Course Title:
Software Engineering

Submitted By:
Muhammad Furqan
Roll No: **30052**

1.1

Data Types: What They Are and Why They Matter in Programming

Data types define the kind of data a variable can hold, like numbers, text, or true/false values. They're crucial because they help organize memory use, prevent errors, and make code easier to understand. With data types, developers can better manage, manipulate, and secure data, making programming more efficient.

Real-World Examples in Software Development

1. **Integer**: Used for whole numbers. Example: `age = 25`. In a school system, student age is stored as an integer.
2. **Float**: For decimals. Example: `price = 19.99`. In an e-commerce app, prices use floats for precision.
3. **Character**: For single characters. Example: `grade = 'A'`. Schools might store grades as characters.
4. **String**: For text. Example: `name = "Alice"`. Used to store names or addresses.
5. **Boolean**: For true/false. Example: `isLoggedIn = true`. Useful in login systems to track user status.

1.2

Categories of Data Types

1. Primitive Data Types

Basic types built into most programming languages.

- **Integer (`int`)**
  - **Usage**: Holds whole numbers.
  - **Size**: Usually 4 bytes.

```
int age = 25;
```

  -
- **Floating-point (`float`, `double`)**
  - **Usage**: Stores decimal numbers.
  - **Size**: `float` is 4 bytes; `double` is 8 bytes.

```
float price = 19.99;
```

  -
- **Character (`char`)**
  - **Usage**: Holds a single character.
  - **Size**: 1 byte.

```
char grade = 'A';
```

  -
- **Boolean (`bool`)**
  - **Usage**: Represents true or false values.
  - **Size**: 1 byte.

```
bool isActive = true;
```

Derived Data Types

These types are built from primitive types.

- **Array**
    - **Usage**: Stores multiple values of the same type.

Example:

```
int scores[] = {90, 85, 78};
```

- **Pointer**
    - **Usage**: Holds the memory address of a variable.

Example:

```
int x = 10;
int *ptr = &x;
```

-
    -
- **Structure (`struct`)**
    - **Usage**: Groups different data types together.

Example:

```
struct Person {
  char name[50];
  int age;
};
```

-
    -
- **Enumeration (Enum)**
    - **Usage**: Lists named integer constants.
    - Example:
      `enum Day { MON, TUE, WED };`

1.3

Rules and Constraints of Data Types

1. Declaration Rules

Each data type has specific syntax rules for declaring variables.

- **Integer (`int`)**: Declare with `int` keyword, e.g., `int age = 25;`
- **Float (`float`)**: Declare with `float`, e.g., `float price = 19.99;`
- **Character (`char`)**: Use `char`, e.g., `char grade = 'A';`
- **Boolean (`bool`)**: Use `bool`, e.g., `bool isActive = true;`
- **Array**: Declare type followed by `[]`, e.g., `int scores[5];`

2. Type Limits and Overflow Issues

Each data type has limits, and exceeding these can cause overflow, leading to unpredictable results.

Integer Overflow: For a typical 4-byte `int`, the range is $-2,147,483,648 to 2,147,483,647$. Exceeding this range wraps around, causing the value to become negative.

- **Floating-Point Precision**: Floating-point numbers like `float` or `double` have limited precision. Small rounding errors can accumulate in calculations, especially with very large or very small numbers.

1.4

Type Conversion (Typecasting)

Type conversion is the process of converting a value from one data type to another. It can be classified into two main categories: implicit conversion and explicit conversion.

1. Implicit Conversion

- **Definition**: This type of conversion occurs automatically by the compiler without the programmer's intervention.
- **When It Happens**: Typically happens when you mix different data types in an expression. The compiler promotes the smaller type to a larger type to prevent data loss.
- **For Example:**

```
int num = 10;

float result = num + 5.5; // num is implicitly converted to float

// result is 15.5 (float)
```

2. Explicit Conversion (Typecasting)

- **Definition**: This conversion requires the programmer to specify the desired type using type casting syntax.
- **When It's Used**: When you want to convert a variable to a specific type, particularly when you know it won't lead to data loss.
- **For Example**:

```
double pi = 3.14;

int intPi = (int)pi; // Explicitly converting double to int

// intPi is 3 (fractional part is discarded)
```

Key Points

- **Implicit conversion** is safe and convenient but can lead to unexpected results if not understood.
- **Explicit conversion** gives the programmer control but should be used carefully to avoid losing important data.

1.5

Usage of Constants

Constant Variables

Constant variables, or simply constants, are variables whose values cannot be changed once they are defined. They are useful for representing fixed values that remain the same throughout the program's execution, enhancing code readability and maintainability. Constants help prevent accidental modifications and make the code easier to understand.

Why Constants Are Useful

1. **Readability**: Using `PI` makes the formula clearer than using `3.14159` directly.
2. **Maintainability**: If the value of `PI` needs to change (e.g., more precision), you only need to update it in one place.
3. **Error Prevention**: Prevents accidental modifications to important values throughout the code, reducing the risk of bugs.

1.6

Common Errors Related to Data Types

1. Type Mismatch Errors

Type mismatch errors occur when a variable is assigned a value that doesn't match its declared data type. For example, if you try to assign a float value to an integer variable, the compiler will raise an error or produce unexpected behaviour.

Example:

```
int num;

num = 3.14; // Error: assigning a float to an int
```

In this case, trying to assign `3.14` (a float) to `num` (an int) causes a type mismatch error since the data types don't align.

2. Precision Loss During Conversion

When converting between data types, particularly from a float to an int, you can lose precision. This happens because the fractional part of the float is discarded when it's converted to an integer.

Example:

```
float pi = 3.99;

int intPi = (int)pi; // intPi becomes 3
```

Here, converting `3.99` to an int results in `3`, losing the `.99` and thus, some precision.

3. Overflow and Underflow Problems

Overflow and underflow occur when a calculation exceeds the limits of the data type.

Overflow: This happens when a value exceeds the maximum limit of a data type.

Example:

```
int maxInt = 2147483647; // Max value for a 4-byte int

maxInt += 1; // Causes overflow
```

```
// maxInt now becomes -2147483648
```

Underflow: This occurs when a value goes below the minimum limit.

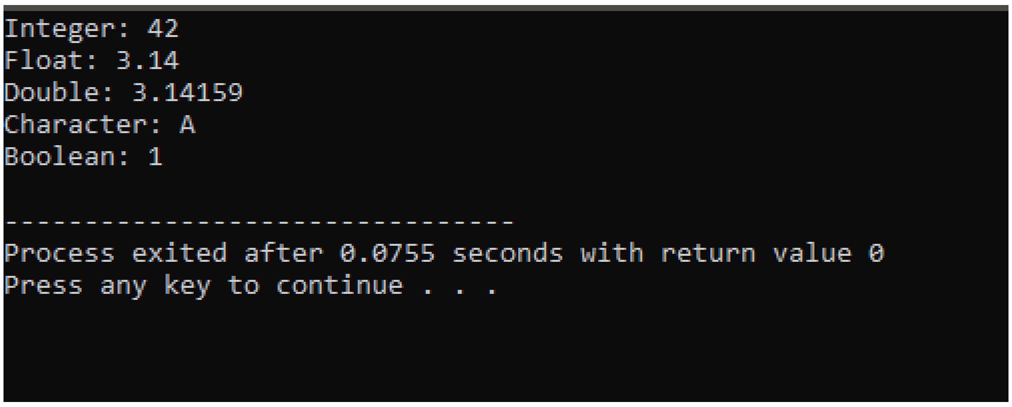
Example:

```
float minFloat = -3.4e38; // Minimum value for a float

minFloat -= 1.0; // Causes underflow

// minFloat might stay at -3.4e38 or become positive (undefined
behaviour)
```

Summary

Understanding these common errors helps programmers avoid pitfalls related to data types, ensuring that code runs perfectly and accurately.

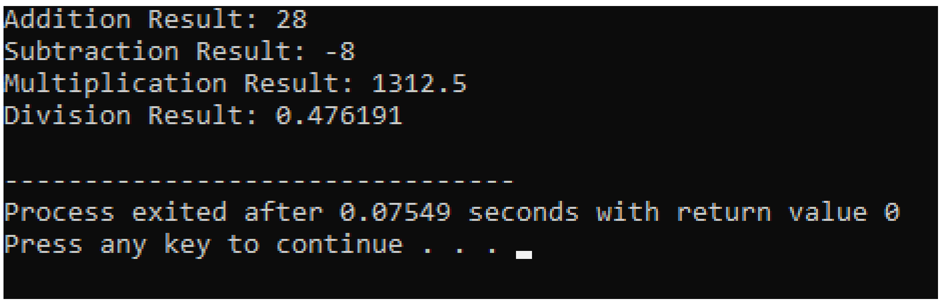
2.1. Task 1: Data Type Declaration and Initialization



```
Integer: 42
Float: 3.14
Double: 3.14159
Character: A
Boolean: 1

--------------------------------
Process exited after 0.0755 seconds with return value 0
Press any key to continue . . .
```

2.2. Task 2: Arithmetic Operations with Different Data Types

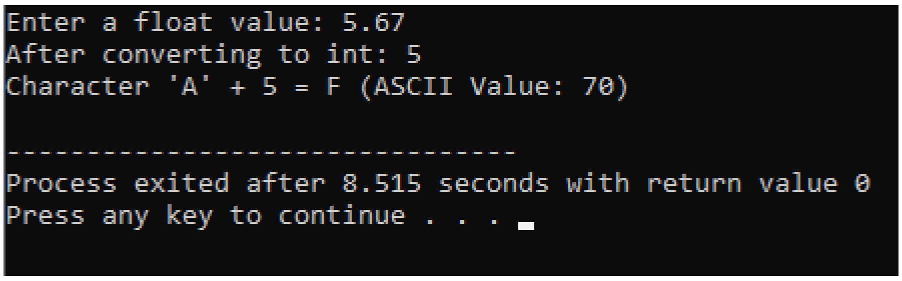
Input



```
Addition Result: 28
Subtraction Result: -8
Multiplication Result: 1312.5
Division Result: 0.476191

--------------------------------
Process exited after 0.07549 seconds with return value 0
Press any key to continue . . .
```

2.3. Task 3: Demonstrate Type Conversion (Typecasting):

Input



```
Enter a float value: 5.67
After converting to int: 5
Character 'A' + 5 = F (ASCII Value: 70)

--------------------------------
Process exited after 8.515 seconds with return value 0
Press any key to continue . . .
```