ARRAYS, ADDRESSES & POINTERS (ptrs)

some syntax: addresses & pointers

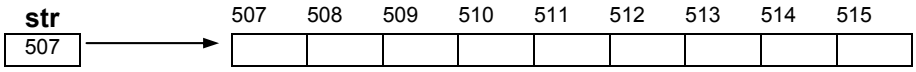
&x the memory address of x
*x the location addressed/referenced by x

memory allocation for arrays

eg: char str[Asize]

1. find Asize bytes of continuous memory
2. set up str as a variable of type *memory ptr to char*
3. set the value of str to the address of the 1st byte of the Asize block

layout of allocated memory eg: char str[9]



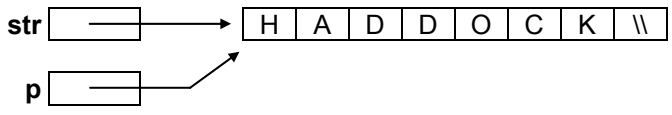
with the array str
str is equivalent to &str[0]
*str is equivalent to str[0]

When an arrays is passed as an argument or assigned only the ptr to the array structure is passed, ie: the address of its 1st element (element zero). Consider the following example:

after the following statements:

```
char str[] = "HADDOCK";
char *p;

p = str;
```

the underlying structures of **str** and **p** are:



some examples

1.	char *p;	declares p as same type as str, a ptr to a char structure
2.	p = str; p = &str[0];	these are legal operations, both set p to the same value
3.	str[0] = 'X'; *p = 'X';	two more legal operations both have the same result
4.	c = *p; c = str[0];	another two - both have the same result assume declaration: char c;
5.	c = str[1]; c = *(p + 1)	equivalent statements, C converts array indexes into ptr references
6.	c = str[i]; c = *(p + i)	equivalent statements

Using pointers to access arrays, EG: calculating the sum of an array of numbers

Using array indexing

```
int calcsum( int *a, int size )
{   int i, sum = 0;
    for( i=0; i < size; i++ )
        sum += a[i];
    return sum;
}
```

Using pointers (version 1)

```
int calcsum( int *a, int size )
{   int i, sum = 0;
    for( i=0; i < size; i++ )
    {   sum += *a;                  }
        a++;                        }   sum += *a++;
    }
    return sum;
}
```

Using pointers (version 2)

```
int calcsum( int *a, int size )
{   int *endp, sum = 0;
    endp = a + size;
    while( a < endp )
        sum += *a++;
    return sum;
}
```

Comparing index & pointer access for efficiency

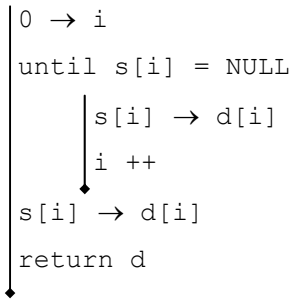
`// using array indexing`
 	
`int calcsum( int *a, int size )`
`{  int i, sum = 0;`
`  for( i=0; i < size; i++ )`
`  sum += a[i];`
`  return sum;`
`}`
	`// using pointers (version 2)`
	
	`int calcsum( int *a, int size )`
	`{  int endp, sum = 0;`
	`  endp = a + size;`
	`  while( a < endp )`
	`  sum += *a++;`
	`  return sum;`
	`}`

`sum = 0`
`0 → i`
`LOOP`
`  i < size`
`  sum + [ [a]+i ] → sum`
`  i + 1 → i`
`LOOPEND`
	`sum = 0`
	`a + size → endp`
	`LOOP`
	`  a < endp`
	`  sum + [a] → sum`
	`  a + 1 → i`
	`LOOPEND`

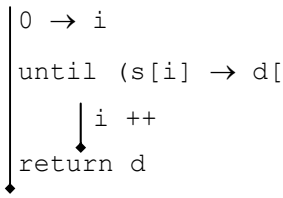
Another example: strcpy

char *strcpy(char *d, char *s)
copy source string s to destination string d, return d

Using array indexing



Array indexing with an embedded assignment:



In C

```
char *strcpy( char *d, char *s )
{   int i = 0;
    while( (d[i] = s[i]) != NULL )
        i++;
    return d;
}
```

NB: the **!= NULL** can be omitted since NULL == 0

Using pointers

```
d → dcopy
until ( s↑ → d↑ ) = NULL
    s++
    d++
return dcopy
```

In C

```
char *strcpy( char *d, char *s )
{   char *dcopy = d;
    while( *d = *s )
    {   s++;
        d++;
    }
    return dcopy;
}
```

Or more concisely:

```
char *strcpy( char *d, char *s )
{   char *dcopy = d;
    while( *d++ = *s++ )
        ;
return dcopy;
}
```

Comparison of efficiency

// using indexing
 	
while(d[i] = s[i])
  i++;
	// using pointers
	
	while(*d++ = *s++)
	  ;

LOOP
  [[s]+i] → [[d]+i]
  i + 1 → i
LOOPEND
	LOOP
	  [s] → [d]
	  s + 1 → s
	  d + 1 → d
	LOOPEND

A third (& final) example: strcmp

Using indexing:

```
0 → i
until (a[i] ≠ b[i]) or (a[i] = NULL)
    i++
return a[i] - b[i]
```

In C:

```
int strcmp( char *a, char *b )
{   int i;
    for( i=0; a[i] && a[i] == b[i]; i++ )
        ;
    return a[i] - b[i];
}
```

Using pointers

```
until (a↑ ≠ b↑) or (a↑ = NULL)
    a++
    b++
return a↑ - b↑
```

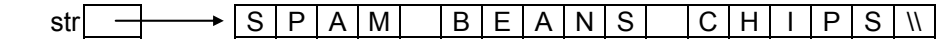
```
int strcmp( char *a, char *b )
{   for( ; *a && *a == *b; a++, b++ )
        ;
    return *a - *b;
}
```

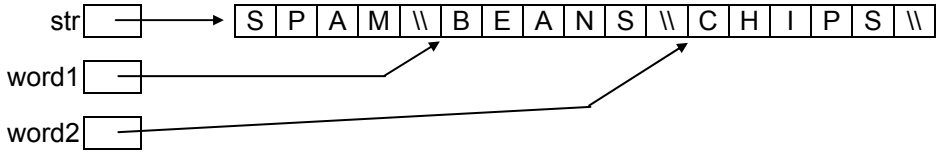
What is wrong with this....

```
int strcmp( char *a, char *b )
{   while( *a && *a++ == *b++ )
        ;
    return *a - *b;
}
```

Slicing Strings

Remember: a string is a pointer to a sequence of characters which is terminated with null.

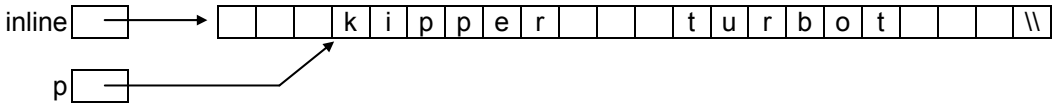




The following example is based on the attached program.

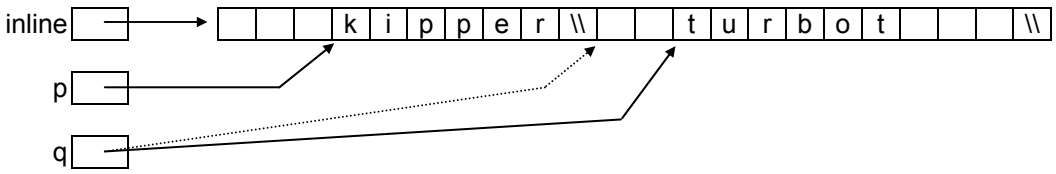
STAGE 1

program output:
```
inline  '   kipper   turbot   '
p       'kipper   turbot   '
```



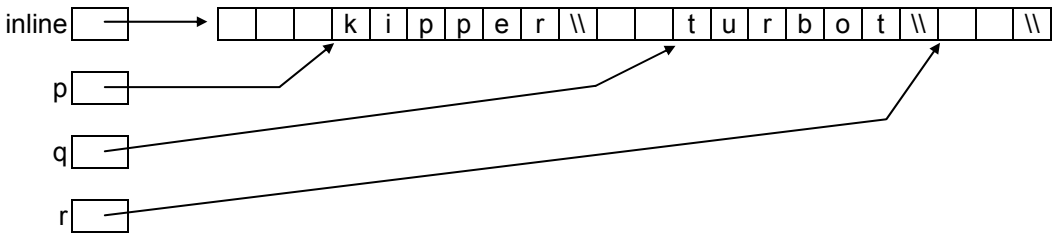
STAGE 2

program output:
```
inline  '   kipper'
p       'kipper'
q1      '  turbot   '
q2      'turbot   '
```



STAGE 3

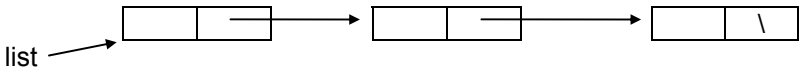
program output:
```
inline  '   kipper'
p       'kipper'
q       'turbot'
r       '  '
```



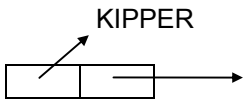
```
#include <stdio.h>

#define SPACE ' '

void display( char *name, char *val );

int main()
{   char inline[] = "   kipper   turbot   ";
    char *p;
    char *q;
    char *r;

    p = inline;

    while( *p == SPACE )
           p++;

    // stage 1
    display( "\ninline", inline );
    display( "p     ", p );

    q = p;

    while( *q != SPACE )
           q++;

    *q++ = NULL;

    // stage 2
    display( "\ninline", inline );
    display( "p     ", p );
    display( "q1    ", q );

    while( *q == SPACE )
           q++;

    display( "q2    ", q );

    r = q;

    while( *r != SPACE )
           r++;

    *r++ = NULL;

    // stage 3
    display( "\ninline", inline );
    display( "p     ", p );
    display( "q     ", q );
    display( "r     ", r );

    return 0;
}

void display( char *name, char *val )
{    printf( "%s  '%s'\n", name, val );
}
```

Dynamic Data Structures & Linked Lists

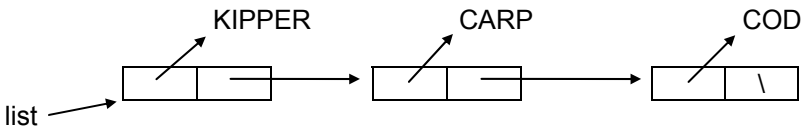
A linked list is a series of nodes chained together by pointers



A node has two parts: a data part & a link part. Eg a node with a string as its data part:



A list of strings:



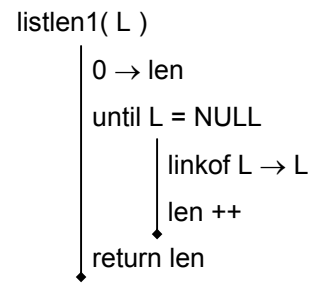
Structure type for nodes:

```
#define DATASIZE 10

typedef char Nstring[ DATASIZE ];

typedef struct Nstruct
{   Nstring data;
    struct Nstruct *link;
} Node;

Node *list;
```

Node access	Ptr refernece	Equivalent using ->
data part of 1st node	`(*list).data`	`list->data`
link part of 1st node	`(*list).link`	`list->link`
data part of 2nd node	`(*(*list).link).data`	`list->link->data`

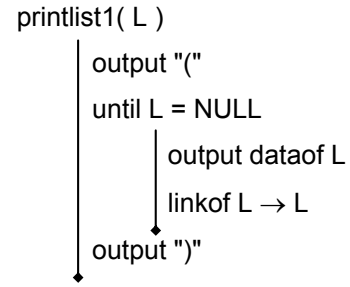
Functions which work through a complete list

listlen counts the number of elements in a list




```
int listlen1( Node *l )
{   int len = 0;
    while( l )
    {   l = l->link;
        len++;
    }
    return len;
}
```

printlist prints the data part of each element of a list




```
void printlist1( Node *l )
{   printf( "( " );
    while ( l )
    {   printf( "%s ", l->data );
        l = l->link;
    }
    printf( ")\n" );
}
```

Allocating memory at run-time

Given the declarations:

```
#define SIZE 10
int *table;
```

Memory can be allocated using malloc

```
table = (int *) malloc( SIZE * sizeof( int ));
if (table == NULL)
{   // malloc failed
    printf( "\nOut of memory, program terminates\n" );
    return 1;
}
```

Note malloc does not initialise allocated memory

```
printf( "contents of table\n" );
for( i=0; i < SIZE; i++ )
    printf( "\t%i\n", table[i] );
```

```
contents of table
    3
    0
    15
    0
    981
    0
    4096
    0
    -6400
    3
```

Adding new list elements

newnode builds a new list element

```
Node *newnode( Nstring word, Node *lnk )
{   Node *n;
    if (n = (Node *) malloc( sizeof(Node) ))
    {   strcpy( n->data, word );
        n->link = lnk;
        return n;
    }
    else /* memory allocation fault */
        return NULL;
}
```

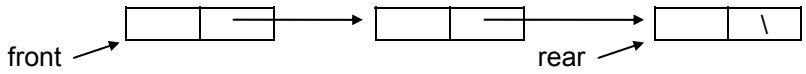
Removing list elements

destroy frees the first element of a list & returns the rest of the list

```
Node *destroy( Node *l )
{   Node *p;
    if (l)
    {   p = l->link;
        free( l );
        return p;
    }
    else
        return NULL;
}
```

A Queue

A queue can be represented by a list & 2 pointers indicating the front & rear of the queue.




```
typedef struct Qstruct
{   Node *front;
    Node *rear;
} Queue;
```

Add an item to a queue.

```
void add_queue( Queue *Q, char *name )
{   if (Q->front == NULL)
        // Q is empty
        Q->front = Q->rear = newnode( name, NULL );
    else
        // non empty Q, at least 1 node
        Q->rear = Q->rear->link = newnode( name, NULL );
}
```

Remove an item from a queue.

```
char *off_queue( Queue *Q, )
{   if (Q->front == NULL)
        // Q empty
        return NULL;
    else
    {   char *name;
        Node *tmp;
        if (name = (char *)
               malloc( strlen( Q->front->data ) * sizeof(char) ))
        {   strcpy( name, Q->front->data );
            tmp = Q->front;
            Q->front = Q->front->link;
            free(tmp);
        }
        return name;
    }
}
```