POINTERS

typedef *pointername

Pointers use 8 bytes of data and ALWAYS initialize the pointer by pointing it to a var

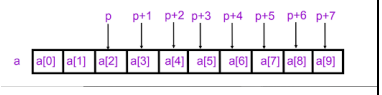
You can use **malloc()** and **free()** to dynamically allocate and release memory

When a pointer variable points to an array element, there is a notion of adding or subtracting an integer to/from the pointer.

```
int a[ 10 ], *p;
p = &a[2];
*p = 10;
*(p+1) = 10;
printf("%d", *(p+3));
```

```
int a[ 10 ], *p;

a[2] = 10;
a[3] = 10;
printf("%d", a[5]);
```



- If two pointers point to elements of a same array, then there are notions of subtraction and comparisons between the two pointers.

```
int a[10], *p, *q , i;
p = &a[2];
q = &a[5];
i = q - p;      /* i is 3*/
i = p - q;      /* i is -3 */
a[2] = a[5] = 0;
i = *p - *q;    /* i = a[2] – a[5]  */
p < q;          /* true */
p == q;         /* false */
p != q;         /* true */
```

POINTER TO POINTERS

int x;
int *px;
int ** ppx;
ppx = &px;
px = &x; /*ie *ppx = &x*/
****ppx = 10;** /*ie *px = 10; and x = 10; */

```
struct part {
  float price ;
  char name [10] ;
} ;
struct part *p , thing;
p = &thing;
/* The following three statements are equivalent */
thing.price = 50;
(*p).price = 50;   /* () around *p is needed */
p -> price = 50;
```

```
void free_list(List list)
{
  if (list != NULL) {
    free_list(list->next);
    free(list);
  }
}
```

- list.h

```
typedef int Data;
struct nodestr {
   Data data;
   struct nodestr *next;
};
typedef struct nodestr node;
typedef node* List;
```

```
BStree tree_ini(void) {
 BStree bst;
 bst = (BStree) malloc( sizeof(node*));
 *bst = NULL;
 return bst;
}
```

```
type definition for Key;
type definition for Data;

typedef struct node {
   Key *key;
   Data *data;
   struct node *left, *right;
} node;
 ypedef node **BStree;
```

```
/* insert a data d into list */
void add(List list; Data d) {
  Data * in_list;
  in_list = search(  list, d );
  if (in_list != NULL) return;  /* No need to change */
  else {
     node * newhead;
     newhead = malloc( sizeof(node) );
     newhead -> data = d;
     newhead -> next= list->next;
     list->next = newhead;
```

```
/* Search List by a data d */
Data * search(List list, Data d)
{
  node *head=list->next;
  for( head != NULL; head = head->next)
    if ( head -> data == d)  return &(head->data);
  return NULL;
}
```

```
void tree_free(BStree  bst) {
  btree_free(*bst);
  free(bst);
}
```

```
e_free(node *bt) {
if ( bt == NULL ) return;
btree_free(bt->left);
btree_free(bt->right);
(free bt->key and bt->data here, if needed)
free(bt);
```

FILES IN C

FILE *fopen(const char *filename, const char *mode); opens a file and returns a pointer to file. If fopen fails, NULL is returned

r	Open a file for reading. NULL if it does not exist
w	Create for writing. Destroy old if file exists
a	Open for writing. Create if not there. Start at end of file
r+	Open for update (NULL if it doesn't exist)
w+	Create the r/w. Destroy old if there
a+	Open for r/w. Create if not there. Start at end of file

fscanf(stdin, "%d", &int_var);
fprintf(stdout, "Hello there!\n");
fprintf(stderr, "An error has occurred!\n") - good for debugging
fscanf(FILE *fp, const char *fmt, …); for formatted file I/O

Read or write a single byte from or to a file

int fgetc(FILE *fp); returns the character that was read, converted to an integer
int putc(int c, FILE *fp); returns the same value parameter c if it succeeds, EOF otherwise

Read or write a string from or to a file

char *fgets(char *s, int n, FILE *fp); - reads an entire line into s, up to n-1 characters, returns a pointer s on success, NULL if an error or EOF is reached
int fputs(char *s, FILE *fp) returns the number of characters written if successful, EOF otherwise

Binary file reading and writing functions

int fwrite(void *buf, int size, int count, FILE *fp); - returns the number of items actually written
fwrite(fread) - returns the number of items actually written (read)
int fread(void *buf, int size, int count, FILE *fp)
fclose(FILE *fp) returns 0 if it works, -1 if it fails
int fflush (FILE *fp) - clear a buffer without closing it
void rewind(FILE *fp); - 'rewind' and start reading from the beginning of the file
long ftell(FILE *fp) determine where the position indicator is. Returns -1 if error occurs
int fseek(FILE *fp, long offset, int origin); sets the file position of the stream to the given offset

- **SEEK_SET** - beginning of file
- **SEEK_CUR** - current position of the file pointer
- **SEEK_END** - end of file

while ((c = fgetc(fp) != EOF) - reads characters until it encounters EOF
int feof(FILE *fp) - binary mode files; only realizes end of file after a reading fail

```
char *myStrchr(const char *s, char ch) {
char *cp=s;

  while (*cp!='\0') {
    if (*cp == ch) break;
    cp++;
  }
  if (*cp == '\0') cp = NULL;
  return cp;
}
```

```
char *findLast(const char *s, const char *searchStr) {

char *sp=NULL;
int ls= strlen(s), lss=strlen(searchStr);

  for (int i=ls-lss; i>=0; i--) {
    if (strncmp(s+i, searchStr, lss)==0) {
      sp = s+i;
      break;
    }
  }
  return sp;
}
```

STRING COMMANDS

scanf("%d", &x); skips leading white spaces and searches for character that are apart of numeric data and stops at the first character that is not part of the numeric data.
scanf("%c", &x); leading white spaces will not be skipped, to skip the leading white spaces, put a space character in the format string
scanf("%ns", &ch); scans the next n characters or up to the next white space
int puts(const char *str) - more efficient than printf
char *gets(char *str) - gets a line from stdin and is defined in stdio.h. Returns NULL upon failure and returns str otherwise
***fgets(char *buffer, int size, FILE *stream);** - safer than gets()
fgets() and gets() read a line whereas scanf() reads up to the next white space
#include <string.h>
size_t strlen(char *str); returns length of a NULL terminated character string
char *strcpy(char *destination, char *source); a copy of source is made at destination

- Source should be NULL terminated and return value points at destination

char *strcat(char *str1, char *str2); - appends a copy of str2 to the end of str1

- A pointer equal to str1 is returned

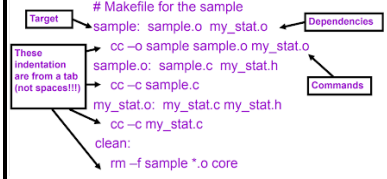
int strcmp(char *str1, char *str2); - does an ASCII comparison
int strncmp(char *str1, char *str2, size_t n); - compares n chars of str1 and str2
strcasecmp() and **strncasecomp()** - does the same but ignores cases in letters
char *strchr(char *str, int ch); - searches str until ch is found or NULL character is found. If found, ch is returned otherwise, NULL is returned
char *strstr(char *str, char *query); - searches str until query is found or a NULL character is found instead. If found, a pointer to str is returned otherwise NULL returned
int sprintf(char *s, const char *format, ..); prints out a string instead of stdout

- Can be used to convert integers or floating point numbers to strings

MAKEFILES

The make Utility (1)

- Programs consisting of many modules are nearly impossible to maintain manually.
- This can be addressed by using the make utility.



A Makefile Example (2)

- Stack contains StackImplementation.c and the following Makefile:

```
export: StackImplementation.o

StackImplementation.o: StackImplementation.c \
                       ../Include/StackTypes.h \
                       ../Include/StackInterface.h
        gcc -I../Include -c StackImplementation.c
# substitute a print command of your choice for lpr below
print:
        lpr StackImplementation.c
clean:
        rm -f *.o
```

The –I option for the C compiler specifies one or more paths on which the compiler can look in order to find .h files that have been referenced in the .c files with #include statements.

To invoke the make use command:

- *make target*: where target is one of the targets listed in the makefile
- If not target is specified when make is involved, it will build the target of the first rule

SHELL PROGRAMMING **find -executable** prints all executable files in cd

For Loop

- for loops allow the repetition of a command for a specific set of values
- Syntax:

```
for var in value1 value2 ...
do
  command_set
done
```

– command_set is executed with each value of var (value1, value2, ...) in sequence

While Loop

- While loops repeat statements as long as the next Unix command is successful.
- For example:

```
#!/bin/sh
i=1
sum=0
while [ $i -le 100 ]; do
  sum=`expr $sum + $i`
  i=`expr $i + 1`
done
echo The sum is $sum.
```

Until loop

- Until loops repeat statements until the next Unix command is successful.
- For example:

```
#!/bin/sh
x=1
until [ $x -gt 3 ]; do
  echo x = $x
  x=`expr $x + 1`
done
```

We can use backquotes to assign the output of a command to a variable
read var1 var2 var3… reads a line of input from standard input and assigns first word to var1, second word to var2, and the last variable gets any excess words on the line

Positional parameters
$1 - first argument after command
$2 - second argument after command … up to **$9**
$0 - name of the command running
$* - all the arguments
$# - the number of arguments
shift - shifts all arguments to the left
$$ - process id of the current shell
$? - return value of the last foreground process to finish

Test
-eq : equal to
-ne : not equal to
-gt : greater than
-lt : less than
-le : less than or equal to
-n stg : string stg is not a null string
-z stg: string stg is assigned and not null
Testing if two statements are equivalent **test $x -eq $y** or **[$x -eq $y]**

expr : performs four basic arithmetic operations and the modulus function

- Can determine the length of the string and return a substring by putting two strings separated by a colon

> - standard output can be redirected the same
< - standard input is redirected the same
2> - redirecting stderr
2>&1 - redirecting stderr and standard output

int remove(const char *filename) - erases a file. Returns 0 if deleted, -1 otherwise
int rename(const char *oldname, const char *newname) returns 0 if successful, -1 if error occurs
char *tmpnam(char *s) creates a filename that does not conflict with any other existing files.

grep command

TABLE 10.1 grep Options

Option	Significance
-i	Ignores case for matching
-v	Doesn't display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Displays list of filenames only
-e *exp*	Specifies expression *exp* with this option. Can use multiple times.
	Also used for matching expression beginning with a hyphen.
-x	Matches pattern with entire line (doesn't match embedded patterns)
-f *file*	Takes patterns from *file*, one per line
-E	Treats pattern as an extended regular expression (ERE)
-F	Matches multiple fixed strings (in **fgrep**-style)
-*n*	Displays line and *n* lines above and below *(Linux only)*
-A *n*	Displays line and *n* lines after matching lines *(Linux only)*
-B *n*	Displays line and *n* lines before matching lines *(Linux only)*

```
size_t strlen(const char *s){
  size_t nl
  for (n=0l *s != '\0'; s++){
    n++;
  }
  return n;
}
char *strcat(char *s1, char *s2){
  char *p = s1;
  while(*p)
    p++;
  while ((*p)) = *s2++));
  return s1;
}
int strcmp(char *s1, char *s2){
  int i;
  for (i = 0; s1[i]==s2[i]; i++){
    if (s1[i] == '\0')
      return 0
  }
  return *s1 - *s2;
}
```

-a dot . matches any single character
-**[..]** matches a range of characters
-**[..]*** match repeated sets of the characters in range
-**\(..\)** refers to the most recent search
Match any of wxyz and [u-z]: match char in range u-z: **grep '[wxyz]'**
Match any number of vowels: grep '[aeiou]*'
Pattern must not contain any character in set: **grep '[^...]'**

[:lower:]	**For a-z**	Match any non-vowel :**grep ' [^aeiou]'**
[:upper:]	**For A-Z**	Matches any line containing no lower case letter: **grep '^[^a-z]*$'**
[:alpha:]	**For A-Za-z**	
[:digit:]	**For 0-9**	Ignores the special meaning of the character following it: **\(backslash)**:
[:alnum:]	**For A-Za-z0-9**	

^KEYWORD: match beginning of line
KEYWORD$: match the end of a line
Matching part of words
Match the beginning of a word with \<
Matches Fo if it appears at the beginning of a word: **\<Fo**:
Match the end of a word with \>
Matches ox if it, appears at the end of the word: **ox\>**
*Can match whole words (\<Fox\>)
X\{m,n\}: matches m -- n repeats of the one character regular expression X
Match all sequences of 2 to 10 lower case letters: **grep '[a-z]\{2,10\}'**
X\{m\}: matches exactly m repeats of the one character regular expression X
Match 23 #s: **grep '#\{23\}'**
X\{m,\}: matches at least m repeat of the one character regular expression X
Match at least 2 vowels in a row at the beginning of a line: **grep '^[aeiou]\{2,\}'**
Match more than 0 characters: **grep '\{1,\} '**
Find all the lines in a file that contains at least one alphabetic character: **grep '[[:alpha:]]'**
Find all the lines in a file that contain at least one non-alphabetic character: **grep '[^a-zA-Z]'**
Find all the lines in a file that do not contain even a single non-alphabetic character: **grep -v '[a-zA-Z]' filename**
Find all the lines in a file that do not contain even a single non-alphabetic character: **grep -v '[^a-zA-z]' filename**
Find all the lines in a file, display all the lines found, preceded by their line numbers: **grep -c '^$' filename**
How many words in usr/dict/words end in ing? **grep -c 'ing$' /usr/share/dict/words**
How many words start with un and end with g? **grep -c '^un.*g$' /usr/share/dict/words**
How many words begin with a vowel? **grep -ic '^[aeiou]' /usr/share/dict/words**
How many words have triple letters in them? **grep -ic '\(.\)\1\1' /usr/share/dict/words**
How many words start and end w same 3 letters? **grep -c '^\(..\).*1$' /usr/share/dict/words**
What are the 5 palindromes present in…? **grep -ic '^\(.\)\(.\).\2\1$; /usr/share/words**
How many words have a y as their only vowel? **grep '^[aAeEiIoOuU]*$' /usr/words**
How many words don't start and end w the same 3 letters? **grep -ivc '^\(...\);.*\1$ /usr/share/dict/words**

EXAM POINTER QUESTIONS

```
typedef struct thing{
        char *who;
        int marks[10];
        char code;
}Thing;
void fnOne(int k, int *iptr);
void fn
```

FALSE
fnOne(t.marks[2], p->marks[2]);
fnOne(int(p->code), &((*p).arks[2]));
fnOne(t->marks, p->marks);
fnTwo(t.who, p->who);
fnTwo(t.who[0], p->who[0]);
fnTwo(*((*p).who, &(p->who));
fnTwo(p->who->, &(p->code));
TRUE
fnOne((*p).marks[2], &(t.marks[2]]));
fnOne(617, (*p).marks);
fnTwo(t.code, p->who);
fnTwo(*(t.who), &(t.code));

Write a bash that counts the number of regular files and number of subdirectories

```
#!/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 $countd direc
```

```
size_t strlen(const char *s){
  const char *p = s;
  while (*s)
    s++;
  return s-p;
}
char *strcpy (char *s1, char *s2){
  char *p = s1;
  while (*s2 != '\0'){
    *p = *s2;
    p++;
    s2++;
  }
  *p = '\0';
  return s1;
}
int strcmp(char *s1, char *s2){
  while (*s1++ == *s2++){
    if (!*s1)
      return 0;
  }
  return *(--s1) - *(--s2);
}
```

INTERACTING WITH UNIX

#include <unistd.h>
char *getcwd(char *buf, size_t size); get the current directory
int chdir(const char *path); changing the current directory
<time.h>
time_t time(time_t *time);
struct tm *localtime(const time_t *time);
char *asctime(const struct tm *timeptr);
Piping to ad from other programs
FILE *popen(const char * command, const char *mode);
int pclose(FILE *fp) r mode returns file pointer that can be used to read the standard output of command. W mode returns a file pointer that can be used to write to the standard input of command. Returns NULL on error
execl() loads a new program and uses it to replace the current process
int execl(const char *path, const char *arg(), … const char *argn, char */*NULL*/);
Multi-process programming
pid_t fork() creates a new process. The new process (child process) is an exact copy of the calling process (parent process). Only difference is return value. Child process gets 0 if successful, parent gets process id of child or -1 on errors

I/O Redirection
stdin: default place where program read **</0<**
stdout: default place where program write **>/1>**
stderr: default place where errors recorded **2>**

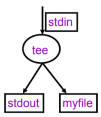
ps: list the current processes
-ps -l: gives you a long listing of what is going on
-u loginid: tells you about loginid's processes

Redirecting standard output
cat file1 file2 > file3: concatenates fil2 and file2 into file3 (file3 is created if it does not exist)
cat file1 file2 >| file3: file3 is overridden if it exists
cat file1 file2 >> file3: if file3 exists, it is appended, gives a warning if file3 does not exist
cat > file3: file3 is created from whatever user provides from stdin

Filters
Filters read from stdin, modify it, write result on stdout
ex: tr(translate) read from stdin, echo to stdout'
grep "unix is easy" < myfile1 > myfile2 - write all lines of myfile1 containing the phrase **unix is easy** to myfile2
tr read stdin echo to stdout translating some specified characters
nl numbers the line
grep patternstr read stdin and write lines containing stdout
wc: count the number of chars/words/lines on stdin. Writes the resulting statistics to stdout
sort: sort all the input lines in alphabetical order and write to the standard output

Quoting
Double quotes: stops interpretation of some shell special characters (or whitespace)
Single quotes: stops interpretation of even more specials
-stops variable expansion ($HOME)

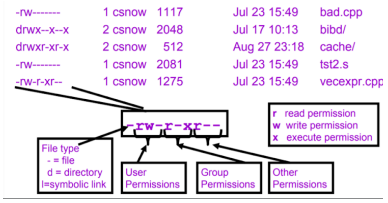
The tee Command
- tee - replicate the standard output
  - cat readme.txt | tee myfile



Pipes
Stdout of command 1 used as stdin for command 2 **command1 | command2**
Name of the command always comes first on the line
readme.txt > grep unix | wc -l WRONG alt:
cat readme.txt | grep unix | wc -l
grep unix < readme.txt | wc -l
grep unix readme.txt | wc -l
grep -c unix readme.txt

FILE SECURITY AND PERMISSIONS

ls -l view permissions
Split to three categories
1)User - owner of the file
2) Group users - your group and yourself
3)Others - everyone else
chmod: change permissions
-chmod [ugoa][+-=][rwx]file[...]
Gives user read, write and execute
chmod u+rwx file
Gives the group read and execute
chmod g+rx file
Gives all execute permission: **chmod a+x file**
Gives group read permission and make sure it has nothing else: **chmod g=r file**
Symbolic mode: **chmod u=rwx, g-w,o-rwx file**



1	2	3	4	5	6	7
--x	-w-	-wx	r--	r-x	rw-	rwx

Shell script checks files in cd and files that contain string specified by the first argument will be deleted

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

Shell script checks the script arguments in order. If it is a file in cd, print it to stdout

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