C

Come let's see how deep it is!!

Team Emertxe

Functions

Embedded C

Functions – What?

An activity that is natural to or the purpose of a person or thing.

"bridges perform the function of providing access across water"

A relation or expression involving one or more variables.

"the function (bx + c)"

Source: Google

- In programing languages it can something which performs a specific service
- Generally it can have 3 parameters like
  - Input
  - Operation
  - Output

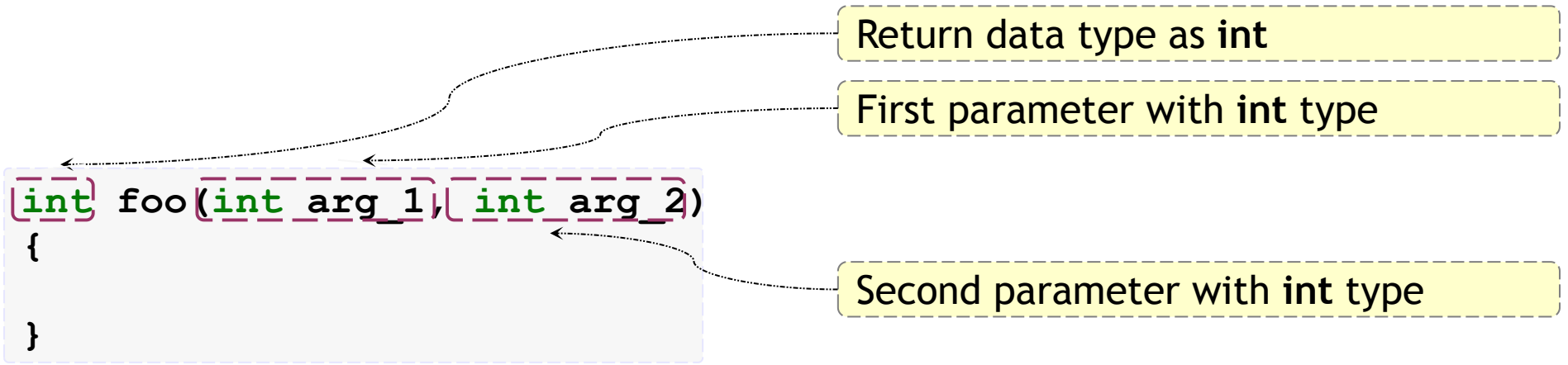
Embedded C

Functions - How to write

Syntax

```
return_data_type function_name(arg_1, arg_2, ..., arg_n)
{
    /* Function Body */
}
```

Example



Embedded C

Functions - How to write

$$y = x + 1$$

Example

```
int foo(int x)
{
    int ret = 0;

    ret = x + 1;

    return ret;
}
```

Return from function

Embedded C

Functions - How to call

Example

```c
#include <stdio.h>

int main()
{
    int x, y;

    x = 2;

    y = foo(x);
    printf("y is %d\n", y);

    return 0;
}
```

The function call

```c
int foo(int x)
{
    int ret = 0;

    ret = x + 1;

    return ret;
}
```

Embedded C

Functions – Why?

- Re usability
  - Functions can be stored in library & re-used
  - When some specific code is to be used more than once, at different places, functions avoids repetition of the code.
- Divide & Conquer
  - A big & difficult problem can be divided into smaller sub-problems and solved using divide & conquer technique
- Modularity can be achieved.
- Code can be easily understandable & modifiable.
- Functions are easy to debug & test.
- One can suppress, how the task is done inside the function, which is called Abstraction

Embedded C

Functions - A complete look

Example

```c
#include <stdio.h>

int main()
{
    int num1 = 10, num2 = 20;
    int sum = 0;

    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}

int add_numbers(int num1, int num2)
{
    int sum = 0;

    sum = num1 + num2;

    return sum;
}
```

The main function

The function call

Actual arguments

Return type

Formal arguments

Function

Return from function

Embedded C

Functions - Ignoring return value

Example

```c
#include <stdio.h>

int main()
{
    int num1 = 10, num2 = 20;
    int sum = 0;

    add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
```

```c
int add_numbers(int num1, int num2)
{
    int sum = 0;

    sum = num1 + num2;

    return sum;
}
```

Ignored the return from function
In C, it is up to the programmer to capture or ignore the return value

Embedded C

Functions – Parameter Passing Types

Pass by Value	Pass by reference
• This method copies the actual value of an argument into the formal parameter of the function. • In this case, changes made to the parameter inside the function have no effect on the actual argument.	• This method copies the address of an argument into the formal parameter. • Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Embedded C

Functions - Pass by Value

Example

```c
#include <stdio.h>

int add_numbers(int num1, int num2);

int main()
{
    int num1 = 10, num2 = 20, sum;

    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
```

```c
int add_numbers(int num1, int num2)
{
    int sum = 0;

    sum = num1 + num2;

    return sum;
}
```

Embedded C

Functions - Pass by Value

Example

```c
#include <stdio.h>

void modify(int num1);

int main()
{
    int num1 = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num1);

    modify(num1);

    printf("After Modification\n");
    printf("num1 is %d\n", num1);

    return 0;
}
```

```c
void modify(int num1)
{
    num1 = num1 + 1;
}
```

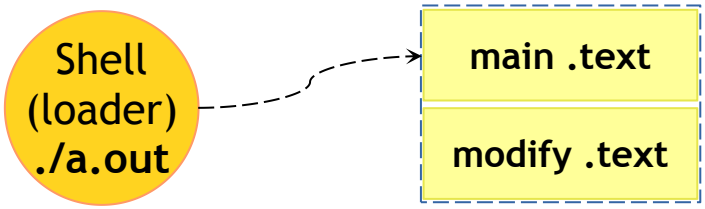
Embedded C

Functions - Pass by Reference

Example

```c
#include <stdio.h>

void modify(int *iptr);

int main()
{
    int num = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num);

    modify(&num);

    printf("After Modification\n");
    printf("num1 is %d\n", num);

    return 0;
}
```

```c
void modify(int *iptr)
{
    *iptr = *iptr + 1;
}
```



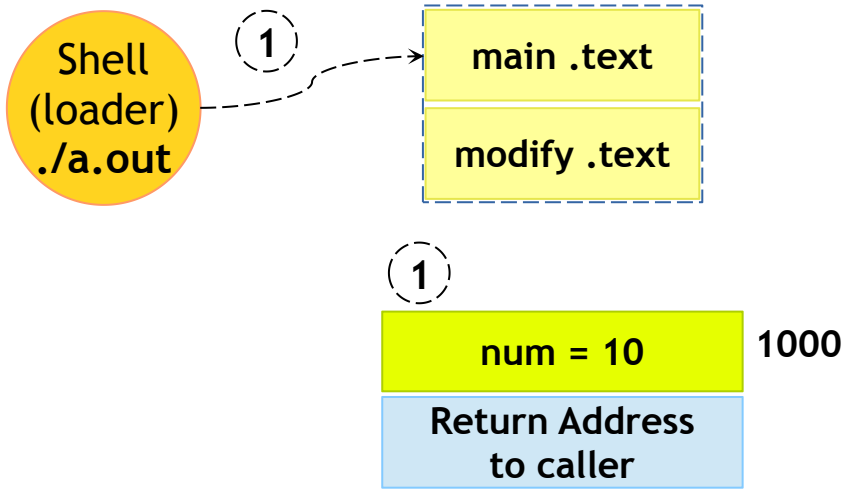
Embedded C

Functions - Pass by Reference

Example

```c
#include <stdio.h>

void modify(int *iptr);

int main()
{
    int num = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num);

    modify(&num);

    printf("After Modification\n");
    printf("num1 is %d\n", num);

    return 0;
}
```

```c
void modify(int *iptr)
{
    *iptr = *iptr + 1;
}
```



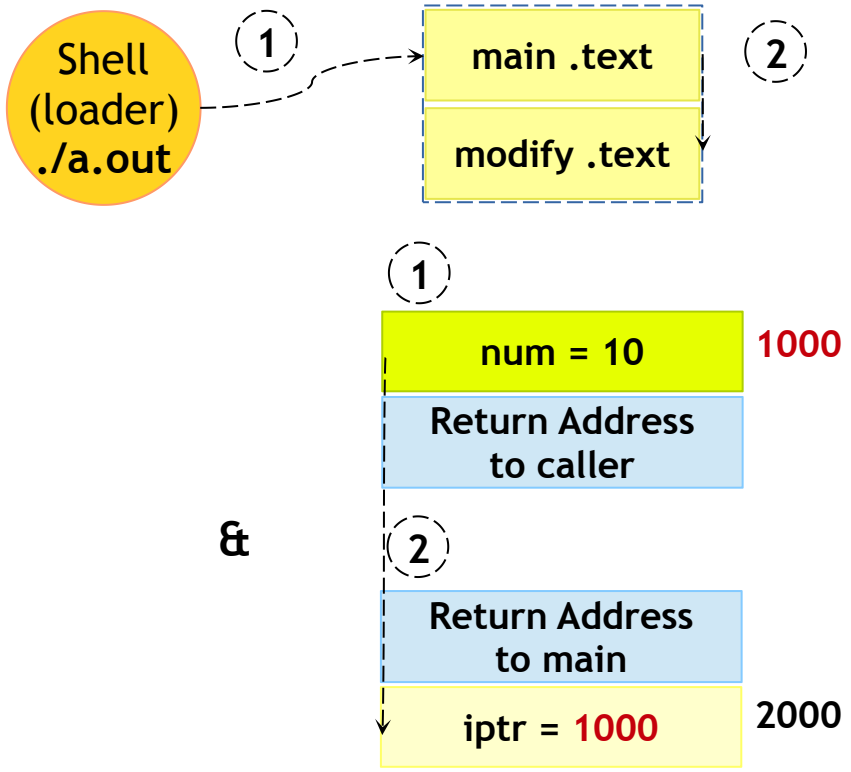
Embedded C

Functions - Pass by Reference

Example

```c
#include <stdio.h>

void modify(int *iptr);

int main()
{
    int num = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num);

    modify(&num);

    printf("After Modification\n");
    printf("num1 is %d\n", num);

    return 0;
}
```

```c
void modify(int *iptr)
{
    *iptr = *iptr + 1;
}
```



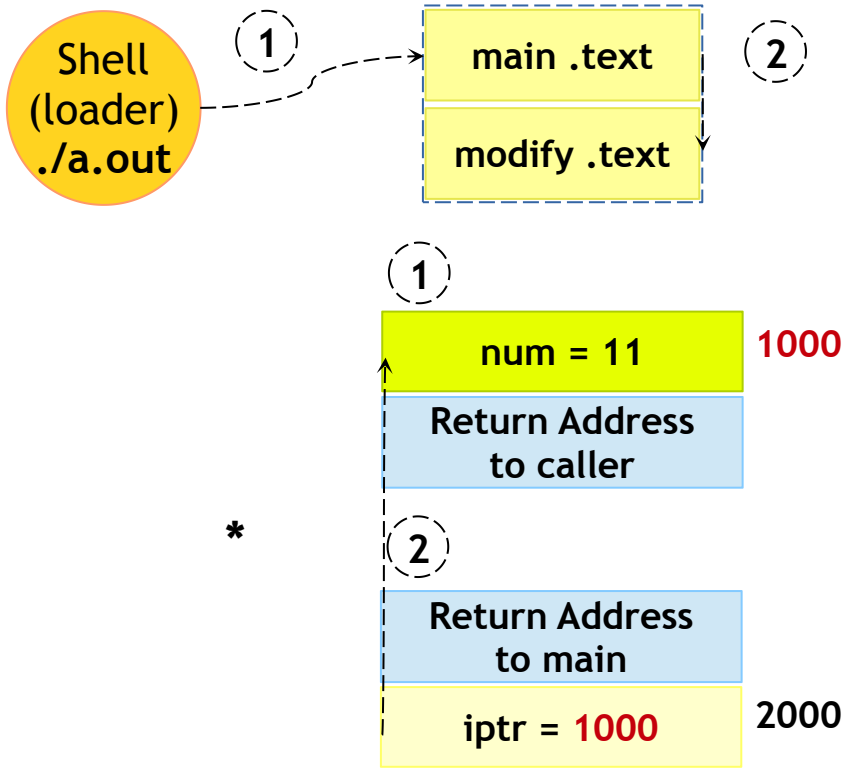
Embedded C

Functions - Pass by Reference

Example

```c
#include <stdio.h>

void modify(int *iptr);

int main()
{
    int num = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num);

    modify(&num);

    printf("After Modification\n");
    printf("num1 is %d\n", num);

    return 0;
}
```

```c
void modify(int *iptr)
{
    *iptr = *iptr + 1;
}
```



Embedded C

Functions – Pass by Reference - Advantages

- Return more than one value from a function

Embedded C

Functions - Passing Array

Example

```
#include <stdio.h>

void print_array(int array[]);

int main()
{
     int array[5] = {10, 20, 30, 40, 50};

     print_array(array);

     return 0;
}
```

```
void print_array(int array[])
{
     int i;

     for (i = 0; i < 5; i++)
     {
          printf("Index %d has Element %d\n", i, array[i]);
     }
}
```

Embedded C

Functions - Returning Array

Example

```c
#include <stdio.h>

int *modify_array(int *array, int size);
void print_array(int array[], int size);

int main()
{
     int array[5] = {10, 20, 30, 40, 50};
     int *iptr;

     iptr = modify_array(array, 5);
     print_array(iptr, 5);

     return 0;
}
```

```c
int *modify_array(int *array, int size)
{
     int  i;

     for (i = 0; i < size; i++)
     {
          *(array + i) += 10;
     }

     return array;
}
```

```c
void print_array(int array[], int size)
{
     int  i;

     for (i = 0; i < size; i++)
     {
          printf("Index %d has Element %d\n", i, array[iter]);
     }
}
```

Embedded C

Functions - Returning Array

Example

```c
#include <stdio.h>

int *return_array(void);
void print_array(int *array, int size);

int main()
{
     int *array_val;

     array_val = return_array();
     print_array(array_val, 5);

     return 0;
}
```

```c
int *return_array(void)
{
     static int array[5] = {10, 20, 30, 40, 50};

     return array;
}
```

```c
void print_array(int *array, int size)
{
     int  i;

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

Embedded C

Functions – Recursive

- Recursion is the process of repeating items in a self-similar way
- In programming a function calling itself is called as recursive function
- Two steps

Step 1: Identification of base case

Step 2: Writing a recursive case

Embedded C

Functions - Recursive - Example

Example

```c
#include <stdio.h>

/* Factorial of 3 numbers */

int factorial(int number)
{
    if (number <= 1)
    {
        return 1;
    }
    else
    {
        return number * factorial(number - 1);
    }
}

int main()
{
    int result;

    result = factorial(3);
    printf("Factorial of 3 is %d\n", result);

    return 0;
}
```

Embedded C

Pointers - Pitfalls - Segmentation Fault

- A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed.

Example

```c
#include <stdio.h>

int main()
{
    int num = 0;

    printf("Enter the number\n");
    scanf("%d", num);

    return 0;
}
```

Example

```c
#include <stdio.h>

int main()
{
    int *num = 0;

    printf("num = %d\n", *num);

    return 0;
}
```

Embedded C

Pointers - Pitfalls - Wild Pointer

- An uninitialized pointer pointing to a invalid location can be called as an wild pointer.

Example

```c
#include <stdio.h>

int main()
{
    int *iptr_1; /* Wild Pointer */
    static int *iptr_2; / Not a wild pointer */


    return 0;
}
```

Embedded C

Pointers - Const Qualifier

Example

```c
#include <stdio.h>

int main()
{
    int const *num = NULL;

    return 0;
}
```

The location, its pointing to is constant

Example

```c
#include <stdio.h>

int main()
{
    int * const num = NULL;

    return 0;
}
```

The pointer is constant

Embedded C

Strings

> A set of things tied or threaded together on a thin cord.
>
> Source: Google

Embedded C

Strings

- Contiguous sequence of characters
- Easily stores ASCII and its extensions
- End of the string is marked with a special character, the null character '\0'
- '\0' is implicit in strings enclosed with “”
- Example

“You know, now this is what a string is!”

Embedded C

Strings - Initializations

Examples

```
char char_array[5] = {'H', 'E', 'L', 'L', 'O'};
```
← Character Array

```
char str[6] = {'H', 'E', 'L', 'L', 'O', '\0'};
```
← String

```
char str[] = {'H', 'E', 'L', 'L', 'O', '\0'};
```
← Valid

```
char str[6] = {“H”, “E”, “L”, “L”, “O”};
```
← Invalid

```
char str[6] = {“H” “E” “L” “L” “O”};
```
← Valid

```
char str[6] = {“HELLO”};
```
← Valid

```
char str[6] = “HELLO”;
```
← Valid

```
char str[] = “HELLO”;
```
← Valid

```
char *str = “HELLO”;
```
← Valid

Embedded C

Strings - Size

Examples

```c
#include <stdio.h>

int main()
{
    char char_array_1[5] = {'H', 'E', 'L', 'L', 'O'};
    char char_array_2[] = "Hello";

    sizeof(char_array_1);
    sizeof(char_array_2);

    return 0;
}
```

The size of the array Is calculated So,

5, 6

```c
int main()
{
    char *str = "Hello";

    sizeof(str);

    return 0;
}
```

The size of pointer is always constant so,

4 (32 Bit Sys)

Embedded C

Strings - Size

Example

```
#include <stdio.h>

int main()
{
    if (sizeof("Hello" "World") == sizeof("Hello") + sizeof("World"))
    {
        printf("WoW\n");
    }
    else
    {
        printf("HuH\n");
    }


    return 0;
}
```

Embedded C

Strings – Library Functions

Purpose	Prototype	Return Values
Length	**size_t strlen(const char *str)**	String Length
Compare	**int strcmp(const char *str1, const char *str2)**	str1 < str2 → < 0 str1 > str2 → > 0 str1 = str2 → = 0
Copy	**char *strcpy(char *dest, const char *src)**	Pointer to dest
Check String	**char *strstr(const char *haystack, const char *needle)**	Pointer to the beginning of substring
Check Character	**char *strchr(const char *s, int c)**	Pointer to the matched char else NULL
Merge	**char *strcat(char *dest, const char *src)**	Pointer to dest

Storage Classes

Embedded C

Memory Segments

The Linux OS is divided into two major sections

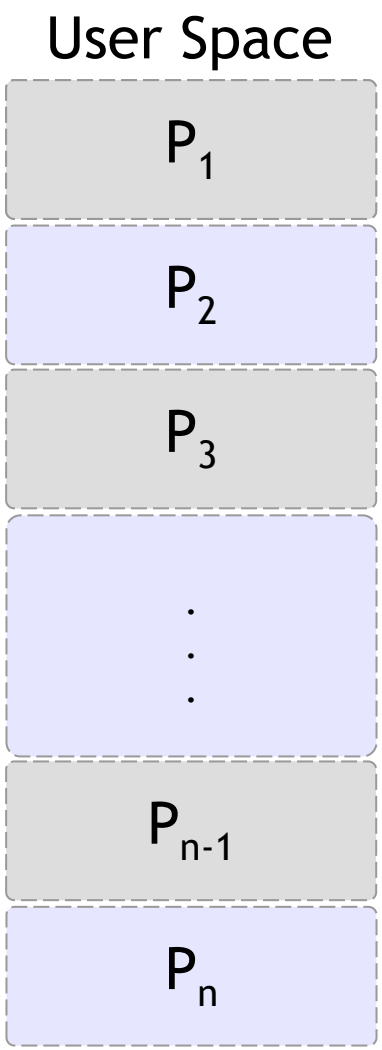
- User Space
- Kernel Space

The user programs cannot access the kernel space. If done will lead to segmentation violation

Let us concentrate on the user space section here

Embedded C

Memory Segments



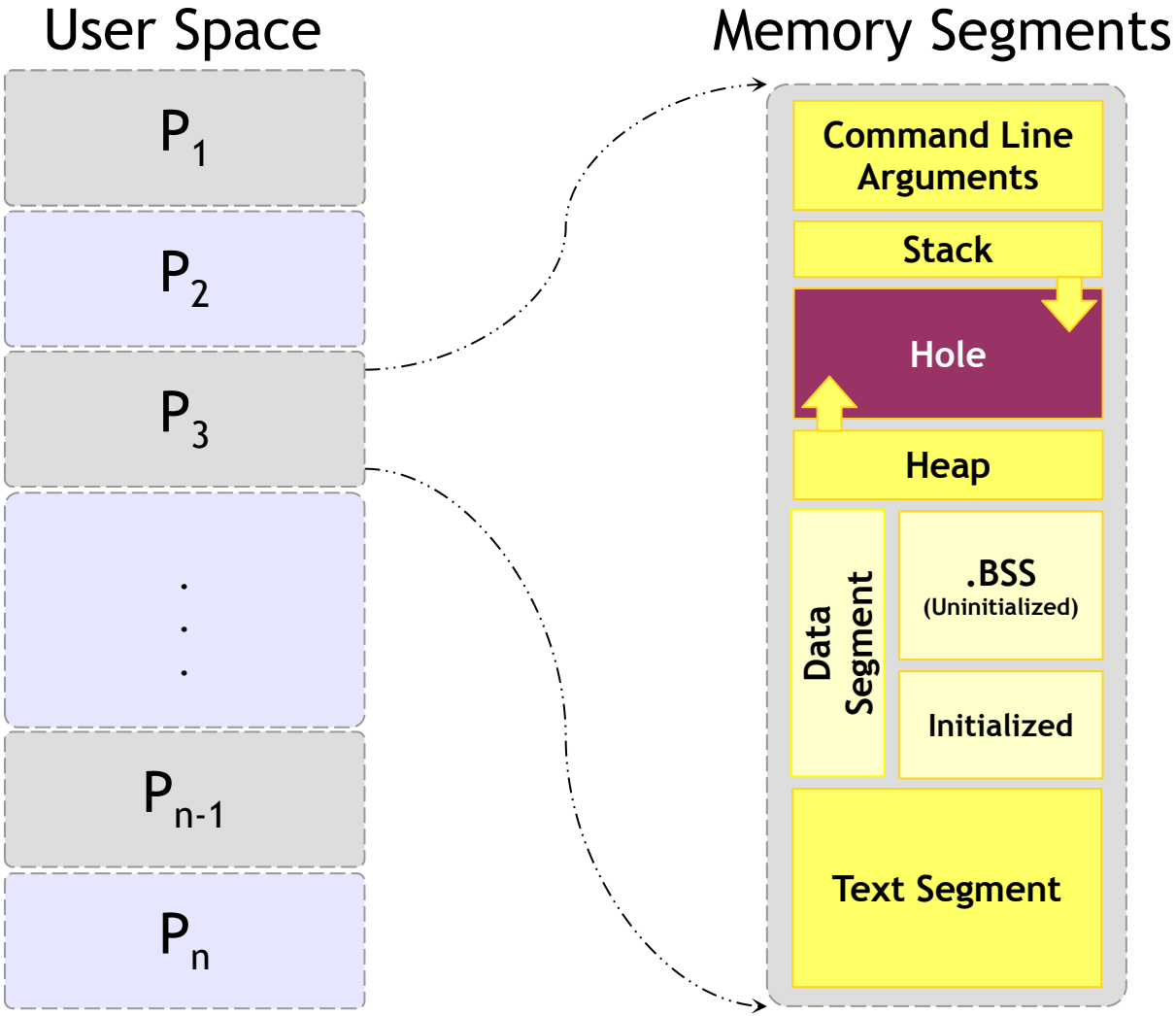
The User space contains many processes

Every process will be scheduled by the kernel

Each process will have its memory layout discussed in next slide

Embedded C

Memory Segments



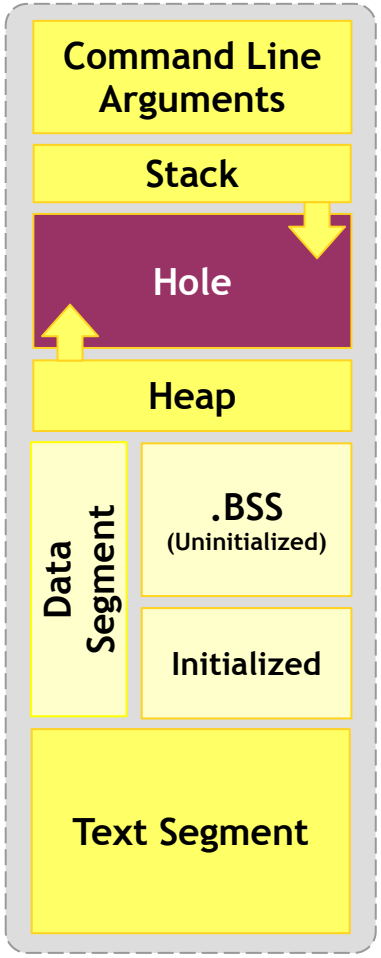
The memory segment of a program contains four major areas.

- Text Segment
- Stack
- Data Segment
- Heap

Embedded C

Memory Segments – Text Segment

Memory Segments

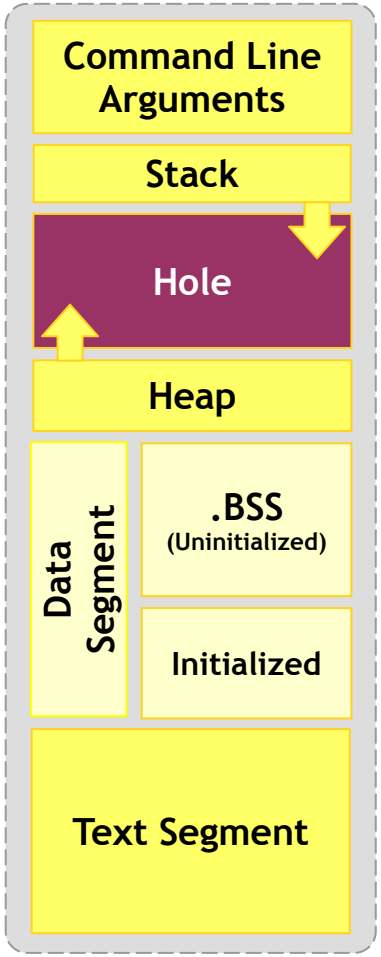


- Also referred as Code Segment
- Holds one of the section of program in object file or memory
- In memory, this is place below the heap or stack to prevent getting over written
- Is a read only section and size is fixed

Embedded C

Memory Segments – Data Segment

Memory Segments

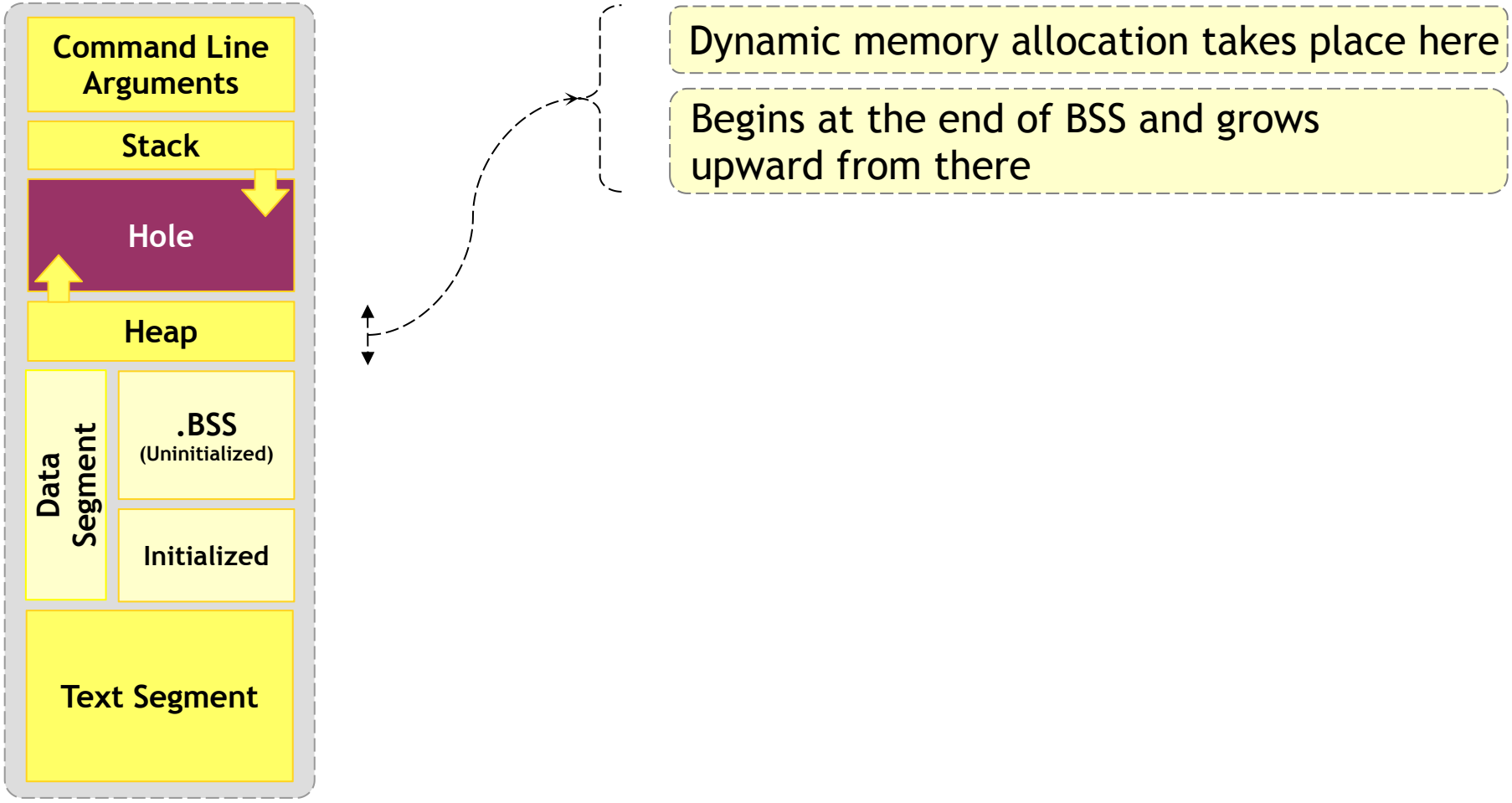


- Contains 2 sections as initialized and uninitialized data segments
- Initialized section is generally called as Data Segment
- Uninitialized section is referred as BSS (Block Started by Symbol) usually filled with 0s

Embedded C

Memory Segments - Heap

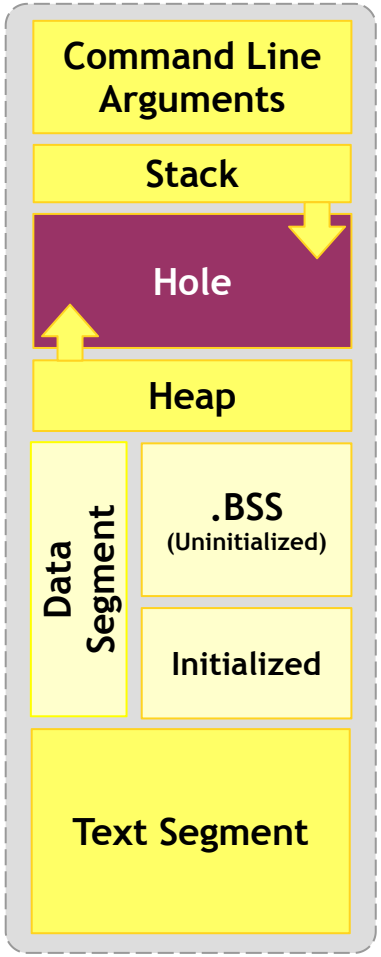
Memory Segments



Embedded C

Memory Segments – Stack Segment

Memory Segments



- Adjoins the heap area and grow in opposite area of heap when stack and heap pointer meet (Memory Exhausted)
- Typically loaded at the higher part of memory
- A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack
- The set of values pushed for one function call is termed a “stack frame”

Embedded C

Storage Classes

Storage Class	Scope	Lifetime	Memory Allocation
auto	Within the block / Function	Till the end of the block / function	Stack
register	Within the block / Function	Till the end of the block / function	Register
static local	Within the block / Function	Till the end of the program	Data Segment
static global	File	Till the end of the program	Data segment
extern	Program	Till the end of the program	Data segment

Embedded C

Storage Classes

Example

```c
#include <stdio.h>

int global_1;
int global_2 = 10;

static int global_3;
static int global_4 = 10;

int main()
{
    int local_1;
    static int local_2;
    static int local_3 = 20;

    register int i;
    for (i = 0; i < 0; i++)
    {
        /* Do Something */
    }

    return 0;
}
```

Variable	Storage Class	Memory Allocation
global_1	Global	.BSS
global_2	Global	Initialized data segment
global_3	Static global	.BSS
global_4	Static global	Initialized data segment
local_1	auto	stack
local_2	Static local	.BSS
local_3	Static local	Initialized data segment
iter	Register	Registers

Embedded C

Declaration

```
extern int num1;
extern int num1;

int main();

int main()
{
    int num1, num2;
    char short_opt;

    ...
}
```

Declaration specifies type to the variables

Its like an announcement and hence can be made 1 or more times

Declaration about num1

Declaration about num1 yet again!!

Declaration about main function

Embedded C

Storage Classes - extern

file1.c

```c
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
        func_2();
    }

    return 0;
}
```

file2.c

```c
#include <stdio.h>

extern int num;

int func_1()
{
    printf("num is %d from file2\n", num);

    return 0;
}
```

file3.c

```c
#include <stdio.h>

extern int num;

int func_2()
{
    printf("num is %d from file3\n", num);

    return 0;
}
```


Preprocessor

Embedded C

Preprocessor – Header Files

- A header file is a file containing C declarations and macro definitions to be shared between several source files.
- Has to be included using C preprocessing directive '**#include**'
- Header files serve two purposes.
  - Declare the interfaces to parts of the operating system by supplying the definitions and declarations you need to invoke system calls and libraries.
  - Your own header files contain declarations for interfaces between the source files of your program.

Embedded C

Preprocessor – Header Files vs Source Files

VS

- Declarations
- Sharable/reusable
  - #defines
  - Datatypes
- Used by more than 1 file

- Function and variable definitions
- Non sharable/reusable
  - #defines
  - Datatypes

Embedded C

Preprocessor – Header Files - Syntax

Syntax

```
#include <file.h>
```

- System header files
- It searches for a file named ***file*** in a standard list of system directories

Syntax

```
#include "file.h"
```

- Local (your) header files
- It searches for a file named ***file*** first in the directory containing the current file, then in the quote directories and then the same directories used for <file>

Embedded C

Preprocessor - Header Files - Once-Only

- If a header file happens to be included twice, the compiler will process its contents twice causing an error
- E.g. when the compiler sees the same structure definition twice
- This can be avoided like

Example

```
#ifndef NAME
#define NAME

/* The entire file is protected */

#endif
```

Embedded C

Preprocessor - Macro - Object-Like

- An object-like macro is a simple identifier which will be replaced by a code fragment
- It is called object-like because it looks like a data object in code that uses it.
- They are most commonly used to give symbolic names to numeric constants

Syntax

```
#define SYMBOLIC_NAME      CONSTANTS
```

Example

```
#define BUFFER_SIZE        1024
```

Embedded C

Preprocessor - Macro - Arguments

- Function-like macros can take arguments, just like true functions
- To define a macro that uses arguments, you insert parameters between the pair of parentheses in the macro definition that make the macro function-like

Example

```
#include <stdio.h>

#define SET_BIT(num, pos)      (num | (1 << pos))

int main()
{
    SET_BIT(0, 2);

    return 0;
}
```

Embedded C

Preprocessor - Macro - Multiple Lines

- You may continue the definition onto multiple lines, if necessary, using backslash-newline.
- When the macro is expanded, however, it will all come out on one line

Example

```
#include <stdio.h>

#define SWAP(a, b)                  \
{                                   \
    int temp = a;                   \
    a = b;                          \
    b = temp;                       \
}

int main()
{
    int num1 = 10, num2= 20;

    SWAP(num1, num2);

    printf("%d %d\n", num1, num2);

    return 0;
}
```

Embedded C

Preprocessor – Macro – Standard Predefined

- Several object-like macros are predefined; you use them without supplying their definitions.
- Standard are specified by the relevant language standards, so they are available with all compilers that implement those standards

Example

```
#include <stdio.h>

int main()
{

    printf("Program: \"%s\" ", __FILE__);
    printf("was compiled on %s at %s. ", __DATE__, __TIME__);
    printf("This print is from Function: \"%s\" at line %d\n", __func__, __LINE__);

    return 0;
}
```

Hope you enjoyed the session.
Thank You