## Assignment\_20-08-2024

Author: ThanhTH10

Date: 20/08/2024

1. Implementation of memcpy and strcmp with void pointers

[https://elixir.bootlin.com/linux/v6.10/source/lib/string.c](https://elixir.bootlin.com/linux/v6.10/source/lib/string.c" \t "https://lms.vectorinstitute.in/mod/assign/_blank)

#include <stdio.h>

// Implementation of memcpy

void \*memcpy(void \*dest, const void \*src, int n)

{

    char \*dest\_ptr = (char \*)dest;

    const char \*src\_ptr = (const char \*)src;

    for (int i = 0; i < n; i++)

    {

        dest\_ptr[i] = src\_ptr[i];

    }

    return dest;

}

// Implementation of strcmp

int strcmp(const void \*s1, const void \*s2)

{

    const char \*str1 = (const char \*)s1;

    const char \*str2 = (const char \*)s2;

    while (\*str1 != '\0' && \*str2 != '\0')

    {

        if (\*str1 < \*str2)

            return -1;

        else if (\*str1 > \*str2)

            return 1;

        str1++;

        str2++;

    }

    if (\*str1 == '\0' && \*str2 != '\0')

        return -1;

    else if (\*str1 != '\0' && \*str2 == '\0')

        return 1;

    else

        return 0;

}

int main()

{

    char src[] = "Hello, World!";

    char dest[20];

    // Test memcpy

    memcpy(dest, src, sizeof(src));

    printf("Copied string: %s\n", dest);

    // Test strcmp

    char str1[] = "Hello";

    char str2[] = "World";

    int result = strcmp(str1, str2);

    printf("Comparison result: %d\n", result); // output -1

    return 0;

}

Output:



1. what is dynamic initalization in C++, how it is possible in C++ during runtime

Dynamic initialization in C++ refers to the process of initializing variables during runtime, as opposed to compile-time initialization. This is particularly useful when the value of a variable is not known until the program executes.

Dynamic initialization is possible in C++ through various mechanisms, such as:

* **Constructor Initialization**: Objects can be dynamically initialized when their constructors are called, which might include reading data from files, user input, or performing calculations.
* **Dynamic Memory Allocation**: Using dynamic memory allocation (new and delete), you can create and initialize objects or variables at runtime.

#include <iostream>

class Example {

public:

    int x;

    Example(int val) : x(val) {  // Constructor initializing dynamically

        std::cout << "Initialized x to " << x << std::endl;

    }

};

int main() {

    int userInput;

    std::cout << "Enter a value: ";

    std::cin >> userInput;

    Example obj(userInput);  // Object is dynamically initialized with runtime input

    return 0;

}

1. Implement stack data structure by using template class.

#include <iostream>

using namespace std;

template <class T>

class Stack

{

private:

    int top;

    int capacity;

    T \*arr;

public:

    Stack(int size = 10) : top(-1), capacity(size), arr(new T[capacity]) {}

    ~Stack() { delete[] arr; }

    void push(T st);

    T pop();

    T topElement();

    bool isEmpty();

    bool isFull();

};

template <class T>

void Stack<T>::push(T st)

{

    if (isFull())

    {

        T \*newArr = new T[capacity \* 2];

        if (!newArr)

        {

            throw runtime\_error("Memory allocation failed");

        }

        for (int i = 0; i < capacity; i++)

        {

            newArr[i] = arr[i];

        }

        delete[] arr;

        arr = newArr;

        capacity \*= 2;

    }

    arr[++top] = st;

}

template <class T>

T Stack<T>::pop()

{

    if (isEmpty())

    {

        throw runtime\_error("Stack is empty");

    }

    return arr[top--];

}

template <class T>

T Stack<T>::topElement()

{

    if (isEmpty())

    {

        throw runtime\_error("Stack is empty");

    }

    return arr[top];

}

template <class T>

bool Stack<T>::isEmpty()

{

    return top == -1;

}

template <class T>

bool Stack<T>::isFull()

{

    return top == capacity - 1;

}

int main()

{

    try

    {

        Stack<int> intStack;

        Stack<double> doubleStack;

        Stack<char> charStack;

        // Push some elements onto the stacks

        intStack.push(1);

        intStack.push(2);

        intStack.push(3);

        doubleStack.push(3.14);

        doubleStack.push(2.71);

        charStack.push('a');

        charStack.push('b');

        charStack.push('c');

        // Pop elements from the stacks

        cout << "Popped from intStack: " << intStack.pop() << endl;

        cout << "Popped from doubleStack: " << doubleStack.pop() << endl;

        cout << "Popped from charStack: " << charStack.pop() << endl;

    }

    catch (const runtime\_error &e)

    {

        cerr << "Error: " << e.what() << endl;

        return 1;

    }

    return 0;

}

Output:



4. C/C++ files vs Database software tools usage

|  |  |  |
| --- | --- | --- |
| **Feature** | **C/C++ Files** | **Database Software Tools** |
| **Data Storage Method** | Directly stores data in text or binary files | Stores data in tables with structured queries (SQL/NoSQL) |
| **Complexity** | Simple, direct file I/O (Input/Output) | More complex, requires setup and management of DBMS |
| **Data Size** | Suitable for small or moderate datasets | Designed for large-scale datasets |
| **Data Integrity** | No built-in data integrity checks | Built-in mechanisms for data integrity (ACID properties) |
| **Concurrency** | Limited support, prone to conflicts | Strong concurrency handling, supports multiple users |
| **Security** | Requires custom implementation for encryption and access control | Built-in security features such as authentication and access control |
| **Efficiency** | Efficient for simple, small-scale applications | Optimized for complex queries and large datasets |
| **Scalability** | Poor scalability for large datasets and multi-user environments | Highly scalable, suited for growing datasets and concurrent users |
| **Querying Data** | Manual parsing and handling | Advanced querying capabilities (SQL/NoSQL languages) |
| **Setup & Maintenance** | Minimal setup, no external software needed | Requires installation, configuration, and maintenance of a DBMS |
| **Use Case** | Best for small applications with simple data storage needs | Best for applications that require complex data management and multi-user support |
| **When to use** | Small and simple data storage requirements.  Minimal data manipulation and no need for advanced queries.  Single-user applications or lightweight systems. | Applications with large datasets.  Need for data integrity, security, and multi-user access.  Complex data relationships and querying requirements. |