Gorgo C for C++ Programmers

April 11, 2026

9. Numbers and Casting

To the CPU, everything is just a number. It has no concept of characters, strings, pointers, or objects. The CPU only knows about the numeric values in registers and memory addresses.

The types we assign to variables in C do not change the underlying bits; they simply tell the *compiler* how we want to interpret and use those numbers. Different numeric types provide different sizes (which determine the range of values they can hold) and different semantics (like whether they are signed or unsigned, integer or floating-point).

Everything is a Number

A `char` is just a small integer (usually 8 bits). Assigning `'A'` is exactly the same as assigning the number `65` (in ASCII). You can perform math on characters just as you would on any integer:

```c
#include <stdio.h>

int main(void) {
    char grade = 'A';
    int score = 65;

    // Both variables hold the exact same numeric value
    printf("grade as char: %c, as int: %d\n", grade, grade);
    printf("score as char: %c, as int: %d\n", score, score);

    char next_grade = grade + 1; // 65 + 1 = 66
    printf("Next grade: %c\n", next_grade); // 'B'
```

```
    return 0;
}
```

A pointer is also just a number. Specifically, it is an integer that represents a memory address. When you provide a pointer type like `int *`, you are telling the compiler: "Treat this address number as the location of an `int`." C allows you to print the address just like a number, usually formatted as hexadecimal using the `%p` specifier for pointers:

```
int val = 1986; // Year "Danger Zone" charted
int *p = &val;

printf("Address: %p\n", (void *)p); // e.g., 0x7ffd9b8
```

Strings are Not Special

Because everything is just a number, C has no native understanding of "strings." A string in C is merely an array of characters (small integers) that ends with a special `0` byte (the null terminator, `'\0'`).

When you write a string literal like `"Africa"`, the compiler simply lays out an array of numbers (`65, 102, 114, 105, 99, 97, 0`) in read-only memory and gives you a `char *` pointer to the first number (`65`). The standard libraries (`<string.h>`) are what give us the illusion of strings, by processing these arrays of numbers until they hit that `0` byte.

```
#include <stdio.h>

int main(void) {
    char word[] = "Hola";

    printf("String: %s\n", word);
    printf("Bytes: ");
    for (int i = 0; i < (int)sizeof(word); i++)
        printf("%d ", word[i]);
    printf("\n");

    return 0;
}
// Output:
// String: Hola
// Bytes: 72 111 108 97 0
```

The five bytes are the ASCII values for `H`, `o`, `l`, `a`, and the null terminator.

Converting Strings to Numbers

Because strings are really arrays of characters, the text `"1986"` is entirely different from the integer `1986`. Unsurprisingly, converting between the text representation of a number and its purely numeric form is a very common task.

To translate a string representation into an actual integer, use `strtol` (string to long) from `<stdlib.h>`:

```
long strtol(const char *str, char **endptr, int base);
```

The first argument is the string to parse. The second argument, `endptr`, is an optional pointer that `strtol` sets to point to the first character after the parsed number (pass `NULL` if you don't need it). The third argument is the numeric base (10 for decimal, 16 for hex, 0 to auto-detect from the prefix).

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

```c
int main(void) {
    char *year_str = "1986";

    // 10 is the base (base-10 decimal)
    long year = strtol(year_str, NULL, 10);

    printf("Year: %ld\n", year);
    return 0;
}
```

For floating-point conversions, use `strtod` (string to double) in the same way — it takes a string, an optional `endptr`, but no base argument since floating-point literals are always decimal.

Later on when we discuss Standard I/O, we will cover `sprintf` and `sscanf`, which provide convenient formatting routines to convert back and forth between strings and numbers.

Integer Types

Integers are the most common type of number you will use in C. The CPU can do arithmetic on integers very quickly, and integers are also used for pointer arithmetic and array indexing. Integers can be signed or unsigned and can be of different sizes. Here is a table of the common sizes, ranges, and literal suffixes of integers on most 64-bit systems:

Type	Bytes	Range	Suffix
`signed char`	1	-128 to 127	
`unsigned char`	1	0 to 255	
`short`	2	$-32{,}768$ to $32{,}767$	
`unsigned short`	2	0 to $65{,}535$	
`int`	4	-2^{31} to $2^{31}-1$	
`unsigned int`	4	0 to $2^{32}-1$	`U`
`long`	8	-2^{63} to $2^{63}-1$	`L`
`unsigned long`	8	0 to $2^{64}-1$	`UL`
`long long`	8	-2^{63} to $2^{63}-1$	`LL`
`unsigned long long`	8	0 to $2^{64}-1$	`ULL`

Note that the sizes and ranges specified here can vary. The C standard's rules about sizes are so vague they aren't worth quoting here :'(. Notice that the range is 1 number larger for negative numbers than for positive numbers. This is because most systems use **two's complement** representation for signed integers. The top bit indicates the sign: if it is set, the number is negative. But the remaining bits are not a simple magnitude — two's complement encodes negative values so that addition and subtraction work the same way for signed and unsigned numbers. A consequence of this encoding is that there is one more negative value than positive values, and zero is represented only one way (with the sign bit clear). While this may not make your math teacher happy, it gets worse when we talk about floating-point numbers!

You can explore the actual sizes and ranges on your machine using `<limits.h>`:

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

int main(void) {
    printf("char:      %zu byte,  range %d to %d\n",
           sizeof(char), CHAR_MIN, CHAR_MAX);
    printf("short:     %zu bytes, range %d to %d\n",
```

```
           sizeof(short), SHRT_MIN, SHRT_MAX);
    printf("int:       %zu bytes, range %d to %d\n",
           sizeof(int), INT_MIN, INT_MAX);
    printf("long:      %zu bytes, range %ld to %ld\n",
           sizeof(long), LONG_MIN, LONG_MAX);
    printf("long long: %zu bytes, range %lld to %lld\n",
           sizeof(long long), LLONG_MIN, LLONG_MAX);

    return 0;
}
```

Trap: Most integer types are always signed by default, but `char` is different — its signedness is implementation-defined. Sadly, on x86_64 CPUs, `char` is signed by default, and on ARM CPUs, `char` is unsigned by default! Watch out!

Integer Promotion

When you use an integer type in an expression that is equal to or less than the size of `int`, it is automatically promoted to `int` if `int` is large enough to hold all the values of the type. Otherwise, it is promoted to `unsigned int`. This is called integer promotion.

For expressions that involve larger integer types, the rules generally promote to the signedness and the size of the larger type.

Trap: One place where this can cause problems is when using `strlen`. `strlen` returns a `size_t` which is an unsigned integer type. If you subtract two `size_t` values, the result will be a `size_t`. If the first value is smaller than the second value, the result will be a large positive number. This can cause problems when using the result in other expressions.

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

int main(void) {
    char *a = "Jump";       // strlen = 4
    char *b = "Jump!!!!";   // strlen = 8

    // size_t is unsigned, so 4 - 8 wraps around!
    size_t diff = strlen(a) - strlen(b);
    printf("strlen(a) - strlen(b) = %zu\n", diff);

    // cast to a signed type to get the correct result
    long sdiff = (long)strlen(a) - (long)strlen(b);
    printf("Signed difference:      %ld\n", sdiff);

    return 0;
}
// Output:
// strlen(a) - strlen(b) = 18446744073709551612
// Signed difference:      -4
```

Floating Point Types

Floating point numbers are used for decimal numbers that may have fractional components. They also vary in size and precision, but they are always signed. Here is a table of the common sizes, ranges, and literal suffixes of floating point numbers on most 64-bit systems:

Type	Size	Range	Literal Suffix
`float`	4 bytes	-3.4e+38 to 3.4e+38	`F`
`double`	8 bytes	-1.7e+308 to 1.7e+308	(*none, default*)
`long double`	16 bytes	-1.2e+4932 to 1.2e+4932	`L`

Floating point numbers also have a sign bit, but unlike integers, the sign bit can be set on zero. This means floating point numbers have two zeros: positive zero and negative zero. (They compare as equal to each other though.) Floating point numbers can also be used to represent infinity and NaN (Not a Number).

Floating point numbers cannot always exactly represent a value. The fractional part of a number is stored as a sum of negative powers of 2 (1/2, 1/4, 1/8, ...). Values like `0.5` (which is 1/2) and `0.25` (which is 1/4) can be represented exactly, but values like `0.1` and `0.2` cannot. Rounding is used to get as close as possible to the value you want. This can cause surprising results: for example, `0.1 + 0.2` is not exactly equal to `0.3`.

Observe the following code:

```c
#include <stdio.h>
int main(void) {
    float f = 1.2;
    if (f != 1.2) printf("what?!?\n");
    if (f == 1.2f) printf("ok\n");
    return 0;
}
```

You would expect only the `ok` message to print, but the `what` message prints too! Why? The literal `1.2` is a `double`. When it is assigned to `f`, the value is rounded to fit in a `float`, losing some precision. In the first `if`, `f` is promoted back to `double` for the comparison, but the precision lost during the assignment is not recovered, so `f` and `1.2` are not equal. In the second `if`, `1.2f` is a `float` literal, which was rounded the same way, so the comparison succeeds.

Casting

A **cast** is a way of forcing the compiler to treat a value of one type as another type. The syntax is `(type) value`.

Casts in C are much simpler than in C++. They are also much less magical! C has a single unified cast syntax `(type)value`. You can only cast between scalar types (integers, floating-point, and pointers). Remember that `char` is an integer type. The following table summarizes the allowed casts:

From / To	Integer	Floating-Point	Pointer
Integer	Yes	Yes	Yes
Floating-Point	Yes	Yes	No
Pointer	Yes	No	Yes

Casting from floating point to integer drops the decimal part (it does not round). As long as the numbers fit into the target type, the conversion works well. But if the number is too large, you get the dreaded "undefined behavior" — anything can happen.

Casting from a floating point to a pointer doesn't make sense, but you can do it if you cast to an integer type first. Why would you ever want to do that though? Keep your coworkers happy and don't do it.

By using a cast, you are telling the compiler: "I know what I am doing, suppress any warnings, and just treat this as the type I specified."

```c
double pi = 3.14159;
int roughly_pi = (int)pi; // truncates to 3
```

Because C trusts you implicitly, casting can be dangerous. Magic *does not* happen when you cast.

Trap: A classic beginner mistake is trying to convert a string to an integer by casting the pointer.
```c
char *movie_year = "1985";  // The Goonies
int bad_year = (int)movie_year; // THIS IS A BUG! not 1985!!
```
Casting a `char *` string to an `int` does not convert the text `"1985"` into the number `1985`. It tells the compiler to take the *memory address* where the string is stored and chop or pad it to fit inside an `int`. On a 64-bit system, the pointer is 8 bytes and the `int` is 4 bytes, so compiling this code will actually result in a warning that you are casting a pointer to an integer of different size! Always use functions like `strtol` to parse strings into numbers.

Casting pointers to other pointers

`void *` is a pointer that can point to anything. It is a pointer to a generic memory location. This is why you don't have to cast pointers returned by `malloc` to a specific pointer type.

Take care when casting pointers to other pointer types. You must understand the memory layout of the structures you are working with. One common pattern is using a `char *` for byte-level arithmetic on a base address plus an offset, and then casting the result to the desired structure pointer type.

```c
#include <stdio.h>

int main(void) {
    int nums[] = {1984, 1985, 1986, 1987};

    void *vp = nums;            // any pointer converts to void *
    int *ip = (int *)vp;        // cast back to use it
    printf("First: %d\n", ip[0]);

    // byte-level access with char *
    char *bp = (char *)nums;
    printf("First byte of nums[0]: 0x%02x\n", (unsigned char)bp[0]);

    return 0;
}
// Output:
// First: 1984
// First byte of nums[0]: 0xc0
```

Try It: Numbers Starter

This program exercises the key concepts from this chapter: characters as numbers, string-to-number conversion, integer sizes, and casting.

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

int main(void) {
    // Characters are just numbers
    char ch = 'A';
    printf("'%c' is %d\n", ch, ch);
    printf("'%c' + 3 = '%c' (%d)\n", ch, ch + 3, ch + 3);
```

```c
    // Strings are arrays of numbers
    char title[] = "Rio";
    printf("\"%s\" bytes: ", title);
    for (int i = 0; i < (int)sizeof(title); i++)
        printf("%d ", title[i]);
    printf("\n");

    // String to number conversion
    char *bpm_str = "120";
    long bpm = strtol(bpm_str, NULL, 10);
    printf("\"%s\" as a number: %ld\n", bpm_str, bpm);

    // Hex string to number
    long color = strtol("FF8000", NULL, 16);
    printf("0x%lX = %ld\n", color, color);

    // Integer sizes on this machine
    printf("\nsizeof(char)  = %zu\n", sizeof(char));
    printf("sizeof(short) = %zu\n", sizeof(short));
    printf("sizeof(int)   = %zu\n", sizeof(int));
    printf("sizeof(long)  = %zu\n", sizeof(long));
    printf("UINT_MAX      = %u\n", UINT_MAX);

    // Casting: float to int truncates
    double tempo = 120.7;
    int whole = (int)tempo;
    printf("\n(int)%.1f = %d\n", tempo, whole);

    // The classic trap: casting a pointer
    char *year_str = "1982";
    long bad  = (long)year_str;          // address, not 1982!
    long good = strtol(year_str, NULL, 10);
    printf("(long)\"1982\"        = %ld (an address!)\n", bad);
    printf("strtol(\"1982\", ...) = %ld\n", good);

    // void * round-trip
    int val = 42;
    void *vp = &val;
    int *ip = (int *)vp;
    printf("\nvoid * round-trip: %d\n", *ip);

    int hex_val = 0xbadd00d;
    printf("%d %x\n", hex_val, hex_val);
    // let's look at the bytes of hex_val (int has 4 bytes)
    unsigned char *raw = (unsigned char *)&hex_val;
    printf("little-endian: %02x %02x %02x %02x\n",
           raw[0], raw[1], raw[2], raw[3]);

    return 0;
}
```

Key Points

- Under the hood, everything (characters, pointers) is just a number.
- C does not have native "strings," only arrays of numbers ended with a `0`.
- The type of a variable tells the compiler how you intend to interpret its numeric bits.
- C casts `(type) value` assert your intent to the compiler. It assumes you know what you are doing.
- Casting a pointer to an integer simply gives you the raw numeric memory address, not the parsed contents of a string. Use `strtol` to parse strings.

Exercises

1. **Think about it:** C++ provides several different cast operators (`static_cast`, `reinterpret_cast`, etc.) whereas C provides only one. What are the advantages of C++'s approach over C's single cast syntax?
2. **What does this print?**

   ```c
   char letter = 'C';
   printf("%c %d\n", letter + 2, letter + 2);
   ```

3. **Calculation:** Assuming a 64-bit system where pointers are 8 bytes and `int` is 4 bytes, what is the output of `sizeof("Danger")` and what is the output of `sizeof((int)0)`?
4. **Where is the bug?**

   ```c
   #include <stdio.h>

   int main(void) {
       char *score_str = "100";
       int score = (int)score_str;
       printf("You got a %d percent!\n", score);
       return 0;
   }
   ```

5. **Write a program** that declares a `double` variable with a fractional component and uses casting to separate the integer part from the fractional part. Print both pieces.