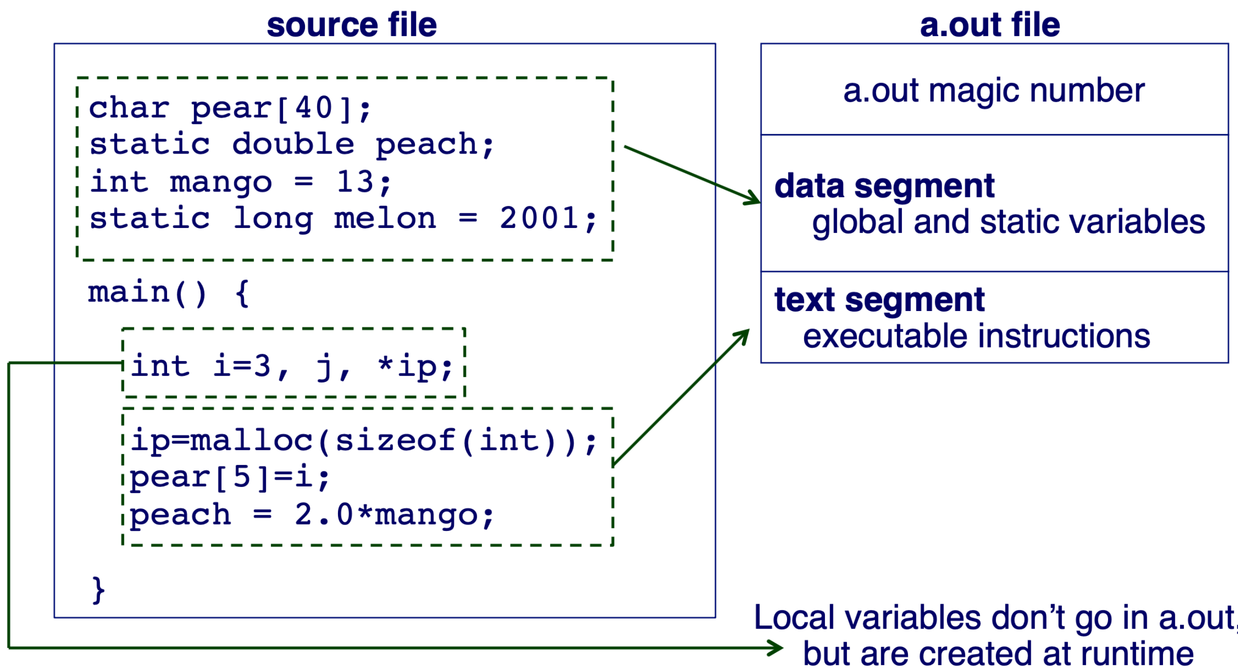
C Programming:
(part 3)

Topics

- Address Space
- Memory Segments
- Space Allocation and Activation Records
- Dynamic Allocation
- Preprocessing
- Handling Command Line Arguments
- System Calls
- File I/O

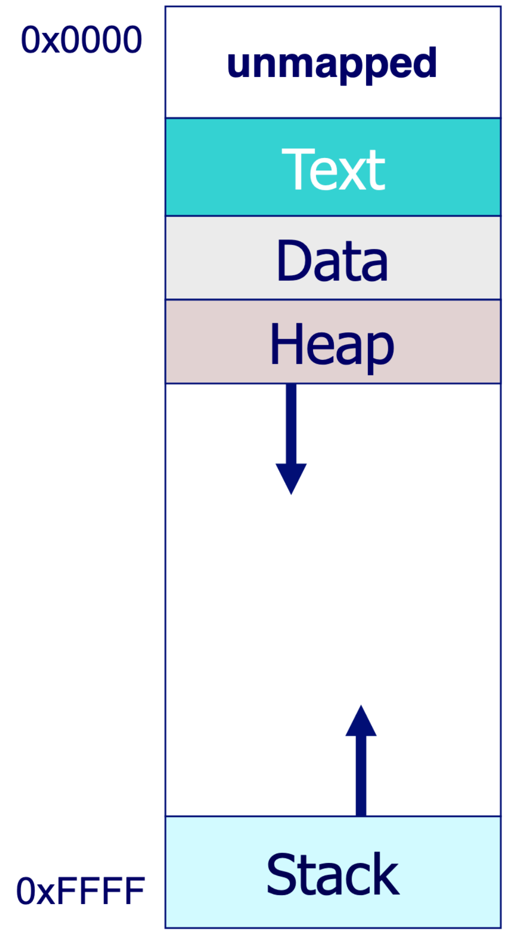
Address Space and Segments

- The address space represents the memory space of a program
- Recall all instructions and data for a program is stored in memory
- The address space consists of various segments
- A segment section of related stuff in memory in a binary



Segments

- Segments of an executable are laid out in memory
- An application/program's memory has 4 segments
  - Text: instructiona of the program
  - Data: global and static data
  - Heap: dynamic allocation
  - Stack: function calls and local data

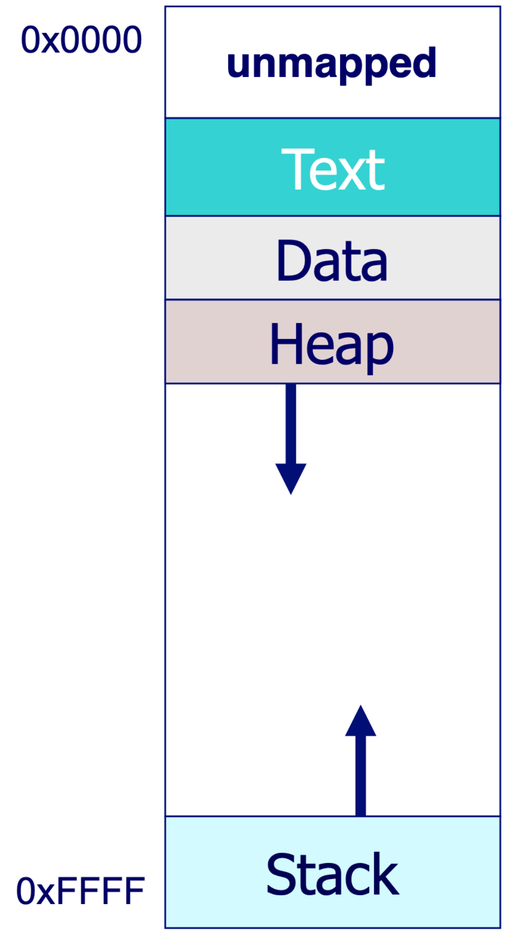


Stack

- When a function call is performed in a program, the run-time system must allocate resource to execute
  - Memory for any local variables, arguments, and result
  - This is stored on the stack segment
- The same function can be called many times
  - Ex. Recursion
  - Each function instance will require resources
- The state associated with a function is called an activation record

Allocating Space for Variables

- Allocation records are allocation of a call stack
- Function calls leads to a new activation record pushed on top of the stack
- Activation record is popped off the stack when the function returns
- Lets see an example..



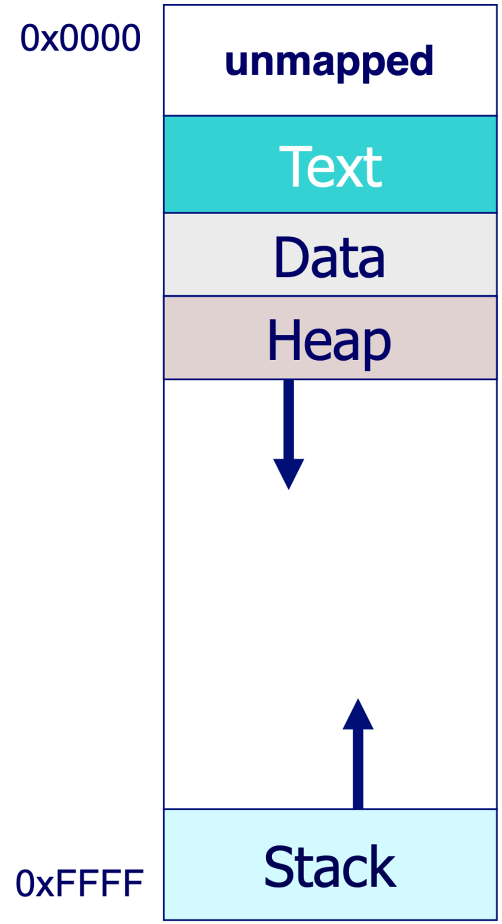
Allocating Space for Variables

- Compute the sum of numbers from 1 to N

```
int summation (int n){
        if(n == 0){
                return 0;
        }
        return n + summation(n-1);
}
```

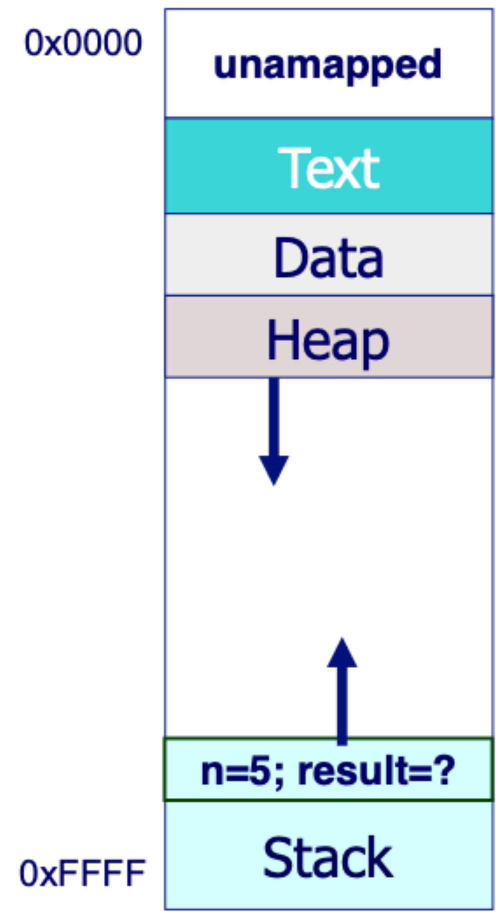
- Recall that the activation record for a function contains state for all arguments, local variables, and result

int n; int result;



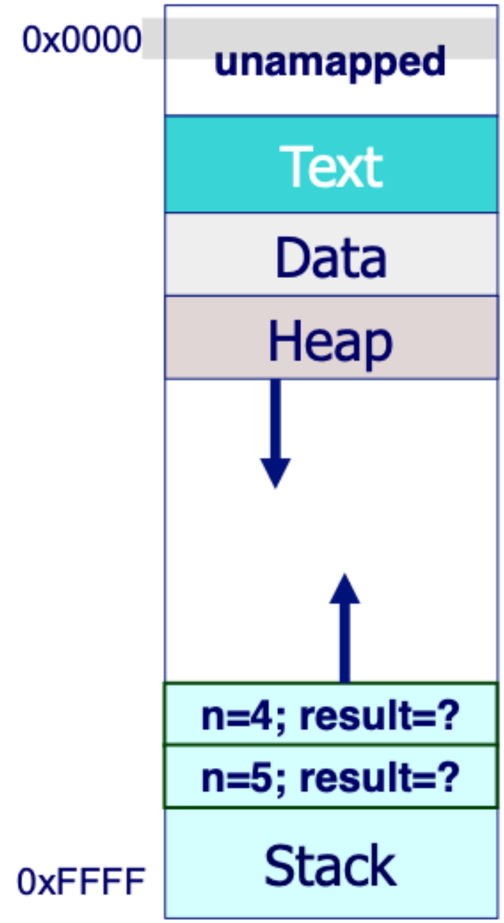
Allocating Space for Variables

- Lets calculate N = 5
- Execution sequence
  - summation(5);



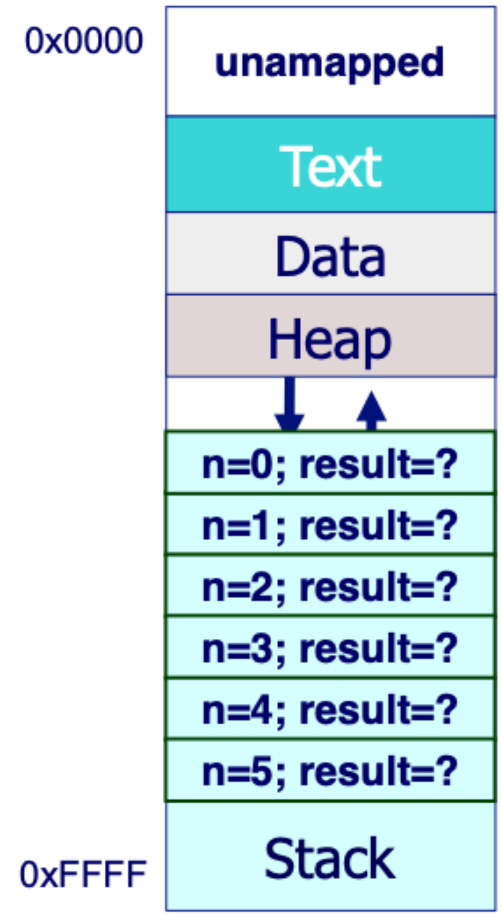
Allocating Space for Variables

- Lets calculate N = 5
- Execution sequence
  - summation(5);
  - summation(4);



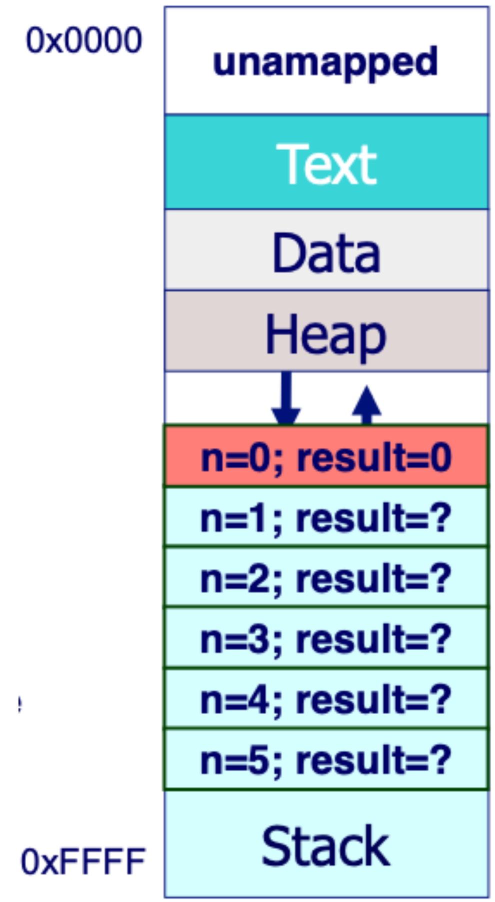
Allocating Space for Variables

- Lets calculate N = 5
- Execution sequence
  - summation(5);
  - summation(4);
  - summation(3);
  - summation(2);
  - summation(1);
  - summation(0);



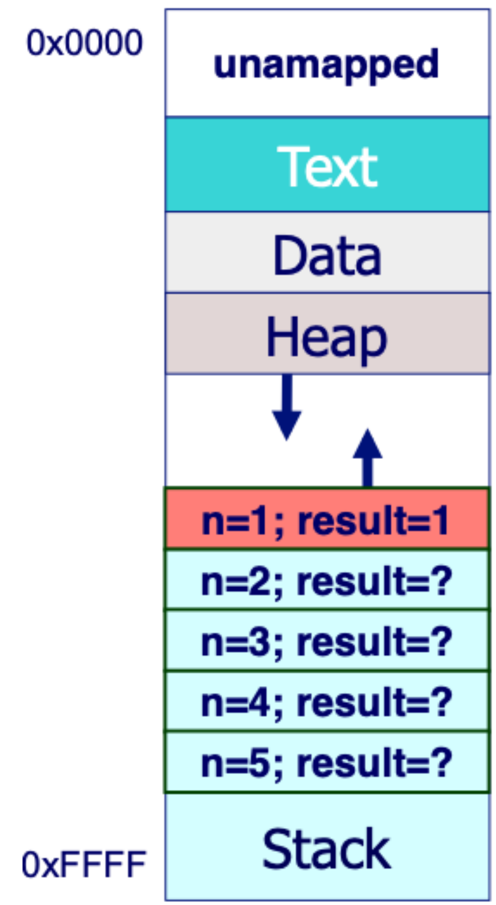
Allocating Space for Variables

- Lets calculate N = 5
- Execution sequence
  - summation(5);
  - summation(4);
  - summation(3);
  - summation(2);
  - summation(1);
  - summation(0);
- As function return, their activation records are removed
- Important: The state in a function call can be accessed safely only so long as its activation record is still active on the stack



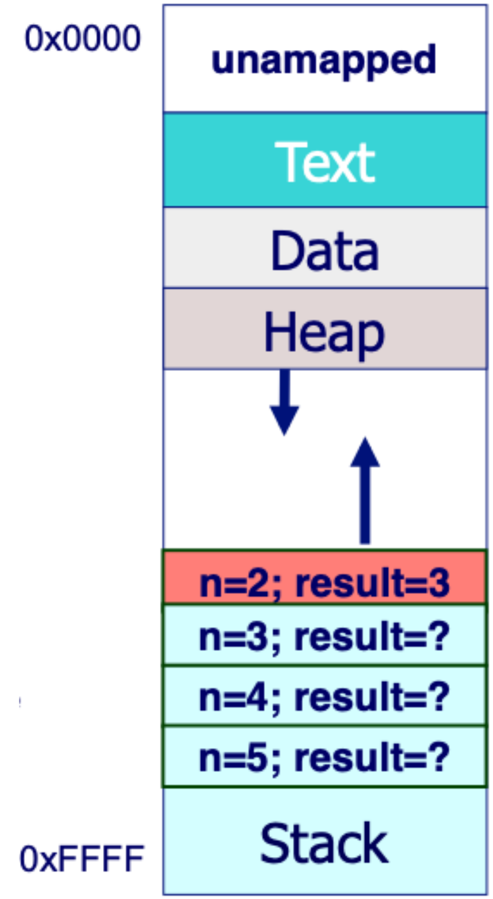
Allocating Space for Variables

- Lets calculate N = 5
- Execution sequence
  - summation(5);
  - summation(4);
  - summation(3);
  - summation(2);
  - summation(1);
- As function return, their activation records are removed
- Important: The state in a function call can be accessed safely only so long as its activation record is still active on the stack



Allocating Space for Variables

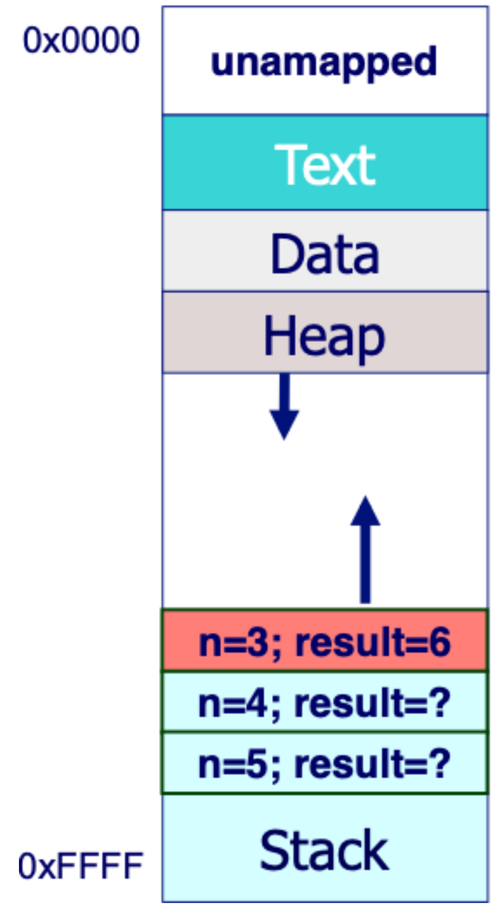
- Lets calculate N = 5
- Execution sequence
  - summation(5);
  - summation(4);
  - summation(3);
  - summation(2);



- As function return, their activation records are removed
- Important: The state in a function call can be accessed safely only so long as its activation record is still active on the stack

Allocating Space for Variables

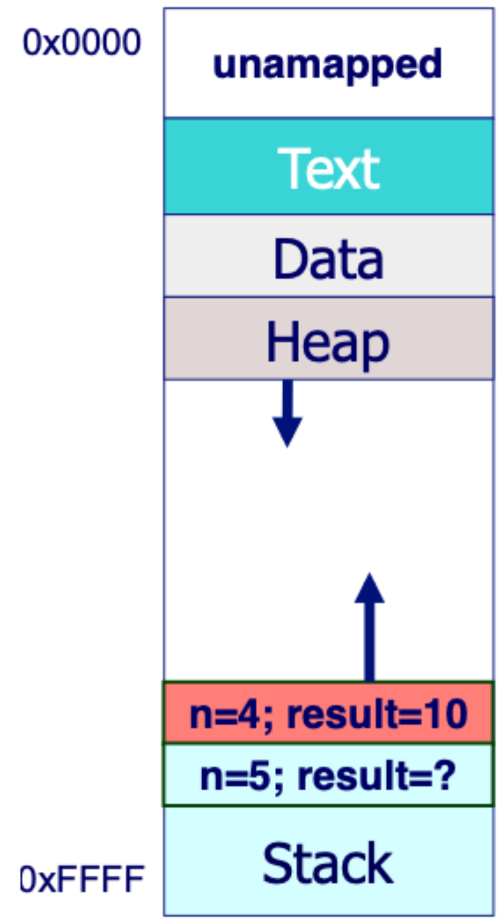
- Lets calculate N = 5
- Execution sequence
  - summation(5);
  - summation(4);
  - summation(3);



- As function return, their activation records are removed
- Important: The state in a function call can be accessed safely only so long as its activation record is still active on the stack

Allocating Space for Variables

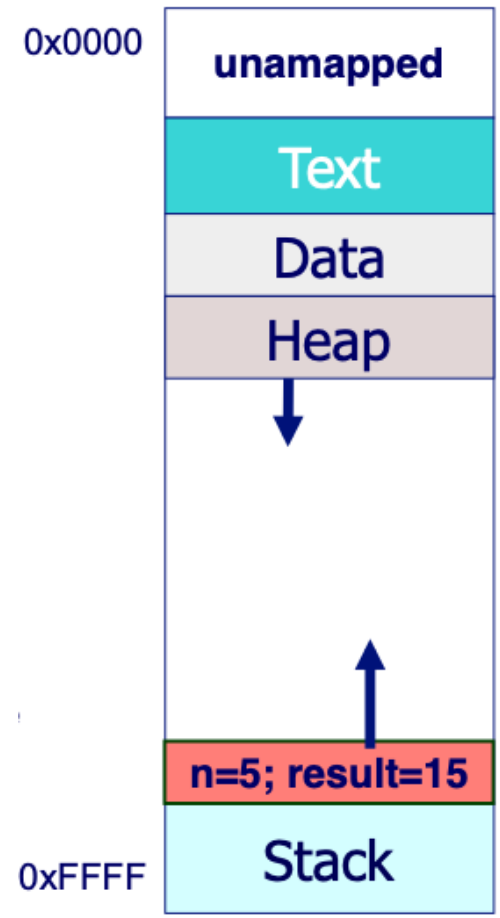
- Lets calculate N = 5
- Execution sequence
  - summation(5);
  - summation(4);



- As function return, their activation records are removed
- Important: The state in a function call can be accessed safely only so long as its activation record is still active on the stack

Allocating Space for Variables

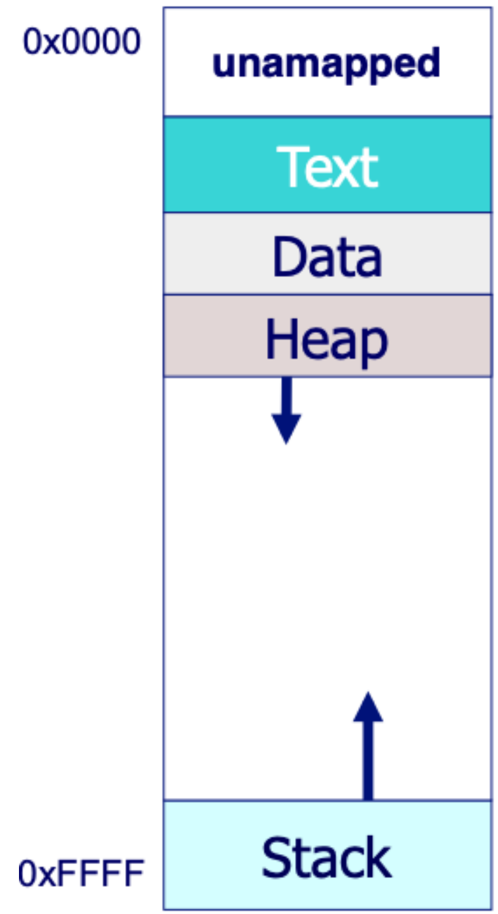
- Lets calculate N = 5
- Execution sequence
  - summation(5);



- As function return, their activation records are removed
- Important: The state in a function call can be accessed safely only so long as its activation record is still active on the stack

Allocating Space for Variables

- Lets calculate N = 5
- Execution sequence



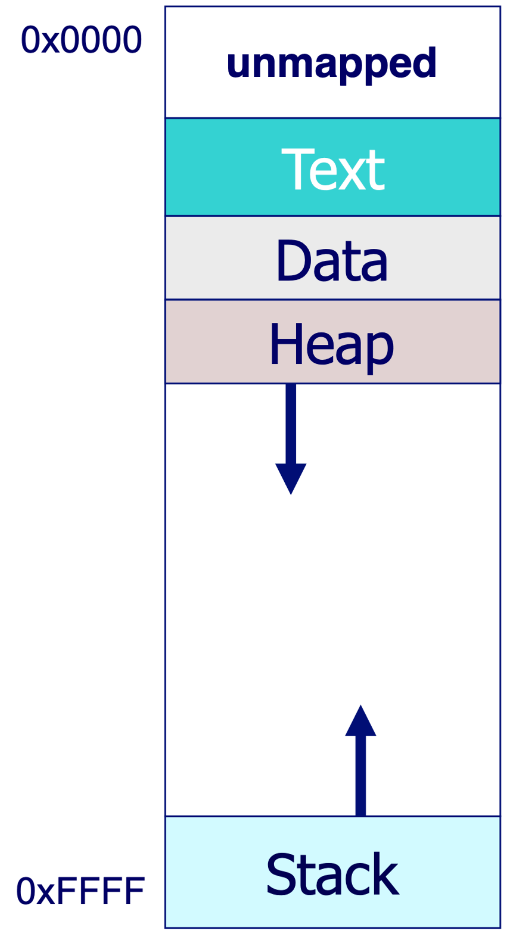
- As function return, their activation records are removed
- Important: The state in a function call can be accessed safely only so long as its activation record is still active on the stack

Dynamic Allocation

- What is we want to write a program to handle a variable amount of data?
  - Ex. Arbitrary list of numbers to sort
  - Can’t allocate an array because we don’t know how many numbers we will get
  - Naïve Solution: Allocate a very large array
    - Inflexible and inefficient
- Memory area whose lifetime does not match any particular function?
  - Remember Local variables are don’t live long
    - Placed in stack
    - Lives and dies with containing function
  - Naïve solution: Global variable
    - Placed in global area (data segment) to be accessible from anywhere
    - Lives forever throughout the whole program execution
    - However takes up space and lasts forever
- Answer: Dynamic Memory Allocation
  - Similar to “new” in Java

Dynamic Memory

- Another area of region of memory exists called the heap
- Dynamic requests for memory are allocated in this region
- Managed by the run-time system
- Memory can be allocated in the heap using malloc()
  - void *malloc(int numBytes);
  - malloc() allocates a given number of bytes within the heap and returns a pointer to the start of the bytes

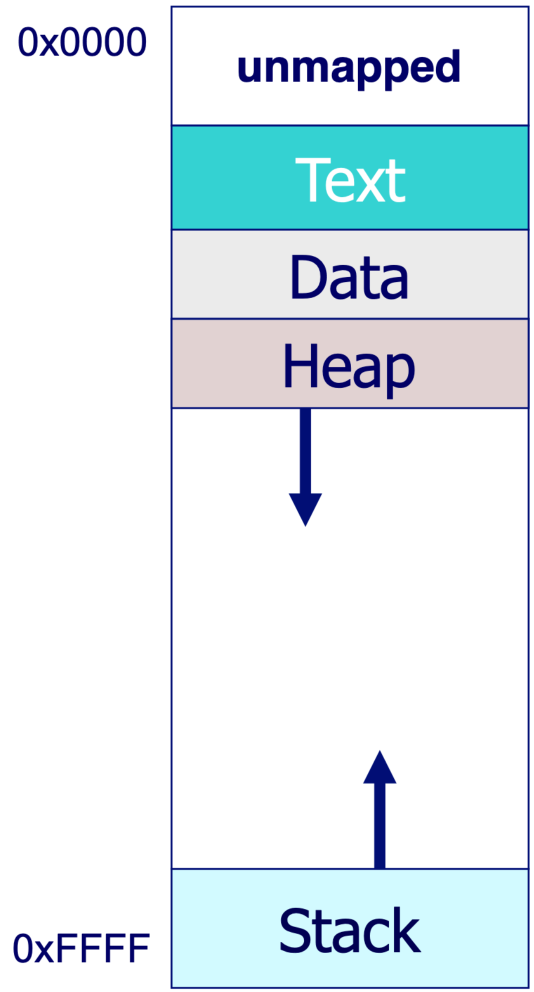


Dynamic Allocation

```
program_x () {

    int x = function_x ();
}


function_x() {
  malloc....
}
```

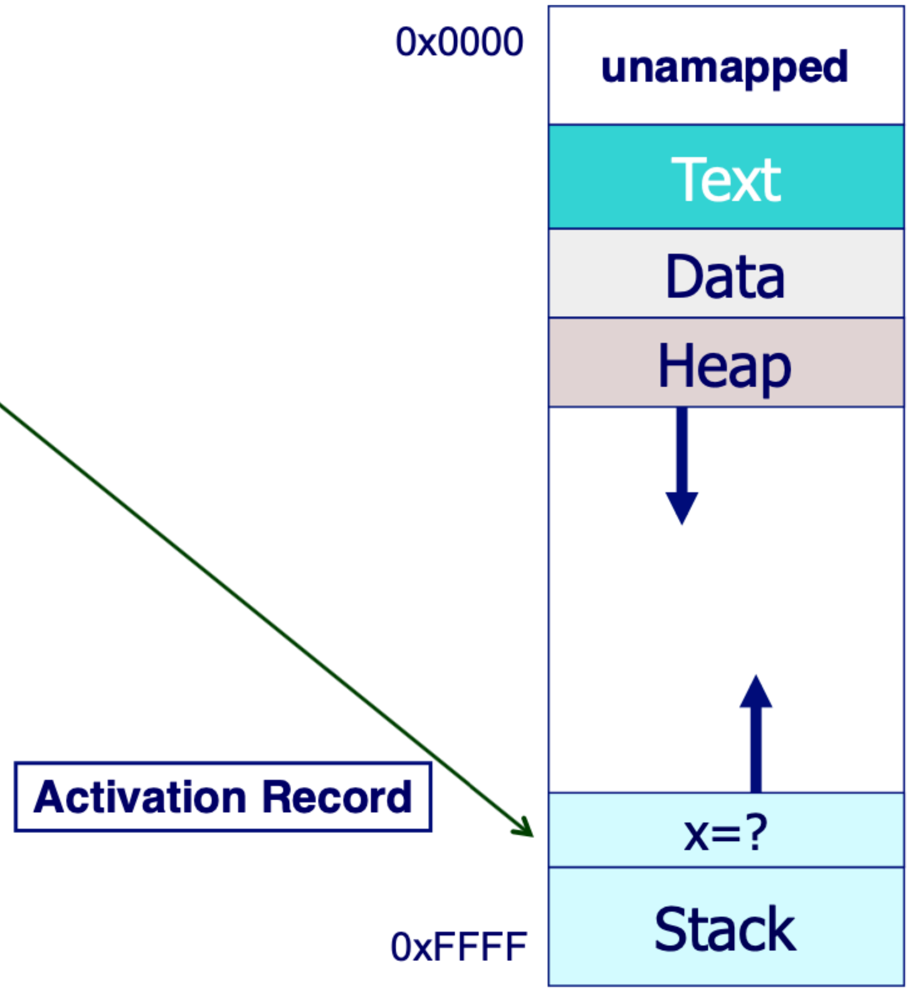


Dynamic Allocation

```
program_x () {

      int x = function_x ();
}


function_x() {
   malloc….
}
```



Dynamic Allocation

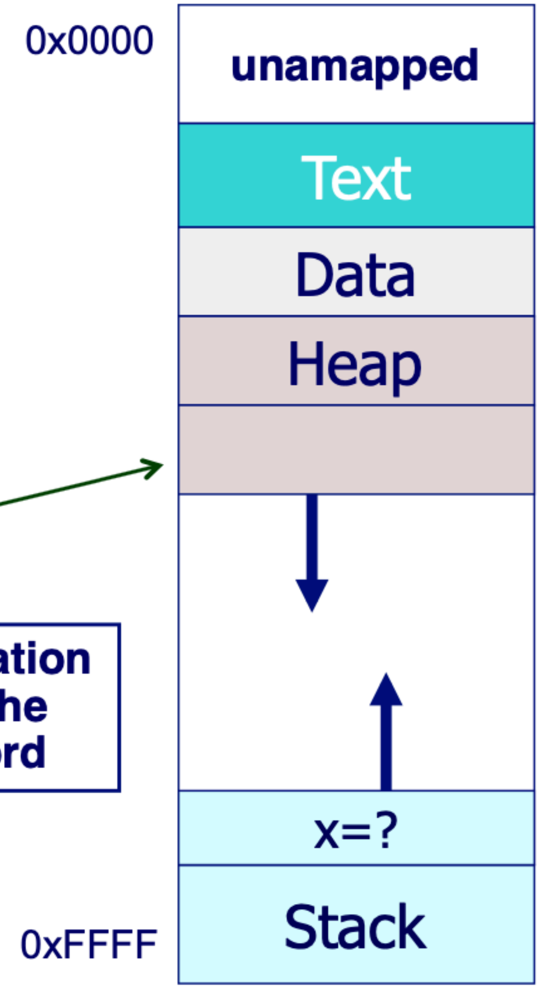
```
program_x () {

    int x = function_x ();
}


function_x() {
  malloc....
}
```

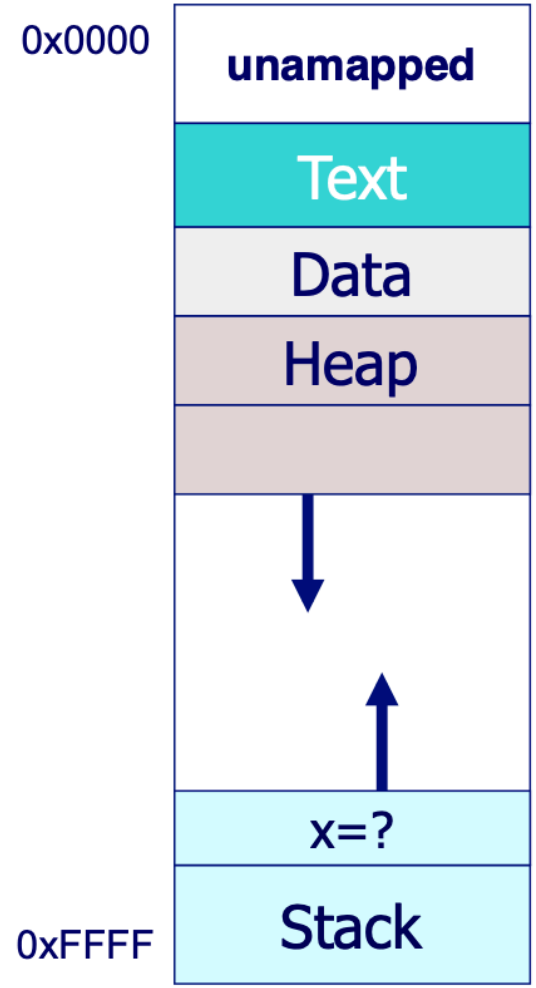
function_x asks for a chunk of memory

NOTE: This allocation is NOT part of the activation record



Dynamic Allocation

- After function returns, memory is still allocated
- Remains allocated until it is released using free()



malloc()

- The Standard C library provides a function for dynamic memory allocation

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

- System tries to allocate a contiguous region of memory of size numBytes
- If there is enough free memory, it returns a pointer to the beginning of this region
- Returns NULL if there is insufficient free memory
- Why is return type void*?
  - malloc is designed to be generic to handle any type, so returns a generic pointer (e.g. void*)

Using malloc()

- How many know how many bytes to allocate?
- Use sizeof() function
  - size_t sizeof(type)
  - given a data type, it will return how many bytes it needs
  - Ex. sizeof(int)
    - returns 4 since an type int is 4 bytes long
- We can also get the size of structs
  - Ex.

```
struct myStruct{
    int x;
    char y;
}
```

  - sizeof(struct myStruct) will return 5
    - 4 byte int + 1 byte char = 5 bytes

Using malloc()

- How do we use the void* returned from malloc()?
  - ex. ? = malloc(sizeof(int));
- Typecast the void pointer to start using it
  - ex. ? = (int *) malloc(sizeof(int));
- Use the typecast pointer to start using the allocated space
  - int *x = (int *) malloc(sizeof(int));
    *x = 5;
- We can also allocate multiple of a type (like an array)
  - int *nums = (int *) malloc(sizeof(int) * n);
    - returns the address to a space large enough to hold n integers
  - We can then start using the space like an array:
    nums[0] = 1;
    nums[1] = 2;

free()

- Remember dynamically allocated memory will remain allocated until it is manually freed using free()
- It is important to free() allocated memory when not in use any more
  - We only have finite amount of memory
  - If we don't release, we'll eventually run out of heap space
- void free(void *ptr)
  - Takes in a pointer pointing to the start of the heap allocation
  - Frees memory associated with that allocation
- Example:

```
int *x = (int *) malloc(sizeof(int));
free(x);
```

malloc() and free() (example)

```
int airbornePlanes;
struct flightType *planes;

printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);

planes =
 (struct flightType*)malloc(sizeof(struct flightType) *
                  airbornePlanes);
if (planes == NULL) {
  printf("Error in allocating the data array.\n");
  ...
}
planes[0].altitude = ...
...

free(planes);
```

If allocation fails, malloc returns NULL.

Note: Can use array notation or pointer notation.

typedef

- typedef is used to name types
  - typedef <type> <name>;
- Useful for clarity and ease of use
- Examples:
  - typedef int Color;
  - typedef struct flightType plane;
  - typedef struct ab_type{
    int a;
    double b;
    } ABGroup;

Preprocessor

- C compilation uses a preprocess called cpp
- The preprocessor manipulates the source code in various ways before the code is passed through the compiler
  - Preprocessor is controlled by directives
  - Directives start with #
- Examples:
  - #include <stdio.h>
  - #include ”myHeader.h”
  - #define MAX 100   // replace every “MAX” with 100
  - #ifdef MAX   // if MAX is defined include following code

        …
    #endif

Standard C Library

- Standard C Library provides useful basic functionality
  - A collection of functions and macros that must be implemented by any ANSI standard implementation
    - Ex. Libraries for I/O, string handling, etc.
  - Automatically linked with every executable
  - Implementation depends on processor, operating system, etc, but interface is standard
- Since they are not part of the language, compiler must be told about function interfaces
- Standard header files are provided, which contain declarations of functions, variables, etc.
  - Ex. stdio.h

Command Line Arguments

- When using shell we might want to pass arguments to the our programs
  - Ex. $ hello 5
- Entire command line would be given to the program as a sequence of strings
  - White space are typically the separator characters
- Lets take a look at the typical main function:
  - int main(int argc, char *argv[]){
      ….
    }
- argc is the number of strings passed in
  - includes program name
  - In our example: argc = 2 (“hello” and “5”)
- argv is the strings themselves
  - In our example: argv[0] = “hello” and argv[1] = “5\0”

System Calls

- The operating system extends the functionality of the underlying hardware
  - OS functionality is exported as a set of system calls
  - In C, system calls are "wrapped" by C functions
    - System calls look like typical C function calls
  - Can look at full list of system calls here:
    - https://man7.org/linux/man-pages/man2/syscalls.2.html#:~:text=The%20system%20call%20is%20the,or%20perhaps%20some%20other%20library).
- In some instances, the C standard library adds functionality on top of system calls
  - Ex. File I/O

File I/O

- A file is a contiguous set of bytes
  - Has a name
  - Files can be created, removed, read from, written to, and appended to
- Unix/Linux supports persistent files stores on disk
  - Access using system calls:
    - open(), read(), write(), close(), creat(), lseek()
- C supports extended interface to UNIX files:
  - Wrapper to i/o system calls
    - fopen(), fscanf(), fprintf(), fgetc(), fputc(), fclose()
  - views files as streams of bytes
  - Learn more here:
    - https://www.tutorialspoint.com/cprogramming/c_file_io.htm

fopen()

- fopen() associates a physical file with a stream
- FILE *fopen(char *name, char*mode);
- The name argument
  - Name/path of the physical file
- Second argument: "mode"
  - How the file will be used
ex.
"r" – open file for reading
"w" – open file for writing
"a" – open file for appending

fprintf() and fscanf()

- Once a file is opened, it can be read of written to using fscanf() and fprintf()
- Just like scanf() and printf() except with an additional argument specifying the file pointer
- Examples:
  - fprintf(outfile, “The answer is %d\n”, x);
  - fscanf(infile, “%s %d/%d/%d %lf”,
    &name, &month, &day, &year, &gpa);
- When a program starts, there are three standard streams open for input, output, and errors
  - stdin, stdout, stderr

fclose()

- fclose() closes a file stream
  - flushes all buffers to file
- int fclose(FILE *stream)
- Example:
  - FILE *outfile = fopen(”outfile.txt”, “w”);
    fprintf(outfile, “HelloWorld”);
    fclose(outfile);
- Make sure to close any files that you are done reading and writing to
  - OS can have a finite number of files open
  - If program exits abonormally, buffered data may not be flushed, resulting in data not reaching file