C PROGRAMMING
Lecture 12

1st semester 2023 - 2024

Inline Function

In C programming, the inline keyword is used as a suggestion to the compiler to perform inlining of a function.

Inlining is a compiler optimization technique where the code of a function is inserted directly at the call site, rather than being invoked through a function call.

Most of the Inline functions are used for small computations. They are not suitable for large computing.

An inline function is similar to a normal function. The only difference is that we place a keyword inline before the function name.

```
inline function_name (){
   //function definition }
```

Inline Function

Function Inlining:

When a function is declared as inline, the compiler may choose to replace function calls with the actual code of the function at the call site.

This can result in faster execution because it eliminates the overhead of a function call.

Inline Function

Syntax:

The inline keyword is used before the function declaration or definition.

Example:

```
inline int add(int a, int b) {
    return a + b;
}
```

Inline Function

Header Files:

It is common to define inline functions in header files to allow the compiler to inline the function in different translation units.

Linkage:

Functions declared as inline should generally be defined in the same translation unit where they are used.

If an inline function is defined in multiple translation units, it may violate the One Definition Rule (ODR) and lead to linker errors.

Compiler-Specific Behavior:

Different compilers may have different strategies for inlining functions, and the effectiveness of inlining can vary.

Inline Function

```
#include <stdio.h>
// Declaration of an inline function
inline int add(int a, int b);
int main() {
    int result = add(3, 4);
    printf("Result: %d\n", result);
    return 0;
}
// Definition of the inline function
inline int add(int a, int b) {
    return a + b;
}
```

Inline Function

```
// myfunctions.h
#ifndef MYFUNCTIONS_H
#define MYFUNCTIONS_H

// Declaration of an inline function
inline int add(int a, int b) {
    return a + b;
}

#endif // MYFUNCTIONS_H
```

Inline Function

```
// myfunctions.c
#include "myfunctions.h"
#include <stdio.h>

// Definition of the inline function
// Note: The definition is included in the source file
inline int add(int a, int b) {
    return a + b;
}

// Additional non-inline function
int multiply(int a, int b) {
    return a * b;
}
```

Inline VS Macros

Scope:

Inline functions have their own scope and can contain local variables.

Macros don't have their own scope and are textually replaced during preprocessing.

Type Safety:

Inline functions provide type safety since they operate on typed parameters.

Macros operate on text and may not provide type safety.

Debugging:

Debugging **inline** functions is usually easier because they have their own scope.

Debugging **macros** can be challenging due to text substitution.

Preprocessing:

Inline functions are part of the C language and undergo the usual compilation process.

Macros are processed during the preprocessing phase and result in text substitution.

Inline VS Macros

```
// Inline function
inline int add(int a, int b) {
    return a + b;
}

// Inline macro
#define ADD(a, b) ((a) + (b))

int main() {
    int result_func = add(3, 4);
    int result_macro = ADD(3, 4);
    return 0;
}
```

Inline VS Function

Size of the Function:

Inline functions are more suitable for small, simple functions.

Regular functions may be more appropriate for larger functions or functions with complex logic.

Frequency of Invocation:

Inline functions are beneficial when a function is called frequently, and the function call overhead is a concern.

Regular functions are suitable for less frequently called functions.

Code Readability and Maintainability:

Inline VS Function

Code Readability and Maintainability:

Regular functions can enhance code readability and maintainability by encapsulating logic in separate units.

Inline functions may be more readable for very short and straightforward logic.

Compiler Optimization:

Compiler optimizations, including **inlining** decisions, depend on the compiler and its settings.

Profiling and performance testing may be necessary to determine the impact of inlining on your specific codebase.

Recall

```
// dynamic_allocation.h

#ifndef DYNAMIC_ALLOCATION_H
#define DYNAMIC_ALLOCATION_H

// Function declaration for dynamic allocation
int* allocateArray(int size);

#endif // DYNAMIC_ALLOCATION_H
```

Recall

```
// dynamic_allocation.c
#include "dynamic_allocation.h"
#include <stdlib.h>

// Function definition for dynamic allocation
int* allocateArray(int size) {
    int* array = (int*)malloc(size * sizeof(int));
    return array;
}
```

Recall

```
// main.c
#include "dynamic_allocation.h"
#include <stdio.h>

int main() {
    int size;

    // Get the size of the array from the user
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    // Call the function to dynamically allocate an array
    int* dynamicArray = allocateArray(size);

    // Check if memory allocation was successful
    if (dynamicArray == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Exit with an error code
    }
```

Recall

```
    // Initialize the array with values
    for (int i = 0; i < size; i++) {
        dynamicArray[i] = i * 2; // Just an example initialization
    }

    // Display the values of the dynamically allocated array
    printf("Dynamic Array Values: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", dynamicArray[i]);
    }
    printf("\n");

    // Free the dynamically allocated memory
    free(dynamicArray);

    return 0; // Exit successfully
}
```

Recall-Struct

```
// dynamic_allocation_struct.h

#ifndef DYNAMIC_ALLOCATION_STRUCT_H
#define DYNAMIC_ALLOCATION_STRUCT_H

// Structure declaration
struct Person {
    char name[50];
    int age;
};

// Function declaration for dynamic allocation
struct Person* allocatePersons(int numPersons);

#endif // DYNAMIC_ALLOCATION_STRUCT_H
```

Recall-Struct

```
// dynamic_allocation_struct.c
#include "dynamic_allocation_struct.h"
#include <stdlib.h>

// Function definition for dynamic allocation
struct Person* allocatePersons(int numPersons) {
    // Allocate memory for an array of structures
    struct Person* persons = (struct Person*)malloc(numPersons * sizeof(struct Person));
    return persons;
}
```

Recall-Struct

```
// main_struct.c
#include "dynamic_allocation_struct.h"
#include <stdio.h>

int main() {
    int numPersons;

    // Get the number of persons from the user
    printf("Enter the number of persons: ");
    scanf("%d", &numPersons);

    // Call the function to dynamically allocate an array of structures
    struct Person* people = allocatePersons(numPersons);

    // Check if memory allocation was successful
    if (people == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Exit with an error code
    }
```

Recall-Struct

```
    // Initialize the array of structures with values
    for (int i = 0; i < numPersons; i++) {
        // Just an example initialization
        sprintf(people[i].name, "Person%d", i + 1);
        people[i].age = 25 + i;
    }

    // Display the values of the dynamically allocated array of structures
    printf("People Information:\n");
    for (int i = 0; i < numPersons; i++) {
        printf("Name: %s, Age: %d\n", people[i].name, people[i].age);
    }

    // Free the dynamically allocated memory
    free(people);

    return 0; // Exit successfully
}
```