# Advanced Functions

# Command Line Arguments

# Advanced C

## Functions - Command Line Arguments

**Example**

```c
#include <stdio.h>

int main(int argc, char *argv[], char *envp[])
{

    return 0;
}
```

Environmental Variables

Passed Arguments on CL

Arguments Count

**Usage**

```
user@user:~] ./a.out 5 + 3
```

4th argument

3rd argument

2nd argument

1st argument

Total counts of the args stored in **argc**

All stored in **argv**

EMERTXE

# Advanced C

## Functions - Command Line Arguments

**Example**

```c
#include <stdio.h>

int main(int argc, char **argv)
{

    printf("No of argument(s): %d\n", argc);

    printf("List of argument(s):\n");
    for (i = 0; i < argc; i++)
    {
        printf("\t%d - \"%s\"\n", i + 1, argv[i]);
    }

    return 0;
}
```

# Advanced C

## Functions – Command Line Arguments - DIY

- WAP to calculate average of numbers passed via command line

# Function Pointer

# Advanced C

## Functions – Function Pointers

- A variable that stores the address of a function. Therefore, points to the function.

**Syntax**

```
return_datatype (*foo)(list of argument(s) datatype);
```

# Advanced C

## Functions - Function Pointers

**Example**

```c
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    printf("%p\n", add);
    printf("%p\n", &add);

    return 0;
}
```

- Every function code would be stored in the text segment with an address associated with it
- This example would print the address of the **add** function

EMERTXE

# Advanced C

## Functions - Function Pointers

**Example**

```c
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    int *function;

    function = add;

    printf("%p\n", add);
    printf("%p\n", function);
    printf("%p\n", &function);

    return 0;
}
```

- Hold on!!. Can't I store the address on the normal pointer??
- Well, Yes you can! But how would you expect the compiler to interpret this?
- The compiler interprets this as a pointer to normal variable and not the code
- Then how to do it?

# Advanced C

## Functions - Function Pointers

**Example**

```c
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    int (*function)(int, int);

    function = add;

    printf("%p\n", add);
    printf("%p\n", function);
    printf("%p\n", &function);

    return 0;
}
```

- The address of the function should be stored in a function pointer
- Not to forget that the function pointer is a variable and would have address for itself

# Advanced C

## Functions - Function Pointers

**Example**

```c
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    int (*function)(int, int);

    function = add;

    printf("%d\n", function(2, 4));
    printf("%d\n", (*function)(2, 4));

    return 0;
}
```

- The function pointer could be invoked as shown in the example

# Advanced C

## Functions - Func Ptr - Passing to functions

### Example

```c
#include <stdio.h>

int main()
{
    int (*func)(int, int);

    func = add;
    printf("%d\n", oper(func, 2, 4));

    func = sub;
    printf("%d\n", oper(func, 2, 4));

    return 0;
}
```

```c
int add(int num1, int num2)
{
    return num1 + num2;
}

int sub(int num1, int num2)
{
    return num1 - num2;
}

int oper(int (*f)(int, int), int a, int b)
{
    return f(a, b);
}
```

EMERTXE

# Advanced C

## Functions - Array of Function Pointers

**Example**

```c
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int sub(int num1, int num2)
{
    return num1 - num2;
}

int main()
{
    int (*f[])(int, int) = {add, sub};

    printf("%d\n", f[0](2, 4));
    printf("%d\n", f[1](2, 4));

    return 0;
}
```

# Advanced C

## Functions - Array of Function Pointers

### Example

```c
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int sub(int num1, int num2)
{
    return num1 - num2;
}

int oper(int (*f)(int, int), int a, int b)
{
    return f(a, b);
}

int main()
{
    int (*f[])(int, int) = {add, sub};

    printf("%d\n", oper(f[0], 2, 4));
    printf("%d\n", oper(f[1], 2, 4));

    return 0;
}
```

EMERTXE

# Advanced C

## Functions - Func Ptr - Std Functions - atexit()

### Example

```c
#include <stdio.h>
#include <stdlib.h>

static int *ptr;

int main()
{
    /*
     * Registering a callback
     * Function
     */
    atexit(my_exit);

    /* Allocation in main */
    ptr = malloc(100);

    test();

    printf("Hello\n");

    return 0;
}
```

```c
void my_exit(void)
{
    printf("Exiting program\n");

    if (ptr)
    {
        /* Deallocation in my_exit */
        free(ptr);
    }
}

void test(void)
{
    puts("In test");

    exit(0);
}
```

# Advanced C

## Functions - Func Ptr - Std Functions - qsort()

### Example

```c
#include <stdio.h>
#include <stdlib.h>

int sa(const void *a, const void *b)
{
    return *(int *) a > *(int *) b;
}

int sd(const void *a, const void *b)
{
    return *(int *) a < *(int *) b;
}

int main()
{
    int a[5] = {9, 2, 6, 1, 7};

    qsort(a, 5, sizeof(int), sa);
    printf("Ascending: "); print(a, 5);

    qsort(a, 5, sizeof(int), sd);
    printf("Descending: "); print(a, 5);

    return 0;
}
```

```c
void print(int *a, unsigned int size)
{
    int i = 0;

    for (i = 0; i < size; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");
}
```

EMERTXE

# Variadic Functions

# Advanced C

## Functions - Variadic

- Variadic functions can be called with any number of trailing arguments
- For example,

  printf(), scanf() are common variadic functions
- Variadic functions can be called in the usual way with individual arguments

**Syntax**

```
return_data_type function_name(parameter list, ...);
```

# Advanced C

## Functions – Variadic – Definition & Usage

- Defining and using a variadic function involves three steps:

  Step 1: Variadic functions are defined using an ellipsis (‘…’) in the argument list, and using special macros to access the variable arguments.

Example

```
int foo(int a, ...)
{
    /* Function Body */
}
```

Step 2: Declare the function as variadic, using a prototype with an ellipsis (‘…’), in all the files which call it.

Step 3: Call the function by writing the fixed arguments followed by the additional variable arguments.

# Advanced C

## Functions – Variadic – Argument access macros

- Descriptions of the macros used to retrieve variable arguments
- These macros are defined in the header file stdarg.h

| Type/Macros | Description |
|---|---|
| va_list | The type va_list is used for argument pointer variables |
| va_start | This macro initializes the argument pointer variable ap to point to the first of the optional arguments of the current function; last-required must be the last required argument to the function |
| va_arg | The va_arg macro returns the value of the next optional argument, and modifies the value of ap to point to the subsequent argument. Thus, successive uses of va_arg return successive optional arguments |
| va_end | This ends the use of ap |

# Advanced C

## Functions - Variadic - Example

### Example

```c
#include <stdio.h>
#include <stdarg.h>

int main()
{
    int ret;

    ret = add(3, 2, 4, 4);
    printf("Sum is %d\n", ret);

    ret = add(5, 3, 3, 4, 5, 10);
    printf("Sum is %d\n", ret);

    return 0;
}
```

```c
int add(int count, ...)
{
    va_list ap;
    int iter, sum;

    /* Initilize the arg list */
    va_start(ap, count);

    sum = 0;
    for (iter = 0; iter < count; iter++)
    {
        /* Extract args */
        sum += va_arg(ap, int);
    }

    /* Cleanup */
    va_end(ap);

    return sum;
}
```

EMERTXE