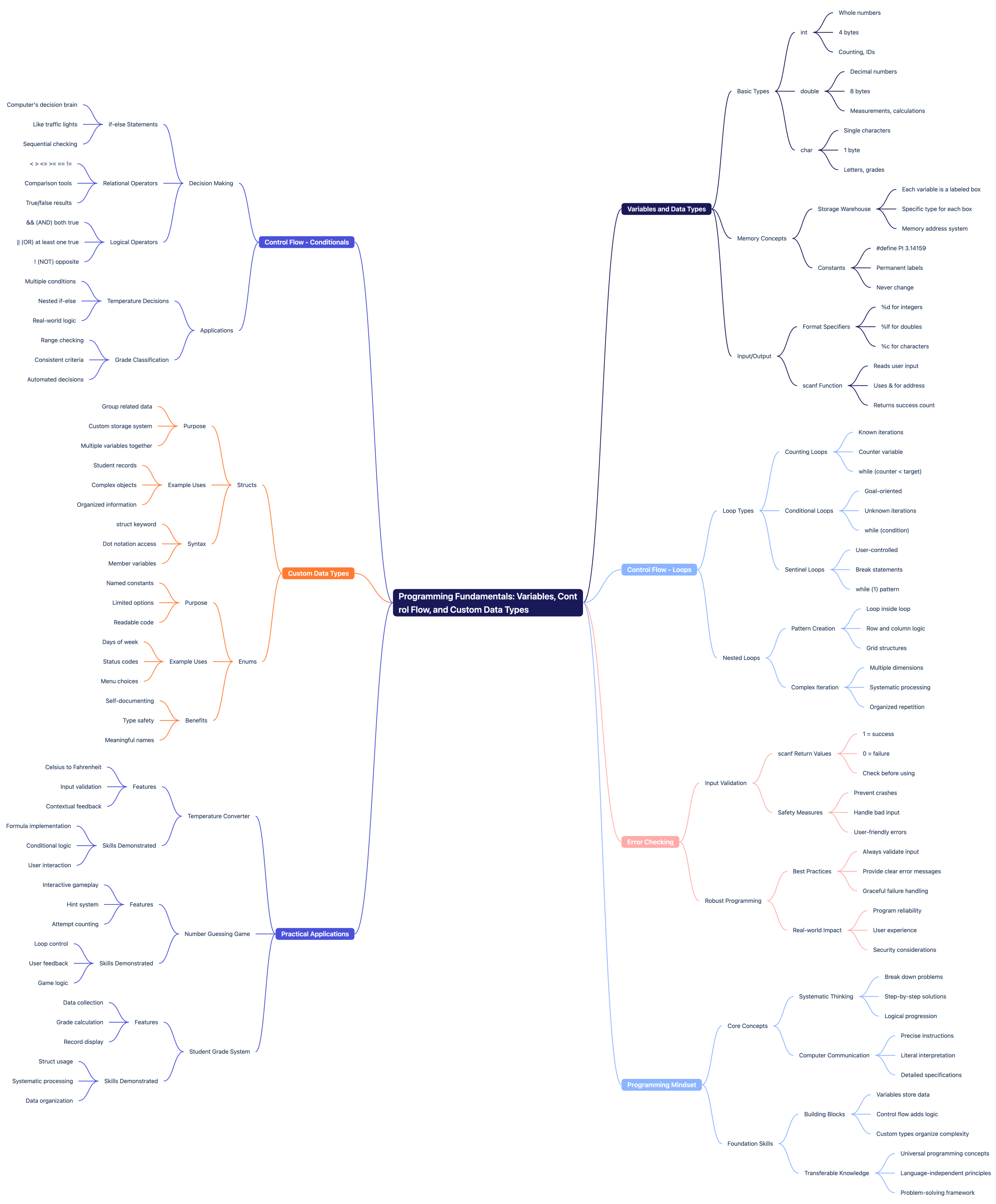
Lecture 2-4: Variables, Control Flow, and Custom Data Types



Why These Topics Matter

These foundational concepts transform simple sequential code into powerful, dynamic programs. Variables let programs store and manipulate data, control flow enables decision-making and repetition, and custom data types help organise complex information. Together, they form the building blocks that appear in virtually every programming language.

Discussion: What's the difference between a calculator and a computer program?

Ask students to think about this - we'll return to it after covering variables and control flow.

1. Variables and Data Types: The Computer's Memory Warehouse

The Computer's Memory System

Imagine your computer's memory as a massive warehouse filled with storage boxes. Each box has a label (the variable name) and can hold a specific type of item. When you create a variable in C, you're essentially reserving a box in this warehouse and telling the computer what kind of stuff you plan to store in it. It's like having a filing cabinet where each drawer is labelled and can only hold certain types of documents - you wouldn't put photos in a drawer marked "receipts."

Discussion: How many different types of data can you think of that a program might need to store?

Basic Data Types Table

Data Type	Size	Range	Example	Use Case
`int`	4 bytes	-2,147,483,648 to 2,147,483,647	`int age = 25;`	Counting, whole numbers
`double`	8 bytes	±1.7×10^308 with 15-17 digits precision	`double height = 175.5;`	Scientific calculations, measurements
`char`	1 byte	-128 to 127 (or 0 to 255 unsigned)	`char grade = 'A';`	Single characters, text

Understanding Data Types

Integer Variables (`int`) are like sturdy metal boxes designed to hold whole numbers - perfect for counting students in a lecture hall, but useless if you want to store someone's height down to the centimetre. Think of counting people in a queue - you can't have 3.7 people, only whole numbers make sense.

Double Variables (`double`) are precision storage containers that can hold decimal numbers with incredible precision. Think of it as the difference between a rough estimate ("about 175-ish") and a precise measurement ("exactly 175.5 centimetres"). When you're cooking and need exactly 2.5 cups of flour, you need that decimal precision.

Character Variables (`char`) are tiny single-letter containers. Computers secretly store numbers that represent each character (ASCII values), but they're polite enough to show you the actual letter when you ask. It's like how your phone stores your contacts as numbers internally, but shows you the actual names when you look them up.

Demo: Let's see what happens when we try to store the wrong type of data.

Constants and Input/Output

```c
// Constants - like permanent labels in your warehouse
// These never change throughout the program
#define PI 3.14159
#define MAX_STUDENTS 100

// Input/Output with format specifiers
// %d = integer, %lf = double, %c = character
printf("Age: %d, Height: %lf, Grade: %c\n", age, height, grade);

// scanf reads input and stores it at the memory address of 'number'
// & means "address of" - tells scanf where to put the data
scanf("%d", &number);
```

Exercise: What will this print?

```c
int x = 5;
double y = 3.14;
char z = 'A';
printf("x=%d, y=%.2lf, z=%c\n", x, y, z);
```

2. Control Flow - Conditional Statements: The Computer's Decision-Making Brain

Decision Making in Programs

Think of conditional statements as your computer's way of being a really pedantic bouncer at a pub. The computer looks at each condition you give it and makes decisions based on very strict rules. Just like how a traffic light makes decisions - if the light is red, you stop; if it's green, you go; if it's yellow, you prepare to stop.

Discussion: What decisions does your phone make automatically? (brightness, notifications, etc.)

Relational and Logical Operators Table

Operator	Meaning	Example	Result
`<`	Less than	`5 < 10`	True
`>`	Greater than	`10 > 5`	True
`<=`	Less than or equal	`5 <= 5`	True
`>=`	Greater than or equal	`5 >= 5`	True
`==`	Equal to	`5 == 5`	True

Operator	Meaning	Example	Result
`!=`	Not equal to	`5 != 10`	True
`&&`	AND (both must be true)	`x > 0 && x < 10`	True if x is 1-9
`||`	OR (at least one true)	`x == 0 || x == 10`	True if x is 0 or 10
`!`	NOT (opposite)	`!(x < 0)`	True if x is not negative

Understanding Conditional Logic

If-Else Statements work like a series of questions you might ask when deciding what to wear. When you wake up and check the temperature, you're essentially running an if-else statement in your head. Your brain automatically thinks: "If it's raining, take an umbrella; else if it's cloudy, maybe take a jacket; else, just a t-shirt is fine."

Relational Operators are the computer's way of comparing things, much like how you might compare prices at different shops. When you're shopping and see "Buy 2, get 1 free" - you're using relational operators to decide if you have enough items in your cart.

Logical Operators are like the computer's way of asking compound questions. The `&&` (AND) operator is like saying "both of these things must be true" - imagine checking if someone can vote: they must be 18 OR OLDER AND be an Australian citizen. It's like checking if you can enter a club: you need to be over 18 AND have valid ID.

Example: Temperature Decision Maker

```c
// This program helps you decide what to wear based on temperature
// It demonstrates how computers can make decisions based on conditions
if (temperature <= 10) {
    // If temperature is 10 or below, it's cold
    printf("It's cold! Wear a jacket.\n");
} else if (temperature < 25) {
    // If temperature is between 11 and 24, it's moderate
    printf("Just right! A jumper should do.\n");
} else {
    // If temperature is 25 or above, it's hot
    printf("It's hot! T-shirt weather.\n");
}
```

Challenge: Write a condition to check if someone is a teenager (13-19 years old).

3. Control Flow - Loops: The Computer's Repetitive Tasks

Repetition in Programming

Loops are like giving your computer a set of instructions for repetitive tasks, much like teaching a very obedient (but not particularly bright) assistant how to do boring jobs. Think of a washing machine - you set it to run a cycle, and it repeats the same washing, rinsing, and spinning actions until the cycle is complete.

Discussion: What repetitive tasks do you do every day that could be automated?

Three Types of While Loops Table

Loop Type	When to Use	Example Scenario	Pattern
Counting	Known number of iterations	Print numbers 1-10	`while (counter < target)`
Conditional	Working towards a goal	Add numbers until sum > 100	`while (total < 100)`
Sentinel	Unknown number of inputs	Process input until -1	`while (1)` with `break`

Understanding Loop Types

Counting Loops are like asking someone to count out loud from 1 to 10. You set up a counter, give them the task, and tell them to keep going until they reach the target. It's like a microwave timer - you know exactly how many seconds it will count down.

Conditional Loops are more like telling someone "keep adding money to the tip jar until we have enough for a pizza." You don't know exactly how many donations it'll take, but you know the condition that needs to be met. It's like filling a bucket with water - you keep pouring until it's full, but you don't know exactly how many cups it will take.

Sentinel Loops are like having a conversation that continues until someone says the magic word "goodbye." You set up an infinite loop but include a secret escape hatch. It's like a vending machine that keeps asking for money until you press the "cancel" button.

Nested Loops are like organizing a filing system where you have to check every folder, and inside every folder, check every document. It's like reading a book - you go through each chapter (outer loop), and within each chapter, you read every page (inner loop).

Example: Pattern Printing

```c
// This program prints a 5x5 grid of asterisks
// Demonstrates nested loops - one loop inside another
int row = 0;  // Track which row we're on
while (row < 5) {  // Outer loop: do 5 rows
    int col = 0;  // Track which column we're on
    while (col < 5) {  // Inner loop: do 5 columns per row
        printf("* ");  // Print an asterisk and space
        col++;  // Move to next column
    }
    printf("\n");  // New line after each row
    row++;  // Move to next row
}
```

Exercise: What pattern would this create?

```
int i = 1;
while (i <= 5) {
    int j = 1;
    while (j <= i) {
        printf("%d ", j);
        j++;
    }
    printf("\n");
    i++;
}
```

4. Custom Data Types: Building Your Own Storage Systems

Organising Complex Data

Custom data types are like designing your own specialised storage solutions when the standard boxes just don't cut it for your specific needs. It's like creating a custom recipe card that has specific sections for ingredients, cooking time, and instructions - instead of just writing everything on a blank piece of paper.

Discussion: What information would you need to store about a student? How would you organise it?

Structs vs Enums Comparison Table

Feature	Structs	Enums
Purpose	Group related data	Define named constants
Memory	Multiple variables bundled together	Single integer with names
Use Case	Student records, complex objects	Days of week, status codes
Example	`struct student { char name; int age; }`	`enum days {MON, TUE, WED}`

Understanding Custom Types

Structs are like creating a custom filing folder for specific types of information. Instead of having separate loose papers floating around for each piece of data, you create a special folder that always contains the same types of information in the same order. It's like a passport - it always has the same sections (name, photo, date of birth, nationality) in the same places, making it easy to find information.

Enums are like creating a custom set of named options for situations where you have a limited list of possibilities. Think of them as designing your own multiple-choice system. It's like the buttons on a washing machine - instead of numbers 1, 2, 3, you have meaningful labels like "Delicate," "Normal," and "Heavy Duty."

Example: Student Management System

```
// Define a custom data type for storing student information
// This groups related data together in one structure
```

```c
struct student {
    char first_initial;  // Student's first name initial
    char last_initial;   // Student's last name initial
    int age;            // Student's age
    double lab_mark;    // Student's lab mark (can have decimals)
};

// Define named constants for days of the week
// Makes code more readable than using numbers 0-6
enum weekdays {MON, TUE, WED, THU, FRI, SAT, SUN};

int main(void) {
    // Create a student variable using our custom type
    struct student alice;

    // Fill in Alice's information
    alice.first_initial = 'A';
    alice.last_initial = 'S';
    alice.age = 20;
    alice.lab_mark = 8.5;

    // Use enum to make day checking more readable
    enum weekdays today = FRI;
    if (today == SAT || today == SUN) {
        printf("Weekend! No classes.\n");
    }
}
```

Challenge: Design a struct for a book. What fields would it need?

5. Error Checking: The Computer's Safety Net

Building Robust Programs

Error checking with scanf is like having a bouncer at a nightclub who checks IDs. When you ask someone to enter their age, you're not just blindly trusting that they'll enter a valid number. It's like a spell-checker in your word processor - it doesn't just accept whatever you type, it checks if the words make sense and warns you about mistakes.

Discussion: What happens when programs don't check for errors? (Think: crashes, security issues)

Error Checking Table

Input	scanf Return Value	What It Means	Action
`"25"`	`1`	Successfully read 1 integer	Process the number
`"abc"`	`0`	Failed to read any integers	Show error message
`"25.5"`	`1`	Read 1 integer (25), ignored decimal	Process 25
`""` (empty)	`0`	No input provided	Show error message

Example: Safe Input Handling

```c
int number;  // Variable to store the user's input
int inputs_read;  // Variable to store how many values scanf successfully read

// Try to read an integer from user input
inputs_read = scanf("%d", &number);

// Check if scanf was successful
if (inputs_read != 1) {
    // scanf failed - user probably entered invalid data
    printf("Error: Please enter a valid number\n");
} else {
    // scanf succeeded - we can safely use the number
    printf("You entered: %d\n", number);
}
```

Exercise: What would happen if we didn't check scanf's return value?

6. Practical Applications: Real-World Scenarios

Temperature Converter: The Helpful Weather Assistant

```c
#include <stdio.h>

int main(void) {
    double celsius, fahrenheit;  // Variables to store temperatures

    // Get temperature from user
    printf("Enter temperature in Celsius: ");
    scanf("%lf", &celsius);

    // Convert Celsius to Fahrenheit using the formula
    fahrenheit = (celsius * 9.0 / 5.0) + 32;

    // Display the conversion result
    printf("%.1lf°C = %.1lf°F\n", celsius, fahrenheit);

    // Provide helpful context based on the temperature
    if (celsius <= 0) {
        printf("Water freezes at this temperature!\n");
    } else if (celsius >= 100) {
        printf("Water boils at this temperature!\n");
    }

    return 0;
}
```

You're a helpful weather station assistant. Someone gives you a temperature in Celsius, and you convert it to Fahrenheit. But you're not just a calculator - you're thoughtful and provide context about what that temperature means.

Question: What other temperature-related decisions could this program make?

Number Guessing Game: The Patient Teacher

```c
#include <stdio.h>
#define SECRET_NUMBER 42  // The number the user needs to guess

int main(void) {
    int guess;      // Store the user's current guess
    int attempts = 0;  // Keep track of how many guesses they've made

    // Keep asking for guesses until they get it right
    while (1) {  // Infinite loop - we'll break out when they guess
correctly
        printf("Enter your guess: ");

        // Try to read their guess and check if it was valid
        if (scanf("%d", &guess) != 1) {
            printf("Invalid input! Game over.\n");
            break;  // Exit the loop if input was invalid
        }

        attempts++;  // Count this attempt

        // Check if they guessed correctly
        if (guess == SECRET_NUMBER) {
            printf("Correct! You got it in %d attempts\n", attempts);
            break;  // Exit the loop - they won!
        } else if (guess < SECRET_NUMBER) {
            printf("Too low! Try higher.\n");
        } else {
            printf("Too high! Try lower.\n");
        }
    }

    return 0;
}
```

You're a patient teacher playing a guessing game. You have a secret number and guide the guesser with helpful feedback, keeping track of attempts and celebrating when they succeed.

Challenge: How could we make this game more interesting? (Random numbers, difficulty levels, etc.)

Student Grade System: The Consistent Registrar

```c
#include <stdio.h>

// Define a structure to hold all student information together
struct student {
    int id;           // Student ID number
    char name_initial; // First letter of student's name
    double mark;      // Student's numerical mark
    char grade;       // Letter grade (F, P, C, D, H)
};

int main(void) {
    struct student s1;  // Create a student record

    // Collect student information from user
    printf("Enter student ID: ");
    scanf("%d", &s1.id);

    printf("Enter name initial: ");
    scanf(" %c", &s1.name_initial);  // Space before %c skips whitespace

    printf("Enter mark: ");
    scanf("%lf", &s1.mark);

    // Apply grading policy consistently - no human bias!
    if (s1.mark < 50) {
        s1.grade = 'F';  // Fail
    } else if (s1.mark < 65) {
        s1.grade = 'P';  // Pass
    } else if (s1.mark < 75) {
        s1.grade = 'C';  // Credit
    } else if (s1.mark < 85) {
        s1.grade = 'D';  // Distinction
    } else {
        s1.grade = 'H';  // High Distinction
    }

    // Display the complete student record
    printf("Student %d (%c): %.1lf%% - Grade %c\n",
           s1.id, s1.name_initial, s1.mark, s1.grade);

    return 0;
}
```

You're a university registrar processing student enrollments. You collect essential information and apply the grading policy consistently, never making mistakes in the criteria.

Interactive Discussion: What are the advantages of automated grading vs human grading?

Summary: The Big Picture

Key Concepts Table

Concept	What It Does	Real-World Analogy	Why It Matters
Variables	Store data in memory	Storage boxes in a warehouse	Programs need to remember information
Control Flow	Make decisions and repeat tasks	Traffic lights and assembly lines	Programs need to be dynamic, not static
Custom Types	Organise related data	Filing systems and labelled containers	Complex problems need structured solutions
Error Checking	Handle unexpected input	Safety nets and validation	Robust programs don't crash on bad input

The Programming Mindset

Programming is really just giving very detailed instructions to a helpful but literal-minded assistant (the computer) who will follow your directions exactly as you give them. The key is breaking down complex problems into simple, systematic steps that a computer can understand and execute reliably. It's like writing a recipe for someone who has never cooked before - you need to be very specific about every step, because they'll follow your instructions exactly as written.

Final Question: Now that we've covered these concepts, what's the difference between a calculator and a computer program?

These foundational concepts form the building blocks for more advanced programming topics and are transferable to virtually all programming languages, making them essential for any programmer's toolkit.