C Programming For Absolute Beginners & Question Set I

Sahas Talasila

Contents

Let's look at what a programming language is and what is the point of learning C.

A programming language is a way of interacting with a computer in a language it can understand. We can control microprocessors and embedded systems with the right language. C allows for easy **low-level control**

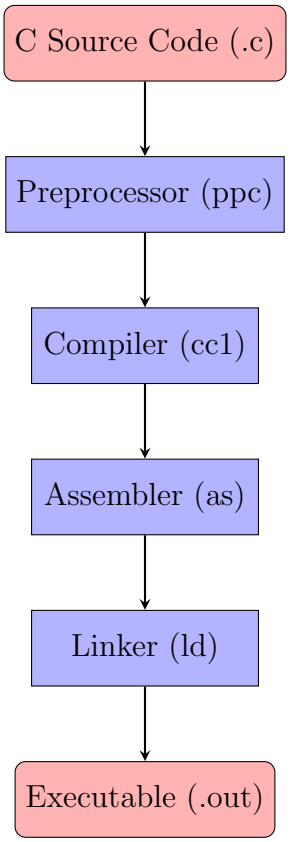
C is a **low level, compiled** and **procedural** language. I will explain what all of these words in bold mean, starting with **compiled**.

1.1 What does compiled mean?

Compiled means that the code you have written is saved then converted into binary (or machine code), which is easily understood by the computer, before you 'run' the code. This is important because it allows us to create code that uses **less memory** and it is very **fast**.

Here is a simpler list below, which combines some of the steps together:

- Preprocessing: This is the first step where the preprocessor processes the source code. It includes tasks like expanding **macros** (macros are preprocessor directives that define code snippets or constants, which are replaced before the actual compilation process begins), including **header files**, and performing text substitutions. The result is a modified source code that is ready for compilation.
- Compilation: The compiler then translates the preprocessed source code into assembly language, which is a low-level language that is easier for machines to understand than high-level languages. This step involves parsing the source code and generating assembly code.
- Assembly: The assembly code generated by the compiler is then converted into object code by an assembler. Object code is a low-level representation that is closer to machine code but still not executable by the computer.
- Linking: Finally, the linker combines the object code with any necessary libraries to create an executable file. This step resolves any external references and creates a complete executable program.



1.2 So how does it work (in detail)?

The following steps are completely **OPTIONAL**, but I think an in-depth understanding of the compiler will help you write code much better. It also includes some easy to understand examples. If you want to get tutored by me for the Advanced C/C++ course, I would suggest reading this, as it forms the foundation for the course.

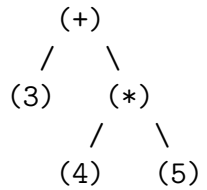
Step 0: Preprocessing (optional in some languages)
Before lexical analysis, some languages (e.g., C/C++) have a *preprocessing* step. This phase expands macros, includes header files, and processes conditional compilation. It essentially transforms all of the #include, #define directives, etc., so the compiler's next stage (Lexical Analysis) sees a simplified, ready-to-tokenize version of the code.

Step 1: Lexical Analysis
The compiler breaks the code into individual 'tokens', such as keywords, identifiers, literals, and symbols from top to bottom. This is like reading a sentence and breaking it down into individual words. An example would be: `Input = Hi, I am Sahas!` - would be split into `'Hi'`, `','` `'I'`, `'am'`, `'Sahas'`, `'!'`, and as we can see, all of the words or 'tokens' have been identified correctly.

Step 2: Syntax Analysis:
The compiler checks the 'tokens' for syntax errors (or spelling mistakes), ensuring that they follow the rules of the programming language. This is like checking if the sentence is grammatically correct. A grammatically incorrect example would be: `Input = Hi, I m Sahas!`. If you read that, you could see that all of the words have been spelt correctly, but it doesn't quite make any sense, as we have missed out an 'a' in 'am', the computer would just see a random 'm' in the middle of the sentence.

Symbol Table:
During the Syntax and Semantic Analysis phases, the compiler creates and maintains a **symbol table** that maps identifiers (**variables, functions, classes**, etc.) to information such as **data types, scopes, memory locations**, and usage counts.

Abstract Syntax Tree (AST):
After ensuring the tokens match the grammar of the language (*Syntax Analysis*), the compiler typically constructs an **Abstract Syntax Tree (AST)**. The AST is a **tree data structure** (watch out for data structures later!) where each node represents a construct in the language. For example, consider the expression `3 + 4 * 5`, an AST might look like:



This tree helps the compiler understand operator precedence and the overall program structure for subsequent steps (semantic checks, optimizations, etc.).

Step 3: Semantic Analysis:
The compiler analyses the syntax-checked tokens to ensure that they make sense in the context of the program. This is like understanding the meaning of the sentence. An incorrect example would be: `Input = I! am Hi, Sahas`. The words have been spelt correctly, getting past **step 2**, but the order doesn't make any sense.

Step 4: Intermediate Code Generation:
The compiler generates intermediate code, which is a platform-independent representation of the source code. We translate it into a language the computer understands. This is like the brain (or neurons) converting the text to electrical signals or impulses.

Step 5: Optimisation of The Compiler:
optimizes the intermediate code to improve its performance, such as reducing memory usage or reordering instructions. This is like editing the sentence to make it more concise and efficient. This is the most 'random' part of the process, because the compiler can optimise some things and not others. The compiler improves the intermediate representation (IR) through various optimizations:

- **Constant Folding**: Evaluating constant expressions at compile time.
- **Dead Code Elimination**: Removing code that never executes or whose results are never used.
- **Loop Optimisations**: Such as *loop unrolling* or *strength reduction*.
- **Inlining**: Replacing a function call with the function body to reduce function-call overhead.

These optimizations vary in aggressiveness depending on compiler settings (e.g., `-O1`, `-O2`, `-O3` in GCC).

Step 6: Code Generation:
The compiler generates machine code that can be executed directly by the computer's processor. This is like translating the sentence into the native language of the computer, which is binary (1s and 0s). Think of it like the impulses in your brain checking before giving you an output, starting to create the words in your mind.

Step 7: Output:
The compiler produces an executable file or object code that can be linked with other object files to create a final executable program.

Summary
Think of it like our brain, when it is processing something, like reading a book and think about every detail our brain would check when reading.

1.3 Low Level and High Level languages

To actually understand what low level languages are, come up with an **algorithm** (set of instructions) using **pseudocode (OPTIONAL)** to describe a morning routine. I have included examples of **High level** and **Low level** below. First, let's look at **High Level** below:

Algorithm 1 High Level Morning Routine

```
1: procedure MorningRoutineHighLevel
2:     Wake up when the alarm rings
3:     Turn off the alarm
4:     if still feeling drowsy then
5:         Perform a quick stretching or breathing exercise
6:     end if
7:     Make the bed
8:     Brush teeth and wash face
9:     Take a shower and get dressed
10:    Prepare and eat a healthy breakfast
11:    for each major task planned for the day do
12:        review the task details and estimate completion time
13:    end for
14:    Check your bag for essentials (keys, phone, wallet, etc.)
15:    Leave the house on time for work or school
16: end procedure
```

We can see that there aren't a lot of steps and it doesn't go into too much detail. High level means a **higher** amount of **abstraction** (distance away from machine code, closer to readable). Low level, is unsurprisingly the opposite of that, closer to machine code (harder to read for us, easier for the computer). Here's an example of that on the next page:

Algorithm 2 Low Level Morning Routine

procedure MorningRoutineLowLevel
    Wake up at 6:00 AM
    Turn off the alarm clock
    Swing legs off the bed and place feet on the floor
    Stand up slowly, stretching arms overhead for 10 seconds
    Walk 12 steps to the bathroom
    Pick up toothbrush from the holder
    Squeeze a pea-sized drop of toothpaste onto the toothbrush
    Brush teeth for 2 minutes (30 seconds for each quadrant of the mouth)
    Rinse mouth and toothbrush
    Wash face with lukewarm water and mild cleanser
    Pat face dry with a clean towel
    Take a shower:
    - **Turn on** the water and wait until warm
    - **Wet hair** thoroughly
    - **Apply shampoo**, massage for 30 seconds
    - **Rinse hair**
    - **Apply conditioner**, leave in for 1 minute
    - **Rinse conditioner** thoroughly
    - **Use body wash** or soap, lather entire body
    - **Rinse off** completely

    Dry off and **dress** in appropriate clothing
    Go to the kitchen and fill a glass of water
    Prepare breakfast (e.g., scrambled eggs and toast)
    if time permits **then**
        Sit down and eat breakfast at the table
    else
        Pack breakfast in a container to eat on-the-go
    end if
    Check bag or briefcase for all essentials (phone, keys, wallet, etc.)
    Turn off any unnecessary lights or devices
    Leave home by 7:30 AM to ensure timely arrival at work/school
end procedure

Building your first program

Let's look at the structure of a simple mathematical program (Listing 1) to understand how the compiler processes everything.

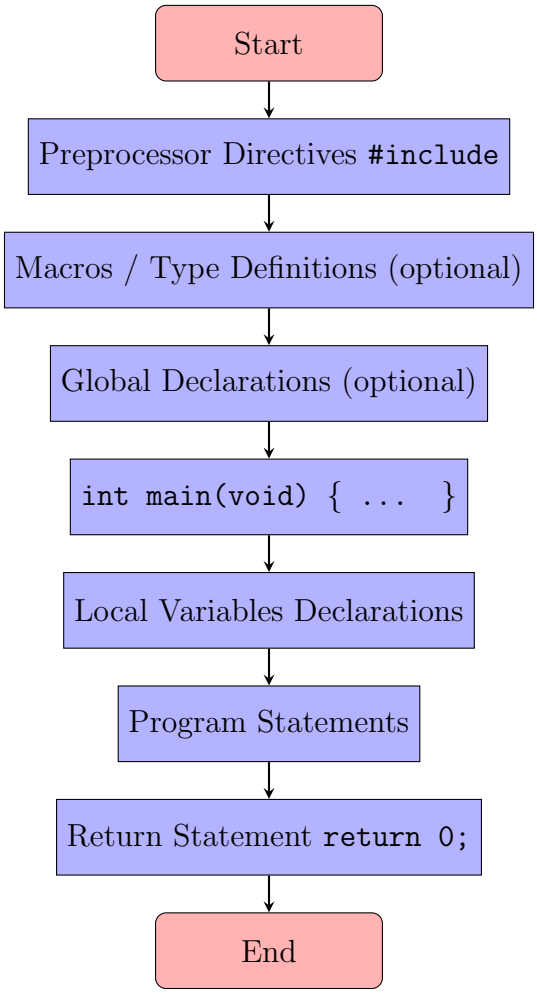
```c
#include <stdio.h>  // Preprocessor directive to include library

#define PI 3.14159  // Macro definition - creates a constant

// Function declaration - tells compiler about the function's existence
int add(int a, int b);

int main() {
    // Main function - program's entry point
    int result = add(5, 3);  // Call the add function
    printf("Result: %d\n", result);  // Print the result
    return 0;  // Exit the program successfully
}

// Function definition - actual implementation of the function
int add(int a, int b) {
    return a + b;  // Simple addition and return of the result
}
```

Listing 1: First program

Now, lets look at what each comment ('//' used for single line comments) means. This is the comment in green in the snippet above.

- **Preprocessor: '#'**
  This allows us to use the **standard library**, or standard packages that we might install, with the help of the keyword 'include'.
- Standard Library:
- **Keyword 'define':**
  We use this for header files and defining global constants (key values that we will use through out a program).
- **The main() function:**
  This is the entry point for the program to start executing. Remember the compiler? This is where it starts converting the code after checking it for errors.
- **Functions:**
  They are small snippets of code that are not in the main() function, but I will explain this later.

Below, I have added a simple flow chart to show what should be in the standard C program.



2.1 Basic Types, Operators and Format Specifiers

Now that we have an idea of what our program should look like, lets start looking at what we can use. We will start with the basic types that you might see in a simple program (Listing 2).

```c
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

int main() {
// Integer: whole numbers
int age = 25;  // Typically 4 bytes, range: -2,147,483,648 to 2,147,483,647

float height = 1.75f;  // Single-precision, 4 bytes (Floats)
double precise_pi = 3.14159;  // Double-precision, 8 bytes

char grade = 'A';  // Characters use single quotes, typically 1 byte

Bool is_student = 1;  // 1 is true, 0 is false (Boolean)

bool isAdmin = true;  // equivalent to int isAdmin = 1; // We can do this as well
bool isUser = false;  // equivalent to int isUser = 0;

// Type casting: converting between types
int x = 10;
float y = (float)x / 3;  // Explicit conversion to avoid integer division

return 0;
}
```

Listing 2: Basic types and type-casting

Quick Refresher:
(A **bit** is simply a 1 or a 0, a **byte** is **8 bits**.)

C has several basic data types:

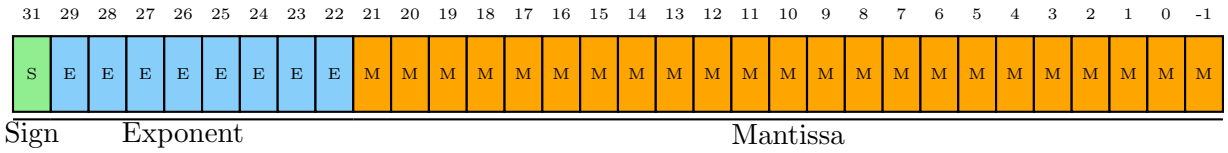
Integers: `int, short, long, long long` (with optional unsigned variants)
Floating-point: `float, double, long double`

In C, the `float` type is commonly represented using the IEEE 754 single-precision format, which occupies 32 bits. These bits are divided into three components:

- **Sign bit** (1 bit): 0 for positive, 1 for negative.
- **Exponent** (8 bits): Stored with a bias of 127, representing the power of 2.
- **Mantissa** (23 bits): The fractional part, normalized with an implicit leading 1.

In C, the `float` type is commonly represented using the IEEE 754 single-precision format, which occupies 32 bits. These bits are divided into three components:

- **Sign bit** (1 bit): 0 for positive, 1 for negative.
- **Exponent** (8 bits): Stored with a bias of 127, representing the power of 2.
- **Mantissa** (23 bits): The fractional part, normalized with an implicit leading 1.



The diagram represents the 32-bit layout as an array of bits:

- Bit 31 (S, green): Sign bit.
- Bits 30–23 (E, blue): Exponent (8 bits).
- Bits 22–0 (M, orange): Mantissa (23 bits).

Characters: `char` (often 8-bit, can be signed or unsigned)
Boolean (in C99 and later): `#include <stdbool.h>` provides bool (with values true/false), shown in Listing 3

The single-precision format offers about 7 decimal digits of precision due to its 23-bit mantissa. This can result in rounding errors for certain calculations. For greater precision, C provides the `double` type, which uses 64 bits.

```c
int age = 25;
float height = 5.9f;
char initial = 'A';
_Bool isStudent = 0; // or bool isStudent = false; if <stdbool.h> is included
```

Listing 3: Numerical data types

Let's look at the operators:

- Arithmetic: `+, -, *, /, %`, (used for simple maths, addition, subtraction, etc.)
- Assignment: `=, +=, -=, *=, /=, %=`, (used for declarations or specific, simple actions that need to be done.)
- Increment/Decrement: `++, --`, (adding or subtracting by a specified amount or 1 until a condition has been satisfied). Will be useful for the algorithms and data structures section!
- Relational: `==, !=, <, >, <=, >=`. Compares two things.
- Logical: `&&, !, AND, OR, NOT` operators.
- Bitwise: `&, ^, ~, <<, >>`, very fast, low-memory use comparison operators.
- Ternary: `?:` is a shorthand for an if-else statement, allowing conditional expressions in a compact form. All of the above shown in Listing 4.

```
int x = 10, y = 3;
int sum = x + y;       // sum = 13
int prod = x * y;      // prod = 30
int remainder = x % y; // remainder = 1
if (x > 5 && y < 5) {
    // ...
}
```

Listing 4: Example output

Type-casting:
If we look at the last example in the code above, we see `float y = (float)x / 3`. This means that we will **temporarily** convert an integer into a float so we get the correct answer. This is generally not recommended, but if you want to convert temporarily, then it is fine.

Format Specifiers and Escape Sequences:
Next, we will look at format specifiers tell the compiler about the type of data to expect, preventing errors and ensuring correct interpretation.

`\%d` for integers
`\%f` for floating-point numbers
`\%s` for strings, etc.

Precision and Alignment: They allow customization of output, such as setting decimal precision or field width. (Example on the next page).

Example: `printf("%.2f", 3.14159); outputs 3.14`
Type Safety: Using the wrong specifier can lead to undefined behavior. For instance, passing a float to `%d can yield incorrect results`.

Escape Sequences:
Represent Non-Printable Characters: They allow inclusion of characters that cannot be typed directly, such as newlines `(\n) or tabs (\t)`.

Example: `printf("Hello\tWorld\n");` outputs:
`Hello  World`
String Formatting: They enable precise control of text layout, aiding in creating visually structured outputs.

Example:
Adding a 'newline' or 'tab' for better readability in console output.

Escape Special Characters:
Some characters, like the double quote (") and backslash (\), have special meanings in C. Escape sequences allow their literal inclusion in strings. In C, when you declare a string literal using double quotes ("..."), the compiler automatically adds a null character (\0) at the end of the string. This null character marks the end of the string and is used by the C runtime library to determine the length of the string, adding +1 to the length of a string.

Example: `printf("She said, \"Hello!\"\n");`.

In C, user input and output are typically handled via the standard I/O library,
included with `#include <stdio.h>`. The function `printf()` displays information on the screen.
It uses a format string with placeholders which we looked at before (e.g., `%d, %f, %c`) to show variables in the desired form.

The function `scanf()` reads data from the user (keyboard) into variables. Like `printf()`, `scanf()` uses format specifiers, but you usually pass the variable's address using `&`. For instance, `scanf("%d", &x)` reads an integer and stores it in x. Strings are an exception `scanf("%s", str)` does not require the `&` if `str` is an array.

Always check the return value of scanf() to ensure input was read successfully. These I/O functions form the basis of user interaction in command-line C programs, such as the one in Listing 5.

```c
#include <stdio.h>

int main(void) {
    int number;
    char name[50];

    printf("Enter your first name: ");
    scanf("%s", name);

    printf("Enter your age: ");
    scanf("%d", &number);

    printf("Hello %s, you are %d years old.\n", name, number);
    return 0;
}
```

Listing 5: I/O in C

Enumerations: Enums in C provide a way to define a set of named integer constants, making code more readable and maintainable.

An `enum` (short for enumeration) is a user-defined data type in C that assigns names to integer values. Instead of using raw numbers, `enum` allows working with meaningful names.

The syntax of an `enum` is shown in Listing 6:

```c
#include <stdio.h>

enum Day {
    SUNDAY,    // 0
    MONDAY,    // 1
    TUESDAY,   // 2
    WEDNESDAY, // 3
    THURSDAY,  // 4
    FRIDAY,    // 5
    SATURDAY   // 6
};

int main() {
    enum Day today = WEDNESDAY;

    printf("Today is %d\n", today); // Output: Today is 3

    return 0;
}
```

Listing 6: ENUM usage example

The values in an `enum` start at `0` by default and increment by `1`, unless explicitly assigned. It is possible to specify values manually, as demonstrated below:

```c
enum Status {
    SUCCESS = 1,
    FAILURE = -1,
    PENDING = 0
```

Enums are internally stored as integers, meaning they can be used just like integer variables.

Using `enum` improves code readability, prevents the use of magic numbers, and encourages consistency in defining related constants.

Common use cases of `enum` include defining days of the week, error codes (`SUCCESS`, `FAILURE`), and state management (`IDLE`, `RUNNING`, `STOPPED`).

2.2 Control Flow and Loops

2.2.1 If-Else and Switch Statements

If-Else and Switch Statements:

The if-else control flow is a way of checking conditions to see if something happens or not. **If** something happens, do x, **else** do y. You might be wondering why we have switch-case statements, because they are better at checking for **values**, whereas `if-else` is used for **conditions**. `Switch-case` statements are a bit faster and memory efficient if used correctly (Listing 7).

```c
#include <stdio.h>

int main() {
int score = 75;

// If-else Ladder: Checks multiple conditions
if (score >= 90) {
    printf("Grade A: Excellent performance!\n");
} else if (score >= 80) {
    printf("Grade B: Very good work!\n");
} else if (score >= 70) {
    printf("Grade C: Good job!\n");
} else {
    printf("Needs improvement. Keep studying!\n");
}

// Switch Case: Efficient for multiple discrete values
switch (score / 10) {
    case 9:  // 90-99
        printf("Top tier performance\n");
        break;  // Prevents falling through to next case
    case 8:  // 80-89
        printf("Strong performance\n");
        break;
    case 7:  // 70-79
        printf("Satisfactory performance\n");
        break;
    default:  // Catches all other cases
        printf("Needs work\n");
}

return 0;
}
```

Listing 7: If-Else control flow example

I have included flowcharts to visually show how each control flow/conditional logic method works. Here is a flowchart for `if-else` and `switch-case`, with `if-else` first, shown on the next page for clarity:

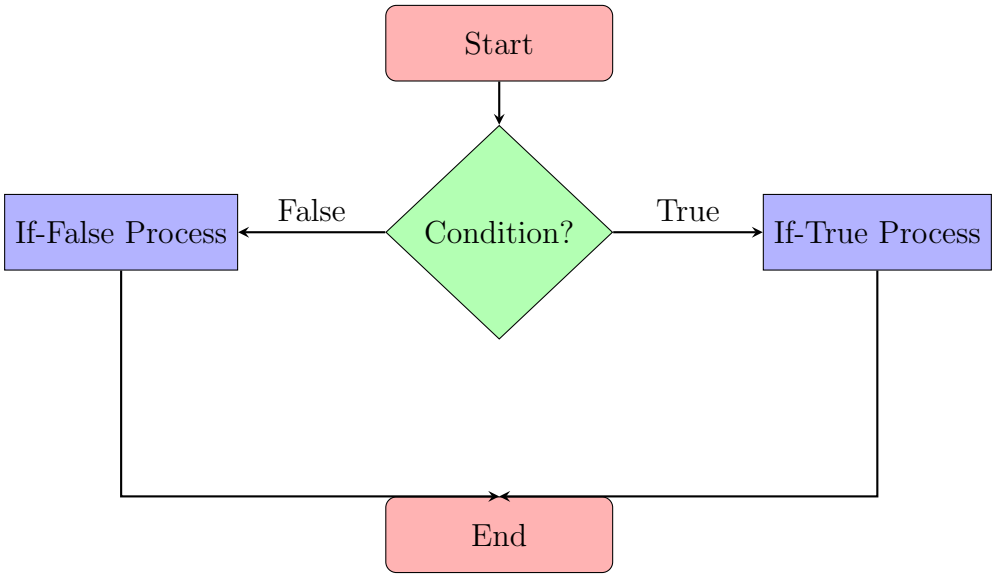


Figure 1: This is an example of an If-Else flowchart

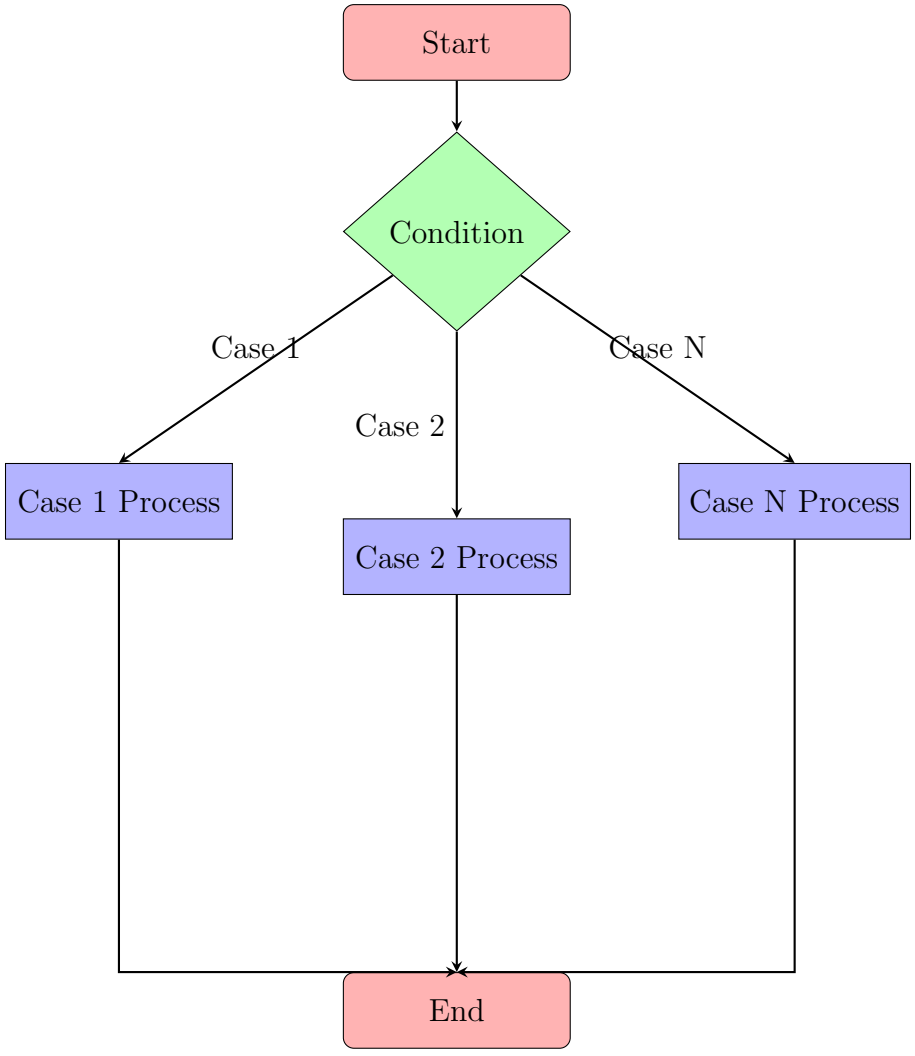


Figure 2: This is an example of a Switch-Case flowchart

2.2.2 For, While, Do-While Loops

What do these loops do?

For loop:
This loop is used iterate over a known number or a sequence, until we have reached the end of said sequence. This is one of the most useful and common loops that we will use, providing the basis for a lot of the searching functions you will create. Below the code, there is a flowchart (Figure 3) for for loops and Listing 8 shows the C syntax.

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

    int main() {
    // For Loop: Used when iteration count is known (1)
    printf("For Loop: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", i);  // Prints: 0 1 2 3 4
    }
    printf("\n");

    // While Loop: Continues while condition is true (2)
    int count = 0;
    while (count < 3) {
        printf("While Loop Count: %d\n", count);
        count++;  // Must increment to avoid infinite loop
    }

    // Do-While Loop: Guarantees at least one execution (3)
    int x;
    do {
        x = rand() % 10;  // Generate random number
        printf("Random number: %d\n", x);
    } while (x != 0);  // Repeat if x is not zero

    return 0;
}
```

Listing 8: For loop example

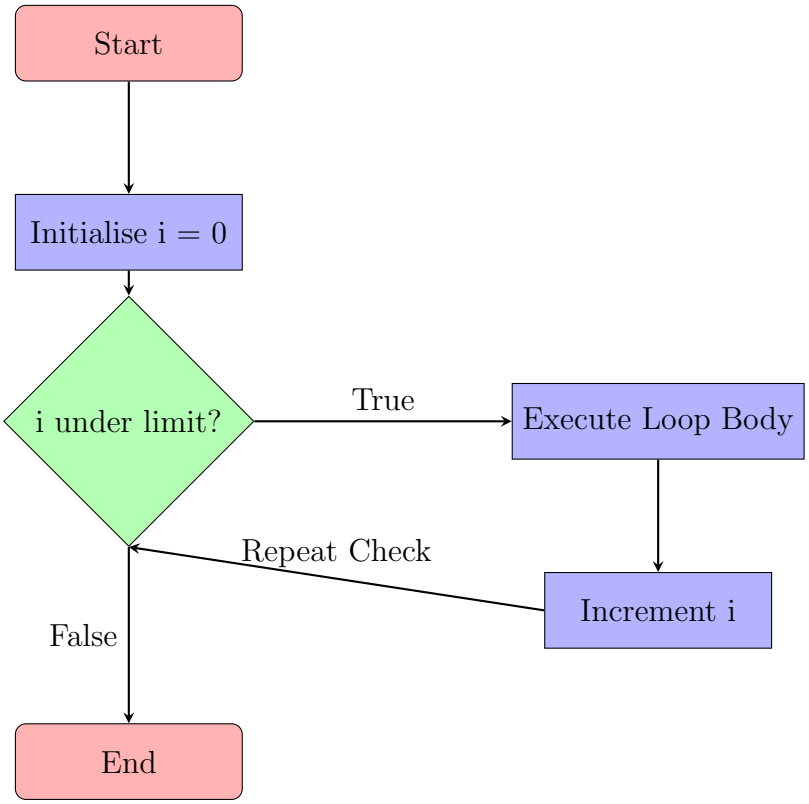


Figure 3: This is an example of a For Loop flowchart

While loop:
This loop is somewhat similar, but instead of looking for a discrete value, it iterates until a specific **condition** has been fulfilled. We have to be careful to end the statement properly or we get an infinite loop. This can cause problems if we aren't careful. Look at the code below and try it out for yourself. Below, I will explain the importance of infinite loops and the errors they can cause. There is a flowchart for easier understanding (Figure 4). Listings 9, 10 and 11 show the potential mistakes and solutions.

```c
#include <stdio.h>

    int main() {
        while (1) {
            printf("This is an infinite loop.\n");
        }

        return 0;
    }
```

Listing 9: While loop: infinite loop error

And another example (Will crash the computer/machine):

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

int main() {
    while (true) {
        printf("This is another infinite loop.\n");
    }

    return 0;
}
```

Listing 10: Infinite loop with Boolean values

And here's a better solution:

```c
#include <stdio.h>

int main() {
    int count = 0;

    while (1) {
        printf("Count: %d\n", count);
        count++;
        if (count > 10) {  // Stop the loop after 10 iterations
            break;
        }
    }

return 0;
}
```

Listing 11: Good use of while loop

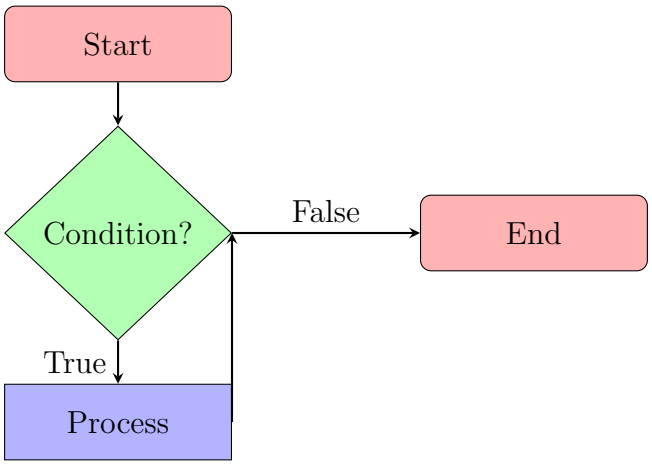


Figure 4: This is an example of a While Loop flowchart

Do-While loops:
Lastly, Do-while loops are used to perform an **action** before checking the **condition**. It is similar to a while loop, but the condition is evaluated after the loop body, whereas in a while loop, the condition is evaluated before the loop body. They can be thought of as a 'safety measure', because they run the loop at least once. Figure 5 and Listing 12 show the flowchart and code.

```c
#include <stdio.h>
#include <stdlib.h> // Not necessary for now.

int main() {

// Do-While Loop: Guarantees at least one execution (3)
    int x;
    do {
        x = rand() % 10;  // Generate random number
        printf("Random number: %d\n", x);
    } while (x != 0);  // Repeat if x is not zero

    return 0;
}
```

Listing 12: Do-while loop

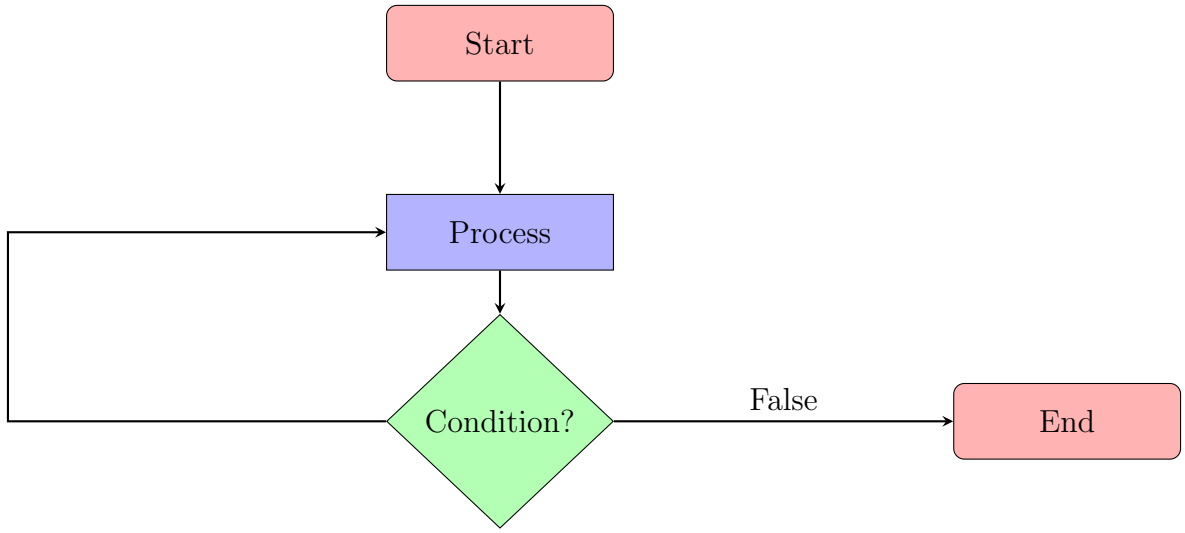


Figure 5: This is an example of a Do-While Loop flowchart

Adding some complexity (Functions, Return Types and Arrays)

Functions are really useful code 'snippets', which we can reuse in other parts of the code or in different classes (in C++). They work by 'interrupting' the compiler's process and completes the function half way through the main() function when it has been called. I have the code below to show some examples.

```c
#include <stdio.h>

// Function with a single return type STEP (1)
int square(int x) {
    // Pure function: always returns same output for same input
    return x * x;
}

// Function demonstrating pass by value STEP (2)
void increment_value(int x) {
    x = x + 1;  // This change does NOT affect the original variable
}

// Function demonstrating pass by reference STEP (3)
void swap_numbers(int *a, int *b) {
    // Uses pointers to modify original variables
    // We will get onto this later
    int temp = *a;  // Dereference to get actual value
    *a = *b;        // Modify first variable
    *b = temp;      // Modify second variable
}

// Function with multiple return values (using pointers) STEP (4)
void calculate_stats(int arr[], int size, int *min, int *max) {
    *min = arr[0];
    *max = arr[0];

    for (int i = 1; i < size; i++) {
        if (arr[i] < *min) *min = arr[i];
        if (arr[i] > *max) *max = arr[i];
    }
}

int main() {
    // Demonstrating function usage STEP (5)
    int result = square(4);  // result is 16
    printf("Square of 4: %d\n", result);

    int num = 10;
    increment_value(num);  // num remains 10
    printf("Original number: %d\n", num);

    int x = 5, y = 10;
    swap_numbers(&x, &y);  // Pass address of variables
    printf("After swap: x = %d, y = %d\n", x, y);

    // Multiple return values example
    int numbers[] = {5, 2, 9, 1, 7};
    int minimum, maximum;
    calculate_stats(numbers, 5, &minimum, &maximum);
    printf("Min: %d, Max: %d\n", minimum, maximum);

    return 0;
}
```

Listing 13: Function example

Lets go through this step by step on the next page:

Step (1):
This function squares an input value of x, and it is called a 'pure' function. This means the output will be constant for the input the function receives.

Step (2):
This is a special type of function called '`void`'. This is unique because there isn't a `return`. Why would you use something like this? We can use it when we want to create a function that doesn't need to return anything, we could use it to print a statement or modify an array or object when we call this function.

Step (3):
"**passing by value**" means that when a function is called with a variable as an **argument**, a copy of the variable's value is created and passed to the function. The function operates on this copy, and any changes made to the variable within the function do not affect the original variable outside the function. This is OK when the program is small, but it uses a lot of memory if we have to keep regenerating values.

Step (4):
This is where pointers come in (the next set of notes will focus on them). This is called "passing by reference" means that when a function is called with a variable as an argument, the function operates on the original variable itself, rather than a copy of its value. This means that any changes made to the variable within the function affect the original variable outside the function. We use much less memory and we can organise our code better rather than worry about local variables with the same name.

Step (5):
We have combined all of the principles above to create a running program.

Scope:
We first mentioned this in example (4). What does it actually mean?

Variables declared **inside** a block (e.g., inside a function) are **local** to that block. They cannot be accessed outside the block where they are defined. Local variables are stored in the stack and are created/destroyed when the block is entered/exited. There is an example below to show this.

```c
#include <stdio.h>

void exampleFunction() {
    int x = 10; // Local to exampleFunction
    printf("x inside exampleFunction: %d\n", x);
}

int main() {
    exampleFunction();
    // printf("%d", x); // Error: x is not accessible here
    return 0;
}
```

Listing 14: Void example

File Scope (Global Scope):
Variables declared outside any function are global and can be accessed by all functions in the file. Global variables are stored in the data segment and retain their values throughout the program's execution. There is an example below (Listing 15).

We need to think about scope as this could be the reason why certain functions or variables aren't found by the compiler.

```c
#include <stdio.h>

int globalVar = 100; // Global variable

void exampleFunction() {
    printf("Accessing globalVar in exampleFunction: %d\n", globalVar);
}

int main() {
    printf("Accessing globalVar in main: %d\n", globalVar);
    exampleFunction();
    return 0;
}
```

Listing 15: Issues with scope

In C, the *scope* of a variable determines where it can be accessed in a program. Variables can have global scope (declared outside any function, accessible everywhere), function scope (declared inside a function, accessible only within it), or block scope (declared inside curly braces {}, accessible only within that block). A common problem is *variable shadowing*, where a variable in an inner scope (e.g., block or function) has the same name as one in an outer scope (e.g., function or global), hiding the outer variable and potentially causing confusion.

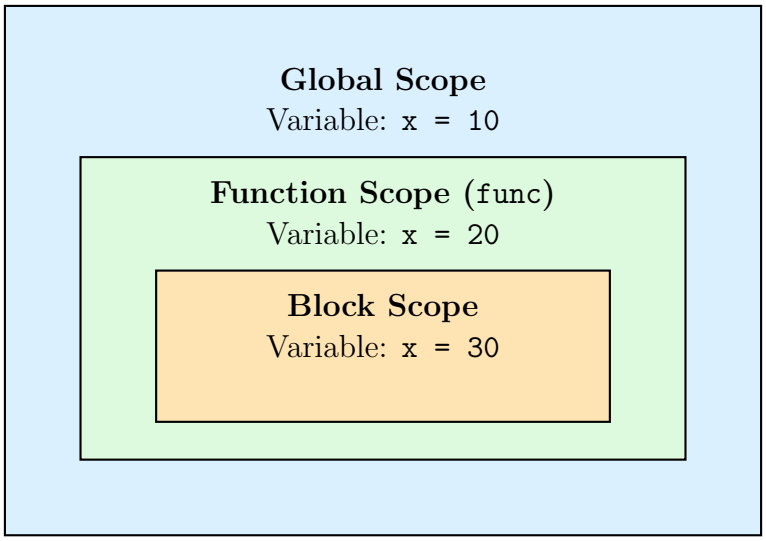
Here is a C program showing variable shadowing:

```c
#include <stdio.h>

int x = 10; // Global scope

void func() {
    int x = 20; // Function scope, shadows global x
    printf("Function scope x: %d\n", x); // Prints 20
    {
        int x = 30; // Block scope, shadows function scope x
        printf("Block scope x: %d\n", x); // Prints 30
    }
    printf("Function scope x after block: %d\n", x); // Prints 20
}

int main() {
    func();
    printf("Global x: %d\n", x); // Prints 10
    return 0;
}
```

Listing 16: Variable shadowing

In this program, the global variable `x` (value 10) is shadowed by the function-scope `x` (value 20) inside `func`, and the function-scope `x` is further shadowed by the block-scope `x` (value 30) inside the block. Each `printf` accesses the `x` from the innermost scope, demonstrating how shadowing hides outer variables.

The following diagram visualises the scope hierarchy as nested boxes, with global scope in blue, function scope in green, and block scope in orange:



The diagram shows how scopes are nested: the global scope contains the function scope, which contains the block scope. Each scope has its own `x`, and the innermost scope's `x` takes precedence, illustrating shadowing.

To avoid scope problems, use unique variable names to prevent shadowing and declare variables in the smallest scope needed (e.g., block scope for temporary variables). Understanding scope helps write clearer, bug-free C programs.

3.1 Intro to Data Structures: Arrays

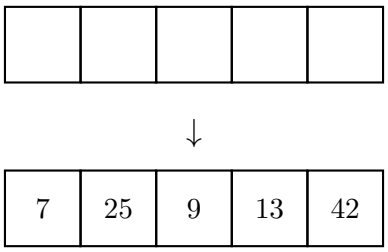
1. Contiguous Memory:
   All elements are stored next to each other in memory. This makes array indexing very efficient because the address of the n-th element can be calculated directly from the starting address of the array.

2. Fixed Size:
   Once an array's size is declared (for example, `int myArray[10]`), the size cannot change at runtime in standard C. You must know the required size beforehand or allocate dynamically using pointers (for example, malloc), but that's a more advanced topic, which I cover in the next set of notes.

3. Indexing:
   C uses zero-based indexing: the first element is `myArray[0]` and the last element is `myArray[size - 1]`. Accessing an array out of its bounds (e.g., myArray[10] when the array size is 10) is undefined behavior and can cause serious program errors.

4. Initialization:
   If you do not initialise elements explicitly, they will typically hold indeterminate (uninitialised) values (for local arrays).
   Global or static arrays are automatically zero-initialized if not explicitly initialized.

5. Memory Layout Each element in the array is laid out in memory one after another. If each element is `sizeof(int)` bytes, then the element `myArray[i]` is located at
   `base_address_of_myArray + i * sizeof(int)`.



In this section, I will give a a lot of questions relating to the content covered in this set of notes.

Most of the questions will be multiple choice or short form (one word) answers as they will most likely appear during the 1st section of the exam. Answers will be on a separate pdf file.

4.1 Multiple Choice and Short Answer Questions

1. Which of the following operators is the bitwise AND operator in C?
   (a) `&`
   (b) `&&`
   (c) `||`
   (d) `^`
2. Which of the following data types is suitable for storing a decimal value (e.g., 3.14)?
   (a) `int`
   (b) `float`
   (c) `char`
   (d) `void`
3. Which keyword is used for a function that does not return any value?
   (a) `int`
   (b) `float`
   (c) `void`
   (d) `char`
4. Which symbol is used to terminate (end) a statement in C?
   (a) `.`
   (b) `;`
   (c) `:`
   (d) `!`
5. Which operator is used to increment a variable by 1?
   (a) `+`
   (b) `++`
   (c) `+=`
   (d) `--`
6. Which of the following is a valid C variable name?
   (a) `1stVariable`
   (b) `count`
   (c) `float`
   (d) `avg-num`

7. Which data type is typically used to store a single character in C?
   (a) `char`
   (b) `int`
   (c) `float`
   (d) `double`
8. Which function is used to display output on the screen in C?
   (a) `scanf()`
   (b) `puts()`
   (c) `printf()`
   (d) `main()`
9. Which of the following is *not* a valid escape sequence in C?
   (a) `\n`
   (b) `\t`
   (c) `\s`
   (d) `\\`
10. Which comparison operator checks for equality between two operands?
    (a) `=`
    (b) `==`
    (c) `!=`
    (d) `===`
11. Which of the following lines is a valid preprocessor directive?
    (a) `#import <stdio.h>`
    (b) `#include <stdio.h>`
    (c) `include <stdio.h>`
    (d) `(include) <stdio.h>`
12. Which statement about global variables in C is **true**?
    (a) They must be declared inside the `main()` function.
    (b) They are visible only within the function that declares them.
    (c) They are accessible to all functions in the same file (and possibly beyond if declared `extern`).
    (d) They cannot be modified once declared.
13. Which of the following best describes what a preprocessor directive does?
    (a) It compiles the code directly.
    (b) It instructs the compiler to link external libraries at runtime.
    (c) It instructs the preprocessor to modify or include code *before* compilation.
    (d) It only handles memory allocation automatically.

14. What is the correct syntax for type casting from `float` to `int` in C?
    (a) `(int) myFloat`
    (b) `[int] myFloat`
    (c) `cast(int) myFloat`
    (d) `int(myFloat)`
15. Consider the following code snippet:

```
#define SIZE 10
int arr[SIZE];
```

    Which statement is **true** about `#define SIZE 10`?
    (a) It declares a global variable named `SIZE`.
    (b) It replaces all instances of `SIZE` with `10` before compilation.
    (c) It allocates memory for `SIZE`.
    (d) It creates a constant pointer to `SIZE`.

Short/One-Word Answer Questions

1. Which keyword is used to create a constant variable in C?
2. What operator is used to access members of a structure via a pointer?
3. Which statement is used to exit a function and optionally return a value?
4. How do you write a single-line comment in C?
5. Which function is commonly used to read input from the user?
6. On most 32-bit systems, how many bytes is an `int` typically?
7. Which operator is used to obtain the memory address of a variable?
8. What arithmetic operator gives the remainder of a division?
9. Which format specifier is used to print an integer using `printf()`?
10. Which header file is required to use `printf()` and `scanf()`?
11. In C, which preprocessor directive is used to conditionally compile sections of code (e.g., only on certain platforms)?
12. Write a single statement to declare a global integer variable named `counter` (initialized to zero).
13. What keyword can be used before a global variable declaration in one file to indicate that it is defined in a different file?
14. How would you cast the integer variable `x` to a double in a mathematical expression (provide the exact syntax)?
15. Which preprocessor directive is used to replace code with a macro definition (e.g., `PI` as 3.14159) before the compilation process?

4.2 Open Ended Coding Questions

1. Simple Calculator

Build a calculator that reads two numbers and an operator from the user, then performs addition, subtraction, multiplication, or division based on the operator entered.
(Hint: look back at **operators**!)

2. Factorial Using Recursion

Write a program that calculates the factorial of a non-negative integer using a **recursive function**.
(Hint: look back at **do-while loops** and **functions**!)

3. Reverse a String (Difficult)

Read a string from the user and print the reversed version of it.
(Hint: **Check string methods and incrementing!**)

4. Global and Local Variables Demonstration

Show the difference between **global** and **local** variables by modifying a global counter from two different functions.
(Hint: Give an example of a simple counting or **incrementing** program with both local and global variables!)

5. Convert Temperature (Type Casting)

Prompt the user for a temperature in Fahrenheit, then display the equivalent temperature in Celsius. Demonstrate type casting to `double` if needed.
(Hint: I would suggest quickly writing down the **control flow** for this.)

6. Check Prime Number

Prompt the user for an integer and determine whether it's prime. (Hint: Similar input structure to Q1 (and come up with a diagram) and think about the **modulus** operator and its meaning.)

4.3 Spot The Error Questions

1. Spot the errors if there are any with this program, then explain where it is and why it causes an error.

```c
#include <stdio.h>

int main() {
    printf("Hello, World!")
    return 0;
}
```

2. Spot the errors if there are any with this program, then explain where it is and why it causes an error.

```c
#include <stdio.h>

int main() {
    int age = "25";
    printf("Age: %d\n", age);
    return 0;
}
```

3. Spot the errors if there are any with this program, then explain where it is and why it causes an error.

```c
#include <stdio.h>

int checkNumber(int num) {
    if (num > 0)
        printf("Positive\n");
    else
        printf("Negative or Zero\n");
}
```

4. Spot the errors if there are any with this program, then explain where it is and why it causes an error.

```c
#include <stdio.h>

int main() {
    int num = 10;
    printf("Value: %f\n", num);
    return 0;
}
```

5. Spot the errors if there are any with this program, then explain where it is and why it causes an error.

```c
#include <stdio.h>

int main() {
    int x;
    printf("Value of x: %d\n", x);
    return 0;
}
```

6. Spot the errors if there are any with this program, then explain where it is and why it causes an error.

```c
#include <stdio.h>

int main() {
    int i = 0;
    while (i < 5) {
        printf("Iteration %d\n", i);
    }
    return 0;
}
```

7. Spot the errors if there are any with this program, then explain where it is and why it causes an error.

```c
#include <stdio.h>

int main() {
    int arr[3] = {1, 2, 3};
    printf("Fourth element: %d\n", arr[3]); // Index out of bounds
    return 0;
}
```

Test
