C for Python Programmer

EECS 678 Staff
Yuying Li

Compilers vs. Interpreters

Python: Uses an **interpreter** to execute code directly.

No separate compilation step, but compiled C code can run faster.

C: Uses a **compiler** to translate code into machine-native executable.

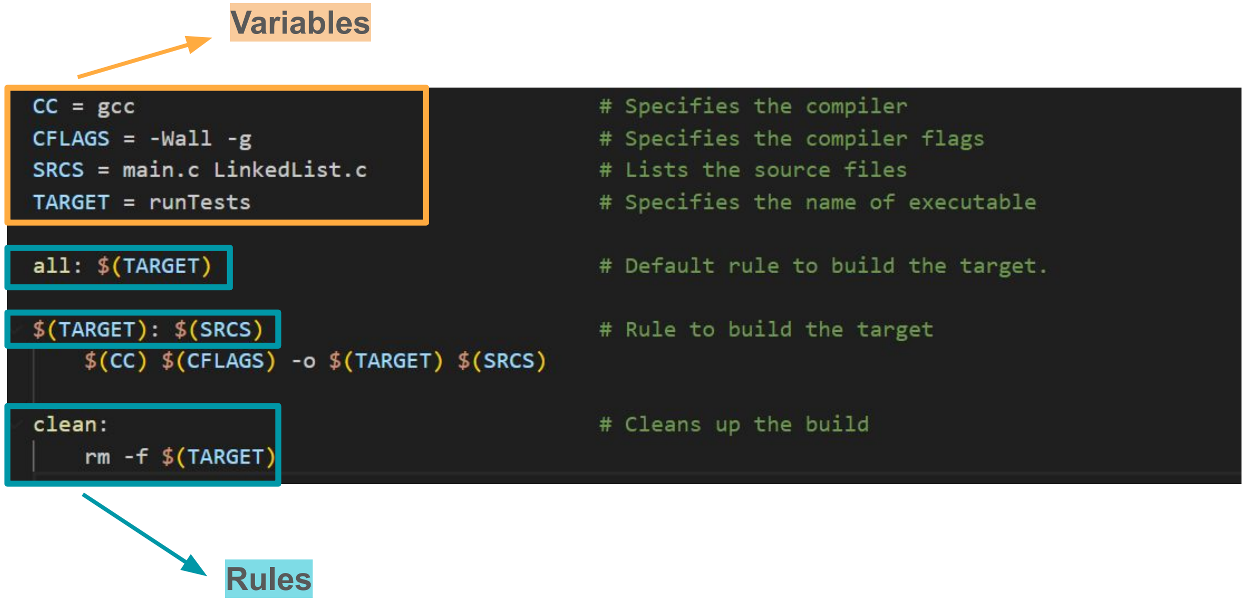
Execution is a two-step process:

- ➔ Compile the program (*gcc my_program.c*)
- ➔ Run the executable (*./a.out*)

Makefile in C Projects

What is a Makefile?

- **Purpose:** Automates the build process for C projects, managing compilation and linking.
- **Why Use It:** Simplifies handling multiple source files and dependencies, making projects easier to manage.



```
CC = gcc                                    # Specifies the compiler
CFLAGS = -Wall -g                           # Specifies the compiler flags
SRCS = main.c LinkedList.c                  # Lists the source files
TARGET = runTests                           # Specifies the name of executable

all: $(TARGET)                              # Default rule to build the target.

$(TARGET): $(SRCS)                          # Rule to build the target
	$(CC) $(CFLAGS) -o $(TARGET) $(SRCS)

clean:                                      # Cleans up the build
	rm -f $(TARGET)
```

Header Files in C

C: Header files in C are used to declare the interfaces of functions, macros, and data types, allowing code to be shared across multiple source files.

- ➔ **Content:**
  - Declares functions so they can be used across different .c files.
  - Defines macros that can be reused.
  - Declares structures, enums, and typedefs.
- ➔ **Include:**
  - *#include <header.h>:* Used for system libraries.
  - *#include "header.h":* Used for user-defined headers.
- ➔ **Benefit:** Enhances code organization and reusability.

Variable Declarations

Python: Variables are dynamically typed and do not require declarations.

C: Requires **explicit** variable declarations with type specification (e.g., double x;).

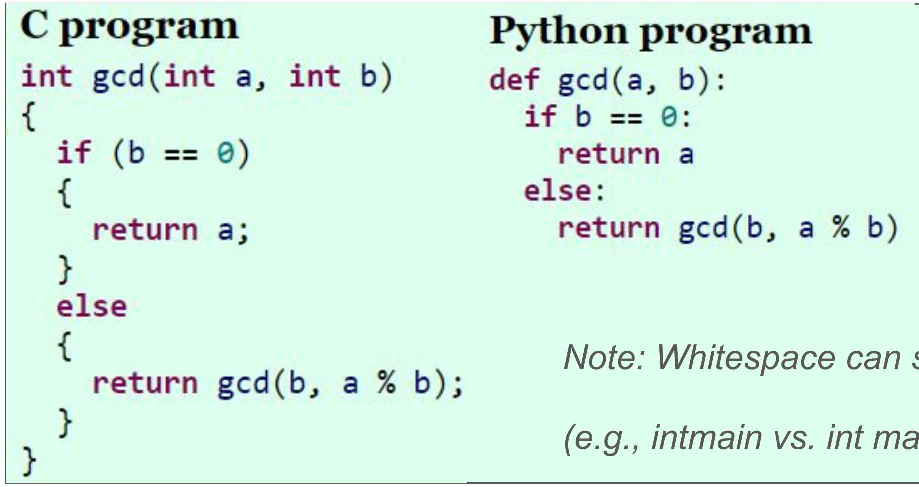
➔ Variables must be declared before use.
➔ Compiler checks for undeclared variables and catches misspellings.
➔ If the assigned value type differs from the declared type, the compiler will generate an error.

int	integer
char	character
double	double-precision floating-point number
_Bool	A boolean type that can hold true or false (typically 0 or 1)

Whitespace

Python: Uses indentation and line breaks to define blocks and separate statements.

C: Ignores most whitespace. Blocks are defined using “{ }” and statements are terminated with “;”.



C program

```c
int gcd(int a, int b)
{
  if (b == 0)
  {
    return a;
  }
  else
  {
    return gcd(b, a % b);
  }
}
```

Python program

```python
def gcd(a, b):
  if b == 0:
    return a
  else:
    return gcd(b, a % b)
```

Note: Whitespace can still affect interpretation in certain cases.

(e.g., intmain vs. int main)

The printf() Function

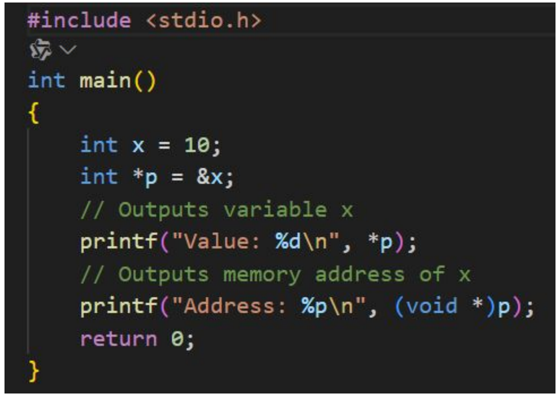
C: Uses *printf()* to display output, similar to Python's *print*.

- ➔ **Format Strings:** Use “%” followed by a character to format output (e.g., “%d” for integers, “%f” for floats).
- ➔ **Escape Characters:** Use “\n” for newline, “\t” for tab, etc.
- ➔ **Example:** printf("# solns: %d\n", num_sol); prints formatted output.

Pointers

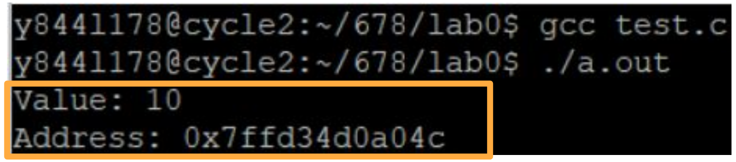
C: Pointers store memory addresses, allowing direct access to variables and memory.

- ➔ **Declaration:** *int* p;* declares a pointer to an integer.
- ➔ **Assignment:** *p = &x;* stores the address of x in p.
- ➔ **Dereferencing:** **p* accesses the value at the address stored in p
- ➔ **Benefits:** Enables dynamic memory allocation and efficient array handling.



```c
#include <stdio.h>

int main()
{
    int x = 10;
    int *p = &x;
    // Outputs variable x
    printf("Value: %d\n", *p);
    // Outputs memory address of x
    printf("Address: %p\n", (void *)p);
    return 0;
}
```



```
y8441178@cycle2:~/678/lab0$ gcc test.c
y8441178@cycle2:~/678/lab0$ ./a.out
Value: 10
Address: 0x7ffd34d0a04c
```

Note: Be careful with memory management! Incorrect use of pointers can lead to memory leaks, segmentation faults, or other issues.

Functions

C: All executable code must be inside functions, unlike Python where top-level code can exist outside function.

Functions cannot be nested. (e.g., you cannot declare a function inside another function).

- ➔ **Main Function:** The entry point of every C program is the main function, which typically returns int.
- ➔ **Custom Functions:** Define specific tasks within a program.
  - ◆ **Function Types:** Every function has a return type that must be specified (e.g., int, double, void).
  - ◆ **Return Type:** Specifies what type of value the function returns. Use void if the function doesn't return a value.

Function Calls

Pass by Value: Function receives a copy of x and y. Original values are unchanged.

➔ **Syntax:** *result = add(x, y);*

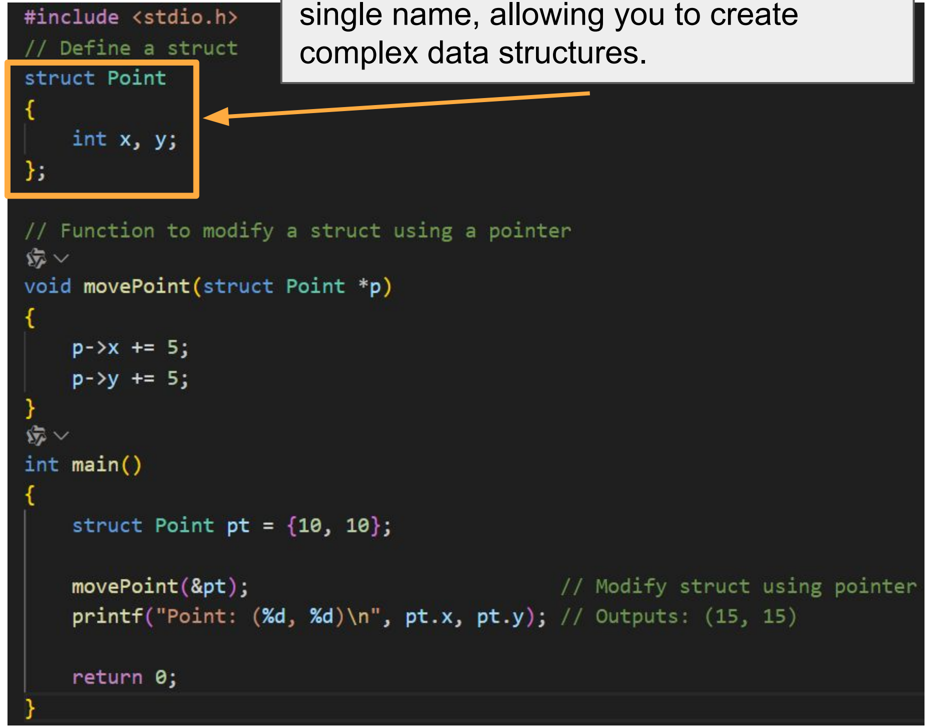
Pass by Reference (Pointers):

➔ **Syntax:** *modify(&x);*
➔ **Effect:** Function receives the address of x. Changes affect the original x.

Accessing Struct Members:

➔ **Dot (.) Operator:** Used with struct variables.
  ◆ Example: *variable.func*.
➔ **Arrow (->) Operator:** Used with pointers to structs.
  ◆ Example: *pointer->func*.

Structs (short for structures) group different data types together under a single name, allowing you to create complex data structures.



```c
#include <stdio.h>
// Define a struct
struct Point
{
    int x, y;
};

// Function to modify a struct using a pointer
void movePoint(struct Point *p)
{
    p->x += 5;
    p->y += 5;
}

int main()
{
    struct Point pt = {10, 10};

    movePoint(&pt);                              // Modify struct using pointer
    printf("Point: (%d, %d)\n", pt.x, pt.y); // Outputs: (15, 15)

    return 0;
}
```

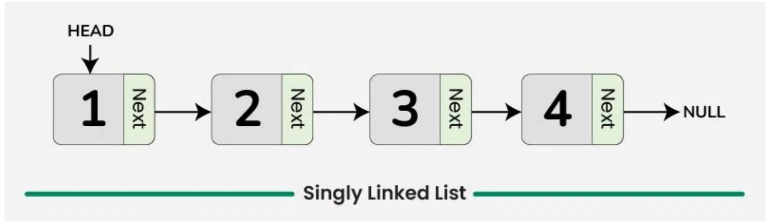
Linked List

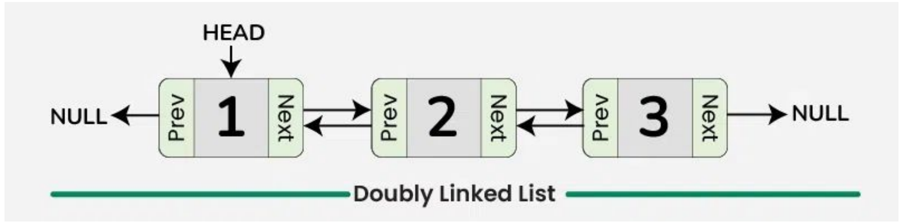
Definition: A linked list is a dynamic data structure consisting of nodes.

- ➔ **Data:** The actual value (e.g., an integer).
- ➔ **Pointer:** A reference (pointer) to the next node in the sequence.

How it Works:

- ➔ **Head Node:** The starting point of the list.
- ➔ **Traversal:** Move from one node to the next using the pointers until you reach a node where the next pointer is NULL (the end of the list).
- ➔ **Insertion/Deletion:** Nodes can be easily inserted or removed without reallocating or reorganizing the entire structure.
- ➔ **Dynamic Size:** Unlike arrays, linked lists can grow and shrink in size dynamically.

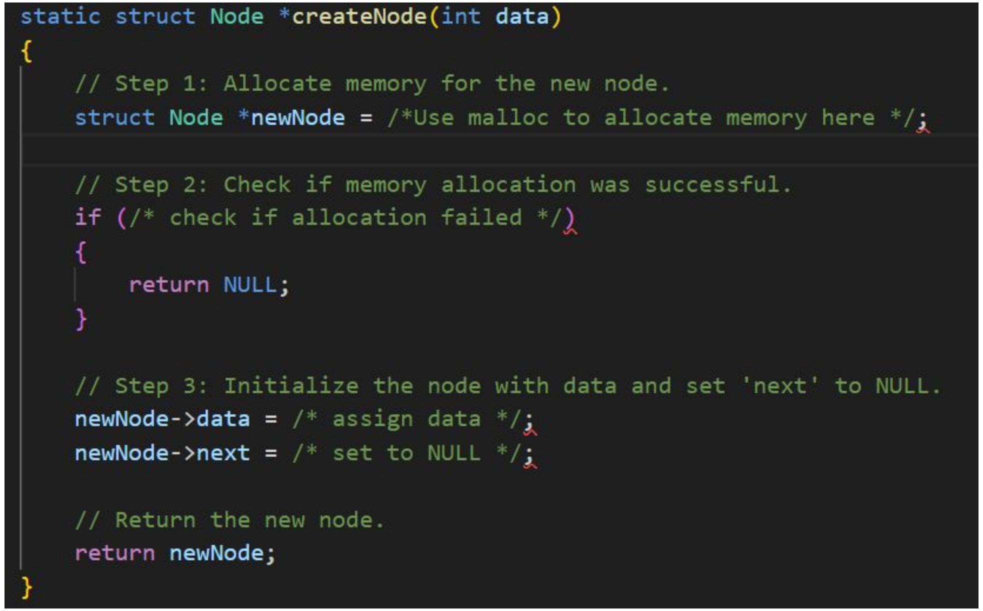




LinkedList.h

Functions:

- ➔ **createNode:** Create a new node for the linked list with a given integer value.
- ➔ **insertAtEnd:** Add a new node with given data to the end of the linked list.
- ➔ **printList:** Traverse and print all the elements in the linked list.
- ➔ **deleteList:** Free all nodes in the linked list to prevent memory leaks.



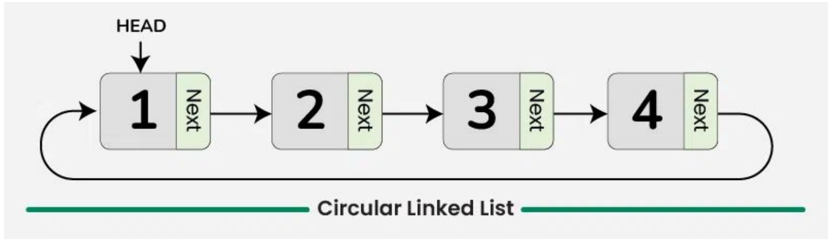
```c
static struct Node *createNode(int data)
{
    // Step 1: Allocate memory for the new node.
    struct Node *newNode = /*Use malloc to allocate memory here */;

    // Step 2: Check if memory allocation was successful.
    if (/* check if allocation failed */)
    {
        return NULL;
    }

    // Step 3: Initialize the node with data and set 'next' to NULL.
    newNode->data = /* assign data */;
    newNode->next = /* set to NULL */;

    // Return the new node.
    return newNode;
}
```

ListQuestions.h



Task 1: Detecting Cycles in a Linked List

- ➔ Problem: If a node's next pointer eventually points back to a previous node, creating an infinite loop, a cycle exists.
- ➔ Approach: Two-Pointer Technique (Fast & Slow): Use two pointers, one moving faster than the other. If they meet, a cycle is detected.

Task 2: Merging Two Sorted Linked Lists

- ➔ Problem: Merge two sorted linked lists into one sorted list without creating new nodes.
- ➔ Approach: 1) Compare nodes from both lists and link them in order. 2) Continue merging by advancing pointers in the lists.

Outcome: Return the head of the newly merged list.

How to Test Your Code

Running the Tests:

- ➔ Compile the Code: Run make all in the terminal.
- ➔ Execute the Tests: Run ./runTests.

Check Results:

- ➔ If your implementation is correct, you should see Total Score: 100/100.
- ➔ This indicates that all test cases passed successfully.

Testing Process:

We have prepared 10 test cases located in the Tests folder, each test case is designed to check different aspects of your linked list implementation, you can review the tests individually to understand what each one is testing.

Helpful Resources

C programming primer (for Python programmers):

https://learnxinyminutes.com/docs/c/

https://www.cs.toronto.edu/~patitsas/cs190/c_for_python.html

Linked List in C:

https://www.geeksforgeeks.org/linked-list-in-c/

More Details of C:

https://www.geeksforgeeks.org/c-programming-language/?ref=shm

Any Questions?