

Ilesha Ham
Lab 2 Data Representation
CET 3126C
September 28, 2025

Exercise 1: Data Type Sizes and Limits

Write a program that:

1. Declares variables of types: int, short, long, unsigned int, float, double, char.
2. Uses the sizeof() function to print how many bytes each type uses.
3. Uses the built-in limits from <limits.h> and <float.h> to show the smallest and largest values they can hold.

```
#include <stdio.h>
#include <limits.h>
#include <float.h>

int main() {
    // Declare variables of different types
    int intVar;
    short shortVar;
    long longVar;
    unsigned int unsignedVar;
    float floatVar;
    double doubleVar;
    char charVar;

    printf("=== Data Type Sizes and Limits ===\n\n");

    // Print sizes using sizeof()
    printf("Size of int: %lu bytes\n", sizeof(intVar));
    printf("Size of short: %lu bytes\n", sizeof(shortVar));
    printf("Size of long: %lu bytes\n", sizeof(longVar));
    printf("Size of unsigned int: %lu bytes\n", sizeof(unsignedVar));
    printf("Size of float: %lu bytes\n", sizeof(floatVar));
    printf("Size of double: %lu bytes\n", sizeof(doubleVar));
    printf("Size of char: %lu bytes\n\n", sizeof(charVar));

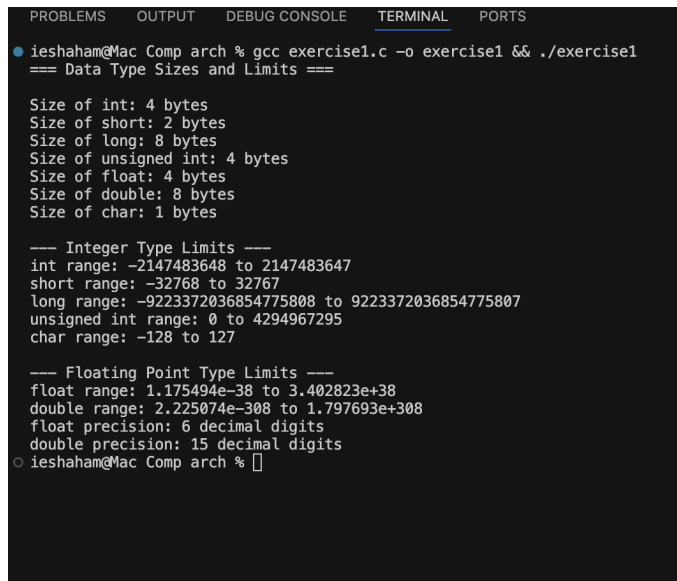
    // Print limits from limits.h and float.h
    printf("--- Integer Type Limits ---\n");
    printf("int range: %d to %d\n", INT_MIN, INT_MAX);
    printf("short range: %d to %d\n", SHRT_MIN, SHRT_MAX);
    printf("long range: %ld to %ld\n", LONG_MIN, LONG_MAX);
    printf("unsigned int range: 0 to %u\n", UINT_MAX);
    printf("char range: %d to %d\n\n", CHAR_MIN, CHAR_MAX);
```

```

printf("--- Floating Point Type Limits ---\n");
printf("float range: %e to %e\n", FLT_MIN, FLT_MAX);
printf("double range: %e to %e\n", DBL_MIN, DBL_MAX);
printf("float precision: %d decimal digits\n", FLT_DIG);
printf("double precision: %d decimal digits\n", DBL_DIG);

return 0;
}

```



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
ieshaham@Mac Comp arch % gcc exercise1.c -o exercise1 && ./exercise1
=== Data Type Sizes and Limits ===

Size of int: 4 bytes
Size of short: 2 bytes
Size of long: 8 bytes
Size of unsigned int: 4 bytes
Size of float: 4 bytes
Size of double: 8 bytes
Size of char: 1 bytes

--- Integer Type Limits ---
int range: -2147483648 to 2147483647
short range: -32768 to 32767
long range: -9223372036854775808 to 9223372036854775807
unsigned int range: 0 to 4294967295
char range: -128 to 127

--- Floating Point Type Limits ---
float range: 1.175494e-38 to 3.402823e+38
double range: 2.225074e-308 to 1.797693e+308
float precision: 6 decimal digits
double precision: 15 decimal digits
ieshaham@Mac Comp arch %

```

Explanation:

This program shows how much memory different basic types in C use and the range of values they can hold. The `sizeof()` operator tells you how many bytes a type takes in memory. The `<limits.h>` and `<float.h>` headers give the smallest and largest values each type can store. Knowing these limits helps avoid errors when a value is too big or too small for a type and helps you pick the right type for your program.

Exercise 2: Type Conversion

Write a program that:

1. Converts an int to a float (implicit conversion)
2. Converts a float to an int using `(int)` (explicit conversion)
3. Converts a char (like 'A') to its ASCII integer value

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

```

int main() {
    printf("=== Type Conversion Examples ===\n\n");

```

```

// 1. Implicit conversion: int to float
int intValue = 42;
float floatResult = intValue; // Implicit conversion
printf("1. Implicit Conversion (int to float):\n");
printf("  Original int value: %d\n", intValue);
printf("  After conversion to float: %.2f\n\n", floatResult);

// 2. Explicit conversion: float to int using cast
float floatValue = 3.14159;
int intResult = (int)floatValue; // Explicit cast
printf("2. Explicit Conversion (float to int):\n");
printf("  Original float value: %.5f\n", floatValue);
printf("  After cast to int: %d\n", intResult);
printf("  Note: Decimal part is truncated, not rounded\n\n");

// 3. Convert char to ASCII integer value
char letter = 'A';
int asciiValue = (int)letter; // Explicit cast
printf("3. Character to ASCII Conversion:\n");
printf("  Character: %c\n", letter);
printf("  ASCII value: %d\n", asciiValue);
printf("  (You can also print without cast: %d)\n", letter);

// Additional examples
printf("\n--- Additional Examples ---\n");
char letter2 = 'Z';
printf("Character 'Z' has ASCII value: %d\n", letter2);

float pi = 3.14159;
printf("Converting %.5f to int gives: %d\n", pi, (int)pi);

return 0;
}

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
ieshaham@Mac Comp arch % gcc exercise2.c -o exercise2 && ./exercise2
=== Type Conversion Examples ===
1. Implicit Conversion (int to float):
  Original int value: 42
  After conversion to float: 42.00

2. Explicit Conversion (float to int):
  Original float value: 3.14159
  After cast to int: 3
  Note: Decimal part is truncated, not rounded

3. Character to ASCII Conversion:
  Character: A
  ASCII value: 65
  (You can also print without cast: 65)

--- Additional Examples ---
Character 'Z' has ASCII value: 90
Converting 3.14159 to int gives: 3
ieshaham@Mac Comp arch %

```

Explanation:

This C program demonstrates how different types of data can be converted in C. It first shows implicit conversion, where an integer 42 is automatically converted to a float 42.00. Then it demonstrates explicit conversion, where a float 3.14159 is manually converted to an integer using (int), truncating the decimal part to produce 3. The program also illustrates how characters are stored as numbers by converting 'A' to its ASCII value 65 and 'Z' to 90. Additional examples reinforce these concepts, such as converting the float 3.14159 to an integer again. Overall, the code highlights how C handles type conversion and how characters can be represented numerically.

Exercise 3: Integer Overflow and Precision Loss

Write a program that:

1. Creates an unsigned int variable and sets it to the biggest value (UINT_MAX)
2. Adds 1 to it and prints the new value (it should "wrap around" to 0)
3. Shows how large float values lose precision (add 1.0 to a very large float like 1.0e20)

```
#include <stdio.h>
#include <limits.h>
```

```
int main() {
    printf("=== Integer Overflow and Precision Loss ===\n\n");

    // 1. Integer overflow with unsigned int
    printf("1. Unsigned Integer Overflow:\n");
    unsigned int maxUnsigned = UINT_MAX;
    printf("  UINT_MAX value: %u\n", maxUnsigned);
    printf("  Adding 1 to UINT_MAX...\n");
    maxUnsigned = maxUnsigned + 1;
    printf("  Result after overflow: %u\n", maxUnsigned);
    printf("  Explanation: Value wraps around to 0\n\n");

    // 2. Precision loss with large floats
    printf("2. Floating Point Precision Loss:\n");
    float largeFloat = 1.0e20;
    printf("  Original large float: %.0f\n", largeFloat);
    printf("  Adding 1.0 to it...\n");
    float result = largeFloat + 1.0;
    printf("  Result: %.0f\n", result);
    printf("  Difference detected: ");
    if (result == largeFloat) {
```

```

        printf("NO - the 1.0 was lost!\n");
    } else {
        printf("YES - values differ\n");
    }
    printf("  Explanation: Float has only ~7 significant digits\n\n");

    // Additional demonstration with smaller numbers
    printf("3. More Precision Loss Examples:\n");
    float f1 = 16777216.0f; // 2^24, at the edge of float precision
    float f2 = f1 + 1.0f;
    printf("  16777216.0 + 1.0 = %.1f\n", f2);
    printf("  Are they equal? %s\n", (f1 == f2) ? "YES (precision lost)" : "NO");

    // Show signed integer overflow (undefined behavior, but demonstrative)
    printf("\n4. Signed Integer Overflow (Caution: Undefined Behavior):\n");
    int maxInt = INT_MAX;
    printf("  INT_MAX: %d\n", maxInt);
    printf("  INT_MAX + 1: %d (wraps to negative)\n", maxInt + 1);

    return 0;
}

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
ieshaham@Mac Comp arch % gcc exercise3.c -o exercise3 && ./exercise3
=== Integer Overflow and Precision Loss ===

1. Unsigned Integer Overflow:
   UINT_MAX value: 4294967295
   Adding 1 to UINT_MAX...
   Result after overflow: 0
   Explanation: Value wraps around to 0

2. Floating Point Precision Loss:
   Original large float: 100000002004087734272
   Adding 1.0 to it...
   Result: 100000002004087734272
   Difference detected: NO - the 1.0 was lost!
   Explanation: Float has only ~7 significant digits

3. More Precision Loss Examples:
   16777216.0 + 1.0 = 16777216.0
   Are they equal? YES (precision lost)

4. Signed Integer Overflow (Caution: Undefined Behavior):
   INT_MAX: 2147483647
   INT_MAX + 1: -2147483648 (wraps to negative)
ieshaham@Mac Comp arch %

```

Explanation:

This C program demonstrates integer overflow and floating-point precision loss. It shows that adding 1 to `UINT_MAX` wraps the value to 0, while very large floats like `1.0e20` can't accurately reflect small additions due to limited precision. It also shows that adding 1 to `INT_MAX` can wrap into negative values, highlighting the limits and unexpected behavior of C's numeric types.

Exercise 4: Mixing Data Types in Math

Write a program that:

1. Adds an int and a float, and prints the result.
2. Adds a char (like 'A') and an int, and prints the result.

Explain what happens. (C automatically converts to a larger type when mixing types.)

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

```
int main() {
    printf("=== Mixing Data Types in Math Operations ===\n\n");

    // 1. Adding int and float
    printf("1. Adding int and float:\n");
    int intNum = 10;
    float floatNum = 3.5;
    float sum1 = intNum + floatNum;
    printf("  int value: %d\n", intNum);
    printf("  float value: %.1f\n", floatNum);
    printf("  Result (int + float): %.1f\n", sum1);
    printf("  Type of result: float\n\n");

    // 2. Adding char and int
    printf("2. Adding char and int:\n");
    char letter = 'A';
    int number = 5;
    int sum2 = letter + number;
    printf("  char value: '%c' (ASCII: %d)\n", letter, letter);
    printf("  int value: %d\n", number);
    printf("  Result (char + int): %d\n", sum2);
    printf("  As a character: '%c'\n", sum2);
    printf("  Explanation: 'A' (65) + 5 = 70 which is 'F'\n\n");

    // Additional examples
    printf("3. More Mixed Type Examples:\n");

    // int + double
    int a = 7;
    double b = 2.5;
    double result1 = a + b;
    printf("  int(7) + double(2.5) = %.1f (promoted to double)\n", result1);

    // int / int vs int / float
    int x = 7, y = 2;
    float z = 2.0;
```

```

printf(" int(7) / int(2) = %d (integer division, truncates)\n", x / y);
printf(" int(7) / float(2.0) = %.1f (float division, precise)\n", x / z);

// char arithmetic
char ch1 = 'a';
char ch2 = ch1 + 1;
printf(" 'a' + 1 = '%c' (moves to next letter)\n", ch2);

return 0;
}

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
ieshaham@Mac Comp arch % gcc exercise4.c -o exercise4 && ./exercise4
=== Mixing Data Types in Math Operations ===

1. Adding int and float:
  int value: 10
  float value: 3.5
  Result (int + float): 13.5
  Type of result: float

2. Adding char and int:
  char value: 'A' (ASCII: 65)
  int value: 5
  Result (char + int): 70
  As a character: 'F'
  Explanation: 'A' (65) + 5 = 70 which is 'F'

3. More Mixed Type Examples:
  int(7) + double(2.5) = 9.5 (promoted to double)
  int(7) / int(2) = 3 (integer division, truncates)
  int(7) / float(2.0) = 3.5 (float division, precise)
  'a' + 1 = 'b' (moves to next letter)
ieshaham@Mac Comp arch %

```

Explanation:

This C program demonstrates mixing different data types in mathematical operations. It shows that adding an int to a float produces a float result, while adding a char to an int uses the ASCII value of the character. It also illustrates type promotion in operations like int + double and the difference between integer division (truncates decimals) and float division (precise). Finally, it demonstrates character arithmetic, like moving from 'a' to the next letter by adding 1.