Chapter 1: Primitives, Variables, and Memory Layout

This chapter explores the fundamental building blocks of data in C++: the primitive types. We will examine how these types are represented in memory, how to declare and initialize variables of these types, and the crucial distinctions between different kinds of initialization.

1.1 C++ Primitive Data Types

Why have different types? C++ is a statically-typed language, meaning the type of every variable must be known at compile time. This allows the compiler to reason about memory usage and to perform type-checking, which prevents a huge class of bugs. The variety of types reflects a core C++ philosophy: you only pay for what you need. You can choose a small type like `char` or `short` if you have millions of them and need to conserve memory, or a larger type like `long long` when you need a wide range of values.

Integer Types

Integer types are for whole numbers. The C++ standard makes very few guarantees about their exact size. It only guarantees a minimum range, and that `sizeof(short) ⩽ sizeof(int) ⩽ sizeof(long) ⩽ sizeof(long long)`.

Type	Typical Size (Bytes)	Guaranteed Minimum Range	Notes
`short`	2	-32,767 to 32,767	Useful for memory-constrained applications.
`int`	4	-32,767 to 32,767	The "natural" integer size for the target machine's architecture. This should be your default choice for general-purpose integers.
`long`	4 or 8	-2,147,483,647 to 2,147,483,647	On modern 64-bit systems (Linux, macOS), this is usually 8 bytes. On 64-bit Windows, it is surprisingly still 4 bytes.
`long long`	8	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,807	Guaranteed to be at least 64 bits. Use this when you need a very large range.

Signed vs. Unsigned: * **`signed`** (the default): Uses one bit (the most significant bit) as a "sign bit" to represent positive or negative values. An `n`-bit signed integer can represent values from roughly -2^(n-1) to 2^(n-1)-1. * **`unsigned`**: Uses all bits to represent the magnitude of the number. It can only be positive or zero. An `n`-bit unsigned integer can represent values from 0 to 2^n-1. This is useful for things that can never be negative, like counts or indices, and for bitwise operations.

Character Types

- **`char`**: The `char` type has a dual nature. It is an integer type, but it's used to store characters from a character set (like ASCII). Its size is guaranteed to be at least 8 bits. **Important:** Whether `char` is `signed` or `unsigned` by default is implementation-defined! If you need to do arithmetic with characters, you should explicitly use `signed char` or `unsigned char`.
- **`wchar_t`**, **`char16_t`**, **`char32_t`**: For wide characters used to represent character sets like Unicode that don't fit in a single `char`.

Floating-Point Types

Floating-point types store numbers with fractional parts. They trade precision for range.

Type	Typical Size (Bytes)	Precision	Notes
`float`	4	~7 decimal digits	Single-precision. Use when memory is tight and high precision is not required.

Type	Typical Size (Bytes)	Precision	Notes
`double`	8	~15 decimal digits	Double-precision. This should be your default choice for floating-point math.
`long double`	10, 12, or 16	Varies	Extended-precision. Offers more precision at the cost of more memory.

1.2 Type Representation and Endianness

Why does this matter? Understanding how data is physically stored in memory is crucial for low-level programming, debugging, network programming, and writing performant code that uses bitwise operations.

Integer Representation: Two's Complement

For `signed` integers, nearly all modern computers use a representation called **Two's Complement**. To get the two's complement of a positive number, you invert all the bits and add one. This scheme has the advantage that addition and subtraction work the same way for both signed and unsigned numbers, simplifying the CPU's design.

Floating-Point Representation: IEEE 754

This is the universal standard for floating-point math. It defines how a number is split into a sign, an exponent, and a mantissa. **Why you should know this:** It explains why floating-point math is not always precise (some decimal fractions cannot be perfectly represented in binary) and why special values like `NaN` (Not a Number) and `Infinity` exist.

Endianness

What is it? Endianness refers to the byte order used to store a multi-byte number in memory. * **Big-Endian:** The "big end" (most significant byte) comes first. The number `0x12345678` is stored in memory as the byte sequence `12 34 56 78`. * **Little-Endian:** The "little end" (least significant byte) comes first. The number `0x12345678` is stored as `78 56 34 12`.

Why does it matter? Most desktop CPUs (Intel, AMD) are little-endian. Many networking protocols and older CPU architectures are big-endian. When you send binary data over a network or write it to a file, you must know the endianness of the data, or you will interpret it as a completely different number on the other end.

Fixed-Width Integers (`<cstdint>`)

Why use them? The variability of `int` and `long` is a problem for portable code. If you are writing a file format or a network protocol, you need to guarantee that a variable is *exactly* 32 bits, not just *at least* 16. The `<cstdint>` header provides this guarantee.

- `int8_t`, `int16_t`, `int32_t`, `int64_t`: Signed integers of exactly 8, 16, 32, or 64 bits.
- `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`: Unsigned versions.
- `int_least_t`, `int_fast_t`: Provide the smallest type with at least `N` bits, or the fastest type with at least `N` bits, respectively. These are less common but useful for optimization.

1.3 Variable Declaration and Initialization

Why is initialization so important? In C++, failing to initialize a variable is a major source of bugs. For local variables, default initialization means the variable holds an indeterminate, or "garbage," value—whatever happened to be in that memory location before. Reading from an uninitialized variable is **Undefined Behavior**.

- **Declaration:** Introduces a name to the compiler. `extern int x;` tells the compiler "the name `x` exists and it's an `int`, but its definition is in another translation unit."
- **Definition:** A declaration that also allocates storage and may provide an initial value. `int x;` or `int x = 5;` are definitions.

The Forms of Initialization

Syntax	Name	Example	Notes
`T object;`	Default Initialization	`int x;`	For local variables, value is indeterminate (garbage). For global/static variables, it's zero-initialization.
`T object = v;`	Copy Initialization	`int x = 5;`	Simple and familiar. Can be slightly less efficient for class types.
`T object(v);`	Direct Initialization	`int x(5);`	Also simple and clear.
`T object{v};`	Direct List Init	`int x{5};`	**Modern C++ default.** Safest and most consistent.
`T object = {v};`	Copy List Init	`int x = {5};`	Similar to above, used in some contexts.

Uniform Initialization ({}) and The Most Vexing Parse

C++11 introduced **uniform initialization** with curly braces `{}` to create a single, unambiguous initialization syntax.

Why it's better: 1. **Prevents Narrowing Conversions:** This is a major safety feature. The compiler will error if you try to initialize a variable with a value it cannot represent, which can hide bugs. `{.cpp}  int x = 3.14; // Bad: Compiles, but `x` silently becomes 3.  // int y{3.14}; // Good: Error! Narrowing conversion is forbidden.` 2. **Solves the "Most Vexing Parse":** This is a famous C++ ambiguity. The syntax for declaring a function can look exactly like the syntax for defining a variable.

```
    // Is this a variable `my_object` of type `MyClass` initialized with a default constructor?
    // OR is it a declaration for a function named `my_object` that takes no arguments
    // and returns a `MyClass` object?
    MyClass my_object(); // <-- This is a function declaration!
    ```
    Uniform initialization removes this ambiguity entirely:
    ``` {.cpp}
    MyClass my_object{}; // Unambiguously a default-initialized object.
    ```
```

## Projects for Chapter 1

### Project 1: The Type Inspector

- **Problem Statement:** Write a C++ program that, for every fundamental type (`char`, `short`, `int`, `long`, `long long`, `float`, `double`, `bool`, etc., in both `signed` and `unsigned` variants), prints out:
  1. The size of the type in bytes, using the `sizeof` operator.
  2. The minimum and maximum representable values for that type.
- **Core Concepts to Apply:** `sizeof` operator, using the `&lt;limits&gt;` header and the `std::numeric_limits` class template.
- **Hint:** `{.cpp}  std::cout << "int size: " << sizeof(int) << " bytes, range: " << std::numeric_limits<int>::min() << " to " << std::numeric_limits<int>::max() << std::endl;`

### Project 2: Endianness Detector

- **Problem Statement:** Write a function `std::string get_endianness()` that programmatically determines if the machine it's running on is Big-Endian or Little-Endian and returns the appropriate string.
- **Core Concepts to Apply:** Understanding memory layout, pointers, and type casting.
- **Hint:** `{.cpp}  Create a 32-bit unsigned integer uint32_t x = 0x01020304;  Create a pointer to it, but cast that pointer to a uint8_t*.  Dereference the character pointer to inspect the value of the *first byte* in memory.  If it's 0x04, the machine is Little-Endian. If it's 0x01, it's Big-Endian.`

### Project 3: Initialization Explorer

- **Problem Statement:** Create a program that demonstrates the difference between uninitialized and initialized variables. Declare a local `int` without initializing it and print its value. Then, declare another local `int` and initialize it to zero using list initialization (`{}`). In a loop, declare an uninitialized `int` and print its value several times to see if it changes. Discuss in comments why reading from the uninitialized variable is dangerous.
```

- **Core Concepts to Apply:** Default initialization, zero initialization, undefined behavior.
- **Hint:** {.cpp} The compiler might warn you about using an uninitialized variable. This is a good thing! You may need to use a specific flag to compile it, or just observe the warning and understand why it's there.