CSE 1320

Week of 04/01/2019

Instructor : Donna French

Pointers to Functions

When a C program is compiled and loaded into memory, the executable code is organized in such a way that an address is associated with every identifier with either external linkage or `static` storage class.

This is true of identifiers that represent variables as well as identifiers that name functions.

When the name of a function is mentioned in C source code, it is interpreted as the address of the code for that function in memory.

just like an array name means the address of the first cell of the array

Pointers to Functions

In C, it is possible to declare a variable that is a pointer to a function.

Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

Unlike normal pointers, we do not allocate/de-allocate memory when using function pointers.

Like normal pointers, we can have an array of function pointers.

Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

Pointers to Functions

```
void MyFunction(int a)
{
    printf("Value of a is %d\n", a);
}


int main(void)
{
    // function_ptr is a pointer to function MyFunction()
    void (*function_ptr)(int) = MyFunction;

    // Invoking MyFunction() using function_ptr
    function_ptr(10);

    return 0;
}
```

functionpointer1Demo.c

```
Breakpoint 1, main () at functionpointer1Demo.c:32
32          void (*function_ptr)(int) = MyFunction;

(gdb) p MyFunction
$1 = {void (int)} 0x400498 <MyFunction>

(gdb) p function_ptr
$2 = (void (*)(int)) 0x400498 <MyFunction>

(gdb) step
35              function_ptr(10);

(gdb) step
MyFunction (a=10) at functionpointer1Demo.c:7
7               printf("Value of a is %d\n", a);

(gdb) p a
$1 = 10

(gdb)
Value of a is 10
```

functionpointer1Demo.c

Pointers to Functions

`MyFunction()` has an `int` parameter and a `void` return type.

```
void MyFunction(int a)
{
    printf("Value of a is %d\n", a);
}
```

`function_ptr` is a function pointer for a function that has an `int` parameter and a `void` return type.

```
void (*function_ptr)(int);
```

The function pointer is assigned the address of `MyFunction()`.

```
function_ptr = MyFunction;
```

functionpointer1Demo.c

Pointers to Functions

```
char *ptr = MyArray;
```

```
char *ptr;
ptr = MyArray;
```

Declaration and assignment can be done separately

```
void (*function_ptr)(int) = MyFunction;
```

```
void (*function_ptr)(int);
function_ptr = MyFunction;
```

Declaration and assignment can be done separately

Pointers to Functions

This statement

```
(*function_ptr)(10);
```

is equivalent to

```
function_ptr(10);
```

If we just remove the ()

```
*function_ptr(20);
```

then we get

```
error: void value not ignored as it ought to be
```

Pointers to Functions

`OurFunction()` has an `int` parameter and a `void` return type.

```
void OurFunction(int a)
{
    printf("Value of a is %d\n", a);
}
```

`our_function_ptr` is a function pointer for a function that has an `int` parameter and a `void` return type.

```
void (*our_function_ptr)(int);
```

The function pointer is assigned the address of `OurFunction()`.

```
our_function_ptr = OurFunction;
```

Call the function using the function pointer

```
our_function_ptr(20);
```

Given this function

```
int Multiplier(int y, int z)
{
    return y*z;
}
```

What would the function pointer look like?

```
int (*MultiPtr)(int, int) = Multiplier;
```

How would it be used?

```
printf("Output from Multiplier %d\n", MultiPtr(2,4));
```

functionpointer1Demo.c

Pointers to Functions

`Multiplier()` has an `int` parameter and an `int` return type.

```
int Multiplier(int z)
{
    return z*z;
}
```

`MultiPtr` is a function pointer for a function that has an `int` parameter and an `int` return type and the function pointer is assigned the address of `Multiplier()`.

```
int (*MultiPtr)(int) = Multiplier;

printf("Output from Multiplier %d\n", MultiPtr(2));
```

functionpointer1Demo.c

Pointers to Functions

`Multiplier()` has two `int` parameters and an `int` return type.

```
int Multiplier(int y, int z)
{
    return y*z;
}
```

`MultiPtr` is a function pointer for a function that has two `int` parameters and an `int` return type and the function pointer is assigned the address of `Multiplier()`.

```
int (*MultiPtr)(int, int) = Multiplier;

printf("Output from Multiplier %d\n", MultiPtr(2,4));
```

functionpointer1Demo.c

```
void Swap(int *a, int *b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```

```
(gdb) p Swap
$1 = {void (int *, int *)} 0x4004e9 <Swap>
(gdb) step

(gdb) p *SwapPtr
$2 = {void (int *, int *)} 0x4004e9 <Swap>
```

```
int SwapA = 20;
int SwapB = 21;

void (*SwapPtr)(int *, int *) = Swap;

printf("SwapA=%d\tSwapB=%d\n", SwapA, SwapB);
SwapPtr(&SwapA, &SwapB);
printf("SwapA=%d\tSwapB=%d\n", SwapA, SwapB);
```

functionpointer1Demo.c

Pointers to Functions

```
int FunAdd(int a, int b)
{
    return a+b;
}

int FunSub(int a, int b)
{
    return a-b;
}

int FunMult(int a, int b)
{
    return a*b;
}

int FunDiv(int a, int b)
{
    return a/b;
}

int (*PtrArray[4]) (int, int) = {FunAdd, FunSub, FunMult, FunDiv};
```

functionpointer2Demo.c

```
int main(void)
{
        int MenuChoice;
        int a, b;

        int (*PtrArray[]) (int, int) = {FunAdd, FunSub, FunMult, FunDiv};

        system("clear");

        printf("Enter first number ");
        scanf("%d", &a);
        printf("\nEnter second number ");
        scanf("%d", &b);

        printf("\n\n0. Add\n1. Subtract\n2. Multiple\n3. Divide\n\nChoice? ");
        scanf("%d", &MenuChoice);

        printf("\n\n%d\n\n", (*PtrArray[MenuChoice])(a,b));

        return 0;
}
```

```
int (*PtrArray[]) (int, int) = {FunAdd, FunSub, FunMult, FunDiv};
```

```
Breakpoint 1, main () at functionpointer2Demo.c:29
29        int (*PtrArray[]) (int, int) = {FunAdd, FunSub, FunMult, FunDiv};
```

```
(gdb) p FunAdd
$25 = {int (int, int)} 0x400528 <FunAdd>
(gdb) p FunSub
$26 = {int (int, int)} 0x40053a <FunSub>
(gdb) p FunMult
$27 = {int (int, int)} 0x40054e <FunMult>
(gdb) p FunDiv
$28 = {int (int, int)} 0x400561 <FunDiv>
```

```
(gdb) p PtrArray
$29 = {0x400528 <FunAdd>, 0x40053a <FunSub>, 0x40054e <FunMult>,
  0x400561 <FunDiv>}
```

functionpointer2Demo.c

```
(gdb) p PtrArray
$29 = {0x400528 <FunAdd>, 0x40053a <FunSub>, 0x40054e <FunMult>,
  0x400561 <FunDiv>}

41          printf("\n\n%d\n\n", (*PtrArray[MenuChoice])(a,b));

(gdb) step
FunMult (a=3, b=4) at functionpointer2Demo.c:16

16          return a*b;

(gdb) p *PtrArray[MenuChoice]
$30 = {int (int, int)} 0x40054e <FunMult>
```

```
Enter first number 2

Enter second number 3

0. Add
1. Subtract
2. Multiple
3. Divide

Choice? 2


6
```

functionpointer2Demo.c

qsort()

- built into C
- designed to sort arrays

```
void qsort(void *base,
           size_t elements,
           size_t size,
           int (*compare)(const void *, const void*));
```

qsort()

```
void qsort(void *base,
           size_t elements,
           size_t size,
           int (*compare)(const void *, const void*));
```

base Pointer to the first element of the array to be sorted.

Set to a `void` pointer so that `qsort()` can handle any type of array.

qsort()

```
void qsort(void *base,
           size_t elements,
           size_t size,
           int (*compare)(const void *, const void*));
```

elements Number of elements in the array.

Using `size_t` as the type allows `qsort()` to accept any unsigned integer type and handle any size array.

qsort()

```
void qsort(void *base,
           size_t elements,
           size_t size,
           int (*compare)(const void *, const void*));
```

size Size in bytes of each element in the array.

Using `size_t` as the type allows `qsort()` to accept any unsigned integer type and handle any size array.

qsort()

```
void qsort(void *base,
           size_t elements,
           size_t size,
           int (*compare)(const void *, const void*));
```

compare Function pointer that compares two elements.

Pointer to a function that compares two array elements.

The function parameters are set to `const` to show that they will not be altered by the function.

The parameters are of type `void` to allow arrays of all types to be sorted.

The compare function returns an `int`.

qsort()

```
void qsort(void *base,
           size_t elements,
           size_t size,
           int (*compare)(const void *, const void*));
```

Return value of compare function

< 0	if the 1st parameter points to the smaller item
0	if the parameters point to identical items
> 0	if the 2nd parameter points to the smaller item

```
Before qsort() : 88 56 100 2  25
After  qsort() : 2  25 56  88 100
```

< 0 if 1st parameter is smaller
0 if parameters are identical
> 0 if 2nd parameter is smaller

qsort()

```
int fnintcmp(const void *a, const void *b)
{
    return (*(int*)a - *(int*)b);
}

int intarray[5] = {88, 56, 100, 2, 25};

qsort(intarray, 5, sizeof(int), fnintcmp);
```

a is a void pointer.
(int*)a is casting a to an int pointer.
(int)a is dereferencing a to get to the value

base

elements

size

compare

qsort1Demo.c

< 0 if 1st parameter is smaller
0 if parameters are identical
> 0 if 2nd parameter is smaller

```
Before qsort() : Z B C E A
After  qsort() : A B C E Z
```

qsort()

```
int fnlistcmp(const void *a, const void *b)
{
   return (strcmp((char *)a, (char *)b));
}

char list[5] = {'Z','B','C','E','A'};

qsort(list, 5, sizeof(char), fnlistcmp);
```

base

elements

size

compare

a is a void pointer.
(char*)a is casting a to a pointer to char.
strcmp takes a pointer to char

qsort1Demo.c

Calling `qsort` using an array of function pointers.

```
#include <stdio.h>
#include <string.h>

#define MAX_MOVIES 4

typedef struct
{
   char name[25];
   char release_date[9];
   char rating[4];
} Movie;

Movie MovieLibrary[MAX_MOVIES] = {{"A New Hope","19770525",'S'},
                                  {"The Empire Strikes Back","19800521",'A'},
                                  {"Return of the Jedi","19830825",'T'},
                                  {"The Phantom Menace","19990519",'R'}};
```

```
void PrintMovieLibrary(Movie MovieLibrary[])
{
  int i;

  for (i = 0; i < MAX_MOVIES; i++)
  {
    printf("%-25s is rated %c and is being released on %s\n",
        MovieLibrary[i].name,
        MovieLibrary[i].rating,
        MovieLibrary[i].release_date);
  }
  printf("\n\n");
}
```

MAX_MOVIES is 4

`%-25s` says to left justify the field and right pad with blanks.

```
A New Hope                  is rated S and is being released on 19770525
The Empire Strikes Back     is rated A and is being released on 19800521
Return of the Jedi          is rated T and is being released on 19830825
The Phantom Menace          is rated R and is being released on 19990519
```

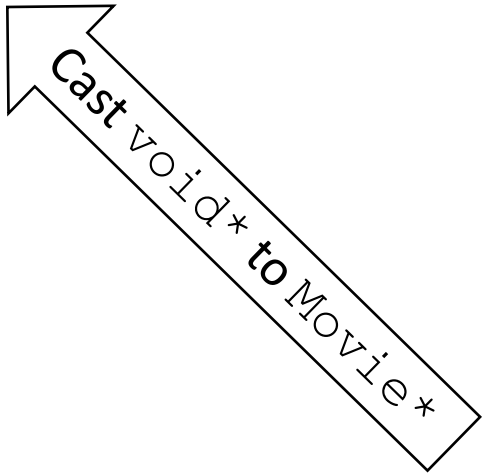
```
int main(void)
{
   int Choice;

   Movie MovieLibrary[MAX_MOVIES] = {{"A New Hope","19770525",'S'},
                                     {"The Empire Strikes Back","19800521",'A'},
                                     {"Return of the Jedi","19830825",'T'},
                                     {"The Phantom Menace","19990519",'R'}};

   PrintMovieLibrary(MovieLibrary);
```

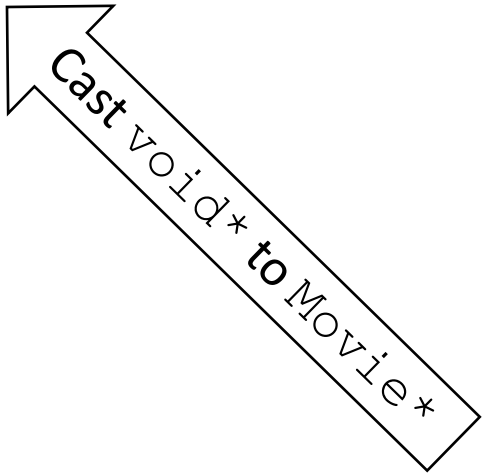
used later in the menu

Did not need struct

```
do
{
   printf("\nHow you want to sort your movie library?\n\n"
        "0. Exit\n"
      "1. By Name?\n"
      "2. By Release Date?\n"
      "Choice : ");

   scanf("%d", &Choice);

   if (Choice)
   {
      call qsort()

      PrintMovieLibrary(MovieLibrary);
   }
}
while (Choice);
```

```
int MovieNameCompare(const void *a, const void *b)
{
   return (strcmp(((Movie*)a)->name, ((Movie*)b)->name));
}
```





`strcmp()` will return < 0 if the `a->name` is alphabetically less than the `b->name`

```
int MovieReleaseDateCompare(const void *a, const void *b)
{
   return (strcmp(((Movie*)a)->release_date, ((Movie*)b)->release_date));
}
```

```
int (*CompareFunctionPtrArray[])(const void *, const void *) =
{MovieNameCompare, MovieReleaseDateCompare};
```

Compare function

Compare function

```
0. Exit
1. By Name?
2. By Release Date?
```

size of a single array element – the structure

elements in the array

array

```
qsort(MovieLibrary, MAX_MOVIES, sizeof(Movie),
     (*CompareFunctionPtrArray[Choice-1]));
```

Choice is 1 or 2 so -1

What if we wanted to add a function to sort by rating?

```
printf("\nHow you want to sort your movie library?\n\n"
      "0. Exit\n"
      "1. By Name?\n"
      "2. By Release Date?\n\n"
      "Choice : ");
```

```
printf("\nHow you want to sort your movie library?\n\n"
      "0. Exit\n"
      "1. By Name?\n"
      "2. By Release Date?\n"
      "3. By Rating?\n\n"
      "Choice : ");
```

What if we wanted to add a function to sort by rating?

```
int MovieNameCompare(const void *a, const void *b)
{
      return (strcmp(((Movie*)a)->name, ((Movie*)b)->name));
}

int MovieReleaseDateCompare(const void *a, const void *b)
{
      return (strcmp(((Movie*)a)->release_date, ((Movie*)b)->release_date));
}

int MovieRatingCompare(const void *a, const void *b)
{
      return (((Movie*)a)->rating > (((Movie*)b)->rating));
}
```

What if we wanted to add a function to sort by rating?

```
int (*CompareFunctionPtrArray[])(const void *, const void *) =
{MovieNameCompare, MovieReleaseDateCompare};


int (*CompareFunctionPtrArray[])(const void *, const void *) =
{MovieNameCompare, MovieReleaseDateCompare, MovieRatingCompare};


qsort(MovieLibrary, MAX_MOVIES, sizeof(Movie),
     (*CompareFunctionPtrArray[Choice-1]));
```

```
How you want to sort your movie library?

0. Exit
1. By Name?
2. By Release Date?
3. By Rating?

Choice : 3

The Empire Strikes Back   is rated A and is being released on 19800521
The Phantom Menace        is rated R and is being released on 19990519
A New Hope                is rated S and is being released on 19770525
Return of the Jedi        is rated T and is being released on 19830825
```

Stack

Stack is a linear data structure which follows a particular order in which the operations are performed. The order will be LIFO (Last In First Out).

Stack

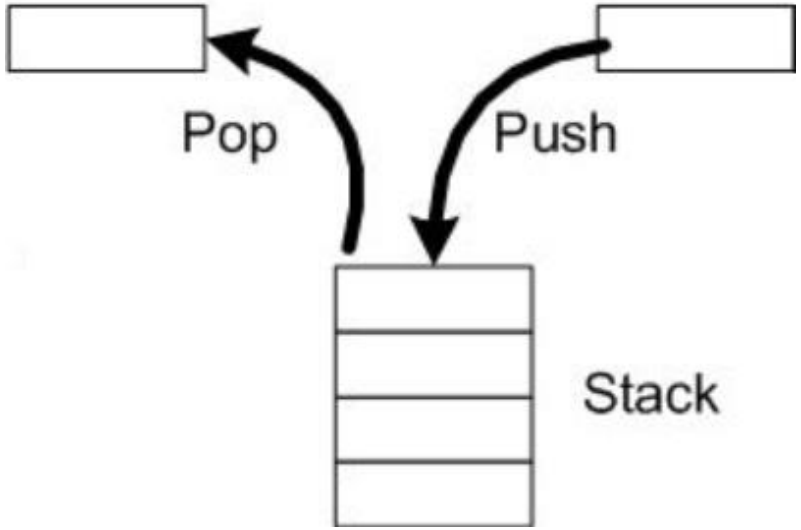
Operations on a Stack

Push Adds an item to the stack.

Pop Removes an item from the stack.

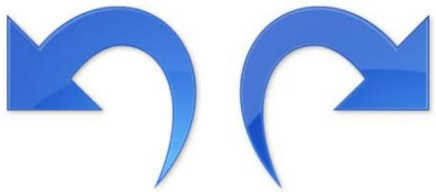
Peek Returns top element of stack.

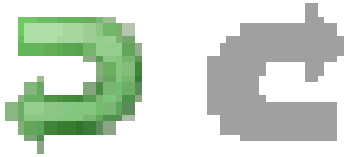
IsEmpty Returns TRUE if stack is empty, else FALSE.

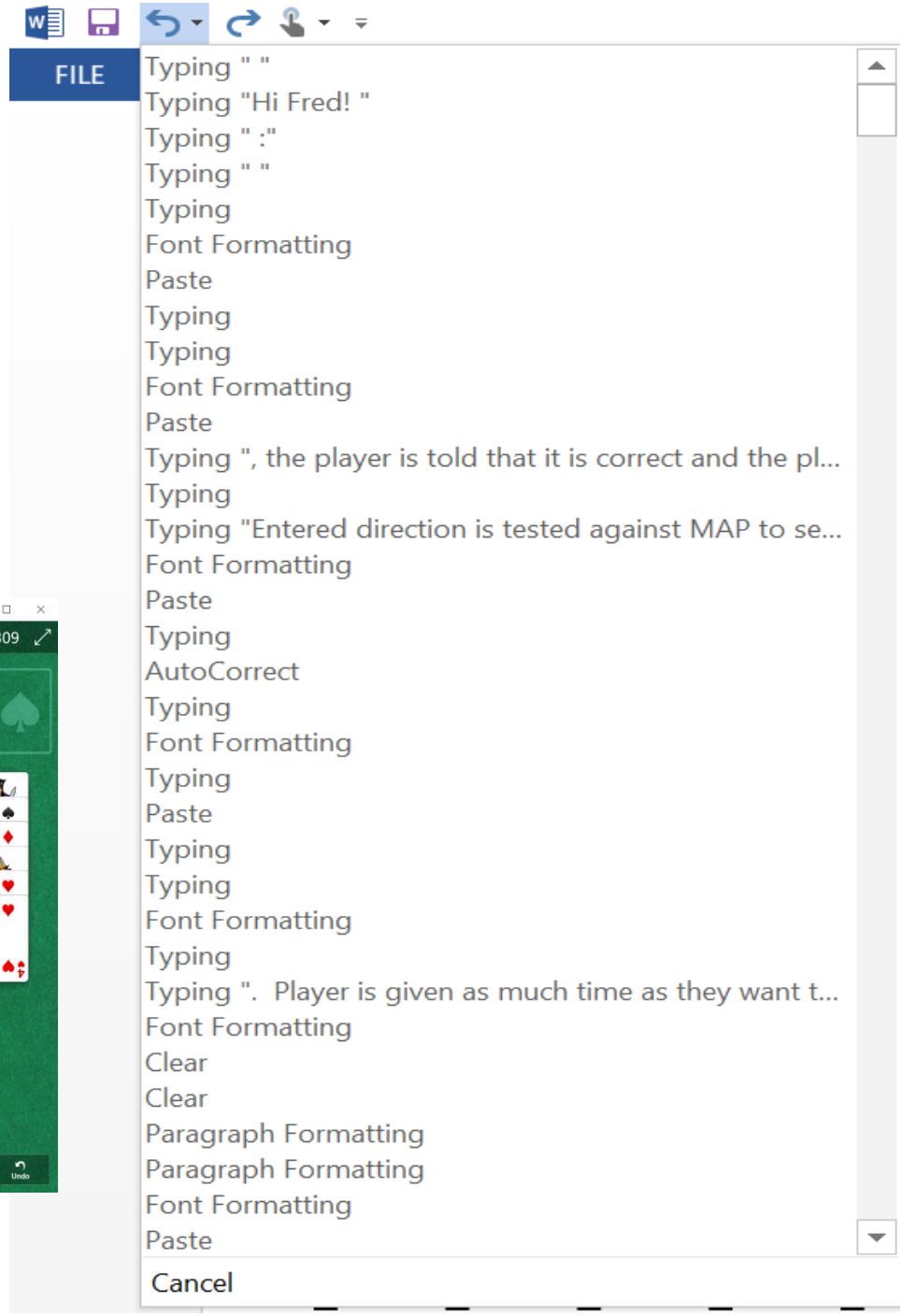


Stack

Word processors, text editors, some games use undo and redo capabilities.







```
node *StackTop = NULL;
```

```
// A structure to represent a stack node
typedef struct node
{
    int node_number;
    struct node *next_ptr;
}node;

void DisplayStack(node *StackTop)
{
      node *TempPtr = StackTop;

      if (StackTop == NULL)
            printf("\nStack is empty");

      while (TempPtr != NULL)
      {
            printf("Stack node %d\n", TempPtr->node_number);
            TempPtr =  TempPtr->next_ptr;
      }

      return;
}
```

stackDemo.c

Queue

Queue is a linear data structure which follows a particular order in which the operations are performed. The order will be FIFO (First In First Out).

When you stand in line waiting for something, the person at the head of the line goes first and anyone new is added to the back of the line.

Queue

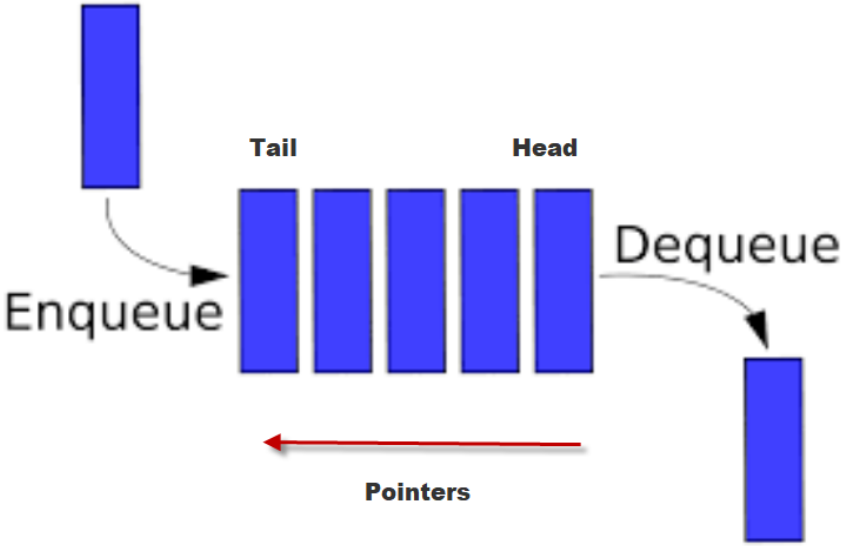
Operations on a Queue

Enqueue Adds an item to the queue.

Dequeue Removes an item from the queue.

Head Get the head of the queue

Tail Get the tail of the queue



```
node *QueueHead = NULL;
```

```
// A structure to represent a queue node
typedef struct node
{
    int node_number;
    struct node *next_ptr;
}node;


void DisplayQueue(node *QueueHead)
{
      node *TempPtr = QueueHead;

      while (TempPtr != NULL)
      {
            printf("Queue node %d\n", TempPtr->node_number);
            TempPtr =  TempPtr->next_ptr;
      }

      return;
}
```

queueDemo.c

Week of 04012019

Build Content | Assessments | Tools | Partner Content

Slides - Week of 04012019

Linked List Concept in C/C++

Watch Video

Linked List Concept in C/C++

Duration: 6:10
User: n/a - **Added:** 5/10/17

Introduction to Stack Data Structure

Watch Video

Introduction to Stack Data Structure

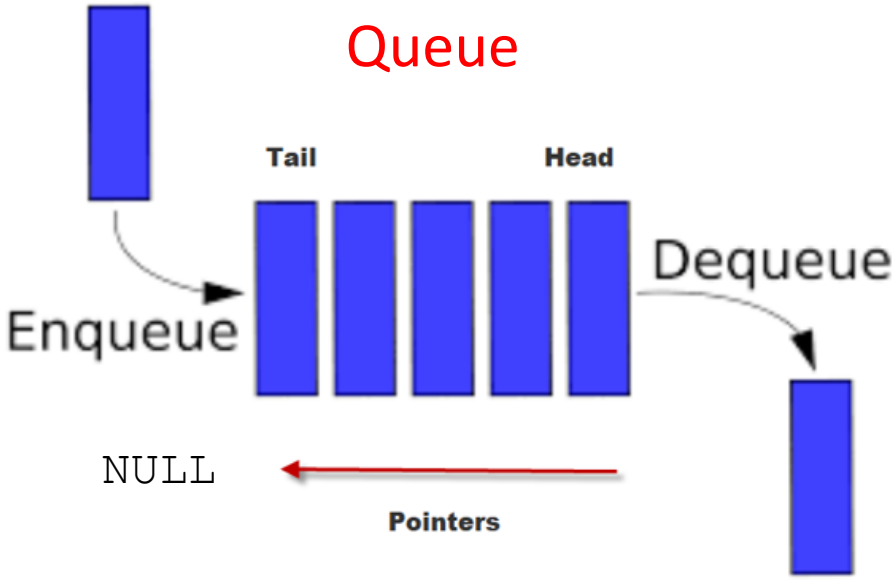
Duration: 5:56
User: n/a - **Added:** 5/13/17

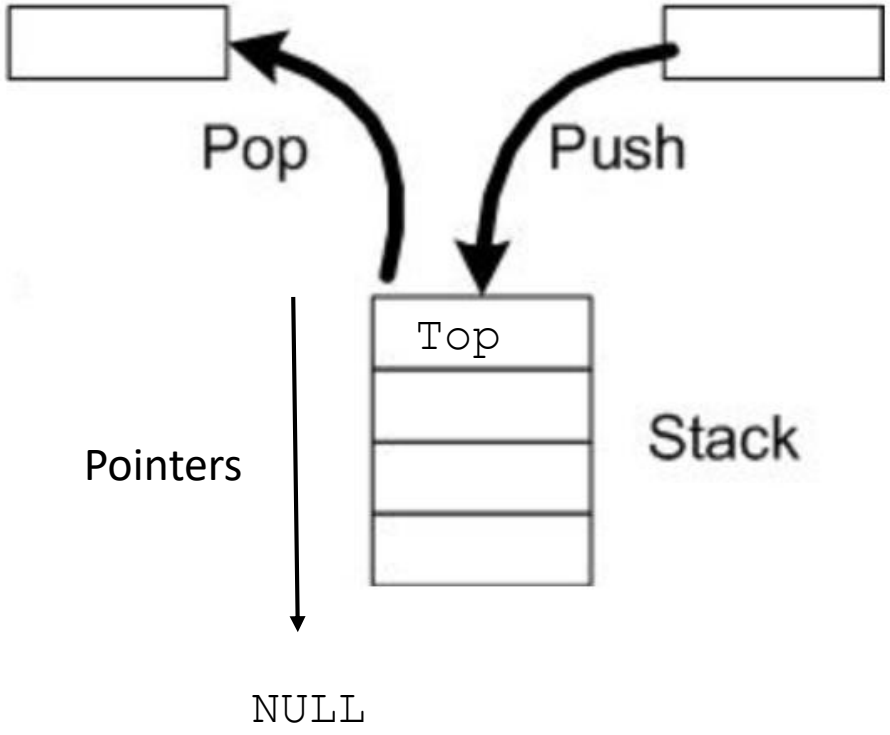
Introduction to Queue DS (Explained With Animation)

Watch Video

Introduction to Queue DS (Explained With Animation)

Duration: 3:39
User: n/a - **Added:** 5/17/17






```
typedef struct node
{
    int node_number;
    struct node *next_ptr;
}node;
```

ICQ10