CS12 CH:Floats, Memory, and Input

Data Types, Memory Size, and User Input

Mr. Gullo

September 15, 2025

Learning Objectives

- Understand and use the `float` data type for numbers with decimals.
- Differentiate between integer division and floating-point division.
- Define fundamental memory concepts: Bit and Byte.
- Use the `sizeof()` operator to determine the memory footprint of various data types.
- Understand binary (base-2) numbers and how to represent them in C++.
- Use `cin` to get input from a user via the console.

Key Concept: Floating-Point Numbers

- In programming, numbers with decimal parts are called floating-point numbers.
- C++ provides the `float` data type to store these values.
- Declaration and initialization is similar to integers:

Example Syntax

```
float pi = 3.14159;
float price = 0.95;
```

- Floats support standard arithmetic operations: addition, subtraction, multiplication, and division.
- Modulo division (%) is not supported for floating-point types.

Essential Equations: Integer vs. Float Division

The data type of your numbers dictates the type of division C++ performs.

- **Integer Division**: If both operands are integers, the result is an integer. Any fractional part is truncated (cut off).
  -
    $$\frac{5}{4} \to 1$$

Essential Equations: Integer vs. Float Division

The data type of your numbers dictates the type of division C++ performs.

- **Integer Division**: If both operands are integers, the result is an integer. Any fractional part is truncated (cut off).
  - $$\frac{5}{4} \to 1$$
- **Floating-Point Division**: If at least one operand is a float, the result is a float, preserving the decimal.
  - $$\frac{5.0}{4} \to 1.25$$

Essential Equations: Integer vs. Float Division

The data type of your numbers dictates the type of division C++ performs.

- **Integer Division**: If both operands are integers, the result is an integer. Any fractional part is truncated (cut off).
  - $$\frac{5}{4} \to 1$$
- **Floating-Point Division**: If at least one operand is a float, the result is a float, preserving the decimal.
  - $$\frac{5.0}{4} \to 1.25$$

This is one of the most common sources of bugs for new programmers!

Code Demo: Division in Action

Demo File: `03_intDivision.cpp` (Interactive - comprehensive demo)
Let's examine how C++ handles different division scenarios.

```
#include <iostream>

using namespace std;

int main()
{
   float a = 5;
   float b = 4;
   float c = 5/4; // Integer division occurs *before* assignment!

   cout << "5/4 = " << 5/4 << endl;         // Integer division
   cout << "c = " << c << endl;             // Result of prior integer division
   cout << "5.0/4 = " << 5.0/4 << endl;     // Floating-point division
   cout << "5/4.0 = " << 5/4.0 << endl;     // Floating-point division
   cout << "a/b = " << a/b << endl;         // Floating-point division (vars)

   return 0;
}
```

Key Concepts: Bits and Bytes

All data in a computer is stored as binary digits, or bits.

- **Bit**:
  - The smallest unit of data in a computer.
  - Can have a value of either 0 or 1.

Key Concepts: Bits and Bytes

All data in a computer is stored as binary digits, or bits.

- **Bit**:
  - The smallest unit of data in a computer.
  - Can have a value of either 0 or 1.
- **Byte**:
  - A group of 8 bits.
  - A common unit for measuring computer memory size.
  - One byte can represent 256 different values (from 0 to 255).

Context: Visualizing a Byte

The terms "bit" and "byte" can be abstract. To make this concrete, the next slide visualizes how 8 individual bits come together to form a single byte, the fundamental unit used to measure the size of data types like `int` and `char`.

Visualization: 8 Bits in 1 Byte

1 Byte

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Figure: A byte is a sequence of 8 bits.

Key Concept: The `sizeof()` Operator

Different data types require different amounts of memory to store their values.

- C++ has a built-in operator called `sizeof()` that tells you how much memory (in bytes) a data type or variable occupies.
- This can vary slightly between computer architectures (e.g., 32-bit vs. 64-bit systems).

Key Concept: The `sizeof()` Operator

Different data types require different amounts of memory to store their values.

- C++ has a built-in operator called `sizeof()` that tells you how much memory (in bytes) a data type or variable occupies.
- This can vary slightly between computer architectures (e.g., 32-bit vs. 64-bit systems).

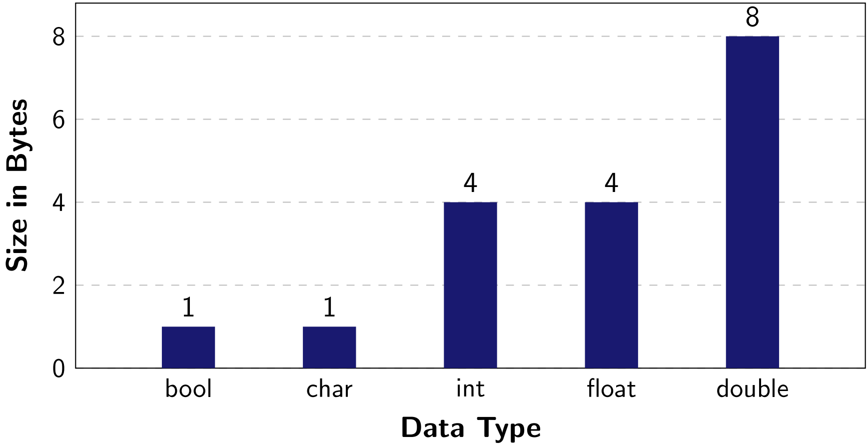
Syntax Examples

```
sizeof(int)      // Returns the size of an integer
sizeof(myAge)    // Returns the size of the variable myAge
```

Context: Visualizing Data Type Sizes

Running a program to see the output of `sizeof()` is useful, but a graph can help us instantly compare the memory footprint of different data types. The next slide shows a bar chart of common data types and their typical sizes in bytes on a 64-bit system. **File:** `03_datatypesSizes.cpp` (Sizes)

Visualization: Typical Data Type Sizes



Key Concept: Binary Numbers

- We typically use the **decimal** (base-10) number system, which has ten digits (0-9).
- Computers use the **binary** (base-2) number system, which has only two digits (0 and 1).
- A number's base indicates how many digits are available.
  - Decimal: $827_{10} = 8 \times 10^2 + 2 \times 10^1 + 7 \times 10^0$
  - Binary: $101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = 5_{10}$

Concept: Binary Literals in C++

- You can write numbers in binary directly in your C++ code by using the `0b` prefix.
- When you print the number, C++ will automatically display it in its decimal (base-10) representation.

Code Example:

```cpp
#include <iostream>

int main()
{
   std::cout << "0b1010011 = " << 0b1010011 << std::endl;
   return 0;
}
```

Concept: Binary Literals in C++

- You can write numbers in binary directly in your C++ code by using the `0b` prefix.
- When you print the number, C++ will automatically display it in its decimal (base-10) representation.

Code Example:

```cpp
#include <iostream>

int main()
{
    std::cout << "0b1010011 = " << 0b1010011 << std::endl;
    return 0;
}
```

Terminal Output

```
0b1010011 = 83
```

Key Concept: Getting Input with `cin`

- To make programs interactive, we need a way to get input from the user.
- In C++, we use the `cin` object (part of `<iostream>`) for this.
- The extraction operator >> is used to get data from the console and store it in a variable.

Key Concept: Getting Input with `cin`

- To make programs interactive, we need a way to get input from the user.
- In C++, we use the `cin` object (part of `<iostream>`) for this.
- The extraction operator >> is used to get data from the console and store it in a variable.

Example: Reading an integer

```
int age; // Declare a variable to store the age

cout << "Please enter your age: "; // Prompt the user
cin >> age; // Read input from the keyboard into 'age'

cout << "You are " << age << " years old." << endl;
```

Why Does Code Formatting Matter?

- **Readability**: Code is read far more often than it is written. Consistent formatting makes it easier for you (and others) to understand what the code is doing.

Why Does Code Formatting Matter?

- **Readability**: Code is read far more often than it is written. Consistent formatting makes it easier for you (and others) to understand what the code is doing.
- **Collaboration**: When working in a team, a shared style guide prevents confusion and makes code reviews more efficient. Everyone is on the same page.

Why Does Code Formatting Matter?

- **Readability**: Code is read far more often than it is written. Consistent formatting makes it easier for you (and others) to understand what the code is doing.
- **Collaboration**: When working in a team, a shared style guide prevents confusion and makes code reviews more efficient. Everyone is on the same page.
- **Maintainability**: It's easier to find bugs and add new features to well-formatted code. Messy code hides problems.

Why Does Code Formatting Matter?

- **Readability**: Code is read far more often than it is written. Consistent formatting makes it easier for you (and others) to understand what the code is doing.
- **Collaboration**: When working in a team, a shared style guide prevents confusion and makes code reviews more efficient. Everyone is on the same page.
- **Maintainability**: It's easier to find bugs and add new features to well-formatted code. Messy code hides problems.
- **Professionalism**: Just like good grammar and spelling in an essay, good formatting is a sign of a careful and professional programmer.

The Golden Rule

Write your code as if the person who has to maintain it is a violent psychopath who knows where you live.

Common C++ Formatting Rules

While style guides vary, most agree on a few key principles:

Bad (Inconsistent)

```
#include <iostream>
int main(){
int x=5;int y=10;
if(x<y){
std::cout<<"x is smaller"<<std::endl;
}
return 0;}
```

Good (Consistent)

```
#include <iostream>

int main() {
    int x = 5;
    int y = 10;

    if (x < y) {
        std::cout << "x is smaller" << std::endl;
    }

    return 0;
}
```

- **Indentation**: Use a consistent number of spaces (e.g., 4) for each level of nesting.
- **Spacing**: Use spaces around operators (‘=‘, ‘+‘, ‘¡‘) to improve readability.
- **Brace Style**: Pick one style for your curly braces ("{}") and stick with it.

The U-P-E-R Problem Solving Method

What is U-P-E-R?

A structured approach to solving programming problems:

- **U - Understand**: Analyze the problem, identify inputs/outputs, and work through examples
- **P - Plan**: Design the logic, identify variables, and create pseudocode
- **E - Execute**: Write the actual code based on your plan
- **R - Review**: Test your code, check for errors, and verify correctness

The U-P-E-R Problem Solving Method

What is U-P-E-R?

A structured approach to solving programming problems:

- **U - Understand**: Analyze the problem, identify inputs/outputs, and work through examples
- **P - Plan**: Design the logic, identify variables, and create pseudocode
- **E - Execute**: Write the actual code based on your plan
- **R - Review**: Test your code, check for errors, and verify correctness

Why Use U-P-E-R?

- Breaks complex problems into manageable steps
- Prevents jumping straight to coding without proper planning
- Encourages systematic testing and debugging
- Builds good programming habits for real-world development

I Do: Grade Calculator - Understand

Problem: Write a program that asks for the total possible score on a test, then calculates and displays the minimum score required to earn grades from A to F based on predefined percentages.

U - Understand the Problem

- **Goal:** Calculate grade cutoffs based on a total score.
- **Inputs:** One integer for the total possible score.
- **Outputs:** Three sentences, each stating the required score for a specific grade (A, B, C-).
- **Example:** If input is 100, output for an A (86%) should be 86. If input is 200, output for an A should be 172.

I Do: Grade Calculator - Plan

P - Plan the Logic

- **Variables:**
  - `int totalScore;` to store user input.
  - Use constants for percentages to avoid "magic numbers":
  - `const int GRADE_A = 86;`
  - `const int GRADE_B = 73;`
  - `const int GRADE_C_MINUS = 50;`
- **Steps (Pseudocode):**
  1. Display a prompt asking for the total possible score.
  2. Read the user's input into the `totalScore` variable.
  3. Calculate the cutoff for an 'A': `totalScore * 86 / 100`.
  4. Print the result for 'A'.
  5. Repeat calculation and printing for 'B' (73%) and 'C-' (50%).

I Do: Grade Calculator - Execute & Review

File: 03_grades.cpp (Answer Key) **E - Execute (Write the Code)**

```
#include <iostream>
using namespace std;
// Grade cutoffs
const int GRADE_A = 86;
const int GRADE_B = 73;
const int GRADE_C_MINUS = 50;

int main() {
   int totalScore;
   cout << "Enter total possible score: ";
   cin >> totalScore;

   cout << "For an A, a mark of " << totalScore * GRADE_A / 100 << " is required." << endl;
   cout << "For a B, a mark of " << totalScore * GRADE_B / 100 << " is required." << endl;
   cout << "For a C-, a mark of " << totalScore * GRADE_C_MINUS / 100 << " is required." << endl;
   return 0;
}
```

I Do: Grade Calculator - Execute & Review

File: 03_grades.cpp (Answer Key) **E - Execute (Write the Code)**

```
#include <iostream>
using namespace std;
// Grade cutoffs
const int GRADE_A = 86;
const int GRADE_B = 73;
const int GRADE_C_MINUS = 50;

int main() {
   int totalScore;
   cout << "Enter total possible score: ";
   cin >> totalScore;

   cout << "For an A, a mark of " << totalScore * GRADE_A / 100 << " is required." << endl;
   cout << "For a B, a mark of " << totalScore * GRADE_B / 100 << " is required." << endl;
   cout << "For a C-, a mark of " << totalScore * GRADE_C_MINUS / 100 << " is required." << endl;
   return 0;
}
```

R - Review and Test

- Compile and run. Does it build without errors?
- Test with example: Input 100. Output is 86, 73, 50. Correct.
- Test with another value: Input 200. Output is 172, 146, 100. Correct.
- What happens if we use floats? The result would be more precise, but here integer truncation is acceptable.

We Do: Arithmetic Sequence - Understand & Plan

Problem (Q5a): Write a code chunk that prompts for n and displays the n^{th} number in the sequence: $11, 15, 19, 23, \ldots$

U - Understand

- **Goal:** Find the value of a term in a sequence.
- **Inputs:** The term number, n.
- **Outputs:** A sentence showing the term and its value.
- **Example:** If $n = 1$, output is 11. If $n = 3$, output is 19.

We Do: Arithmetic Sequence - Understand & Plan

Problem (Q5a): Write a code chunk that prompts for n and displays the n^{th} number in the sequence: $11, 15, 19, 23, \ldots$

U - Understand

- **Goal:** Find the value of a term in a sequence.
- **Inputs:** The term number, n.
- **Outputs:** A sentence showing the term and its value.
- **Example:** If $n = 1$, output is 11. If $n = 3$, output is 19.

P - Plan

- **Variables:** `int n;` for input, `int termValue;` for result.
- **Formula:** The n^{th} term of an arithmetic sequence is $a_n = a + (n - 1)d$.
- Here, first term $a = 11$ and common difference $d = 4$.

We Do: Arithmetic Sequence - Execute & Review

E - Execute the Plan

Based on our plan, how do we translate the formula $a_n = 11 + (n-1) \times 4$ into C++?

```
#include <iostream>
using namespace std;

int main() {
    int n;
    cout << "Enter the term number you want to find: ";
    cin >> n;

    // Calculate the nth term using the formula
    int termValue = _______________________; // What goes here?

    cout << "Term " << n << " is " << termValue << endl;
    return 0;
}
```

We Do: Arithmetic Sequence - Execute & Review

E - Execute the Plan

Based on our plan, how do we translate the formula $a_n = 11 + (n - 1) \times 4$ into C++?

```
#include <iostream>
using namespace std;

int main() {
    int n;
    cout << "Enter the term number you want to find: ";
    cin >> n;

    // Calculate the nth term using the formula
    int termValue = ________________________; // What goes here?

    cout << "Term " << n << " is " << termValue << endl;
    return 0;
}
```

R - Review

You Do: Arithmetic Series Sum

Problem (Q5b): Write a code chunk that prompts for n and displays the *sum* of the first n terms in the series: $2 + 5 + 8 + 11 + \ldots$

Your Task: Use the U-P-E-R Method

1. **Understand**: What are the inputs/outputs? Work out an example for $n = 3$ (sum should be $2 + 5 + 8 = 15$).
2. **Plan**: What variables do you need? What is the formula for the sum of an arithmetic series? ($S_n = \frac{n}{2}(2a + (n-1)d)$).
3. **Execute**: Translate your plan and formula into C++.
4. **Review**: Test your code with your example case. Does it work?

Homework Submission: 03 First C++ Calculations

Instructions

- Submit your completed Jupyter Notebook file named: `firstnameLastname_floats.ipynb`.

Homework Submission: 03 First C++ Calculations

Instructions

- Submit your completed Jupyter Notebook file named: `firstnameLastname_floats.ipynb`.
- Complete all questions in Parts 1 through 6 from the lesson notebook.

Homework Submission: 03 First C++ Calculations

Instructions

- Submit your completed Jupyter Notebook file named: `firstnameLastname_floats.ipynb`.
- Complete all questions in Parts 1 through 6 from the lesson notebook.
- You are encouraged to collaborate with your peers, but the work you submit must be your own.

Homework Submission: 03 First C++ Calculations

Instructions

- Submit your completed Jupyter Notebook file named: `firstnameLastname_floats.ipynb`.
- Complete all questions in Parts 1 through 6 from the lesson notebook.
- You are encouraged to collaborate with your peers, but the work you submit must be your own.
- Please make an effort to organize and format your document similarly to the sample provided in class.

Homework Submission: 03 First C++ Calculations

Instructions

- Submit your completed Jupyter Notebook file named: `firstnameLastname_floats.ipynb`.
- Complete all questions in Parts 1 through 6 from the lesson notebook.
- You are encouraged to collaborate with your peers, but the work you submit must be your own.
- Please make an effort to organize and format your document similarly to the sample provided in class.

Grading Breakdown

- Content Completion (Parts 1-6): **6 pts**
- Formatting and Structure: **1 pt**

Summary

- The `float` data type is used for numbers with decimal points.
- Division with two integers results in an integer (truncation). If a `float` is involved, the result is a `float`.
- The `sizeof()` operator returns the memory size of a data type in bytes.
- Computers store data using the binary (base-2) system. In C++, you can denote a binary number with the `0b` prefix.
- `cin >> variable;` is the standard way to read user input from the console.
- The U-P-E-R method provides a structured approach to solving programming problems.