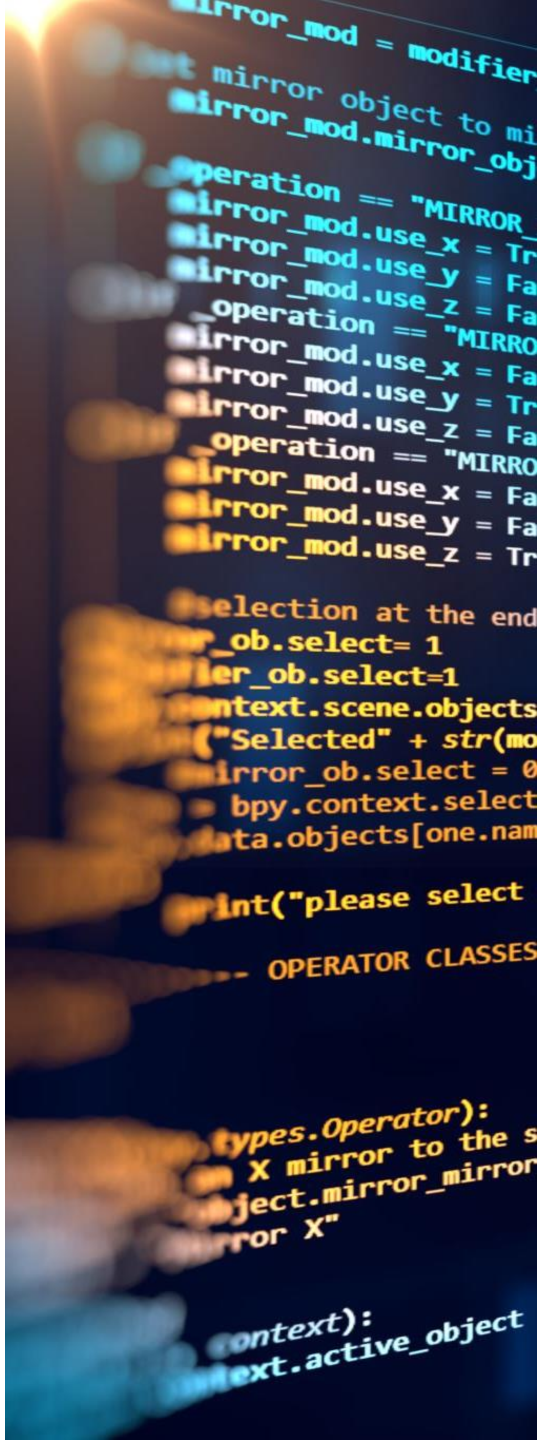
CS 1037
Fundamentals of Computer Science II

C Programming Features

Ahmed Ibrahim



Recap

- **Pointers**
  - Pointers hold memory addresses of variables.
  - Syntax: **data_type** *ptr_name;
  - Assigning address: int x; ptr = &x;
  - Dereferencing: Accessing value at the memory location pointed by the pointer using *ptr.
- **Pointer Operations**
  - Supports arithmetic operations like addition (ptr + k) and subtraction (ptr - k).
  - Increment (ptr++) and decrement (ptr--) adjust the pointer by the size of its data type.
- **Arrays & Pointers**
  - Arrays are collections of same-type data values.
  - Pointers provide efficient tools for array manipulation.
- **Important Notes**
  - Dereferencing uninitialized pointers can cause **runtime errors**.
  - The pointer size is the same for all pointer types, but the data type ensures correct dereferencing.

Question!

- What will be printed by the following program?

```c
#include <stdio.h>

int main() {
    int x = 5;
    int *ptr = &x;
    *ptr = *ptr + 10;
    printf("%d", x);
    return 0;
}
```

A) 5

B) 10

C) 15

D) 20

Question!

```c
#include <stdio.h>

int main() {
    int arr[] = {5, 10, 15};
    int *ptr = arr;
    *(ptr + 1) = 20;
    printf("%d %d", arr[1], *(ptr + 2));
    return 0;
}
```

- What will be the output of the following program?

A) 10 15

B) 20 20

C) 20 15

D) 15 20

Special Pointers

Null Pointers

- A **null pointer** is a special pointer value for not pointing anywhere. This means that a null pointer <u>does not point to any valid memory address</u>.
- C uses the pre-defined macro constant NULL to represent the **null pointer**; it has a value of 0.
  - For example, use NULL to declare and initialize a pointer:

    `int *ptr = NULL;` or equivalently `int *ptr = 0;`
- It is good programming practice to set NULL to a pointer if we don't have a target to point to. Then, we can check if a pointer equals NULL to decide if it is pointing somewhere.
- Example:

  if (ptr == NULL) { **statement block**; }

Case 1: Function Argument Validation

- When writing a function that accepts a pointer as an argument, you can use a NULL pointer to signal an invalid or uninitialized input. This allows the function to perform appropriate checks before proceeding with operations that could cause errors.

```c
void printArray(int *arr, int size) {
    if (arr == NULL) {
        printf("Invalid array pointer!\n");
        return;
    }
    // Print the array elements...
}
```

Typecasting in C

- Type casting refers to converting a variable from one data type to another.
- Two types of type casting:
  - Implicit typecasting (automatic):
    - Done by the compiler automatically.
    - Occurs when a smaller data type is assigned to a larger data type (e.g., int to float).
  - Explicit typecasting (manual):
    - Performed by the programmer using the cast operator (type).
    - Allows conversion between incompatible data types or narrowing data types (e.g., float to int).

Note that explicit casting requires caution because it can lead to data loss (e.g., converting from float to int).

```c
int main() {

  int x = 10;
  float y = x; // Implicit casting: int to float

  return 0;
}
```

Example of Implicit Type Casting

```c
int main() {

  float x = 5.5;

  // Explicit casting: float to int
  int y = (int)x;

  return 0;
}
```

Example of Explicit Type Casting

Generic Pointers

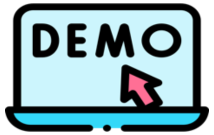
- A generic pointer is a pointer that has **void** as its type.
  - For example, void *ptr // declare ptr a generic pointer
- A generic pointer can point at a variable of any type but needs to be cased to a specific type when doing operations and dereferencing.

```c
int main() {
int a = 10;
  void *ptr = &a;

  // will print 10, (int*)ptr casts ptr to int type pointer
  printf("%d", *(int*)ptr);

  float f = 3.14;
  ptr = &f;
  printf("%f", *(float*)ptr); // will print 3.14

  return 0;
}
```

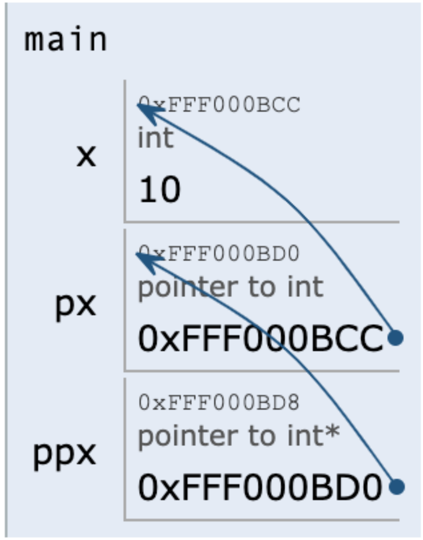
To print *float* value on display

Pointers to Pointers

- C allows a **pointer** to point to another **pointer**. To declare a pointer to a pointer, just add an asterisk (*) for each level of reference.




```c
int main() {

  int x=10;

  // pointer pointing to an int variable
  int *px;

  // pointer pointing to an int pointer
  int **ppx;

  // px pointing to x
  px=&x;

  // ppx pointing to px
  ppx=&px;

  // or *(*ppx) will print 10
  printf("%d\n", **ppx);

  return 0;
}
```



Stop & Think

Does it make sense to have pointers of three asterisks?

- In theory, it allows to have many levels of pointers to pointers. That is, many asterisks can be applied in front of a variable declaration.
  - Example: int a, *p1 = &a, **p2 = &p1, ***P3 = &p2;
- However, more levels of asterisks will make a program hard to understand, so usually at most two levels of asterisks is used in programming.

Question!

- What will be the output of the following program?

A) 10
B) 20
C) 30
D) 40

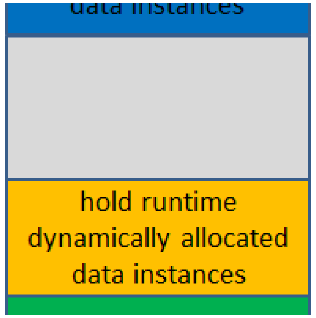
```c
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40};
    int *p = arr + 2;
    printf("%d", *p);
    return 0;
}
```

Memory Management

Memory Allocation

- Memory allocation refers to assigning a memory block to store a data value of a specific type.
- C supports three memory allocation methods:
  - **Static** memory allocation: Memory is allocated at **compile time** and remains **fixed** throughout the program execution.
  - **Automatic** memory allocation: Memory is allocated and dislocated automatically, typically for **local variables** within a **function**.
  - **Dynamic** memory allocation: Memory is allocated at **runtime** using functions like **malloc(), calloc(), realloc()**, and **free()** for flexible memory usage.

Dynamic Memory Allocation



- Dynamic memory allocation uses the **malloc()** function from the **stdlib** library.
- Memory allocated dynamically is stored in the **heap region** and is **NOT** released automatically when the calling function finishes.
- Dynamically allocated memory blocks can be **shared** across different functions.
- If a dynamically allocated memory block is no longer needed, it should be released using the **free()** function to return it to the heap's pool of available memory.
- For example,

Typecasting

```
int *p = (int*) malloc (sizeof(int))
```

  - allocates memory equal to the size of an integer (4 Bytes) in the **heap** and stores the address in the pointer p.
  - The malloc () function returns **void pointer** (**void***) to the allocated memory.

Case 2 of Null Pointers: Checking if Dynamic Memory Allocation Succeeded!

- When you allocate memory dynamically using malloc(), or realloc(), these functions return NULL if they **fail** to allocate memory.
- Checking for NULL allows you to handle such failures.

```
int *ptr = (int*) malloc(sizeof(int) * 10);
if (ptr == NULL) {
printf("Memory allocation failed!\n");
exit(1);}
```

Questions!

What is the correct way to free dynamically allocated memory?

```
int *p = (int*) malloc(5 * sizeof(int));
______;
```

A) realloc(p, 0);

B) free(p);

C) p = NULL;

D) free(*p);

Dynamic Memory Re-Allocation

- The **realloc()** function is used to **resize previously allocated memory blocks** in dynamic memory. It is particularly useful when you are working with <u>dynamic arrays</u> or <u>variable-size</u>d data where the required memory size might change during runtime.
- Syntax:

```
data_type *ptr = (data_type*) realloc(ptr, n);
```

- Here's what each component means:
  - data_type: The type of data stored in the memory block (e.g., int, float, etc.).
  - ptr: A pointer to the previously allocated memory block (which was allocated using **malloc**()).
  - n: The new size (in bytes) that you want to allocate.

- Example:

```
int *p = (int*) malloc (sizeof(int))
p = (int*) realloc(p, 5 * sizeof(int));  // Resize the memory to hold 5 integers
```

How **realloc**() Works

1. **Resizing**: It alters the size of the existing memory block. The block is expanded if the new size is larger than the original size. <u>If the new size is smaller, some memory is freed</u>.
2. **Memory Location**:
    1. **Expansion**: If enough adjacent memory is available, the memory block is expanded in place, and the same pointer (ptr) is returned.
    2. **Relocation**: If there isn't enough space in the current location, the function allocates a new memory block somewhere else in memory. It then copies the contents from the old memory block to the new location. The old block is freed automatically, and a pointer to the new block is returned.
    3. **Failure (NULL Return)**: If **realloc**() cannot resize the memory (e.g., **if there is no more space available to allocate**), it returns **NULL**, indicating **failure**. You should check for **NULL** to avoid memory leaks or undefined behavior.

Important Points

- If the pointer ptr is NULL, **realloc**() behaves like **malloc**() and allocates new memory of size n.
- Some memory is deallocated if the new size (n) is smaller than the original size.
- If the **realloc**() fails, it does not free the original memory block, so you must check if the returned pointer is NULL before using it.
  - This could happen if there is insufficient memory available to resize the block.

Question!

What will be the output of the following C program?

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

int main() {
    int *p = (int*) malloc(3 * sizeof(int));
    for (int i = 0; i < 3; i++) p[i] = i + 1;
    p = (int*) realloc(p, 5 * sizeof(int));
    for (int i = 3; i < 5; i++) p[i] = i + 1;
    for (int i = 0; i < 5; i++) printf("%d ", p[i]);
    free(p);
    return 0;
}
```

A) 1 2 3 4 5

B) 1 2 3 0 0

C) 1 2 3 4 0

D) Compilation Error

Memory leaking

- It is important to keep the address of a dynamically allocated memory block by a pointer.
- If the address is lost, then the memory block cannot be accessed, released, and reused. This situation is called **memory leaking**.

```c
#include <stdio.h>

int main() {

 int *p = (int*) malloc (sizeof(int));

 *p = 3;

 printf("%d", *p);

 p = NULL; // this causes a memory leaking

  return 0;
}
```


Thank you

References

- Data Structures Using C, second edition, by Reema Thareja, Oxford University Press, 2014.