Faculty of Computer Science, Dalhousie University *3-Dec-2018*

CSCI 2132 — Software Development

Lecture 35: Shell Scripting

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lectures

- Examples with writing and reading a binary file
- Shell Scripting: introduction, a basic example
- variables
- arithmetic operations, conditional expressions
- 'if' statement, 'for' loop
- 'case' statement
- **Previous Lecture**
- **Final Exam Review**

Case Statement Example

There is a case statement in shell scripting, which is similar to the switch statement in C. Its syntax is:

```
case var in
  word{|word}*)
    commands
    ;;
  ...
esac
```

This statement specifies multiple actions to be taken when the value of `var` matches one or more values. The following script makes use of this to print whether there is a (2132) lecture today.

```
#!/bin/bash
day=`date | cut -f1 -d" "`

case "$day" in
  Mon|Wed|Fri)
    echo 2132 lectures
    ;;
  Tue|Thu)
    echo no 2132 lectures
    ;;
  Sat|Sun)
    echo No lectures
    ;;
esac
```

The above example makes use of command substitution in an interesting way.

26.6 Conditional Expression for Files

The following conditional expressions can be used on files:
`[ -e file ]` — true if file exists
`[ -f file ]` — true if file exists and is a regular file
`[ -d file ]` — true if file exists and is a directory file
`[ -r file ]` — true if file exists and is readable by the current user
`[ -w file ]` — true if file exists and is writable by the current user
`[ -x file ]` — true if file exists and is executable by the current user

There has to be a space after `[` and before `]`.

26.7 Exit Codes

The exit command can also be used to return an exit code, e.g., we could write exit 1. If no exit code is supplied when using the exit command, then the exit code of the previous command will be returned.

26.8 Example: A backup script

Now let us write a backup script that takes two command-line arguments: one is a source directory, and the other is a destination directory.

For each file in the source directory, the script copies it to the destination directory if and only if this file is a regular file and it does not exist in the destination directory. It also prints out the file name each time a file is copied.

For example, you can run the script by entering:

```
./backup.sh ~/assignment1 ~/backup/a1
```

You can find the code at:

```
/users/faculty/prof2132/public/backup.sh
```

and it is also included here:

```
#!/bin/bash
if [ ! -d $1 ]; then
  echo Source directory does not exist
  exit 1
elif [ ! -d $2 ]; then
  echo Destination directory does not exist
  exit 1
fi

for filename in `ls $1`
do
  if [ -f $1/$filename ]; then
    if [ ! -e $2/$filename ]; then
      cp $1/$filename $2/$filename
      echo $filename
    fi
  fi
done
```

27 Additional Examples: Dynamically Allocated Arrays

We will now focus on allocating array-based structures dynamically. We will start from strings, as string variables are essentially character arrays.

27.1 Dynamically Allocated Strings

We make use of the array-pointer connection to allocate strings dynamically. When we allocate storage, we should always set the size to be the string length $+1$, in order to have one byte for the null character. Strings allocated using dynamic memory allocation can also be used in C library functions for strings.

The following is an example of concatenating two strings without modifying either string. (This is a short example from the textbook). To implement a function for this, instead of storing the result in one string like the standard "strcat" does, we store the result in a new, dynamically-allocated string.

```
char* concat(const char *s1, const char *s2) {
  char *result;

  result = malloc(strlen(s1) + strlen(s2) + 1);
  if (result == NULL) {
    printf("Error: malloc failed in concat\n");
    exit(EXIT_FAILURE);
  }

  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

To use the concat function, we can write something like:

```
char *p;
p = concat("abc", "defg");
...
free(p);
```

Note that we should free `p` when we no longer use it.

Programming Exercise: Implement a function `getline` with the prototype:

```
char* getline();
```

which reads a line from the standard input, including the final new-line character, and returns a dynamically allocated string containing this line, including the new-line character at the end. If an EOF is encountered before reaching end of line, then there is no new-line character to be included.

27.2 More String Examples

Now, let us see a longer example, which is also a good example of C strings. In this example, we are asked to write a function to reverse words in a string. In this string, words are separated by spaces, and punctuation characters are treated the same as letters. For example, if the string is "Do or do not, there is no try.", then after we reverse its

words, we get “try. no is there not, do or Do”. As you can notice, we will consider a word to be any sequence of non-white-space characters. The following is the main idea of the implementation:

1. Scan the string backward to
2. As we identify words, copy them into a temporary buffer
3. Copy the buffer back to the original string

Here, this buffer can be created as a dynamically allocated string.

The fill-in-the blanks code is available at:

```
~prof2132/public/reversewords.c-blanks
```

and it is also included here:

```
/* CSCI 2132 Sample "fill-in blanks" code: reversewords.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define STRLEN 80

int read_line(char line[], int len);
void reverse_words(char* line);

int main(void) {
  char line[STRLEN+1];

  printf("Enter a line of text:\n");
  read_line(line, STRLEN);

  reverse_words(line);

  puts(line);

  return 0;
}

int read_line(char line[], int len) {
  int ch, i = 0;

  while ((ch = getchar()) != '\n' && ch != EOF)
    if (i < len)
      line[i++] = ch;

  line[i] = '\0';

  return i;
}

void reverse_words(char* line) {
  int len;
```

```
  char* buffer;
  int read_pos, write_pos, word_start, word_end;

  len = strlen(line);
  buffer = malloc(__________);
  if (buffer == NULL) {
    printf("out of memory in reverse_words.\n");
    exit(EXIT_FAILURE);
  }

  read_pos = len - 1;
  write_pos = 0;

  while (read_pos >= 0) {
    if (line[read_pos] == ' ') {
      buffer[write_pos++] = line[read_pos--];
    }
    else {
      word_end = read_pos;

      while (read_pos >= 0 && line[read_pos] != ' ')
        read_pos--;

      word_start = read_pos + 1;

      while (word_start <= word_end)
        buffer[write_pos++] = line[_____________];
    }
  }

  buffer[write_pos] = '\0';
  strcpy(line, buffer);

  ____________________;
}
```

It is possible to implement this function without using a buffer, which would improve space efficiency. You can try this for a homework and we will see how to do it later.

Reversing the Words in a Given String, Revisited

Let us revisit now the example `reversewords.c`, of the function that reverses the words in a string, but this time without requiring a temporary buffer.

To achieve this, we need try different ideas and make some observations. One idea is to simply reverse the entire string. This however would not work. For example, doing this for the string “lord of the rings” will give us ”sgnir eht fo drol”. We can however observe that by doing so, the words are in the desired order, and problem is that the letters in each word are not. Thus, we have an idea: let us reverse each word, after we reverse the entire string. This requires a function that can be used to reverse any substring of a given string, which is easy to implement only using an additional temporary character variable. You can check the code (with some blanks) at:

`~prof2132/public/reversewords2.c-blanks`

The code is also available here:

```
/* CSCI 2132 Sample "fill-in blanks" code: reversewords2.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define STRLEN 80

int read_line(char line[], int len);
void reverse_words(char* line);
void reverse_string(char* str, int start, int end);

int main(void) {
  char line[STRLEN+1];

  printf("Enter a line of text:\n");
  read_line(line, STRLEN);

  reverse_words(line);
  puts(line);

  return 0;
}

void reverse_string(char* str, int start, int end) {
  char temp;

  while (_______________) {
    temp = str[start];
    str[start] = str[end];
    str[end] = temp;

    start++;
    end--;
  }
}

void reverse_words(char* line) {
  int start = 0, end = 0, len;

  len = strlen(line);
  reverse_string(line, 0, len-1);

  while (end < len) {
    if (line[end] != ' ') {
      start = end;
      while (end < len && line[end] != ' ')
        end++;

      _______________
```

```
      reverse_string(line, start, end);
    }

    end++;
  }
}

int read_line(char line[], int len) {
  int ch, i = 0;

  while ((ch = getchar()) != '\n' && ch != EOF)
    if (i < len)
      line[i++] = ch;

  line[i] = '\0';

  return i;
}
```

27.3 Dynamically Allocated Arrays and VLAs

The size of an array is often not known at compile time. Again, to address this issue, we can make use of the array/pointer connection to allocate arrays during execution.

In this following short example, we allocate storage for an array of n integers (assume n is an `int` variable that we declared before which has been assigned a positive value), and set all elements to 0:

```
int *array, i;
array = (int*) malloc(n * sizeof(int));
if (array == NULL) {
  ...
}

for (i = 0; i < n; i++)
  array[i] = 0;

...
free(array);
```

We need to pay attention to the `sizeof` operator. It is important as the parameter of `malloc` is the size in bytes. Of course, a better way to allocate and clear the above array is:

```
int *array;
array = (int*) calloc(n, sizeof(int));
if (array == NULL) {
  ... /* error */
}
...
```

Now let us look at the severals lines of the code that we wrote in the last lecture regarding using dynamically allocated arrays. For what we did in that example, we can also use a variable-length array. When should we use a dynamically allocated array and when should we use a VLA?

First, it is important to know that dynamically allocated arrays are stored in heap and VLAs are stored in stacks. Thus large arrays should be declared as dynamically allocated arrays. Particularly, the recursive functions could quickly use all the provided stack space if they include a local large array.

Since stack allocation is faster than heap allocation, it is more efficient to use VLAs for small arrays, which are required only during execution of the function.

Finally, we should also consider the issue of portability. Dynamically allocated arrays have always been in the C programing language, while VLA is a new feature in C99. Even though most compilers support C99, there are exceptions. A notable exception is Visual C++, which is used widely on the Windows platform. Thus, if you need make sure that your program can be compiled using a wide range of compilers, it is a good idea to use dynamically allocated arrays.

Previously, in our implementation of mergesort on arrays, we use two variable-length arrays in the merge function. Would it be better to use dynamically allocated arrays instead, if we choose to release our code as a function in a C library? The answer would be positive, since client modules could use this function to sort very large arrays. To see an implementation based on dynamically-allocated arrays (with fill-in blanks) you can look at:

```
~prof2132/public/mergesort3.c-blanks
```

The code is also available here:

```
/* Program: mergesort3.c */
#include <stdio.h>
#include <stdlib.h>

void mergesort(int* array, int lower, int upper);
void merge(int* array, int lower, int middle, int upper);

int main() {
  int len, *p, *array;

  printf("Enter the length of the array: ");
  scanf("%d", &len);

  if (NULL == (array = (int*) malloc(len*sizeof(int)))) {
    printf("Insufficient memory!\n"); exit(EXIT_FAILURE);
  }

  printf("Enter %d integers:\n", len);

  for (p = array; p < array+len; p++)
    scanf("%d", p);

  mergesort(array, 0, len-1);

  printf("The sorted array is: \n");
  for (p = array; p < array+len; p++) {
    printf("%d", *p);
    if (p < array + len - 1)
      printf(" ");
  }
  printf("\n");
```

```
  return 0;
}

void mergesort(int* array, int lower, int upper) {
  if (lower < upper) {
    int middle = (lower + upper) / 2;
    mergesort(array, lower, middle);
    mergesort(array, middle+1, upper);
    merge(array, lower, middle, upper);
  }
}

void merge(int* array, int lower, int middle, int upper) {
  int len_left = middle - lower + 1;
  int len_right = upper - middle;
  int *left, *right;
  int *p, *q, *r;

  left = malloc(_____________ * len_left);
  if (left == NULL) {
    printf("Out of memory!\n");
    exit(EXIT_FAILURE);
  }

  right = malloc(_____________ * len_right);
  if (right == NULL) {
    printf("out of memory\n");
    exit(EXIT_FAILURE);
  }

  for (p = left, r = array + lower; p < left + len_left; p++, r++)
    *p = *r;

  for (q = right; q < right + len_right; q++, r++)
    *q = *r;

  p = left;
  q = right;
  r = array + lower;

  while (p < left + len_left && q < right + len_right) {
    if (*p <= *q)
      *r++ = *p++;
    else
      *r++ = *q++;
  }

  while (p < left + len_left)
    *r++ = *p++;

  while (q < right + len_right)
    *r++ = *q++;
```

```
   _____________________

   _____________________
}
```

27.4 Dynamic Arrays: Resizable Arrays

We can now use what we have learned to implement dynamic arrays, which support random access, and insertions and deletions of the element at the end of the array. This is similar to the ArrayList in Java, and the vector class in C++.

Initially, we dynamically allocate a memory block of a certain size to store elements in a dynamic array. However, when we keep inserting elements to the end of the array, we will eventually fill this memory block. Thus the challenge is to design an efficient algorithm to handle the case in which the array is full when we try to insert another element. The following is the pseudocode for this case:

```
If array is full
   Resize the array to twice its current capacity using realloc
   Store the new element
```

Why is this algorithm efficient? We will analyze it after we see the code.

With this algorithm, we can see that to implement a dynamic array, we need keep track of the address of its memory block, the capacity of this block, and the number of elements stored in this dynamic array. This suggests the using of a structure:

```
struct vector {
   int *array;
   int capacity;
   int size;
}
```

Here, the field capacity is the maximum number of integers that can be stored in the memory block that array points to, while size is the number of elements currently stored in this dynamic array.

A full fill-in-the-blanks implementation can be found at:

```
~prof2132/public/dynamicarray.c-blanks
```

The code is also available here:

```
/* Program: dynamicarray.c */
#include <stdio.h>
#include <stdlib.h>

#define GROW_FACTOR 2
#define RECIPROCAL_SHRINK_FACTOR 2
#define RECIPROCAL_MIN_LOAD 4

typedef struct {
  int* array;
```

```
  int capacity;
  int size;
} vector;

vector* create(int capacity);
int* at(vector* vec, int i);
void destroy(vector* vec);
void push_back(vector* vec, int value);
int  pop_back(vector* vec);

int main() {
  int i;
  vector* vec;

  vec = create(5);

  for (i = 0; i < 25; i++)
    push_back(vec, i);

  for (i = 0; i < 25; i++)
    ( *at(vec, i) )++;

  printf("The current capactiy of the array is %d\n", vec->capacity);
  for (i = vec->size-1; i >= 0; i--)
    printf("%d ", *( at(vec, i) ));
  printf("\n");

  for (i = 0; i < 21; i++)
    pop_back(vec);

  printf("The current capactiy of the array is %d\n", vec->capacity);
  for (i = vec->size-1; i >= 0; i--)
    printf("%d ", *( at(vec, i) ));
  printf("\n");

  destroy(vec);

  return 0;
}

vector* create(int capacity) {
  vector* vec = malloc(sizeof(vector));
  if (vec == NULL) {
    printf("Out of memory.\n");
    exit(EXIT_FAILURE);
  }

  vec->capacity = capacity;

  vec->array = malloc(__________ * capacity);

  if (vec->array == NULL) {
```

```
    printf("Out of memory.\n");
    exit(EXIT_FAILURE);
  }
  vec->size = 0;

  return vec;
}

int* at(vector* vec, int i) {
  if (i >= vec->size || i < 0) {
    printf("Array out of bounds.\n");
    exit(EXIT_FAILURE);
  }
  return &(vec->array[i]);
}

void destroy(vector* vec) {

  free(_____________);

  free(vec);
}

void push_back(vector* vec, int value) {
  if (vec->size == vec->capacity) {
    vec->array = realloc(vec->array, vec->capacity * GROW_FACTOR * sizeof(int));
    if (vec->array == NULL) {
      printf("Out of memory.\n");
      exit(EXIT_FAILURE);
    }
    vec->capacity *= GROW_FACTOR;
  }

  vec->array[vec->size++] = value;
}

int pop_back(vector* vec) {
  int lowercapacity;
  if (vec->size == 0) {
    printf("Attempting to remove the last element of an empty array.\n");
    exit(EXIT_FAILURE);
  }

  vec->size--;

  lowercapacity = vec->capacity / RECIPROCAL_MIN_LOAD;
  if (vec->size < lowercapacity) {
    vec->array = realloc(vec->array, lowercapacity * sizeof(int));
    if (vec->array == NULL) {
      printf("Unsuccessful vector shrinking!\n");
      exit(EXIT_FAILURE);
    }
```

```
    vec->capacity = ________________ ;
  }
}
```

In this program, the `push_back` function implements the insertion of an element at the end of the array. We adopt the function names used in the C++ vector class.

Why is our algorithm for `push_back` efficient? For a rigorous analysis we would need to use a method called amortized analysis. Instead of that we will describe the intuition behind it. Namely, we see that all operations on this vector are efficient, i.e., they require constant time, except the operation of adding a new element. Adding a new element can be a possibly long operation if we need to resize the array, and if the function `realloc` has to copy the whole current array into the new location. It this case, the operation may take $O(n)$ time, where we use n to represent the current array size. However, if we are using the array within an algorithm, we normally need to fill it from empty to a certain size. Let us say that we add m elements. To make computation easier, let us assume that $m = 2^k$, for some k. While adding these m elements, the `push_back` operation will use an order of $O(m)$ default operations, and when the vector needs to grow, we will be making possibly the following number of copy operations within `calloc`: 1, 2, 4, 8, ..., 2^k, which all add up to $1 + 2 + 4 + 8 + \ldots 2^k = 2^{k+1} - 1 = 2m - 1$. When we add this to the default $O(m)$ operations, we still get the same order of operations $O(m)$. If we "divide" this on the m `push_back` operations, we see that on average we spend only constant time per one `push_back` operation.

The Java implementation also uses this idea, and the only difference is that instead of doubling the capacity when resizing, it increases the capacity by 1/2. The above analysis still applies with minor modifications.

Finally, to support the deletion of the last element in the array, we can use a similar idea, which is to halve the capacity when the array is less than a quarter full. This is also implemented in the code given above. The choice of when to shrink the array is important, since we if decide for example to shrink it as soon as a half of the array is empty, we could be in a situation where adding and removing repeatedly one element could cause repeated expand and shrink operations, which would be expensive.

Having done this implementation, we know better about when to use a dynamic array (or ArrayList in Java and vector in C++). A dynamic array supports fast random access, insertions and deletions of the last element. It however might require the copying of a very large array after an update. Therefore, we need be careful when we use this in real-time programming, in which programs must guarantee response within strict time constraints (the ABS systems of a car is an example).