# Ex1

Convert below JAVA program to C program

# BEGIN StudentApp.java

package Baitapbuoi8\_bai4;

import java.util.Scanner;

public class StudentApp {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter number of students: ");

int n = scanner.nextInt();

scanner.nextLine(); // Consume the left new line character left in the buff

Student [] students = new Student[n];

for (int i = 0; i < students.length; i++) {

students[i] = new Student();

System.out.print("Enter student name: ");

students[i].name = scanner.nextLine();

System.out.print("Enter student age: ");

students[i].age = scanner.nextInt();

scanner.nextLine(); // Consume the left new line character left in the buffer

}

for (int i = 0; i < students.length; i++) {

System.out.println("Student name: " + students[i].name + ", age: " + students[i].age);

}

scanner.close();

}

}

# END StudentApp.java

# BEGIN Student.java

package Baitapbuoi8\_bai4;

public class Student {

String name;

int age;

}

# END Student.java

**Ans1:**

To convert the given Java program into a C program, we need to use the C programming language and its standard input/output functions instead of Java's Scanner class. C does not have built-in support for classes like Java, so we'll use a struct to represent the student information.

Below is the equivalent C program:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define the structure for student

struct Student {

    char name[100];

    int age;

};

int main() {

    int n;

    printf("Enter number of students: ");

    scanf("%d", &n);

    getchar(); // Consume the left new line character left in the buffer

    struct Student \*students = (struct Student\*)malloc(n \* sizeof(struct Student));

    if (students == NULL) {

        printf("Memory allocation failed.\n");

        return 1;

    }

    for (int i = 0; i < n; i++) {

        printf("Enter student name: ");

        fgets(students[i].name, sizeof(students[i].name), stdin);

        students[i].name[strcspn(students[i].name, "\n")] = '\0'; // Remove the trailing newline character

        printf("Enter student age: ");

        scanf("%d", &students[i].age);

        getchar(); // Consume the left new line character left in the buffer

    }

    for (int i = 0; i < n; i++) {

        printf("Student name: %s, age: %d\n", students[i].name, students[i].age);

    }

    // Free the allocated memory

    free(students);

    return 0;

}

In this C program, we use struct Student to represent student information, and malloc to dynamically allocate memory for an array of students. We also use fgets to read student names and scanf to read student ages. After using the allocated memory, we free it using free to prevent memory leaks.

Why don't you code as below in your code snippet ?

    for (int i = 0; i < n; i++) {

        printf("Enter student name: ");

        scanf("%s", students[i].name);

        printf("Enter student age: ");

        scanf("%d", &students[i].age);

    }

Your suggestion to use **scanf** for reading student names directly into **students[i].name** would indeed work in many cases, and it can make the code more concise. However, using **scanf** for reading strings has some potential pitfalls and limitations that should be considered.

The main issue with using **scanf("%s", students[i].name)** directly is that it reads a string up to the first whitespace character, which means it will not work correctly if the student's name contains spaces. For example, if the user enters "John Doe" as the name, **scanf("%s", students[i].name)** would only read "John" into **students[i].name** and leave "Doe" in the input buffer. The next **scanf** call would then encounter "Doe" and may not behave as expected.

To handle names with spaces correctly, the **fgets** function is used in the provided code. **fgets** reads the entire line of input, including spaces, and stops when it encounters a newline character or reaches the specified limit.

Additionally, using **fgets** with **sizeof(students[i].name)** ensures that the input does not exceed the size of the **name** buffer, preventing potential buffer overflows and memory corruption.

In contrast, **scanf("%s", students[i].name)** can be problematic if the user enters a name longer than the buffer size, leading to undefined behavior and security vulnerabilities.

In summary, using **fgets** for reading strings with spaces is generally safer and more robust than using **scanf("%s", students[i].name)**, especially when the input data can be uncontrolled or when input validation is required.

Yes, it is possible to move the **malloc()** function inside the loop. However, doing so is not recommended in most cases, as it will unnecessarily allocate and deallocate memory repeatedly for each iteration of the loop. This can lead to inefficient memory usage and potential performance issues.

It is generally more efficient to allocate memory once before the loop and then use that allocated memory throughout the loop. As shown in the updated code below, the **malloc()** function is called once before the loop to allocate memory for the array of students. The memory is later freed after the loop is complete, ensuring proper memory management.

# Ex2

here's the modified C program with the struct Student changed to use a pointer for the name field:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define the structure for student

struct Student {

    char\* name;

    int age;

};

int main() {

    int n;

    printf("Enter number of students: ");

    scanf("%d", &n);

    getchar(); // Consume the left new line character left in the buffer

    // Allocate memory for the array of students

    struct Student \*students = (struct Student\*)malloc(n \* sizeof(struct Student));

    if (students == NULL) {

        printf("Memory allocation failed.\n");

        return 1;

    }

    // Allocate memory for each student's name

    for (int i = 0; i < n; i++) {

        students[i].name = (char\*)malloc(100 \* sizeof(char));

        if (students[i].name == NULL) {

            printf("Memory allocation failed for student name.\n");

            // Free the previously allocated memory before returning

            for (int j = 0; j < i; j++) {

                free(students[j].name);

            }

            free(students);

            return 1;

        }

        printf("Enter student name: ");

        fgets(students[i].name, 100, stdin);

        students[i].name[strcspn(students[i].name, "\n")] = '\0'; // Remove the trailing newline character

        printf("Enter student age: ");

        scanf("%d", &students[i].age);

        getchar(); // Consume the left new line character left in the buffer

    }

    for (int i = 0; i < n; i++) {

        printf("Student name: %s, age: %d\n", students[i].name, students[i].age);

    }

    // Free the allocated memory for student names

    for (int i = 0; i < n; i++) {

        free(students[i].name);

    }

    // Free the allocated memory for the array of students

    free(students);

    return 0;

}

In this modified version, the name field in struct Student is now declared as a pointer to char\*, and memory is dynamically allocated for each student's name using malloc. Additionally, after using dynamic memory, it is essential to free the allocated memory before the program exits to prevent memory leaks, as shown in the modified code above.

To convert the given Java program to C program using malloc inside the loop, we need to handle memory allocation and deallocation manually, **unlike Java where memory management is handled automatically**.

In the C program, we manually allocate memory for the array of students using malloc. Additionally, we also allocate memory for each student's name individually using malloc inside the loop. After processing, we free the allocated memory using free to prevent memory leaks. **This manual memory management is a key difference between C and Java**.

In your above program, I found you don't free students[i].name. But then, I found you add the code to free students[i].name in the later program. Why ?

// Free the allocated memory

for (int i = 0; i < n; i++) {

free(students[i].name);

}

Reason: use pointer as char in struct Student

    // Allocate memory for each student's name

    for (int i = 0; i < n; i++) {

        students[i].name = (char\*)malloc(100 \* sizeof(char));

        if (students[i].name == NULL) {

            printf("Memory allocation failed for student name.\n");

            free(students);

            return 1;

        }

    }

// Define the structure for student

struct Student {

    char\* name;

    int age;

};

# Ex1

Convert below JAVA program to C program

# BEGIN StudentApp.java

package Baitapbuoi8\_bai3;

public class StudentApp {

public static void main(String[] args) {

Student student1 = new Student();

student1.name = "Nguyen Duy Thai";

student1.age = 35;

Student student2 = new Student();

student2.name = "Tran Van Tuan";

student2.age = 28;

System.out.println("Student 1: " + student1.name + ", age: " + student1.age);

System.out.println("Student 2: " + student2.name + ", age: " + student2.age);

}

}

# END StudentApp.java

# BEGIN Student.java

package Baitapbuoi8\_bai3;

public class Student {

String name;

int age;

}

# END Student.java

ChatGPT

To convert the given Java program to a C program, you need to create a structure to represent the student information and use functions to simulate the behavior of the Java program. Here's the equivalent C program:

c

Copy code

#include <stdio.h>

#include <string.h>

// Define the structure for student

struct Student {

char name[100];

int age;

};

// Function to initialize student information

void initializeStudent(struct Student \*student, const char \*name, int age) {

strcpy(student->name, name);

student->age = age;

}

int main() {

struct Student student1, student2;

// Initialize student information

initializeStudent(&student1, "Nguyen Duy Thai", 35);

initializeStudent(&student2, "Tran Van Tuan", 28);

// Print student information

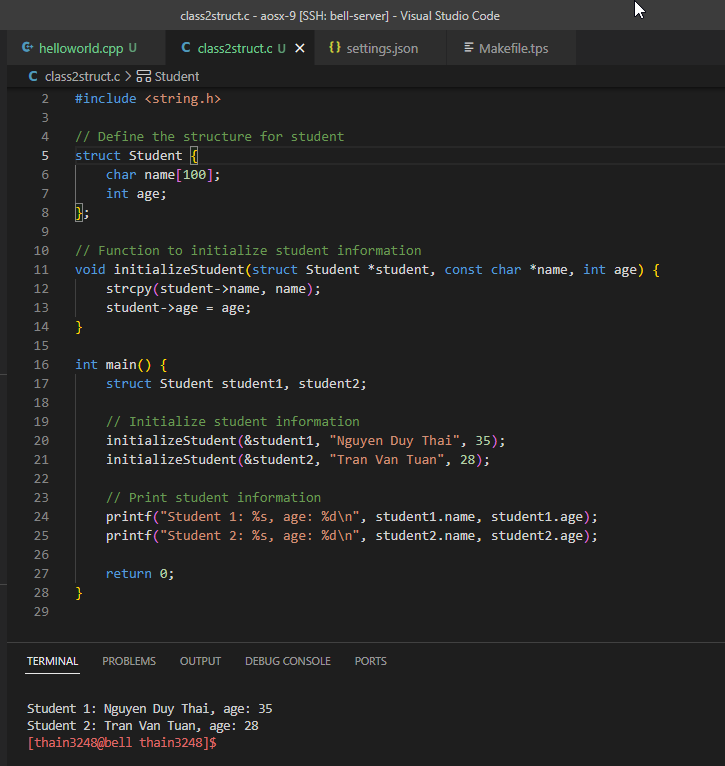
printf("Student 1: %s, age: %d\n", student1.name, student1.age);

printf("Student 2: %s, age: %d\n", student2.name, student2.age);

return 0;

}

In this C program, we use a struct Student to represent student information, which is similar to the Java class Student. The function initializeStudent is used to set the name and age of each student. Then, we print the student information using printf, just like in the Java program.



No, the provided C program does not leak memory. Memory leaks occur when dynamically allocated memory is not properly deallocated (freed) before the program exits. In the given program, there is no dynamic memory allocation using functions like **malloc**, **calloc**, or **realloc**. The **struct Student** objects **student1** and **student2** are declared as automatic variables within the **main** function, and they do not require manual deallocation.

The **student1** and **student2** objects **are allocated on the stack, and their memory will be automatically released** when the **main** function returns, which happens when the program exits. Therefore, there is no need to manually free any memory, and the program does not leak memory.

    struct Student student1, student2;

<https://stackoverflow.com/questions/5060641/does-a-string-created-with-strcpy-need-to-be-freed#>

strcpy itself doesn't allocate memory for the destination string so, no, it doesn't have to be freed.

Of course, if something else had allocated memory for it, then, yes, that memory should be freed eventually but that has nothing to do with strcpy.

That previous statement seems to be the case since your definition is an array of character pointers rather than an array of characters:

char\* buffer[LEN];

and that will almost certainly be done with:

buffer[n] = malloc (length);

# Ex2

**C struct không chứa method (function) như JAVA class**

Convert below JAVA program to C program. Replace class by struct in C.

// BEGIN Rectangular.java

package Baitapbuoi8\_bai2;

import java.util.Scanner;

public class Rectangular {

float width;

float length;

float area;

float perimeter;

Rectangular(float width, float length) {

this.width = width;

this.length = length;

}

Rectangular() {};

public void enterInfo() {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter width: ");

this.width = scanner.nextFloat();

System.out.print("Enter length: ");

this.length = scanner.nextFloat();

}

public void printInfo() {

System.out.println("width=" + this.width);

System.out.println("length=" + this.length);

}

public void calcArea() {

this.area = this.width \* this.length;

System.out.println("Area of rectA: " + this.area);

}

public void calcPerimeter() {

this.perimeter = (this.width + this.length) \* 2;

System.out.println("Perimeter of rectA: " + this.perimeter);

}

}

// END Rectangular.java

// BEGIN RectangularApp.java

package Baitapbuoi8\_bai2;

public class RectangularApp {

public static void main(String[] args) {

Rectangular rectA = new Rectangular();

rectA.enterInfo();

rectA.printInfo();

rectA.calcArea();

rectA.calcPerimeter();

}

}

// END RectangularApp.java

ChatGPT

Here's the equivalent C program using structs and functions:

#include <stdio.h>

// Define the struct for rectangular

struct Rectangular {

float width;

float length;

float area;

float perimeter;

};

// Function to enter information for rectangular

void enterInfo(struct Rectangular \*rect) {

printf("Enter width: ");

scanf("%f", &rect->width);

printf("Enter length: ");

scanf("%f", &rect->length);

}

// Function to print information for rectangular

void printInfo(struct Rectangular rect) {

printf("width=%.2f\n", rect.width);

printf("length=%.2f\n", rect.length);

}

// Function to calculate the area of rectangular

void calcArea(struct Rectangular \*rect) {

rect->area = rect->width \* rect->length;

printf("Area of rectA: %.2f\n", rect->area);

}

// Function to calculate the perimeter of rectangular

void calcPerimeter(struct Rectangular \*rect) {

rect->perimeter = 2 \* (rect->width + rect->length);

printf("Perimeter of rectA: %.2f\n", rect->perimeter);

}

int main() {

struct Rectangular rectA;

enterInfo(&rectA);

printInfo(rectA);

calcArea(&rectA);

calcPerimeter(&rectA);

return 0;

}

In the C program, we have defined a struct Rectangular with the same properties as the Java class. We then created functions enterInfo, printInfo, calcArea, and calcPerimeter to perform the same functionalities as the methods in the Java program. The main function is used to create a struct object rectA, call the functions to enter information, print information, calculate area, and calculate perimeter of the rectangular, similar to the Java program.