Fundamentals of Programming Languages

Unit V: User Defined Functions and Structures

Comprehensive overview of user-defined functions and structures in C progra

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

struct Point {
    int x, y;
};
```

Function Definitions Recursion Structures Memory Management

Course: Fundamentals of Programming Languages
Unit V: User Defined Functions and Structures
C Programming Language

Table of Contents

Unit V: User Defined Functions and Structures

User Defined Functions

Structures

Need for User-Defined Functions

Why User-Defined Functions are Essential

Modular Programming
Breaking down complex problems into smaller, manageable modules that are easier to understand and maintain.

Code Reusability
Write once, use multiple times. Functions eliminate the need to write the same code repeatedly.

Easier Debugging & Testing
Isolating code in functions makes errors easier to find and fix. Each function can be tested independently.

Organized Code Structure
Functions create a logical structure that makes programs more readable and maintainable.

Real-World Example

Consider a graphics program that needs to draw and color shapes:

Without functions:
Complex, repetitive code for each shape

With functions:
Separate `createCircle()` and `color()` functions

Example: Simple function definition

```
/* Function to draw a circle */
void createCircle(int x, int y, int radius) {
    // Circle drawing logic
    // ...
}

/* Function to color shapes */
void color(int r, int g, int b) {
    // Coloring logic
    // ...
}
```

A Multi-Function Program

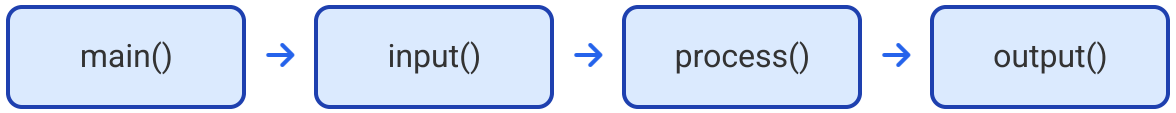
How Multiple Functions Work Together

Division of Tasks

Complex programs are broken down into smaller, specialized functions where each handles a specific task.

Function Relationships

Functions work together through calling each other, passing data between them, and building on each other's results.



Data Flow

Functions communicate by passing arguments and returning values, creating a structured flow of data through the program.

Multi-Function Example: Temperature Converter

A program that converts temperatures between scales using multiple functions:

main()
Controls program flow and user interaction

convert()
Performs temperature conversion calculations

display()
Formats and outputs the results

Example: Temperature Converter Functions

```
float celsiusToFahrenheit(float c) {
    return (c * 9/5) + 32;
}

float fahrenheitToCelsius(float f) {
    return (f - 32) * 5/9;
}

void displayResult(float value, char unit) {
    printf("%.2f %c\n", value, unit);
}
```

Elements of User-Defined Functions

Key Components of Functions in C

Function Prototype (Declaration)

Informs the compiler about the function's name, return type, and parameters without providing the implementation. Also called function signature.

```
return_type function_name(parameter_list);
```

Function Definition

Contains the actual code that implements the function's purpose. Includes the function header and body enclosed in curly braces.

```
return_type function_name(parameter_list) { /* function body */ }
```

Function Call

Transfers program control to the function. Arguments provided in the call are passed to the function parameters. After execution, control returns to the calling function.

```
result = function_name(argument_list);
```

Parameters & Arguments

Parameters are variables in the function declaration/definition. Arguments are the values passed when calling the function.

Function Elements Workflow

1. Declaration (Prototype)
Inform compiler about function

↓

2. Definition
Implement function logic

↓

3. Call
Execute function with arguments

Example: Complete function lifecycle

```
/* Function prototype */
int sum(int a, int b);

/* Main function with call */
int main() {
    int result = sum(5, 3);
    printf("Sum: %d", result);
    return 0;
}

/* Function definition */
int sum(int a, int b) {
    return a + b;
}
```

Function Declaration & Definition

Understanding the difference between function declaration and definition:

Declaration (Prototype) Definition (Implementation)

Function Declaration (Prototype)

```
// Function declaration (prototype)
return_type function_name(parameter_list);

// Examples:
int sum(int a, int b);
void printMessage(char message[]);
float calculateAverage(float arr[], int size);

// Parameter names are optional in declaration
int multiply(int, int);
```

Function declaration tells the compiler:

- Function name
- Return type
- Parameter types
- No function body

Function Definition

```
// Function definition (implementation)
return_type function_name(parameter_list) {
    // function body
    return value; // if applicable
}

// Example:
int sum(int a, int b) {
    int result = a + b;
    return result;
}
```

Function definition contains:

- Complete implementation
- Function body with statements
- Return statement (if not void)
- Parameter names are mandatory

Complete Example: Declaration and Definition

```
// File: math_functions.h
#ifndef MATH_FUNCTIONS_H
#define MATH_FUNCTIONS_H

// Function declarations (prototypes)
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);

#endif

// File: math_functions.c
#include "math_functions.h"

// Function definitions
int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int multiply(int a, int b) {
    return a * b;
}
```

Function Declaration and Definition Best Practices:

- Declare all functions before use (typically in header files)
- Keep declarations and definitions consistent (same return type and parameters)
- Use descriptive function names that indicate their purpose
- Group related functions in the same header and source files

Function Calls & Passing Arguments

Function Call Fundamentals

Function Call Syntax

To call a function, use its name followed by parentheses containing arguments (if any):

```
function_name(arg1, arg2, ...);
```

Actual vs. Formal Parameters

Actual parameters: Values/variables passed during function call
Formal parameters: Variables in function definition that receive values

Actual Parameters (in function call)

sum(a, b);

→

Formal Parameters (in function definition)

int sum(int x, int y) { ... }

Call by Value

The function receives **copies** of the values passed. Changes to parameters inside the function **do not affect** the original variables in the calling function.

Call by Reference

The function receives the **addresses** of variables. Changes to parameters inside the function **do affect** the original variables in the calling function.

Value vs. Reference Comparison

Feature	Call by Value	Call by Reference
Memory	Separate copy	Same location
Effect	No change to original	Changes original
Syntax	Pass variable	Pass pointer (&)
In function	Direct use	Use * operator

Values protected | Values modifiable

Example: Value vs. Reference

```
// Call by Value
void swap_value(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
} // Original values unchanged

// Call by Reference
void swap_ref(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
} // Original values swapped

// Calling the functions
int x = 5, y = 10;
swap_value(x, y);   // x=5, y=10 (unchanged)
swap_ref(&x, &y);   // x=10, y=5 (swa
```

Return Values and Their Types

Functions can return different types of values using the `return` statement:

- void (No value)
- Primitive types (int, float, char)
- Pointers
- Structures

Return Statement Syntax:

`return [expression];` → Returns control and optionally a value to the calling function

void return type

```c
void printMessage(char* msg) {
    printf("%s\n", msg);
    return; // Optional in void functions
}

int main() {
    printMessage("Hello World");
    return 0;
}
```

- **void** functions don't return any value
- return; statement is optional

Primitive return types

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

float calculateAverage(float nums[], int size) {
    float sum = 0.0;
    for(int i = 0; i < size; i++) {
        sum += nums[i];
    }
    return sum / size;
}
```

- Most common return types: `int`, `float`, `char`, etc.
- Return type must match the function declaration

Pointer return types

```c
int* findMax(int arr[], int size) {
    int* max = &arr[0];
    for(int i = 1; i < size; i++) {
        if(arr[i] > *max) {
            max = &arr[i];
        }
    }
    return max; // Returns pointer to max value
}
```

- Returns memory address (pointer)
- Warning: Don't return pointers to local variables!

Structure return types

```c
struct Point {
    int x;
    int y;
};

struct Point createPoint(int x, int y) {
    struct Point p;
    p.x = x;
    p.y = y;
    return p; // Returns entire structure
}
```

- Return complex data types as a single unit
- Creates a copy of the structure when returned

Return Value Best Practices:

- Always ensure the return type matches the declared function type
- Use consistent return values for error conditions (e.g., return -1 or NULL)
- For multiple return values, use pointers as parameters or return structures
- C functions can only return a single value directly

Categories of Functions in C

Functions in C can be categorized based on arguments and return values:

No args, no return | No args, with return | With args, no return | With args, with return

Type 1: No arguments, no return

```c
void displayMessage() {
    // Function with no parameters and no return
    printf("Hello, World!\n");
}

int main() {
    // Function call
    displayMessage();
    return 0;
}
```

- Useful for tasks that don't need input or output values
- Examples: displaying menus, printing headers

Type 2: No arguments, with return

```c
int getRandomNumber() {
    // Function with no parameters but returns value
    return rand() % 100; // Returns 0-99
}

int main() {
    int num = getRandomNumber();
    printf("%d\n", num);
    return 0;
}
```

- Generates and returns data without external input
- Examples: getting system time, random values

Type 3: With arguments, no return

```c
void printSum(int a, int b) {
    // Function with parameters but no return
    int sum = a + b;
    printf("Sum: %d\n", sum);
}

int main() {
    printSum(5, 10);
    return 0;
}
```

- Takes input but doesn't return values
- Examples: printing formatted data, modifying global variables

Type 4: With arguments, with return

```c
int calculateArea(int length, int width) {
    // Function with parameters and return value
    return length * width;
}

int main() {
    int area = calculateArea(5, 3);
    printf("Area: %d\n", area);
    return 0;
}
```

- Takes input and returns processed output
- Examples: calculations, data transformations

Function Selection Guidelines:

- Choose the appropriate function type based on whether you need to pass data in and/or get data out
- Type 4 (args + return) is most common for computational tasks and data processing

Functions Returning Multiple Values

In C, a function can technically return only one value. However, there are several approaches to effectively return multiple values:

■ Using Pointers (Pass by Reference) ■ Using Structures

Using Pointers

```c
// Return multiple values using pointers
void calculateStats(int arr[], int size,
                    int* sum, float* avg, int* max) {
    *sum = 0;
    *max = arr[0];

    for(int i = 0; i < size; i++) {
        *sum += arr[i];
        if(arr[i] > *max) *max = arr[i];
    }

    *avg = (float)*sum / size;
}

// Function call example
int main() {
    int numbers[] = {5, 10, 15, 20, 25};
    int sum, max;
    float avg;

    calculateStats(numbers, 5, ∑, &avg, &max);

    printf("Sum: %d, Avg: %.2f, Max: %d\n",
           sum, avg, max);
    return 0;
}
```

Using pointers allows:

- Modifying variables directly in caller's scope
- Passing memory addresses as parameters
- No need to create custom data types
- Very efficient for primitive data types

Using Structures

```c
// Define a structure to hold multiple values
struct Statistics {
    int sum;
    float average;
    int maximum;
};

// Return a structure containing multiple values
struct Statistics calculateStats(int arr[], int size) {
    struct Statistics stats;
    stats.sum = 0;
    stats.maximum = arr[0];

    for(int i = 0; i < size; i++) {
        stats.sum += arr[i];
        if(arr[i] > stats.maximum) stats.maximum = arr[i];
    }

    stats.average = (float)stats.sum / size;
    return stats;
}

// Function call example
int main() {
    int numbers[] = {5, 10, 15, 20, 25};
    struct Statistics result = calculateStats(numbers, 5);

    printf("Sum: %d, Avg: %.2f, Max: %d\n",
           result.sum, result.average, result.maximum);
    return 0;
}
```

Using structures allows:

- Grouping related values in a single package
- Clear data organization
- Return values with different types
- Simpler function signature

Combining Both Approaches

```c
// Define a structure to hold multiple values
struct Point3D {
    double x, y, z;
};

// Function that both returns a value and modifies parameters
struct Point3D transformPoint(struct Point3D original, double scale, double* distance) {
    struct Point3D result;

    // Scale the point coordinates
    result.x = original.x * scale;
    result.y = original.y * scale;
    result.z = original.z * scale;

    // Calculate distance from origin
    *distance = sqrt(result.x * result.x + result.y * result.y + result.z * result.z);

    return result;
}

// Function call example
int main() {
    struct Point3D p1 = {1.0, 2.0, 3.0};
    double dist;

    struct Point3D p2 = transformPoint(p1, 2.5, &dist);

    printf("New point: (%.2f, %.2f, %.2f), Distance: %.2f\n",
           p2.x, p2.y, p2.z, dist);
    return 0;
}
```

Best Practices for Multiple Return Values:

- Use pointers for simple cases with few values
- Use structures when returning many related values
- Document clearly which parameters are for output
- Always validate pointer parameters to avoid NULL pointer errors

Nesting & Recursion in Functions

Understanding function nesting and recursion in C programming:

Nesting: Calling functions from within functions **Recursion:** A function calling itself

Factorial using Recursion

```c
// n! = n * (n-1)!
// Base case: 0! = 1
int factorial(int n) {
    // Base case
    if (n == 0)
        return 1;

    // Recursive case
    return n * factorial(n - 1);
}

// Example usage
int main() {
    int result = factorial(5);
    printf("5! = %d\n", result);
    return 0;
}
// Output: 5! = 120
```

Fibonacci using Recursion

```c
// F(n) = F(n-1) + F(n-2)
// Base cases: F(0) = 0, F(1) = 1
int fibonacci(int n) {
    // Base cases
    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;

    // Recursive case
    return fibonacci(n - 1) + fibonacci(n - 2);
}

// Example usage
int main() {
    for(int i = 0; i < 7; i++)
        printf("%d ", fibonacci(i));
    // Output: 0 1 1 2 3 5 8
}
```

Nested Function Calls Example

```c
// Helper function
int square(int x) {
    return x * x;
}

// Function that uses the helper
int sumOfSquares(int a, int b) {
    return square(a) + square(b); // Nesting: calling square() inside sumOfSquares()
}

// Recursive function with nested function call
int recursiveSum(int arr[], int n) {
    if (n <= 0)
        return 0;

    return arr[n-1] + recursiveSum(arr, n-1); // Both recursion and nesting
}
```

Call Stack Visualization: factorial(3)

- main()
- factorial(3) → Returns 6
- factorial(2) → Returns 2
- factorial(1) → Returns 1
- factorial(0) → Returns 1

How the Call Stack Works:

1. Each recursive call creates a new stack frame
2. Parameters and local variables are stored in each frame
3. Base case stops the recursion
4. Return values are passed back up the stack
5. Function calls are resolved in LIFO order (Last In, First Out)

Recursion Best Practices:

- Always include a base case to prevent infinite recursion
- Ensure progress toward the base case with each recursive call

Introduction to Structures

What is a Structure in C?

User-defined Data Type

A structure is a user-defined data type that allows you to group items of possibly different types into a single type.

Heterogeneous Collection

Unlike arrays (which store items of the same type), structures can store related data of different types together.

Logical Grouping

Structures enable logical grouping of related data elements under a single name, improving code organization.

Building Block for Data Structures

Serves as a foundation for creating complex data structures like linked lists, trees, and graphs.

Real-World Applications

Structures are widely used to represent real-world entities:

Student Records

Name, ID, grades, courses

Library Management

Book title, author, publication, ISBN

Date & Time

Day, month, year, hour, minute

Example: Structure Definition

```
/* Structure to store student information */
struct Student {
    char name[50];
    int id;
    float gpa;
    char grade;
};

/* Creating a structure variable */
struct Student s1 = {"John", 101, 3.75, 'A'};
```

Structure Declaration and Initialization

Creating and initializing structures in C programming:

Structure Declaration | Structure Initialization

Structure Declaration

```c
// Structure definition (prototype)
struct structure_name {
    data_type1 member1;
    data_type2 member2;
    ...
}; // Don't forget the semicolon!

// Example: Student structure
struct Student {
    char name[50];
    int age;
    float gpa;
};

// Creating structure variables
struct Student s1, s2;
```

Structure declaration notes:

- Defines a custom data type
- Creates template, not memory allocation
- Variables created separately
- Can combine definition with variable declaration

Initialization Using Assignment

```c
// Declaration first
struct Student s1;

// Assignment of members
strcpy(s1.name, "John Smith");
s1.age = 20;
s1.gpa = 3.75;

// Note: Direct string assignment not allowed
// s1.name = "John"; // ERROR
// Must use strcpy() or equivalent
```

Assignment method:

- Initialize after declaration
- Access members using dot (.) operator
- Need string functions for char arrays
- Most flexible but verbose method

Using Initializer List

```c
// Initialize during declaration with list
struct Student s1 = {"Jane Doe", 21, 3.9};

// Partial initialization (remaining set to 0/NULL)
struct Student s2 = {"Bob Jones"};
// s2.age = 0, s2.gpa = 0.0 (default values)

// Initialize all to zero
struct Student s3 = {0};
```

Initializer list:

- Values assigned in order of declaration
- Concise syntax at declaration time
- Missing values are zero-initialized
- Cannot be used after declaration

Designated Initializer (C99)

```c
// Initialize specific members in any order
struct Student s1 = {
    .gpa = 3.5,
    .name = "Alice Wonder",
    .age = 19
};

// Partial initialization with designated initializers
struct Student s2 = {
    .name = "Charlie Brown",
    .gpa = 3.2
    // age will be set to 0
};
```

Designated initializer:

- C99 feature (not in C++)
- Initialize members by name
- Order doesn't matter
- Most readable and maintainable

Complete Structure Example

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

// Define a book structure
struct Book {
    char title[50];
    char author[50];
    float price;
    int pages;
};

int main() {
    // Method 1: Initialize using assignment
    struct Book book1;
    strcpy(book1.title, "C Programming");
    strcpy(book1.author, "Dennis Ritchie");
    book1.price = 29.99;
    book1.pages = 285;

    // Method 2: Initialize using initializer list
    struct Book book2 = {"Data Structures", "Niklaus Wirth", 24.50, 320};

    // Method 3: Initialize using designated initializer
    struct Book book3 = {
        .title = "Algorithms",
        .author = "Robert Sedgewick",
        .price = 45.75,
        .pages = 410
    };

    return 0;
}
```

Structure Declaration and Initialization Best Practices:

- Use descriptive structure and member names for readability
- Use designated initializers (C99) when possible for clarity
- Consider using typedef to create shorter aliases for struct types
- Be aware of structure padding and memory alignment issues

Accessing, Copying, and Passing Structures

Different ways to work with structures in C programming:

Accessing Members | Copying Structures | Passing to Functions

Dot Operator

```c
// Structure definition
struct Student {
    char name[50];
    int roll_no;
    float marks;
};

// Creating structure variable
struct Student s1;

// Accessing members with dot operator
strcpy(s1.name, "John");
s1.roll_no = 101;
s1.marks = 85.5;

// Printing structure members
printf("%s %d %.1f", s1.name, s1.roll_no, s1.marks);
```

Dot operator usage:

- Direct access to structure members
- Used with structure variables
- Syntax: `structure_variable.member_name`
- Cannot be used with structure pointers

Arrow Operator

```c
// Structure definition
struct Student {
    char name[50];
    int roll_no;
    float marks;
};

// Creating structure variable and pointer
struct Student s1;
struct Student *ptr = &s1

// Accessing members with arrow operator
strcpy(ptr->name, "Alice");
ptr->roll_no = 102;
ptr->marks = 92.5;

// Alternative using dereference
// (*ptr).roll_no = 102; // Equivalent
```

Arrow operator usage:

- Used with structure pointers
- Shorthand for `(*ptr).member`
- Syntax: `structure_pointer->member_name`
- Commonly used in linked lists, trees

Copying Structures

```c
// Structure definition
struct Point {
    int x, y;
};

// Method 1: Direct assignment
struct Point p1 = {10, 20};
struct Point p2;
p2 = p1; // Copy all members of p1 to p2

// Method 2: Member by member copy
struct Point p3;
p3.x = p1.x;
p3.y = p1.y;

// Method 3: Using memcpy (requires string.h)
struct Point p4;
memcpy(&p4, &p1, sizeof(struct Point));
```

Structure copying:

- Creates a shallow copy
- Fine for basic data types
- For pointers, only addresses are copied
- Need deep copy for dynamic memory

Passing to Functions

```c
// Structure definition
struct Rectangle {
    float length, width;
};

// Method 1: Pass by value
float calculateArea(struct Rectangle r) {
    return r.length * r.width;
}

// Method 2: Pass by reference (pointer)
void scaleRectangle(struct Rectangle *r, float factor) {
    r->length *= factor;
    r->width *= factor;
}

// Usage example
struct Rectangle rect = {10.0, 5.0};
float area = calculateArea(rect); // Pass by value
scaleRectangle(▭, 2.0);    // Pass by reference
```

Passing structures:

- By value: Makes a complete copy (inefficient for large structures)
- By reference: Passes pointer (efficient, allows modification)
- Const reference: Pass pointer with const qualifier to prevent modification

Best Practices with Structures:

- Use arrow operator with structure pointers for cleaner code
- Pass large structures by reference to improve performance
- Use typedef to create simpler aliases for structure types
- Implement deep copy functions for structures with dynamic memory

Practical Structure Examples & Summary

Key Takeaways

User-Defined Functions
Enable modular programming through code reusability, maintainability, and logical organization of complex tasks.

Function Types
Four categories based on arguments and return values, each serving different programming needs and use cases.

Recursion & Nesting
Functions can call themselves (recursion) or other functions (nesting) to solve complex problems efficiently.

Structures
User-defined data types that group different data types, enabling complex data organization and manipulation.

Practical Examples

Student Record System

```
struct Student {
    int id;
    char name[50];
    float gpa;
};

void displayStudent(struct Student s) {
    printf("ID: %d, Name: %s, GPA: %.2f\n",
           s.id, s.name, s.gpa);
}
```

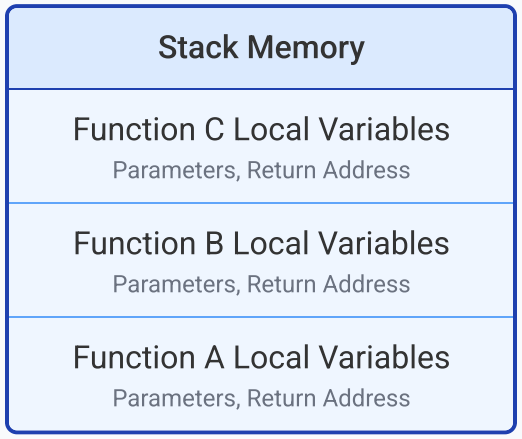
Linked List Node

```
struct Node {
    int data;
    struct Node* next;
};

// Example of recursion for traversal
void printList(struct Node* head) {
    if (head == NULL) return;
    printf("%d ", head->data);
    printList(head->next);
}
```

Memory Layouts

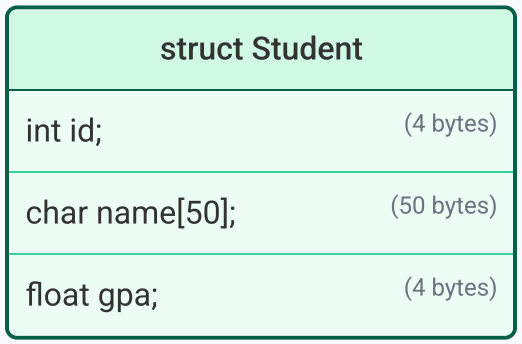
Function Call Stack



The function call stack stores local variables and return addresses for each function call, following LIFO (Last-In-First-Out) order.

Excessive recursion can cause stack overflow when the call stack exceeds its memory limit.

Structure Memory Layout



Structure members are stored sequentially in memory, with possible padding added for alignment requirements.

Structure padding can be controlled using #pragma pack or __attribute__((packed)) for memory optimization.

Implementation Tips

- Use functions to avoid code duplication and improve readability
- Choose appropriate function types based on your specific needs
- Use structures to organize related data logically
- Pass large structures by reference (pointer) for performance