Contents

Malloc, free

Linked lists

Shell programming

Memory Management

Malloc, free

- In <stdlib> header file
- Heap memory

Sr.No.	Function & Description
1	**void *calloc(int num, int size);** This function allocates an array of **num** elements each of which size in bytes will be **size**.
2	**void free(void *address);** This function releases a block of memory block specified by address.
3	**void *malloc(size_t size);** This function allocates an array of **num** bytes and leave them uninitialized.
4	**void *realloc(void *address, int newsize);** This function re-allocates memory extending it upto **newsize**.

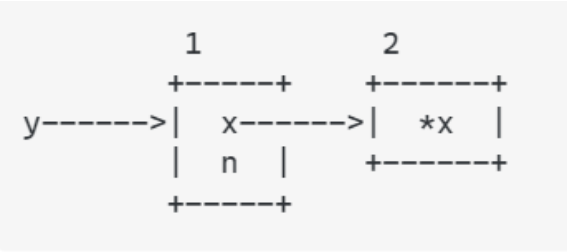
realloc

-可以变大变小

- 如果要变大，但是后面没有空间了，那么之前的pointer memory可能会被free掉然后去找新的，足够大的一块memory

Malloc with structs

```c
struct Vector {
    double *data;
    size_t size;
};
```




```c
struct Vector *vector = malloc(sizeof (struct Vector));
vector->data = malloc(sz * sizeof (double));
```

-如果struct里面有pointer就需要malloc pointers

-malloc先malloc struct

-free先free struct里面的pointer，再free struct

```c
free(vector->data);
free(vector);
```

LinkedList

```c
struct node
{
  int data;
  struct node *next;
};
```

```c
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data=3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

```c
void traverseList()
{
    struct node *temp;

    // Return if list is empty
    if(head == NULL)
    {
        printf("List is empty.");
        return;
    }

    temp = head;
    while(temp != NULL)
    {
        printf("Data = %d\n", temp->data); // Print data of current node
        temp = temp->next;                 // Move to next node
    }
}
```

```c
/* Function to reverse the linked list */
static void reverse(struct Node** head_ref)
{
    struct Node* prev = NULL;
    struct Node* current = *head_ref;
    struct Node* next = NULL;
    while (current != NULL) {
        // Store next
        next = current->next;

        // Reverse current node's pointer
        current->next = prev;

        // Move pointers one position ahead.
        prev = current;
        current = next;
    }
    *head_ref = prev;
}
```

Shell Scripts

Variables and Strings Declaration

```
VariableName='value'
echo $VariableName
or
VariableName="value"
echo ${VariableName}
or
VariableName=value
echo "$VariableName"
```

Note: There should not be any space around the "=" sign in the variable assignment. When you use **VariableName=value**, the shell treats the "=" as an assignment operator and assigns the value to the variable. When you use **VariableName = value**, the shell assumes that **VariableName** is the name of a command and tries to execute it.

Example:

```
yash9274@YASH-PC:~/GFG$ cat geekfile.sh
str1='welcome'
str2="to"
str3=GeeksForGeeks
echo $str1
echo ${str2}
echo "$str3"
yash9274@YASH-PC:~/GFG$ ./geekfile.sh
welcome
to
GeeksForGeeks
yash9274@YASH-PC:~/GFG$ cat geekfile1.sh
num = 100
echo $num
yash9274@YASH-PC:~/GFG$ ./geekfile1.sh
./geekfile1.sh: line 1: num: command not found

yash9274@YASH-PC:~/GFG$
```

Strings Manipulation

2. To print length of string inside Bash Shell: '#' symbol is used to print the length of a string.

Syntax:

```
variableName=value
echo ${#variablename}
```

Example:

```
yash9274@YASH-PC:~/GFG$ cat geekfile.sh
str=GeeksForGeeks
echo ${#str}
yash9274@YASH-PC:~/GFG$ ./geekfile.sh
13
```

5. Extract a substring from a string: In Bash, a substring of characters can be extracted from a string.

Syntax:

```
${string:position}  --> returns a substring starting from $position till end
${string:position:length} --> returns a substring of $length characters starting from $position.
```

Note: $length and **$position** must be always greater than or equal to zero.

If the **$position** is less than 0, it will print the complete string.

If the **$length** is less than 0, it will raise an error and will not execute.

Example:

```
yash9274@YASH-PC:~/GFG$ cat geekfile.sh
str="welcome to GeeksForGeeks"
echo ${str:-100}
echo ${str:7}
echo ${str:0:10}

yash9274@YASH-PC:~/GFG$ ./geekfile.sh
welcome to GeeksForGeeks
to GeeksForGeeks
welcome to
```

If statements

```
if SPACE [ SPACE "$foo" SPACE = SPACE "bar" SPACE ]
```

```
if [ ... ]
then
  # if-code
else
  # else-code
fi
```

Also, be aware of the syntax - the "`if [ ... ]`" and the "`then`" commands must be on different lines. Alternatively, the semicolon "`;`" can separate them:

```
if [ ... ]; then
  # do something
fi
```

You can also use the `elif`, like this:

```
if  [ something ]; then
 echo "Something"
 elif [ something_else ]; then
   echo "Something else"
 else
   echo "None of the above"
fi
```

```
#!/bin/sh
if [ "$X" -lt "0" ]
then
  echo "X is less than zero"
fi
if [ "$X" -gt "0" ]; then
  echo "X is more than zero"
fi
[ "$X" -le "0" ] && \
      echo "X is less than or equal to  zero"
[ "$X" -ge "0" ] && \
      echo "X is more than or equal to zero"
[ "$X" = "0" ] && \
      echo "X is the string or number \"0\""
[ "$X" = "hello" ] && \
      echo "X matches the string \"hello\""
[ "$X" != "hello" ] && \
      echo "X is not the string \"hello\""
[ -n "$X" ] && \
      echo "X is of nonzero length"
[ -f "$X" ] && \
      echo "X is the path of a real file" || \
      echo "No such file: $X"
[ -x "$X" ] && \
      echo "X is the path of an executable file"
[ "$X" -nt "/etc/passwd" ] && \
      echo "X is a file which is newer than /etc/passwd"
```

Note that we can use the semicolon (`;`) to join two lines together. This is often done to save a bit of space in simple `if` statements.
The backslash (\) serves a similar, but opposite purpose: it tells the shell that this is not the end of the line, but that the following line should be treated as part of the current line. This is useful for readability. It is customary to indent the following line after a backslash (\) or semicolon (`;`).

For Loops

```
#!/bin/sh
for i in 1 2 3 4 5
do
  echo "Looping ... number $i"
done
```

```
Looping .... number 1
Looping .... number 2
Looping .... number 3
Looping .... number 4
Looping .... number 5
```

```
#!/bin/sh
for i in hello 1 * 2 goodbye
do
  echo "Looping ... i is set to $i"
done
```

```
Looping ... i is set to hello
Looping ... i is set to 1
Looping ... i is set to (name of first file in current directory)
    ... etc ...
Looping ... i is set to (name of last file in current directory)
Looping ... i is set to 2
Looping ... i is set to goodbye
```

```
foo=string
for (( i=0; i<${#foo}; i++ )); do
  echo "${foo:$i:1}"
done
```

While Loops

```
#!/bin/sh
INPUT_STRING=hello
while [ "$INPUT_STRING" != "bye" ]
do
  echo "Please type something in (bye to quit)"
  read INPUT_STRING
  echo "You typed: $INPUT_STRING"
done
```

Cmd Arguments

$0 = program name

$@ = $1 - $9

$# = number of arguments

$?= exit value of the last command

var3.sh

```
#!/bin/sh
echo "I was called with $# parameters"
echo "My name is $0"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

```
$ ./var3.sh hello world earth
I was called with 3 parameters
My name is ./var3.sh
My first parameter is hello
My second parameter is world
All parameters are hello world earth
```

var4.sh

```
#!/bin/sh
while [ "$#" -gt "0" ]
do
  echo "\$1 is $1"
  shift
done
```

Another special variable is `$?`. This contains the exit value of the last run command. So the code:

```
#!/bin/sh
/usr/local/bin/my-command
if [ "$?" -ne "0" ]; then
  echo "Sorry, we had a problem there!"
fi
```

will attempt to run `/usr/local/bin/my-command` which should exit with a value of zero if all went well, or a nonzero value on failure.

Practice examples

```
#!/bin/sh
for k in *
do
   if [ -f $k ]; then
      str="`grep  $1  $k`"
      if [ -n $str ]; then
         rm –f $k
      fi
   fi
done
```

```
#!/bin/sh
for k in *
do
   if [ -f $k ]; then
      str=”`grep  $1  $k`”
      if [ -n $str ]; then
         rm –f $k
      fi
   fi
done
```

This is a **backtick**. **A backtick is not a quotation sign.** It has a very special meaning. Everything you type between backticks is evaluated (executed) by the shell before the main command (like `chown` in your examples), and the *output* of that execution is used by that command, just as if you'd type that output at that place in the command line.

So, what

```
sudo chown `id -u` /somedir
```

effectively runs (depending on *your user ID*) is:

```
sudo chown 1000 /somedir
  \     \      \      \
   \     \      \      `-- the second argument to "chown" (target directory)
    \     \      `-- your user ID, which is the output of "id -u" command
     \     `-- "chown" command (change ownership of file/directory)
      `-- the "run as root" command; everything after this is run with root privilege
```

```
#!/bin/sh
for k in *
do
   if [ -f $k ]; then
      str="`grep  $1  $k`"
      if [ -n $str ]; then
         rm –f $k
      fi
   fi
done
```

The shell script checks regular files (not a directory) in the current directory. These that contain string specified by the first argument will be deleted.

```
#!/bin/sh
if test $#  –gt  0
then
    if test  –f  $1
    then
        echo $1
    fi
    shift
    $0  $*
fi
```

```
#!/bin/sh
if test $#  –gt  0
then
   if test  –f  $1
   then
      echo $1
   fi
   shift
   $0  $*
fi
```

The shell script checks the script arguments in order. If it is a regular file in the current directory, then print it to the standard output.

```
#!/bin/sh
countf=0
countd=0
for i in *; do
  if test –f $i; then
    countf=`expr $countf + 1`
  fi
  if test –d $i; then
    countd=`expr $countd + 1`
  fi
done
echo Total of $countf regular files.
echo Total of $countd directories.
```

```
# Subtraction
[me@linux ~]$ expr 1 - 1
0
# Addition
[me@linux ~]$ expr 1 + 1
2
# Assign result to a variable
[me@linux ~]$ myvar=$(expr 1 + 1)
[me@linux ~]$ echo $myvar
2
# Addition with a variable
[me@linux ~]$ expr $myvar + 1
3
# Division
[me@linux ~]$ expr $myvar / 3
0
# Multiplication
[me@linux ~]$ expr $myvar \* 3
6
```