Pointers and Input/Output in C

Penn State University

Contents

1 Introduction

This document covers advanced concepts in the C programming language, specifically **pointers** and **input/output (I/O)** operations. Pointers are variables that store memory addresses, allowing for dynamic data manipulation. I/O operations manage data flow between a program and external devices like keyboards, files, and networks.

2 Pointers to Pointers

2.1 Basic Pointer Concept

A pointer is a variable that holds the memory address of another variable. For example, in the following code:

```
int x = 5; int *p = &x; // p points to x
```

Here, `&x` retrieves the address of `x`, and `p` stores this address.

2.2 Function Example

Functions can modify variables using pointers. In the `swap` function below, two integers are swapped:

```
void swap(int *a, int *b) { int temp = *a; *a = *b; *b = temp; }
```

The `*` operator dereferences the pointer, accessing the variable's value.

2.3 Pointers to Pointers

To modify a pointer itself (e.g., `*p`), use a pointer to a pointer. Example:

```
void swapPointers(int **p1, int **p2) { int *temp = *p1; *p1 = *p2; *p2 = temp; }
```

Here, `**p1` accesses the integer pointed to by `p1`.

3 Multiple Levels of Pointers

3.1 Pointers to Pointers to Pointers

You can have multiple levels, like `char ***cpp`. This is useful in complex data structures. Example:

```
char *c[] = { "A", "B", "C" }; char **cp = c; char ***cpp = &cp;
```

Here, `cpp` points to `cp`, which points to the array `c`.

3.2 Practical Example

The following code demonstrates multiple pointer levels:

```
#include <stdio.h> int main() { char *strs[] = { "Hello", "World" }; char **ptrs = strs
```

This prints `Hello` by dereferencing each pointer.

4 Pointers to Functions

Pointers can also point to functions, enabling dynamic behavior. Example:

```
int add(int a, int b) { return a + b; } int main() { int (*funcPtr)(int, int) = add; pr
```

Here, `funcPtr` points to the `add` function.

5 Input/Output Basics

5.1 Definition

Input/Output (I/O) is the process of reading from or writing to devices, like keyboards or files. Common I/O types include:

- **Terminal I/O:** Interaction with the keyboard and screen.
- **File I/O:** Reading from and writing to files.
- **Network I/O:** Communicating with other computers.

5.2 Buffered vs. Unbuffered I/O

- **Buffered I/O:** Data is temporarily stored in memory (a buffer) before being read or written. This improves efficiency.
- **Unbuffered I/O:** Data is read or written directly without buffering, which can be slower.

6 File I/O in C

6.1 Opening Files

The `fopen` function opens a file. Syntax:

```
FILE *fopen(const char *path, const char *mode);
```

- **path:** File name.
- **mode:** How to use the file (e.g., `"r"` for reading).

6.2 Reading and Writing

Example of reading a file:

```
FILE *file = fopen("data.txt", "r"); char buffer[100]; fgets(buffer, 100, file); printf
```

This code opens `data.txt`, reads a line, and prints it.

6.3 Closing Files

Always close files with `fclose` to free resources:

```
fclose(file);
```

7 Terminal I/O in C

7.1 Standard Streams

C has three default streams:

- **STDIN:** Standard input (keyboard).
- **STDOUT:** Standard output (screen).
- **STDERR:** Standard error (also the screen).

7.2 Using `stdin`, `stdout`, and `stderr`

Example:

```
#include <stdio.h> int main() { fprintf(stderr, "Error message\n"); return 0; }
```

This prints an error message to the screen using `STDERR`.

8 Input/Output Redirection

8.1 Redirection Basics

Redirection changes where input and output go. For example:

- **Input Redirection:** Uses a file as input. Example: `./program < input.txt`.
- **Output Redirection:** Sends output to a file. Example: `./program > output.txt`.

8.2 Example of Redirection

Suppose you have a program that reads user input:

```
#include <stdio.h> int main() { char name[50]; printf("Enter your name: "); fgets(name,
```

You can run it like this:

```
./program < names.txt > output.txt
```

This uses `names.txt` as input and saves the output to `output.txt`.

9 Pipes

9.1 What is a Pipe?

A pipe connects the output of one program to the input of another. Example: `ls` more— lists files page by page.

9.2 Chaining Pipes

You can chain multiple pipes: `cat file.txt` grep "text" — sort—. This command reads `file.txt`, filters lines containing "text", and sorts the results.

10 Conclusion

This document introduced advanced C topics: pointers, functions, and I/O. Understanding these concepts is crucial for systems programming and managing hardware interactions.