax of the for statement is

```
for (expr1; expr2; expr3) {

        statement_block

}
```

- *expr1* is the *loop initialiser*. It is evaluated only once, before the start of the loop.

- *expr2* is the *conditional expression* that terminates the loop. The conditional expression in *expr2* behaves just like the ones in while and if statements. If its value is 0 (false), the loop is terminated, and if its value is nonzero, the loop is executed.

- *statement_block* is the collection of statements that is executed if (and when) *expr2* has a nonzero value.

- *expr3* is executed once per iteration of the loop and is executed after the last statement in statement_block is executed.

The following program prints the numbers from 1 to 5 using the *for* statement.

```
#!/usr/bin/perl

for ($count=1; $count <= 5; $count++)
{
        print ("$count\n");
}
```

The first expression defined in the *for* statement, *$count = 1*, is the *loop initialiser*; it is executed before the loop is iterated.

The second expression defined in the *for* statement, *$count <= 5*, tests whether to continue iterating the loop.

The third expression defined in the *for* statement, *$count++,* is evaluated after the last statement in the loop (i.e. *print("$count\n")*) is executed.

Pattern Matching

The =~ operator tests whether a pattern appears in a character string.

For example,

```
$result = $var =~ /abc/;
```

Here *abc* is the pattern to be tested for (i.e. the pattern enclosed in / /) and the result of the =~ operator is

- A nonzero value, or true, if the pattern is found in the string.
- 0, or false, if the pattern is not matched.

Another example,

```
if($command =~ />/)  #if the string contained in $command has a > char.
{
        print("output redirection occurs.\n");
}
elsif($command =~ /</)  #if the string contained in $command has a < char.
{
        print("input redirection occurs.\n");
}
else
{
        print("neither output or input redirection occur.\n");
}
```

The following sample program reads a question and uses the match operator =~ to establish if the question contains the word *'please'*.

```
#!/usr/bin/Perl

print("Ask me a question politely:\n");
$question = <STDIN>;
if($question =~ /please/)        #if "please" is in the question
{
        print("Thank you for being polite.\n");
}
else
{
        print("That was not very polite.\n")
}
```

Note: Perl also allows powerful pattern matching techniques, such as detecting patterns at the start or end of a string and detecting the occurrence of uppercase or lowercase letters. It also allows the use of *wildcard* characters. That is the **+, *** and ***?*** characters.

The special character + means "*one or more of the preceding characters*." For example, the pattern */de+f/* matches *def, deef, deeef, deeeeeef,* and so on.
Note: Patterns containing + always try to match as many characters as possible.
For example, if the pattern */ab+/* is searching in the string *abbc* it matches *abb*, not *ab.*

The * special character matches zero or more occurrences of the preceding character. For example, the pattern */de*f/* matches *df, def, deef,* and so on.

The ***?*** character matches zero or one occurrence of the preceding character. For example, the pattern */de?f/* matches either *df* or *def.* Note that it does not match *deef*, because the ***?*** character does not match two occurrences of a character.

Logical Operators

The following logical operators are defined in PERL:

```
$a || $b        # logical or:  true if $a or $b is true

$a && $b        # logical and:  true only if $a and $b are true

! $a            # logical not:  true if $a is false
```

For Example,

```
if(($number1 > 0)&&($number2 >0))        #true if both numbers are greater
                                          #than zero
{
        print("The two numbers are greater than zero\n");
}


if(($number1 > 0)||($number2 > 0))        #if at least one of the numbers
                                          #is greater than zero
{
        print("At least one of the numbers is greater than zero\n");
}
```

Note that the || operator evaluates its subexpressions in the order given, and evaluates a subexpression only if the previous subexpression is zero. This means that $number2 is evaluated only if $number1 is zero in the previous example.

When the logical OR operator is used in a larger expression, its value is the last subexpression actually evaluated, which is the first subexpression of the logical OR operator that is nonzero. This means that

```
$myval = $a || $b || $c;
```

is equivalent to

```
if ($a != 0)
{
        $myvalue = $a;
}
elsif ($b != 0)
{
        $myvalue = $b;
}
else
{
         $myvalue = $c;
}
```

die Function

PERL has a die function, which terminates the program immediately and prints the message passed to *die*. The syntax of the *die* function is,

```
die(message);
```

For example, the statement

```
die ("Stop this now!\n");
```
prints *"Stop this now!"* on your screen and terminates the program.

Subroutines

The following example shows how a subroutine works.

```
#!/usr/bin/Perl

$total = 0;
&getnumbers;
print("the first number is $number1 \n");
print("the second number is $number2 \n");

sub getnumbers
{
        $number1 = 29;
        $number2 = 3;
}
```

To invoke a subroutine precede its name by the *&* character, as shown in the above example. This then causes the program to jump to the subroutine. The keyword **sub** tells the PERL interpreter that getnumbers is a subroutine. The code enclosed inside the *{ }* is the body of the subroutine.

Sample Program

The following program prompts the user to enter a sentence. The sentence is then split into a list using the split function. The number of words in the sentence is calculated and displayed. Finally a subroutine is called which checks if your sentence begins with THE and ends with ‘!’.

```
#!/usr/bin/Perl

print("please enter a sentence.\n");
$sentence = <STDIN>;

@words = split(/ /,$sentence);          #split sentence into a list of words

$num_of_words = @words;                 #count the number of words in the list
print("Your sentence has ", $num_of_words, " words in it.\n");

&begin_end;                             #call the begin_end subroutine

print(“This is the end of the program.\n”);

sub begin_end                           #this is a subroutine
{
        if($words[$num_of_words - 1] =~ /!/)     #if the last item in the list contains !
        {
                print("Your sentence ends with !\n");
        }
        else
        {
                print("Your sentence does not end with !\n");
        }

        if(($words[0] eq "the") || ($words[0] eq "The")) #the first item in the list is the
        {
                print("Your sentence begins with THE.\n");
        }
}
```

File Handling

As in the C language, files must be opened before they can be accessed. The library function *open* is used to open a file. The syntax for the *open* function is

open (FILEFP, filename);

(Where, *FILEFP* represents the name you want to use in your PERL program to refer to the file. *filename* represents the name of the file on your machine.)

Note it is common practice to use all uppercase letters for your file variable names This makes it easier to distinguish file variable names from other variable names.

For example,

open(FILE1, 'data.txt'); (This opens the file *data.txt* in current directory and associates it with the file variable *FILE1*.)

open(FIN, "/u/jqpublic/input"); (Note here we want to open a file in a different directory, so the complete pathname is specified.)

When you open a file, you must decide how you want to access the file. Two file-access **modes** (or, simply, file modes) available in Perl are:

- ***read mode***: Enables the program to read the existing contents of the file but does not enable it to write into the file.

- ***write mode***: Destroys the current contents of the file and overwrites them with the output supplied by the program.

By default, *open* assumes that a file is to be opened in read mode. To specify write mode, put a > *character* in front of the filename that you pass to open, as follows:

open (FOUT, ">outfile");

This opens the file *outfile* for writing and associates it with the file variable *FOUT.*

Note: when you open an existing file for writing its current contents are destroyed.

If open returns a nonzero value, the file has been opened successfully.

If open returns 0, an error has occurred.

You should always check that open succeeds and take appropriate action if it fails e.g. print an error message and terminate your program.

A program can be terminated using the ***die*** command, which prints a message to the screen and then terminates the program.

Example

die(“cannot open outfile for writing \n”);

File Input

To read from a file, enclose the file variable associated with the file in angle brackets (< and >), as follows:

$line = <MYFILE>;

This statement reads a line of input from the file specified by the file variable *MYFILE* and stores the line of input in the scalar variable *$line.*

File Output

You may use the stanard print function to write to a file:

print OUTFILE ("Any text or message you wish.\n");

writes *“Any text or message you wish.\n”* to the file specified by OUTFILE.

Redirecting Input/Output

The command,

open(STDIN, “file_name”);

redirects the standard input so that it is now being read from the named file (instead of from the keyboard, for example).

The command,

open(STDOUT, “>file_name”);

redirects the standard output so that it is now writing to the named file(instead of writing to the screen say, for example). Note the > character before the file_name, this is necessary to allow you write to the named file.

The standard input and standard output can be redirected back to the keyboard and screen by the commands

close(STDIN)
and

close(STDOUT)

respectively.

Using Command-Line Arguments as Values

When a PERL program starts up, a specially defined array variable called *@ARGV* contains a list consisting of the command-line arguments.

For Example, typing the command

```
Program_name myfile1 myfile2
```

On the command-line, sets *@ARGV* to the list

```
("myfile1", "myfile2").
```

@ARGV can then be used in the same manner as any other array variable.

Systems Programming

The fork Function

The *fork* function creates two copies of your program: the *parent process* and the *child process*. These copies execute simultaneously.

The syntax for the *fork* function is

```
procid = fork();
```

fork returns zero to the *child process* and a nonzero value to the *parent process*. This nonzero value is the *process ID* of the *child process*. (A *process ID* is an integer that enables the system to distinguish this process from the other processes currently running on the machine.)

The return value from *fork* enables you to determine which process is the *child process* and which is the *parent.* For example:

```
$retval = fork();
if ($retval == 0)
{
                # this is the child process
        exit;   # this terminates the child process
}
else
{
               # this is the parent process

}
```

The waitpid Function

The *waitpid* function waits for a particular child process.

The syntax for the *waitpid* function is

```
waitpid (procid, waitflag);
```

procid is the process ID of the process to wait for, and *waitflag* is a special wait flag. By default, *waitflag* is 0 (a normal wait). *waitpid* returns 1 if the process is found and has terminated, and it returns -1 if the child process does not exist.

The following program shows how *waitpid* can be used to control process execution.

```
#!/usr/bin/Perl

$procid = fork();

if ($procid == 0)
{
        # this is the child process

        print ("this line is printed first\n");
        exit(0);
}
else
{
        # this is the parent process

        waitpid ($procid, 0);
        print ("this line is printed last\n");
}
```

The exec Function

The syntax for the *exec* function is

```
exec (list);
```

This function is passed a list as follows: The first element of the list contains the name of a program to execute, and the other elements are arguments to be passed to the program.

exec accepts additional arguments that are assumed to be passed to the command being invoked.

For example, the following statement executes the command vi file1:

```
exec ("vi", "file1");
```

exec often is used in conjunction with fork: when fork splits into two processes, the child process starts another program using exec.