Parsing Command Lines: *Problem Statement*

- parsing command line arguments is a common part of Unix programs, but it is rather boring programming

- we would like to have an automatic way of parsing command lines, a single procedure that we could call that would handle this for us

- as a design exercise we will produce a command line parsing module. Our module isn't complete, but is usable as it is, and can easily be expanded

- we start by examining the basic problem of command line arguments, this problem is complicated by the fact that several different command line syntax's have been used for Unix programs

- Most arguments consist of a flag that identifies the argument and a value for the argument

- the flag usually consists of a prefix character, such as a + or -, that separates flags from file names, and then one or more characters that form the flag itself

- At this point we need to make a decision, if we restrict flags to one character, then we can string several flags together behind one prefix

character, this only works if the corresponding arguments are Boolean, and thus don't need a value

- since restricting ourselves to one character limits us to 26 possible arguments, we will allow multiple character flags. The loss isn't too great, since there are not many Boolean arguments

- thus a flag consists of a prefix, either + or -, and one or more alphabetic characters

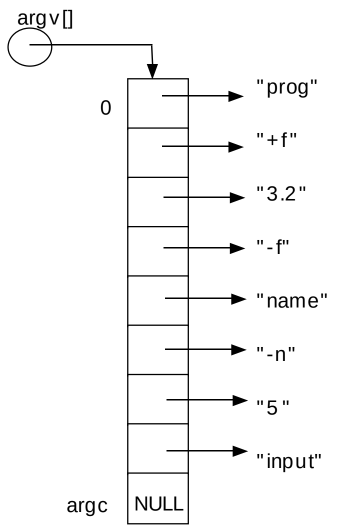
  • the value of an argument follows the flag, in the case of string values there must be a space between the flag and value. For numeric values this space isn't necessary, since digits can't appear in flags

- we must also be able to handle arguments that don't have flags, these are typically file names that contain the input to the program

- with this in mind, the following are examples of our argument syntax

```
prog -n 5 -f name
prog +f 3.2 -f name -n 5 input
```

```
prog +f 3.2 -f name -n 5 input
```



See man getopt for a package to decode command lines

Program Design

- we want one procedure that we can call to process all the arguments, we will call this procedure process_arguments

- what are the parameters to this procedure, what does it need to know

- it obviously needs to have the argc and argv parameters that are passed to our main procedure, but its also needs to know the arguments that it can expect to see on the command line

- how do we communicate the argument descriptions to this procedure, what information do we need to know about each argument?

- for each argument there are three things that we need to know: the flag for the argument, where the argument's value is to be stored, and the type of the argument's value

- we can store this information in a table, or array, with one entry for each parameter. This array is called an argument description table, and is one of the parameters that is passed to the process_arguments procedure

- how will process_arguments return the information that it finds?

* for arguments with flags this is easy, the argument description contains pointers to the variables where the values are to be stored

* in the case of file name arguments, there are no variables to return their values in, so we will return them as the value of process_arguments

* this procedure will return an array of string pointers, one for each file argument, that is terminated by a zero entry. This allows us to handle an arbitrary number of file name arguments

* Now that we have a basic idea of the data structures, how do we organize the code?

* process_arguments is the interface to our module that the rest of the program sees. It also contains the high level structure of the algorithm

* to process the arguments, we need to work our way through the argv array. There are three things that we can expect to see in this array: flags, argument values, and arguments without flags (what we have called file name arguments)

* an argument value will always have a flag before it, so we don't need to recognize them. We can easily find them after we have recognized a flag

* so we will have a while loop that steps along the argv array

* at each entry this loop will determine whether we have a flag, or a file name

* a flag always starts with a + or -, and a file name doesn't so its easy to recognize a flag

* once we have recognized a flag, we need to extract the flag from the argv array, find its entry in the argument table, and then extract the argument value from the argv array

* our high level algorithm is:

```
while (not end of argv) {
    if (current argv entry is a flag) {
        extract the flag from argv
        find flag in argument table
        extract and store the argument value
    } else {
        add argument to file name array
    }
}
```

* extracting the argument value depends upon the type of the argument, if the type is string, then the argument value must be in the next argv entry

* if its a numeric type, the value can be in the current argv entry, or if not there in the next argv entry

* the location of the value can be determined by comparing the length of the flag to the length of the argv entry. If they are the same, the value is in the next argv entry, otherwise it is part of the current entry

Implementation

* the implementation of this module is divided into two files: an include file that contains the main data structure declarations, and a .c file that contains the procedure code

* the .h file contains the declaration of the argument description table, the constants used for argument types, and the declaration of the process_arguments procedure

* the .c file contains the implementation of the process_arguments procedure, along with all the procedures that it calls to process arguments. These helper procedures are all declared static so they can't be called from outside of the module

http://www.cs.ualberta.ca/~tony/C201/Lectures/LecNotes/Examples/Cmds
is the example program

Module Design and Implementation

* a simple example is used to illustrate the design and implementation of modules.
This example is based on a stack of integers

* we want a module to produce a stack of integers.
There may be multiple instances of these stacks

* in this case a module encapsulates a data structure and the functions that are used to manipulate it

* the data structure is hidden in the module, it is only visible to the module's functions. Other functions can only manipulate it by calling stack module functions

* we view the stack module as a state machine, at any point in time each stack is in a particular state. The module's functions are used to move the stack from one valid state to another

* how do we design a module? One approach is to list the operations that we would like to perform, this works well with our example

* since we can have multiple stacks, we need functions to create and destroy stacks. In addition we will need routines to push and pop from the stack

* In addition to this basic capability, we need a few functions that return the stack's state, functions for testing whether the stack is full or empty, and one for peeking at the top element of the stack

* this gives us the following set of functions:

stack_create	stack_destroy
stack_push	stack_pop
stack_empty	stack_full
stack_peek	

* note that all of these functions have been prefixed by "stack". This prevents the names from clashing with any other function names in our program

* these functions will need some parameters, and possibly return values. The Stack type is used to represent a stack. All the other parameters will be integers

* at the same time we will add pre- and post-conditions to our functions

* for create and destroy we have the following:

```
Operator stack_create (int size) -> Stack
    Pre
        size > 0
    Post
        returns a new stack that can contain
        size items
End

Operator stack_destroy (Stack s) -> Stack
    Pre
        s is a valid stack
    Post
        the resources allocated to s are freed
        returns NULL
End
```

* for the push and pop functions we have the following:

```
Operator stack_push (Stack s, int item)
    Pre
        s is a valid stack
        s isn't full
    Post
        the stack increases in size by 1
        item is the new top element of s
End

Operator stack_pop (Stack s) -> int
    Pre
        s is a valid stack
        s isn't empty
    Post
        the returned value is the top element
        of the stack
        the top element of the stack is removed
End
```

* The remaining three functions are specified in the following way

```
Operator stack_full (Stack s) -> int
    Pre
        s is a valid stack
    Post
        returns True if s is full, otherwise
        returns False
End

Operator stack_empty (Stack s) -> int
    Pre
        s is a valid stack
    Post
        returns True if there are no elements
        in s, otherwise False
End

Operator stack_peek (Stack s) -> int
    Pre
        s is a valid stack
    Post
        returns top element of s,
        s is left unchanged
End
```

* Now that we have the specification, we can turn our attention to the implementation

* First we need to define the Stack type, and declare the stack module functions, this will be done in the stack.h include file

* we don't want functions outside of the module to be able to modify Stack, so we will define it in such a way that it can be stored in variables and passed a parameter, but it can't be manipulated

* this can be done by making Stack a struct pointer, but not declaring the underlying struct, this can be done in the following way:

```
typedef struct stack_struct* Stack;
```

* stack_struct prototype is declared in stack.h, but it is defined only in stack.c, so user functions can't manipulate stacks directly

* now we come to stack.c, the file that contains the actual implementation of the stack module

* First, we need to define the stack_struct type. For our simple implementation we will use an array to store the stack, since each stack

could be a different size we will need to dynamically allocate the array and store its size in stack_struct

* the declaration of stack_structure is:

```
typedef struct stack_struct {
      int size;    /* The size of the stack */
      int top;     /* Index of the top of the stack */
      int* data;   /* The actual stack data */
} ;
```

* the most complicated function in the module is stack_create, this function must dynamically allocate the storage required for a stack, and check that the allocation is successful, the other functions are quite easy to write

* Now that the implementation is complete, we need to test the stack module

* there are many types of tests that can be performed, the one we will do is a simple functional test, the basic idea is to show that all the functions operate correctly under normal conditions

* this type of test is used to show some confidence in the correctness of the module, it is usually performed after each modification to the module, and when its transferred to another computer

* in our test we will create a stack, perform a number of operations on it, and then destroy the stack. A more complete test would use multiple stacks to ensure that there is no cross-talk between instances

* after creating a stack, the new stack should be empty. This gives us a chance to test the stack_empty and stack_full procedures

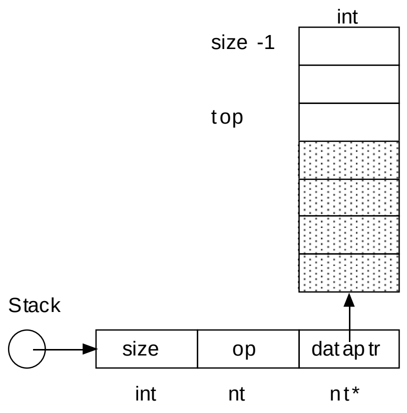
* next the stack is filled with integers, when this is finished the stack should be full, so we can again test stack_empty and stack_full

* next we pop the contents of the stack, and show that the correct sequence of integers is produced

* finally, we have another chance to test stack_empty and stack_full on what should be an empty stack

* the stack is then destroyed, and we check that the correct value is returned

http://www.cs.ualberta.ca/~tony/C201/Lectures/LecNotes/Examples/Stack
to see the example program



```
typedef  struct  stack_struct*  Stack;

typedef  struct  stack_struct  {
   int  size;       /*  The  size  of  the  stack  */
   int  top;        /*  Index  of  the  top  of  the  stack  */
   int*  dataptr;   /*  Pointer  to  stack  of  data  */
};
```