

Introduction to Programming (CS 101)

Spring 2024



Lecture 7:

More about data types, floating point

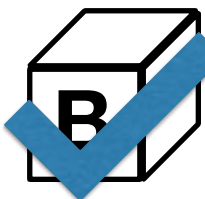
Instructor: Preethi Jyothi

Based on material developed by Prof. Abhiram Ranade and Prof. Manoj Prabhakaran

Recap-I: bool and int types

What is the output of the following program?

 1, 32

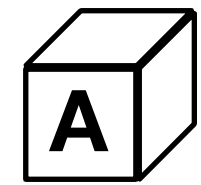
 1, 1

 32, 32

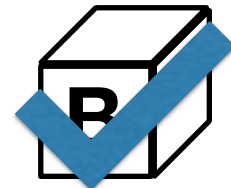
```
main_program {  
    bool b = 32;  
    int i = b;  
    cout << b << ", " << i << endl;  
}
```

Recap-II: char arithmetic

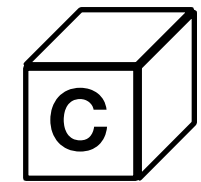
What is the output of the following program?



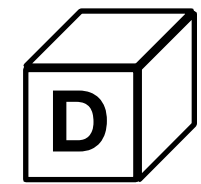
9



18



57 (ASCII code of '9')



48 (ASCII code of '0')

```
main_program {  
    char c = '9' + '9' - '0' - '0';  
    cout << int(c) << endl;  
}
```

Recap-III: Palindrome

Fill in the blanks below to check whether a non-negative integer **n** is a palindrome or not, i.e. is the number the same when read either in forward or backward directions. E.g., 2332 is a palindrome, but 432 is not.

```
main_program {  
    unsigned int n, r = 0;  
    cin >> n;  
  
    for(int q = n; ; q = ) {  
        r = ;  
    }  
  
    if(n == r) cout << "Palindrome" << endl;  
    else cout << "Not Palindrome" << endl;  
}
```

Recap-III: Palindrome

Fill in the blanks below to check whether a non-negative integer **n** is a palindrome or not, i.e. is the number the same when read either in forward or backward directions. E.g., 2332 is a palindrome, but 432 is not.

```
main_program {
    unsigned int n, r = 0;
    cin >> n;

    for(int q = n; q > 0; q = q/10){
        r = (r * 10) + (q % 10);
    }

    if(n == r) cout << "Palindrome" << endl;
    else cout << "Not Palindrome" << endl;
}
```



An aside: lvalues and rvalues

CS 101, 2025

Lvalues and rvalues

- Lvalue always has a defined region of memory, so you can refer to it.
 - Most common example of Lvalue expressions are variable names (including `const` variables). E.g., `int a;`
 - Only an Lvalue can appear on the left side of an assignment. E.g., `a = 5;`
 - `a = b`, `a += b`, and other assignment operations are Lvalue expressions
 - `a, b` (comma expression) where `b` is an Lvalue
- rvalue refer to expressions that evaluate to a value (an rvalue is not an Lvalue!)
 - `bool`, `char`, `int` literals such as `false`, `'A'`, `42`, etc.
 - arithmetic expressions (`a + b`), logical expressions (`a && b`), comparison expressions (`a != b`)
 - `a, b` (comma expression) where `b` is an rvalue
- `++a`, `--a` are Lvalue expressions, but `a++`, `a--` are not. (More about this when we learn about references.)

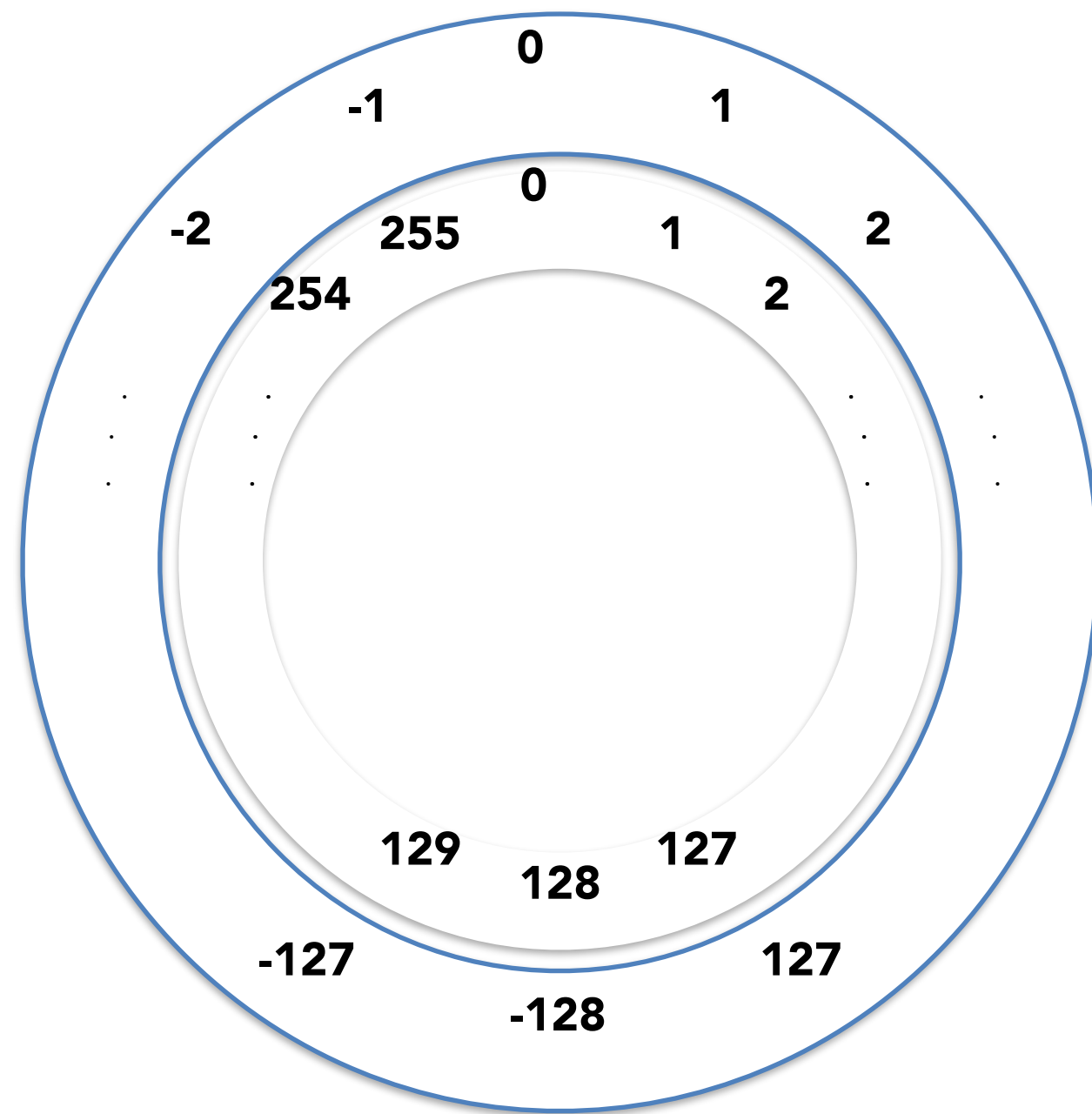


Data types and their representations (contd)

CS 101, 2025

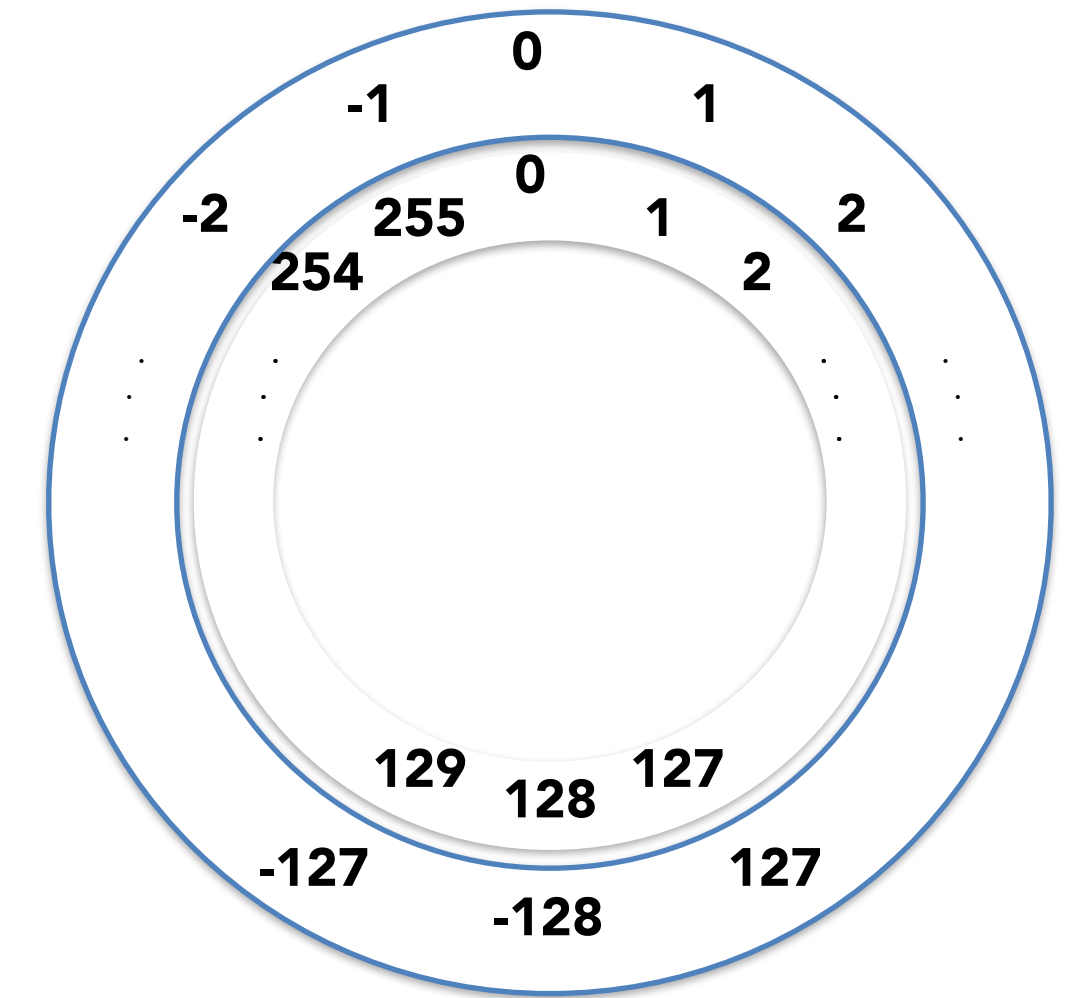
Recap: Binary representations, `bool`, `int`

- Binary representation using `n` bits (two's complement format for signed numbers). Recall the *circular* schematic to help visualize this format.



Recap: Binary representations, `bool`, `int`

- Binary representation using n bits (two's complement format for signed numbers). Recall the *circular* schematic to help visualize this format.
- The bit representation $b_{n-1} \dots b_1 b_0$ stands for the signed number: $-b_{n-1} * 2^{n-1} + \dots + b_1 * 2^1 + b_0 * 2^0$ (note the sign applies only to the most significant bit b_{n-1})
- Type casting: You can explicitly *cast* one data type to another.
 - E.g. `int x; bool b = bool(x);`
 - `int` can be converted to `char` by mod 256
- Bitwise operations (`&`, `|`, `~`, `^`) act bitwise on byte sequences



int: Literal formats

- `int` literal is an integer constant written in a program. E.g., `int a = -1, b = 4;`
- `int` literals are typically in decimal. But literals can also be represented in binary, by starting the literal with `0b`. E.g., `0b11010 == 26`
- `int` literals can also be represented in octal (base 8, starting with `0`) or hexadecimal (base 16, starting with `0x`) formats
 - `032 == 26` ($d_{n-1} * 8^{n-1} + \dots + d_1 * 8^1 + d_0 * 8^0$)
 - `0x1a == 26` ($d_{n-1} * 16^{n-1} + \dots + d_1 * 16^1 + d_0 * 16^0$)
- Hexadecimal format allows for compact representation of long binary sequences.
 - The byte `00011010` can be written as `0x1a` (or `0x1A`)
 - `a, b, c, d, e, f` in hexadecimal format correspond to the numbers `10, 11, 12, 13, 14, 15` respectively
- Note: `cin` reads decimal integers

Internal representations are binary

- Binary representation of integers
 - An n-bit binary representation $b_{n-1} \dots b_1 b_0$ stands for the number $b_{n-1} * 2^{n-1} + \dots + b_1 * 2^1 + b_0 * 2^0$
 - Example: Binary **110** represents the number 6 ($1 * 2^2 + 1 * 2^1 + 0 * 2^0$)
- Binary representation of fractions
 - $b_{n-1} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-m}$ stands for the number $b_{n-1} * 2^{n-1} + \dots + b_1 * 2^1 + b_0 * 2^0 + b_{-1} / 2^1 + b_{-2} / 2^2 + \dots + b_{-m} / 2^m$
 - Example: Binary **110.101** represents the number 6 ($1 * 2^2 + 1 * 2^1 + 0 * 2^0$) + $\frac{5}{8}$ ($1/2 + 0/2^2 + 1/2^3$)

float and double type

- **float** represents floating point numbers that correspond to 4 bytes, i.e., 32 bits
- While representing a real number as a floating point number, we will use some bits for precision, and some for scale (both signed)
 - E.g.: In decimal, $7.8234 \times 10^2 = 782.34$ has 5 digits of precision, and its scale given by the exponent 2 is such that it is between 100 and 999
- 1 bit for sign. Precision of 24 bits (23 bits stored, a leading 1 is implicit). Scale stored using 8-bits: 2^{-126} to 2^{127} (two values of the exponent are used for indicating special values such as **nan**, **inf**, etc.).
 - E.g.: In binary, $0b1.1 \times 2^{-3} = 0b0.0011$
- **double** (for double precision floating number) uses 8 bytes i.e., 64 bits
 - 1 bit for sign, 53 bits of precision (52 bits, one implicit), 11 bits for scale.

float literals

- Format for floating point literals (numbers appearing in the programs) and also as used by `cin/cout`
 - E.g.: We write `num E exp` (with no spaces) to mean `num × 10exp`, where `num` can optionally have a decimal point
 - Note: Exponent is for `10`. Also, the number is in decimal.
 - Examples: `314E-2`, `-0.01`, etc. `E` is optional if `.` present.
 - `6.023e23` (can use either `e` or `E`), `1.5e+2` (+ sign is optional)
- By default, the literal is taken to be a `double`. Suffix `f` or `F` to force `float`, if you want.

Working with real numbers

- For the sake of better precision, use `double` instead of `float`
 - Using `double` can be a little less efficient in large applications: more memory needed, and (hence) slower
- When comparing, allow a “tolerance” (and be prepared for false positives)
 - Instead of `a == b`, check that absolute value of `(a - b)` is less than or equal to `epsilon`
 - Choice of the tolerance value `epsilon` will be application dependent

Floating point arithmetic: Precision issues

- Order matters

```
float f = 5e7; // 50 million > 224
```

```
cout << 1 + f - f << endl; // What is the output?
```

```
cout << f - f + 1 << endl; // What is the output?
```

output

Produces 0 as output!

output

Produces 1 as expected

- Internal representations are binary

```
cout << 1 + 0.001F - 1 << endl; // What is the output?
```

```
cout << 1 + 0.03125 - 1 << endl; // What is the output?
```

output

Not equal to 0.001F!

output

Equal to 0.03125!

Bit shift operators

- Recall that the operators `&`, `|`, `^` and `~` can be used for bit-level manipulations
- Bit shift operators `<<` and `>>` operate on an integral type (`char`, `int`, etc.) variable, and takes a number (how much to shift by) as an additional input
- `(a << n)` shifts the bits in `a` by `n` positions to the left; `n` most significant bits fall off, and `n` least significant bits are set to `0`.
 - Essentially `(a << n)` is the same as `a*2*...*2` (`n` times) done more efficiently
- Similarly `(a >> n)` shifts the bits in `a` by `n` positions to the right; `n` least significant bits fall off, and `n` most significant bits are set to `0` (for unsigned or non-negative `a`) or `1` (for negative signed `a`)
 - Essentially `(a >> n)` is the same as `a/(2*...*2)` (`n` times) if `a` is unsigned or non-negative; for negative `a`, division rounds towards `0`, while `>>` rounds away from `0`

Example: << operator

- Demo in class of code to convert an integer to binary format



Next class: Practice Session
CS 101, 2025