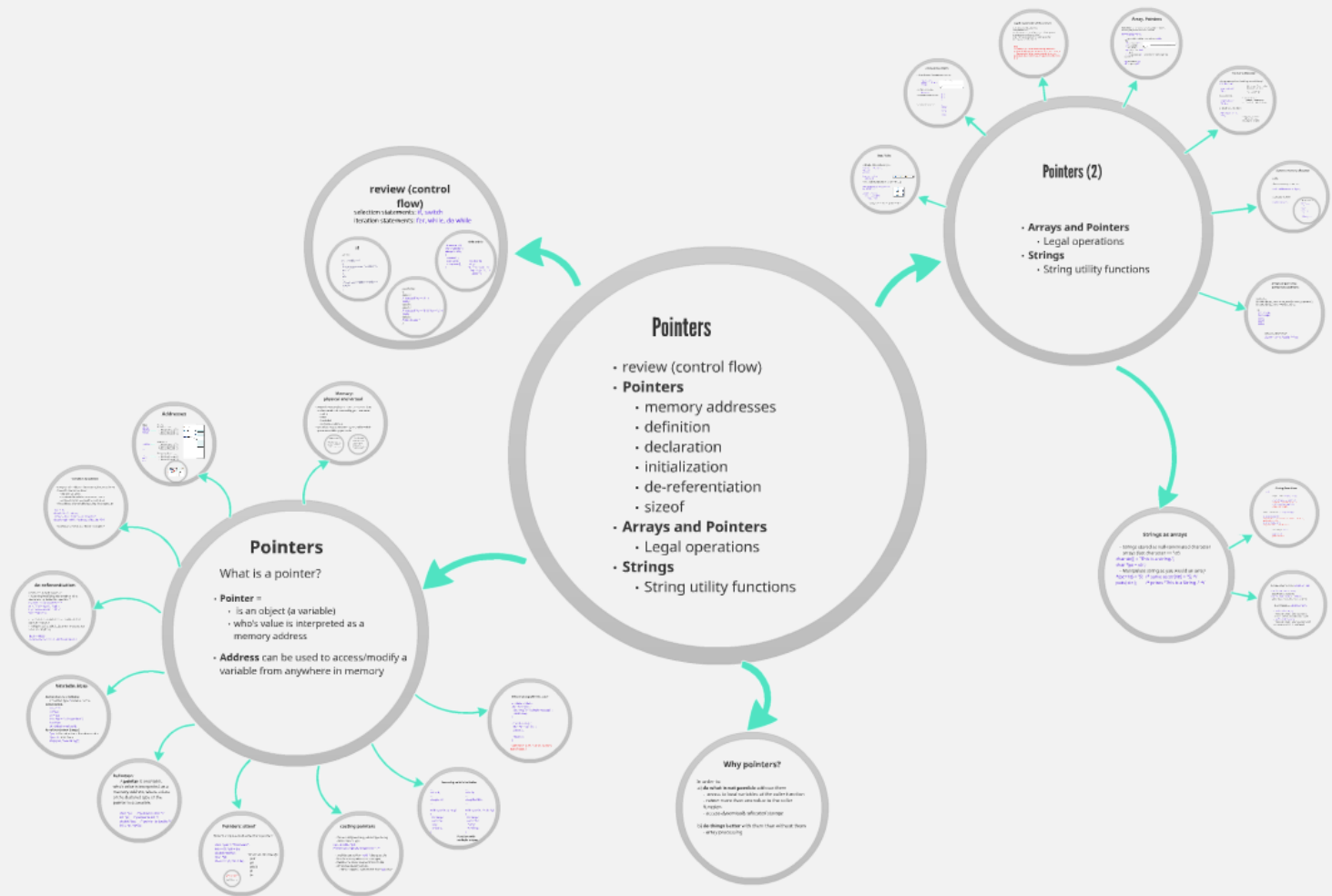


Pointers (1)



Pointers

- review (control flow)
- **Pointers**
 - memory addresses
 - definition
 - declaration
 - initialization
 - de-referentiation
 - sizeof
- **Arrays and Pointers**
 - Legal operations
- **Strings**
 - String utility functions

```
while and for  
int counter = 0;  
char c=getchar();  
while(c != '\n')  
{  
    counter++;  
    putchar(c);  
    c=getchar();  
}  
  
int a[20];  
int i;  
for(i = 0; i < 20; i++)  
    a[i] = i;
```

review (control flow)

selection statements: **if, switch**

iteration statements: **for, while, do while**

if

syntax:

```
if ( <condition> )
{
  /*code is executed if condition is true*/
}
else
{
  /*code is executed if condition is false*/
}
```

while and for

```
int counter = 0;
char c=getchar();
while(c != EOF)
{
  counter++;
  putchar(c);
  c = getchar();
}

int a[20][10];
int i, j;
for(i = 0; i < 20; i++)
  for(j = 0; j < 10; j++)
    a[i][j] = i;
```

switch (Var)

```
{
  case 'A':
    /* execute if Var == 'A' */
    break;
  case 'B':
  case 'C':
    /* execute if Var == 'B' || Var == 'C' */
    break;
  default:
    /*default code */
}
```



if

syntax:

```
if ( <condition> )  
{  
/*code is executed if condition is  
true*/  
}  
else  
{  
/*code is executed if condition is  
false*/  
}
```

S
{
C
/*
L

n is

```
switch (Var)
{
case 'A' :
/* execute if Var == 'A' */
break ;
case 'B' :
case 'C' :
/* execute if Var == 'B' || Var == 'C' */
break;
default :
/*default code */
}
```

while and for

```
int counter = 0;
char c=getchar();
while(c != EOF)
{
    counter++;
    putchar(c);
    c = getchar();
}
```

```
int a[20][10];
int i, j;
for(i = 0; i < 20; i++ )
    for(j = 0; j < 10; j++)
        a[i][j] = i;
```

Pointers

- review (control flow)
- **Pointers**
 - memory addresses
 - definition
 - declaration
 - initialization
 - de-referentiation
 - sizeof
- **Arrays and Pointers**
 - Legal operations
- **Strings**
 - String utility functions

```
while and for
int counter = 0;
char c=getchar();
while (c!='\n')
{
    counter++;
    putchar(c);
    c=getchar();
}
int a[20];
for(i = 0; i < 20; i++)
a[i] = i;
```


Pointers

What is a pointer?

- **Pointer** =
 - is an object (a variable)
 - whose value is interpreted as a memory address
- **Address** can be used to access/modify a variable from anywhere in memory

Memory: physical and virtual

- *Physical memory*: physical resources where data can be stored and accessed by your computer
 - cache
 - RAM
 - hard disk
 - removable storage
- *Virtual memory*: abstraction by OS, addressable space accessible by your code

Physical memory

- Different sizes and access speeds
- Memory management – major function of OS
- Optimization – to ensure your code makes the best use of physical memory available
- OS moves around data in physical memory during execution
- Embedded processors – could have many limits

Virtual memory

- How much physical memory do I have?
 - 2 MB (cache) + 3 GB (RAM) + 160 GB (HDD)
 - ...
- How much virtual memory do I have?
 - <4 GB (32-bit OS), typically 2 GB for Windows, 3-4 GB for Linux
- Virtual memory maps to different parts of physical memory
- Usable parts of virtual memory: **stack** and **heap**
 - stack: where declared variables go
 - heap: where dynamic memory goes

Physical memory

- Different sizes and access speeds
- Memory management – major function of OS
- Optimization – to ensure your code makes the best use of physical memory available
- OS moves around data in physical memory during execution
- Embedded processors – could have many limits

Virtual memory

- How much physical memory do I have?
 - 2 MB (cache) + 3 GB (RAM) + 160 GB (HDD)
+ ...
- How much virtual memory do I have?
 - <4 GB (32-bit OS), typically 2 GB for Windows, 3-4 GB for linux
- Virtual memory maps to different parts of physical memory
- Usable parts of virtual memory: **stack** and **heap**
 - stack: where declared variables go
 - heap: where dynamic memory goes

Addresses

File.c
char a;
short b;
long c;

...

...a+b+c...

...

a=...
b=...
c=...

File.obj

Reserve space for

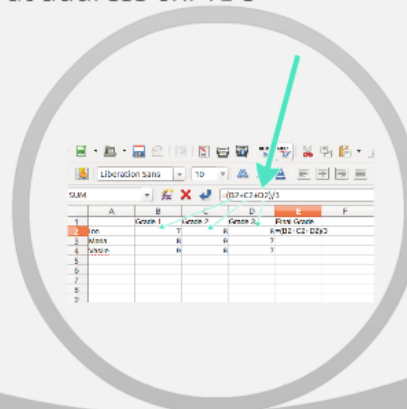
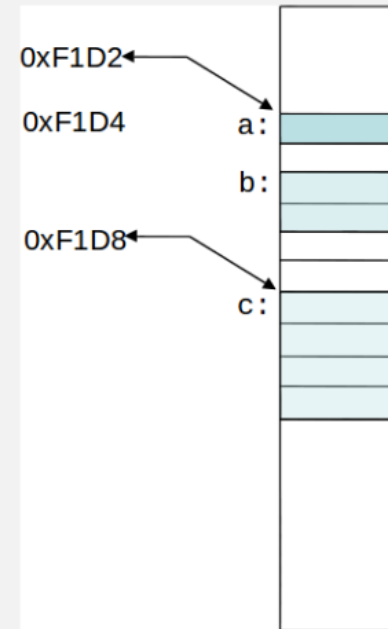
1 byte at address 0xF1D2
2 bytes at address 0xF1D4
4 bytes at address 0xF1D8

Load value of

1 byte at address 0xF1D2
2 bytes at address 0xF1D4
4 bytes at address 0xF1D8

Store new value on

1 byte at address 0xF1D2
2 bytes at address 0xF1D4
4 bytes at address 0xF1D8



The screenshot shows a spreadsheet application interface. At the top, there is a toolbar with various icons for file operations and editing. Below the toolbar, the font settings are set to "Liberation Sans" and size "10". The formula bar displays the formula $=(B2+C2+D2)/3$. Below the formula bar is a table with columns A through F and rows 1 through 9. The table contains the following data:

	A	B	C	D	E	F
1		Grade 1	Grade 2	Grade 3	Final Grade	
2	Ion	7	8	8	$=(B2+C2+D2)/3$	
3	Maria	8	9	7		
4	Vasile	9	8	7		
5						
6						
7						
8						
9						

Variables by address

- Every variable allocated in memory has an address!
- Doesn't have any address:
 - register variables
 - constants/literals/preprocessor defines
 - expressions (unless result is a variable)
- The address of a variable: by using the & operator

```
int n = 4 ;  
double PI = 3.14159 ;  
int *pn = &n ; /*address of integer n* /  
double *ppi = &PI ; /* address of double PI */
```

- Address of a variable of type **t** has type **t ***



de-referentiation

what I can do with a pointer?

- Accessing/modifying addressed variable:
dereferencing/indirection operator *

```
/* prints " p i = 3.14159\n " */  
printf ( "pi = %g\n" , *ppi ) ;  
/* pi now equals 6.14159 */  
*ppi = *ppi + 3 ;
```

- Dereferenced pointer can be considered as any other variable
- null pointer, i.e. 0 (NULL): pointer that does not reference anything

```
if(ppi == NULL)  
/*cannot be accessed - dereferenciated*/
```



Pointer:declare, init, use



declaration (is a variable)

```
[modifier] type *variable_name;
```

initialization

```
int a = 7;
```

```
int *pa;
```

```
pa = &a;
```

```
char *pc = "string pointer";
```

```
float *pf;
```

```
pf = (float *)malloc(4);
```

de-referentiation (usage)

*pa; is the value from the address of a

```
*pa = 5; a will be 5
```

```
strcpy(pc, "new string");
```

Definition:

A **pointer** is *a variable*, who's *value* is interpreted as a *memory address*, where a data of the declared type of the pointer is accessible.

```
char *pc;    /*pointer to char */  
int *pi;    /*pointer to int */  
double *pd; /* pointer to double */  
int a, *p, *q=&a;
```

Pointers: sizeof

How much space is allocated for a pointer?

```
char *pStr = "Timisoara";  
int x = 5, *pX = &x;  
double *pdVal;  
float *pf;  
char c = 'Q', *pc = &c;
```

What returns sizeof():

pStr
pX
pdVal
pf
pc

Answer:

A pointer variable is stored on the number of bytes needed to store an address!

Usually 4 bytes, sometimes 2 bytes or even 1 byte, depending on the machine / type of memory

Answer:

A pointer variable is stored on the number of bytes needed to store an address!

Usually 4 bytes, sometimes 2 bytes or even 1 byte, depending on the machine / type of memory

casting pointers

- Can explicitly cast any pointer type to any other pointer type

```
ppi = (double *)pn;
```

```
/*where pn originally of type ( int *) */
```

- Implicit cast to/from `void *` also possible
- Dereferenced pointer has new type,
- Possible to cause segmentation faults,
- difficult-to-identify errors
 - What happens if we dereference `ppi` now?



Accessing caller's variables

```
...  
int a, b;  
...  
swap(a, b);  
...
```

```
void swap(int x, int y)  
{  
    int temp;  
    temp=x;  
    x=y;  
    y=temp;  
}
```

```
...  
int a, b;  
...  
swap(&a, &b);  
...
```

```
void swap(int *x, int *y)  
{  
    int temp;  
    temp=*x;  
    *x=*y;  
    *y=temp;  
}
```

**Function with
multiple output**



What is wrong with this code?

```
#include <stdio.h>
char *get_str ( ) {
    char msg[] = "A simple message" ;
    return msg ;
}

int main ( void ) {
    char *str = get_str( ) ;
    puts(str) ;

    return 0 ;
}
```

Pointer invalid after variable passes
out of scope !

Why pointers?

In order to:

- a) **do what is not possible** without them
 - *access* to local variables of the caller function
 - *return* more than one value to the caller function
 - *access dynamically allocated storage*

- b) **do things better** with them than without them
 - array processing

Pointers (2)

- **Arrays and Pointers**
 - Legal operations
- **Strings**
 - String utility functions

Incrementation/decrementation

```
p++;  
p--;
```

Comparison of two pointers

```
if(p==q)  
...  
if(p!=q)  
...
```

is equivalent to

```
for(j=0; j<N; j++)  
*(p+j)=...
```

and both are slower than:

```
for(j=0; j<N; j++) p++;  
*p=...
```

Summary of operations:
1 addition, 1 multiplication
and 1 dereferencing / iteration

Summary of operations:
1 addition and 1 de-
referencing/iteration

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
int *p = a;
```

4D...

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

sizeof(a[0])

Declaration
[modifier]
[modifier]

```
E.g.  
TYPE  
TYPE  
ppa=  
ppa+  
ppa+
```

Pointers

What is a pointer?

- **Pointer** =
 - is an object (a variable)
 - whose value is interpreted as a memory address
- **Address** can be used to access/modify a variable from anywhere in memory

Addresses

File.c
char a;
short b;
long c;

...

...a+b+c...

...

a=...
b=...
c=...

File.obj

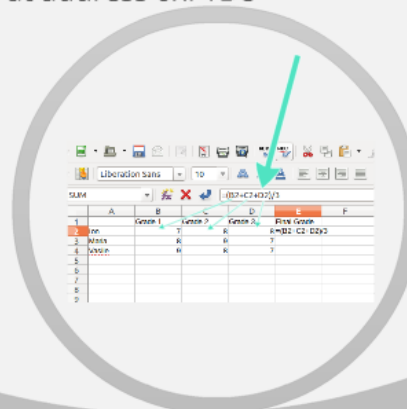
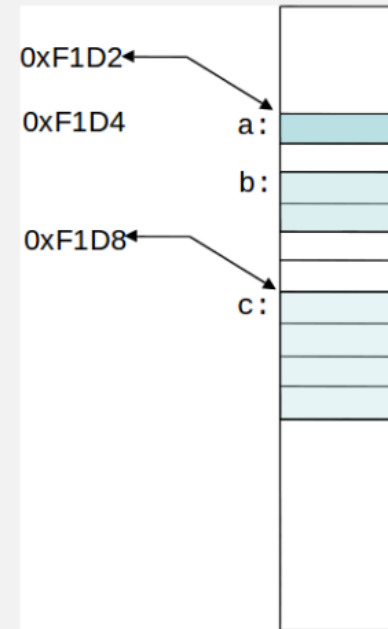
Reserve space for
1 byte at address 0xF1D2
2 bytes at address 0xF1D4
4 bytes at address 0xF1D8

Load value of

1 byte at address 0xF1D2
2 bytes at address 0xF1D4
4 bytes at address 0xF1D8

Store new value on

1 byte at address 0xF1D2
2 bytes at address 0xF1D4
4 bytes at address 0xF1D8



Pointer:declare, init, use



declaration (is a variable)

```
[modifier] type *variable_name;
```

initialization

```
int a = 7;
```

```
int *pa;
```

```
pa = &a;
```

```
char *pc = "string pointer";
```

```
float *pf;
```

```
pf = (float *)malloc(4);
```

de-referentiation (usage)

*pa; is the value from the address of a

```
*pa = 5; a will be 5
```

```
strcpy(pc, "new string");
```

Why pointers?

In order to:

- a) **do what is not possible** without them
 - *access* to local variables of the caller function
 - *return* more than one value to the caller function
 - *access dynamically allocated storage*

- b) **do things better** with them than without them
 - array processing

Pointers (2)

- **Arrays and Pointers**
 - Legal operations
- **Strings**
 - String utility functions

Incrementation/decrementation

```
p++;  
p--;
```

Comparison of two pointers

```
if(p==q)  
...  
if(p!=q)  
...  
if(p<q)  
...
```

is equivalent to

```
for(j=0; j<N; j++)  
*(p+j)=...
```

and both are slower than:

```
for(j=0; j<N; j++)  
*p=...
```

Summary of operations:
1 addition, 1 multiplication
and 1 dereferencing / iteration

Summary of operations:
1 addition and 1 de-
referencing/iteration

```
int a[2][3][4][5][6][7][8];  
4D...);
```

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

```
sizeof(a[0])
```

Declaration
[modifier]
[modifier]

```
E.g.  
TYPE  
TYPE  
ppa=  
ppa+  
ppa+
```

Arrays / Vectors

- single dimensions (1D):

```
int a[10], b[] = {10, 11, 12}, k;  
a[0] = 10;  
a[1] = 11;
```

```
for(k = 0; k < 10; k++)  
    a[k] = 10 + k;
```

10	11	12	13	14	15	16	17	18	19
----	----	----	----	----	----	----	----	----	----

- multidimensions (2D, 3D, 4D,...):

```
short m1[2][3] = {{1, 2, 3}, {4, 5, 6}};  
int A[5][5], i, j;
```

```
for(i = 0; i < 5; i++)  
    for(j = 0; j < 5; j++)  
        A[i][j] = i*5+j;
```

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

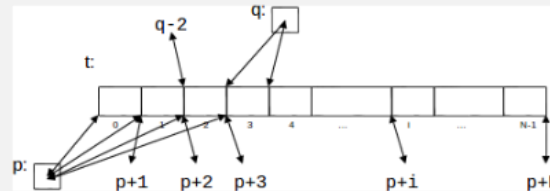
-array size = sizeof(a)/sizeof(a[0])



Array and pointers

- Addition/subtraction of an integer to/from a pointer.

```
TYPE t[N], *p, *q;  
p=&t[0];      /* p=t */  
q=&t[4];
```



- Subtraction of 2 pointers

```
int n = q - p;
```

- Incrementation/ decrementation

```
p++;  
p++;  
p++;  
q--;
```

-Comparison of two pointers

```
...  
if(p==q)  
...  
if(p!=q)  
...  
if(p<q)  
...
```



Legal operations with pointers

1. Initialization and assignment
2. De-referentiation
3. Addition/subtraction of an integer to/from a pointer
4. Incrementation/decrementation
5. Subtraction of a pointer from another pointer
6. Comparison of two pointers

Obs:

- Operations 3 and 4 only make sense (shall only be performed) if the pointer points to an element of an array!
- Operations 5 and 6 only make sense (shall only be performed) if both pointers point to elements of the same array!

Array, Pointers

Remember: the name of an array is a synonym for the *address of its first element* and is constant!

TYPE $t[N]$, $*p=t$;

p is equal to t and both are addresses ($\&t[0]$)

Then

$p+0$ is equal to $t+0$

$p+i$ is equal to $t+i$

$*p$ is equal to $*t$

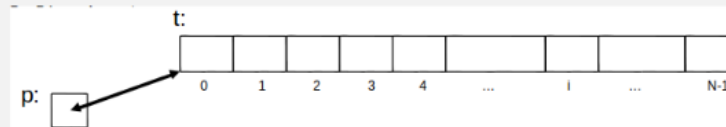
$*(p+i)$ is equal to $*(t+i)$

but

$p++$ is not equal to $t++$! (t is a constant and cannot be modified!)

$t[0]$ is equal to $p[0]$

$t[i]$ is equal to $p[i]$



Pointer's efficiency

Not any use of pointers leads to greater efficiency!

```
TYPE t[N], *p=t;
```

```
for(i=0; i<N; i++)
```

```
t[i]=...;
```

t[i] means *(t+i) which
is equivalent to *(p+i)

*(t+i*sizeof(TYPE))

is equivalent to

```
for(i=0; i<N; i++)
```

```
*(p+i)=...;
```

Summary of operations:

1 addition, 1 multiplication
and 1 de-referentiation / iteration

and both are slower than:

```
for(i=0; i<N; i++, p++)
```

```
*p=...;
```

Summary of operations:

1 addition and 1 de-
referentiation/iteration



dynamic memory allocation

stdlib.h

allocate memory at run time

```
void * malloc(int no_of_bytes);
```

deallocate memory

```
void free(void *);
```

dynamic allocation

```
TYPE1 *p1, *p2;  
TYPE2 *p3, *p4;  
...  
p1=malloc(n*sizeof(TYPE1));  
if( p1 != NULL)  
...  
  
p3=malloc(m*sizeof(TYPE2));  
if(p3 != NULL)  
...  
  
free(p3); /*p3 no more need*/  
p2=malloc(n*sizeof(TYPE1));  
if( p2 != NULL)  
...  
  
free(p1);  
if((p4=malloc(m*sizeof(TYPE2)) == NULL)  
{ /* handle error */ }  
...
```



dynamic allocation

```
TYPE1 *p1, *p2;
TYPE2 *p3, *p4;
...
p1=malloc(n*sizeof(TYPE1));
if( p1 != NULL)
...

p3=malloc(m*sizeof(TYPE2));
if(p3 != NULL)
...

free(p3); /*p3 no more need*/
p2=malloc(n*sizeof(TYPE1));
if (p2 != NULL)

...

free(p1);
if((p4=malloc(m*sizeof(TYPE2))) == NULL)
{ /* handle error */ }
...
```

arrays of pointers; pointers to pointers

Declaration:

```
[modifiers] type_name *array_name[constant_expression];  
[modifiers] type_name **variable_name;
```

E.g.

```
TYPE1 *a[N];  
TYPE2 *b[M];  
ppa=a;  
ppa++;  
ppa++;
```

What is different?

```
char T[n][m], *ap[k], **pp;
```

Strings as arrays

- Strings stored as null-terminated character arrays (last character == '\0')

```
char str[] = "This is a string.";
```

```
char *pc = str ;
```

- Manipulate string as you would an array

```
*(pc+10) = 'S'; /* same as str[10] = "S; */
```

```
puts( str );    /* prints "This is a String ." */
```

String functions

string.h

Copy functions: `strcpy()`, `strncpy()`

```
char * strncpy( strto , strfrom ,n);  
copy n chars from strfrom to strto  
char * strcpy( strto , strfrom );  
copy strfrom to strto
```

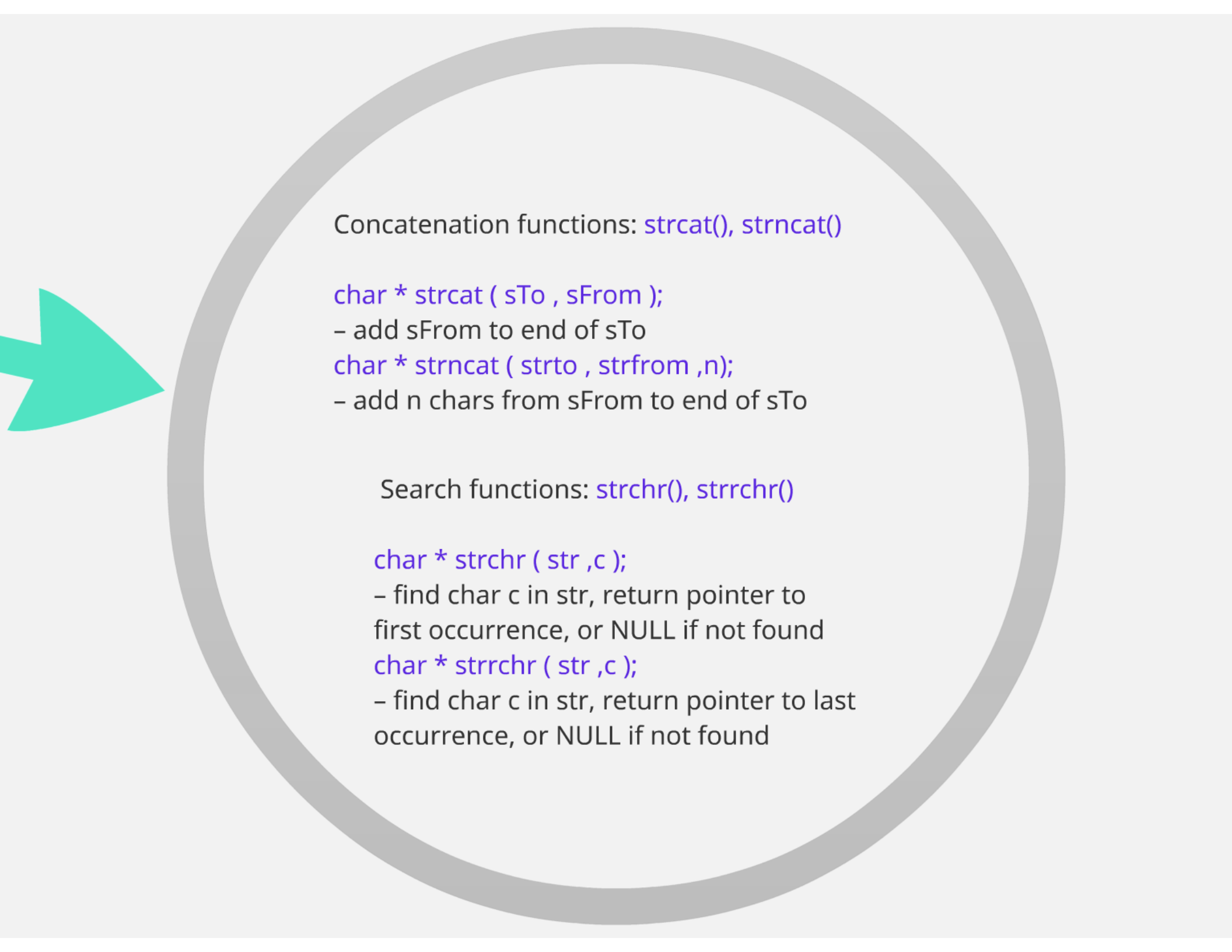
Comparison functions: `strcmp()`, `strncmp()`

```
int strcmp(str1, str2 );  
compare str1, str2; return 0 if equal, positive if str1>str2,  
negative if str1<str2  
int strncmp(str1,str2 ,n);  
compare first n chars of str1 and str2
```

String length: `strlen()`

```
int strlen ( str );  
get length of str
```





Concatenation functions: `strcat()`, `strncat()`

`char * strcat (sTo , sFrom);`

– add sFrom to end of sTo

`char * strncat (strto , strfrom ,n);`

– add n chars from sFrom to end of sTo

Search functions: `strchr()`, `strrchr()`

`char * strchr (str ,c);`

– find char c in str, return pointer to first occurrence, or NULL if not found

`char * strrchr (str ,c);`

– find char c in str, return pointer to last occurrence, or NULL if not found