Session 14

Friday, 1 August 2025 9:34 AM

Let's delve into each type of pointer with illustrative C code examples.

1. Basic Pointer (Type-Specific Pointers)

A pointer that holds the memory address of a variable of a specific data type.

C

```
#include <stdio.h>
int main() {
    int num = 10;
    int *ptr_num; // Declares an integer pointer
ptr_num = &num; // Stores the address of 'num' in 'ptr_num'
printf("Value of num: %d\n", num);
    printf("Address of num: %p\n", &num);
    printf("Value of ptr_num (address it holds): %p\n", ptr_num);
    printf("Value pointed to by ptr_num: %d\n", *ptr_num); // Dereferencing the pointer
*ptr_num = 20; // Modifying the value through the pointer
    printf("New value of num: %d\n", num);
char ch = 'A';
    char *ptr_ch = &ch; // Declares and initializes a character pointer
printf("\nValue of ch: %c\n", ch);
    printf("Value pointed to by ptr_ch: %c\n", *ptr_ch);
return 0;
}
```

2. Null Pointer

A pointer that points to no valid memory location. It's often used to indicate that a pointer is not currently pointing to anything, or to initialize pointers to a known "safe" state.

C

```
#include <stdio.h>
int main() {
    int *ptr_null = NULL; // Declares and initializes a null pointer
if (ptr_null == NULL) {
        printf("ptr_null is a null pointer.\n");
    } else {
        printf("ptr_null is not a null pointer.\n");
    }
// Attempting to dereference a null pointer leads to undefined behavior/segmentation fault
    // if (*ptr_null == 0) { // DO NOT uncomment this line, it will crash
    //     printf("This would crash if uncommented.\n");
    // }
// Null pointers are useful for error checking, e.g., after malloc
    int *dynamic_array = (int *)malloc(5 * sizeof(int));
    if (dynamic_array == NULL) {
        printf("Memory allocation failed!\n");
    } else {
        printf("Memory allocated successfully.\n");
        free(dynamic_array); // Free the allocated memory
    }
return 0;
}
```

3. Void Pointer (Generic Pointer)

A pointer that can hold the address of any data type. It's "generic" because it doesn't know the type of data it's pointing to. Requires typecasting before dereferencing.

C

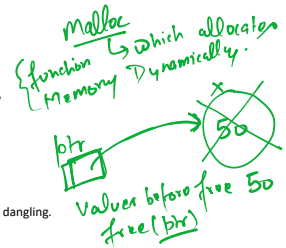
```
#include <stdio.h>
int main() {
    int num = 100;
    float pi = 3.14f;
    void *ptr_void; // Declares a void pointer
ptr_void = &num; // ptr_void now holds the address of an int
    printf("Value of num using void pointer: %d\n", *(int *)ptr_void); // Typecast to int* before
dereferencing
ptr_void = &pi; // ptr_void now holds the address of a float
    printf("Value of pi using void pointer: %f\n", *(float *)ptr_void); // Typecast to float* before
dereferencing
// Void pointers are commonly used with memory allocation functions like malloc
    int *arr = (int *)malloc(5 * sizeof(int)); // malloc returns void*, typecasted to int*
    if (arr != NULL) {
        arr[0] = 10;
        printf("First element of allocated array: %d\n", arr[0]);
        free(arr);
    }
return 0;
}
```

4. Dangling Pointer

A pointer that points to a memory location that has been deallocated (freed). Accessing a dangling pointer leads to undefined behavior.

C

```
#include <stdio.h>
#include <stdlib.h> // For malloc and free
int main() {
    int *ptr_dangling = (int *)malloc(sizeof(int)); // Allocate memory
    if (ptr_dangling == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
*ptr_dangling = 50;
    printf("Value before free: %d\n", *ptr_dangling);
free(ptr_dangling); // Deallocate the memory. ptr_dangling is now dangling.
  // At this point, ptr_dangling points to freed memory.
    // Attempting to access *ptr_dangling is undefined behavior.
    // It might print the old value, print garbage, or crash the program.
    printf("Value after free (DANGEROUS ACCESS): %d\n", *ptr_dangling); // DO NOT RELY ON THIS
OUTPUT
ptr_dangling = NULL; // Good practice: set to NULL after freeing to avoid dangling
    printf("Pointer set to NULL after free: %p\n", (void *)ptr_dangling);
return 0;
}
```



5. Wild Pointer

An uninitialized pointer. It contains an unpredictable, garbage value and points to an arbitrary memory location. Dereferencing it is very risky.
C

```
#include <stdio.h>
int main() {
    int *ptr_wild; // Declares a pointer but doesn't initialize it. It's a wild pointer.
// Attempting to dereference a wild pointer is highly dangerous and can crash your program
    // or corrupt memory.
    // *ptr_wild = 100; // DO NOT UNCOMMENT THIS LINE IN PRODUCTION CODE
printf("Wild pointer value (uninitialized, unpredictable): %p\n", (void *)ptr_wild);
// To make it safe, initialize it:
    int x = 5;
    ptr_wild = &x; // Now it's a valid pointer to x
    printf("Initialized pointer value: %p\n", (void *)ptr_wild);
    printf("Value pointed to by initialized pointer: %d\n", *ptr_wild);
return 0;
}
```

6. Double Pointer (Pointer to Pointer)

A pointer that stores the address of another pointer.
C

```
#include <stdio.h>
int main() {
    int num = 77;
    int *ptr_num = &num;     // ptr_num stores the address of 'num'
    int **ptr_to_ptr = &ptr_num; // ptr_to_ptr stores the address of 'ptr_num'
printf("Value of num: %d\n", num);
    printf("Address of num: %p\n", &num);
    printf("Value of ptr_num (address of num): %p\n", ptr_num);
    printf("Address of ptr_num: %p\n", &ptr_num);
    printf("Value of ptr_to_ptr (address of ptr_num): %p\n", ptr_to_ptr);
    printf("Address of ptr_to_ptr: %p\n", &ptr_to_ptr);
// Dereferencing a double pointer:
    printf("\nValue pointed to by ptr_num: %d\n", *ptr_num); // *ptr_num gets the value of num
    printf("Value pointed to by ptr_to_ptr (indirectly num): %d\n", **ptr_to_ptr); // **ptr_to_ptr gets the
value of num
// Modifying num through double pointer
    **ptr_to_ptr = 88;
    printf("New value of num: %d\n", num);
return 0;
}
```

7. Function Pointer

A pointer that stores the memory address of a function. This allows you to call functions indirectly.
C

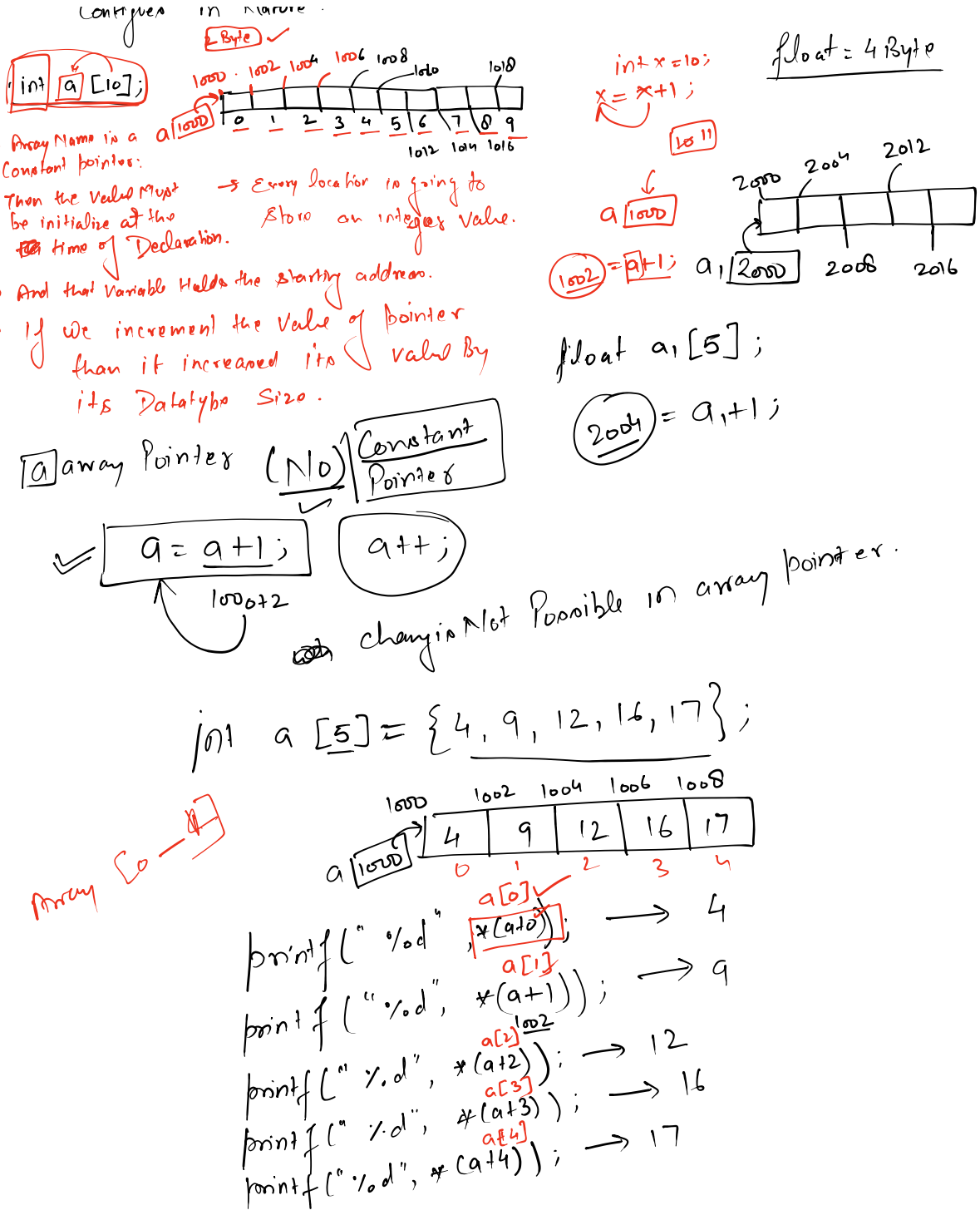
```
#include <stdio.h>
// A simple function
int add(int a, int b) {
    return a + b;
}
// Another function
int subtract(int a, int b) {
    return a - b;
}
int main() {
    // Declare a function pointer that points to a function
    // taking two ints and returning an int.

    int (*op_ptr)(int, int);


// Assign the address of the 'add' function to the function pointer
    op_ptr = &add; // '&' is optional but good practice for clarity
// Call the 'add' function using the function pointer
    int result_add = op_ptr(10, 5);
    printf("Result of addition: %d\n", result_add);
// Assign the address of the 'subtract' function
    op_ptr = subtract; // 'subtract' itself decays to its address
// Call the 'subtract' function using the same function pointer
    int result_subtract = op_ptr(10, 5);
    printf("Result of subtraction: %d\n", result_subtract);
// Function pointers in an array (for a simple menu system, for example)
    int (*operations[2])(int, int);
    operations[0] = add;
    operations[1] = subtract;
printf("Result from array [0] (add): %d\n", operations[0](20, 10));
    printf("Result from array [1] (subtract): %d\n", operations[1](20, 10));
return 0;
}
```

8. Array Pointer

A pointer that points to an entire array. Its type explicitly includes the size of the array it points to. This is different from a pointer to the first element of an array.
C

```
#include <stdio.h>
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
// Pointer to the first element of the array (most common)
    int *ptr_first_element = arr; // 'arr' decays to a pointer to its first element
    printf("Using pointer to first element: %d\n", *ptr_first_element);
    printf("Sizeof ptr_first_element: %zu bytes\n", sizeof(ptr_first_element)); // Size of the pointer itself
(e.g., 8 bytes on 64-bit)
// Pointer to the entire array
    // The syntax: 'int (*ptr_array)[5]' means ptr_array is a pointer to an array of 5 integers.
    int (*ptr_array)[5] = &arr; // Points to the whole array
printf("\nUsing pointer to the entire array:\n");
    printf("Value of the first element using ptr_array: %d\n", (*ptr_array)[0]);
    printf("Value of the third element using ptr_array: %d\n", (*ptr_array)[2]);
printf("Sizeof ptr_array: %zu bytes\n", sizeof(ptr_array)); // Size of the pointer itself
// You can also iterate through the array using ptr_array
    printf("Iterating through array using ptr_array:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", (*ptr_array)[i]);
    }
    printf("\n");
    return 0;
```

Array:- { Array is a datastructure which stores a homogeneous Data. and the array Name is a Constant pointer.
→ The memory allocated to array is




```c
/* Example of Function Pointer */
#include <stdio.h>
int add(int a, int b)
{
    return a + b;
}
int sub(int a, int b)
{
    return a - b;
}
int main()
{
    int (*ptr)(int, int);
    ptr = add;
    int add = ptr(12, 23);
    ptr = sub;
    int sub = ptr(12, 23);
    printf("%d \n", add);
    printf("%d \n", sub);
    return 0;
}
```

```c
//Constant Function Pointer

int (*const ptr)(int, int) = add;
```

In C programming, **storage classes** define the scope, lifetime, visibility, and memory location of variables and functions. Understanding them is crucial for effective memory management and writing efficient, well-structured code.
There are four main storage classes in C:

1. **auto (Automatic)**
2. **register**
3. **static**
4. **extern (External)**

Let's break down each one:

1. auto

- **Default for Local Variables:** auto is the default storage class for all local variables declared inside a function or a block. You usually don't need to explicitly use the auto keyword, as it's implied.
- **Scope:** Local (within the block or function where it's declared).
- **Lifetime:** Created when the block/function is entered, destroyed when the block/function is exited.
- **Memory Location:** Stack (RAM).
- **Default Initial Value:** Garbage value (undefined).

Example:

C

```
#include <stdio.h>
void myFunction() {
  int x; // 'x' is an auto variable by default
  auto int y; // Explicitly declared as auto, but same behavior
  printf("Initial value of x: %d\n", x); // Will be a garbage value
  printf("Initial value of y: %d\n", y); // Will be a garbage value
}
int main() {
  myFunction();
  // printf("%d", x); // Error: 'x' is out of scope here
  return 0;
}
```

2. register

- **Suggestion for CPU Registers:** The register keyword is a request to the compiler to store the variable in a CPU register instead of RAM. This is done to achieve faster access.
- **Compiler Discretion:** The compiler *might* store it in a register if one is available and it deems it beneficial. If not, it will store it in RAM (like an auto variable).
- **Scope:** Local (within the block or function where it's declared).
- **Lifetime:** Created when the block/function is entered, destroyed when the block/function is exited.
- **Memory Location:** CPU Register (if possible), otherwise Stack (RAM).
- **Default Initial Value:** Garbage value (undefined).
- **Restriction:** You cannot take the address of a register variable using the & operator because it might not have a memory address in RAM.

Example:

C

```
#include <stdio.h>
void myFunction() {
  register int counter; // Suggest to store in a register
  for (counter = 0; counter < 5; counter++) {
    printf("%d ", counter);
  }
  printf("\n");
}
int main() {
  myFunction();
  return 0;
}
```

3. static

- **Persistent Value:** static variables retain their value between multiple function calls.
- **Scope:**
  - **Local static:** If declared inside a function, its scope is local to that function, but its lifetime is for the entire program execution. It retains its value between calls.
  - **Global static (File Scope):** If declared outside any function (at the global level), its scope is limited to the file in which it is declared. It cannot be accessed from other files.
- **Lifetime:** Exists throughout the entire program execution.
- **Memory Location:** Data segment (RAM).
- **Default Initial Value:** Zero (0) for integral types, null for pointers.

Example of Local static:

C

```
#include <stdio.h>
void counterFunction() {
  static int count = 0; // 'count' retains its value between calls
  count++;
  printf("Count: %d\n", count);
}
int main() {
  counterFunction(); // Output: Count: 1
  counterFunction(); // Output: Count: 2
  counterFunction(); // Output: Count: 3
  return 0;
}
```

Example of Global static (in file1.c):

C

```
// file1.c
static int global_static_var = 10; // Visible only within file1.c
void print_static_var() {
  printf("From file1.c: %d\n", global_static_var);
}
```

You cannot access global_static_var directly from file2.c.

4. extern

- **Declaration, Not Definition:** The extern keyword is used to declare a variable or function that is *defined* elsewhere (either in the same file or another file). It tells the compiler that the variable/function exists but its memory allocation is handled in another compilation unit.
- **Sharing Global Variables:** It's primarily used to share global variables across multiple source files in a larger project.
- **Scope:** Global (across all files in the program).
- **Lifetime:** Exists throughout the entire program execution.
- **Memory Location:** Data segment (RAM) - where the actual definition is.
- **Default Initial Value:** Zero (0) (from where it's defined).

Example (assuming file1.c and file2.c):

file1.c (Definition of the global variable):

C

```
// file1.c
int shared_variable = 100; // Definition of the global variable
void print_shared_variable() {
  printf("From file1.c, shared_variable: %d\n", shared_variable);
}
```

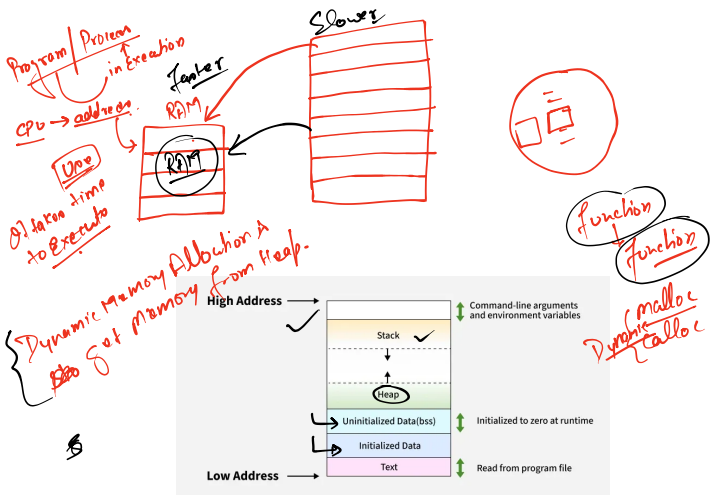
file2.c (Accessing the global variable from file1.c):

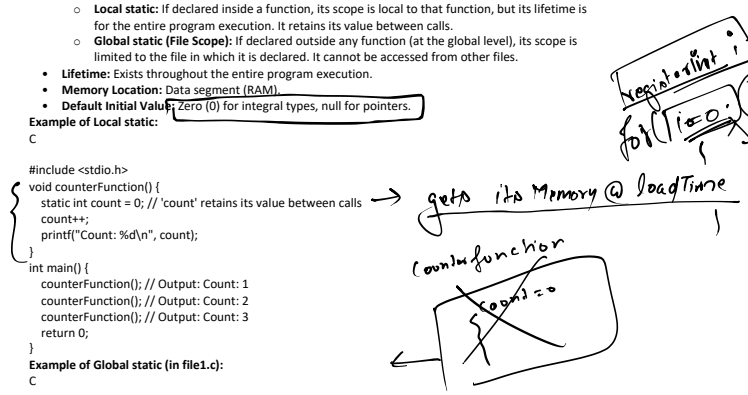
C

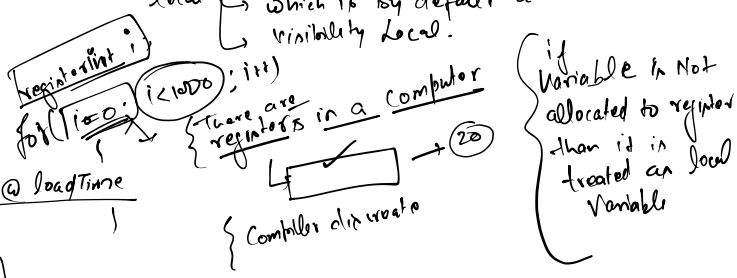
```
// file2.c
#include <stdio.h>
extern int shared_variable; // Declaration: 'shared_variable' is defined elsewhere
int main() {
  printf("From file2.c, shared_variable: %d\n", shared_variable);
  shared_variable = 200; // Can modify the shared variable
  printf("From file2.c (after modification), shared_variable: %d\n", shared_variable);
  return 0;
}
```

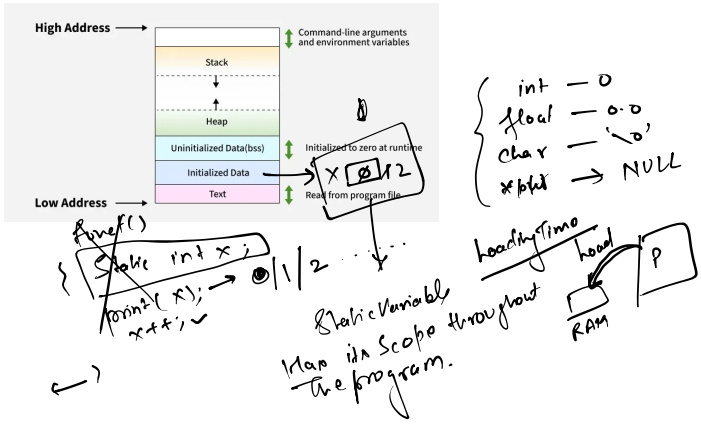
When you compile file1.c and file2.c together, the linker will resolve the reference to shared_variable.

Summary Table:









Storage Class	Scope	Lifetime	Memory Location	Default Initial Value	Usage
auto	Local (within block)	Until block/function exits	Stack	Garbage	Default for local variables
register	Local (within block)	Until block/function exits	CPU Register (or Stack)	Garbage	For frequently accessed variables (loop counters)
static	Local (within function) or File (global)	Throughout program execution	Data segment	Zero	Retains value between function calls, file-level scope for globals
extern	Global (across files)	Throughout program execution	Data segment	Zero	Declares a variable defined elsewhere

Understanding storage classes is fundamental to mastering C, as it directly impacts how your program manages memory and how different parts of your code interact with variables.