C Programming Concepts Tutorial

This tutorial explains the C programming concepts used in the `student_records.c` project. It is intended for students who are new to C programming.

Table of Contents

Basic C Syntax

`main` function

The `main` function is the entry point of every C program. Execution of the program starts from the `main` function.

```c
int main(void) {
    // ...
    return EXIT_SUCCESS;
}
```

Comments

Comments are used to explain the code. They are ignored by the compiler. C supports two types of comments:

- **Single-line comments**: `// ...`

- **Multi-line comments**: `/* ... */`

```
// This is a single-line comment

/*
This is a
multi-line comment
*/
```

Preprocessor Directives

`#include`

The `#include` directive is used to include the contents of a file into the current file. In this project, we use it to include standard library headers like `<stdio.h>`, `<stdlib.h>`, etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
```

`#define`

The `#define` directive is used to define macros. Macros are typically used for constants to make the code more readable and maintainable.

```
#define INITIAL_CAPACITY 8
#define PASS_THRESHOLD 40
#define FILENAME "students.txt"
```

Data Types

The `student_records.c` project uses several data types:

- `int`: Used for storing integer values like roll number and marks.
- `char`: Used for storing single characters. `char *` is used for strings.
- `size_t`: An unsigned integer type used to represent the size of objects in bytes. It is returned by the `sizeof` operator.
- `FILE`: A structure used for file handling.

Enums

`typedef enum`

Enums (enumerations) are used to create a set of named integer constants. `typedef` is used to create a new type name for the enum.

```
typedef enum {
    SUCCESS = 0,
    ERR_MEMORY,
    ERR_NOT_FOUND,
    ERR_DUPLICATE,
    ERR_FILE_IO,
    ERR_INVALID_INPUT
} ErrorCode;
```

Structs

`typedef struct`

Structs (structures) are used to group related data items of different types. `typedef` is used to create a new type name for the struct.

```
typedef struct {
    int roll;
    char *name;
    int marks;
} Student;

typedef struct {
    Student **items;
    size_t size;
    size_t capacity;
    int modified;  // Track unsaved changes
    char *last_filename;  // Remember last used filename
} StudentList;
```

Pointers

Pointers are variables that store the memory address of another variable. They are extensively used in this project for dynamic memory allocation and passing arguments to functions by reference.

- `*`: The dereference operator, used to access the value at a memory address. (Gets the value at an address)
- `**`: The double deference operator. (Gets the value at the address of a pointer)
- `&`: The address-of operator, used to get the memory address of a variable. (Gets the address of a variable)
- `->`: The arrow operator, used to access the members of a struct through a pointer.
- `NULL`: A special pointer value that represents a pointer that doesn't point to any valid memory location.

Dynamic Memory Allocation

Dynamic memory allocation allows you to allocate memory during runtime.

- `malloc`: Allocates a block of memory of a specified size. Returns a pointer to the beginning of the block.
- `calloc`: Allocates a block of memory for an array of elements, initializes all bytes to zero, and returns a pointer to the first element.
- `realloc`: Changes the size of the memory block pointed to by a pointer.
- `free`: Deallocates a block of memory previously allocated by `malloc`, `calloc`, or `realloc`.

```
// Example of malloc
char *copy = malloc(len);

// Example of calloc
list->items = calloc(list->capacity, sizeof(Student*));

// Example of realloc
char *tmp = realloc(buf, capacity);

// Example of free
free(buf);
```

Arrays and Strings

- **Arrays**: A collection of elements of the same type stored in contiguous memory locations.
- **Strings**: In C, strings are represented as arrays of characters, terminated by a null character (`\0`).

This project uses several standard library functions for string manipulation:

- `strlen`: Calculates the length of a string.
- `memcpy`: Copies a block of memory from a source to a destination.
- `strcmp`: Compares two strings.
- `strchr`: Finds the first occurrence of a character in a string.
- `strcspn`: Calculates the length of the initial segment of a string which consists entirely of characters not in another string.
- `strtol`: Converts a string to a long integer.

File I/O

File I/O (Input/Output) is used to read from and write to files.

- `fopen`: Opens a file and returns a `FILE` pointer.
- `fprintf`: Writes formatted data to a file.
- `fgets`: Reads a line from a file.
- `fclose`: Closes a file.
- `errno`: A global variable that is set by system calls and some library functions in the event of an error to indicate what went wrong.
- `strerror`: Returns a pointer to the textual representation of the current `errno` value.

```c
// Opening a file for writing
FILE *f = fopen(filename, "w");

// Writing to a file
fprintf(f, "%d|%d|%s\n", s->roll, s->marks, s->name ? s->name : "");

// Reading from a file
while (fgets(buffer, sizeof(buffer), f)) {
    // ...
}

// Closing a file
fclose(f);
```

Functions

Functions are blocks of code that perform a specific task. They can be reused throughout the program.

- **Function Prototypes**: A function prototype declares a function's name, return type, and the types of its parameters. It allows the compiler to check for errors when the function is called.
- `static` **functions**: A `static` function is only visible within the file it is declared in. This is a way to restrict access to functions and is used for encapsulation.

Control Flow

Control flow statements are used to control the order in which the statements of a program are executed.

- `if`, `else`: Used for conditional execution.
- `while`, `for`: Used for looping.
- `switch`, `case`: Used for multi-way branching.
- `break`: Used to exit a loop or a `switch` statement.
- `continue`: Used to skip the current iteration of a loop and continue with the next one.

Type Casting

Type casting is used to convert a variable from one data type to another.

```c
// Casting a long to an int
int roll = (int)strtol(buffer, NULL, 10);

// Casting a char to an unsigned char
char c = tolower((unsigned char)line[0]);
```

Standard Library Functions

This project uses several functions from the C standard library:

- `printf`: Prints formatted output to the console.

- `fflush`: Flushes the output buffer of a stream.
- `getchar`: Reads a single character from the standard input.
- `isspace`: Checks if a character is a whitespace character.
- `tolower`: Converts a character to lowercase.
- `qsort`: Sorts an array.

Advanced Memory and String Operations

`memmove()`

The `memmove()` function is used to copy a block of memory from a source to a destination. It is similar to `memcpy()`, but it is safer to use when the source and destination memory blocks overlap.

```
// Example of memmove
memmove(&list->items[index], &list->items[index + 1],
        (list->size - index - 1) * sizeof(Student*));
```

`printf` Format Specifiers

The `printf` function uses format specifiers to determine how to format the output.

- `%s`: String
- `%d`: Integer
- `%-5d`: Integer, left-justified, with a minimum width of 5 characters.
- `%-30s`: String, left-justified, with a minimum width of 30 characters.
- `%3d`: Integer, with a minimum width of 3 characters.
- `%zu`: `size_t` (unsigned integer)
- `%.2f`: Floating-point number with 2 digits after the decimal point.
- `\n`: Newline character.

The Null Terminator (`\0`)

The null terminator is a special character (`\0`) that marks the end of a string in C. It is automatically appended to string literals.

C Operators

- `&&`: Logical AND operator. Returns true if both operands are true.
- `!=`: Not equal to operator. Returns true if the operands are not equal.
- `!`: Logical NOT operator. Reverses the logical state of its operand.

Functions Returning Pointers

In C, functions can return pointers. This is useful when a function needs to create and return a dynamically allocated object, such as a `Student` in this project.

```
Student *create_student(int roll, const char *name, int marks) {
    Student *student = malloc(sizeof(Student));
    // ...
    return student;
}
```

The `*` before the function name `create_student` indicates that the function returns a pointer to a `Student` struct.

Best Practices

This project follows several best practices for C programming:

- **Modularity**: The code is organized into functions, each with a specific purpose.
- **Error Handling**: The code checks for errors (e.g., memory allocation failures, file I/O errors) and handles them gracefully.
- **Memory Management**: The code uses dynamic memory allocation and deallocation to manage memory efficiently.
- **Readability**: The code is well-commented and uses meaningful variable names to improve readability.
- **Constants**: The code uses `#define` to create named constants, making the code more maintainable.
- **Input Validation**: The code validates user input to prevent errors.