Outline

- In this presentation, we will:
  - Define literal data
  - Describe:
    - Integers
    - Characters
    - Strings
    - Floating-point numbers or floats
    - Boolean values

Literal data

- Often we must hard-code values into our programs
  - Such values are called literals—they are literally what they represent
- We have seen:
  - The integer 0
  - A literal phrase of text "Hello world!"
- There are five types of literal data:
  - Integers
  - Characters
  - Strings
  - Floating-point numbers
  - Boolean

Integer literals

- We have seen an integer literal
  return 0;
- 0 or any sequence of decimal digits not starting with a 0 is interpreted as a decimal integer literal
  - The integer can be prefixed with either + or -
    - 0
    - +0
    - -42 42 +42
    - -1023 1023 +1023
    - -1048576 1048576 +1048576
### Integer literals

```cpp
#include <iostream>

int main()
{
    std::cout << "The answer to the ultimate question is ";
    std::cout << 42;
    std::cout << std::endl;
    return 0;
}
```

**Output:**

```
The answer to the ultimate question is 42
```

### Character literals

- Books are a sequence of letters, numbers or punctuation
  - All of these symbols are collectively called *characters*
- There are two common representations of characters:
  - ASCII
  - Unicode


---

### Integer literals

```cpp
#include <iostream>

int main()
{
    std::cout << "The answer to the ultimate question is ";
    std::cout << 42;
    std::cout << std::endl;
    return 0;
}
```

### Character literals

- ASCII is limited to 128 characters stored in one byte
  - `@ ABCDEFGHIJKLMNOPQRSTUVWXYZ\[\]`~
  - 33 code for non-printing control characters (e.g., TAB, BS, CR, BELL, DEL)
- Most keyboards include all 95 printable ASCII characters and some control characters

Character literals

• Unicode is designed to encode most writing systems
  – Unicode 11.0:
    • Contains 137,439 characters
    • Covers 146 modern and historic scripts
    • Symbol sets and emojis

$\pi \ \text{音} \ ^\infty$

Character literals

• Printable characters (those on a keyboard) can be literally encoded in C++ source code by using single quotes:

```cpp
#include <iostream>

int main()
{
    std::cout << 'a';
    std::cout << 'b';
    std::cout << 'c';
    return 0;
}
```

Output:

```
a
b
c
```

Character literals

• Printable characters (those on a keyboard) can be literally encoded in C++ source code by using single quotes:

```cpp
#include <iostream>

int main()
{
    std::cout << 'a' << 'b' << 'c' << std::endl;
    return 0;
}
```

Output:

```
abc
```
Escape sequences

- Problem: How do you store a literal single quote?
  - You cannot use '''?
- Solution: escape sequences
  - An escape character indicates that the next character is interpreted
    - For C++ characters, the escape character is \\
    - The compiler sees ‘\’ but treats it as the single ASCII character for an apostrophe
      - A literal backslash \`
      - The TAB character 't'

End-of-line characters

- The ASCII representation was designed for teletype machines
  - Automated typewriters
  - The carriage return CR control character (‘\r’) moved the printing carriage back to the start of the line
  - The line feed LF control character (‘\n’) rotated the roller for the next line
    - The new-line character
- You needed to send both characters: CR LF

End-of-line characters

- Computer screens automatically go to the start of the next line
  - Unix (and now Linux and macOS) chose LF
  - The classic Mac OS chose CR
  - Microsoft DOS kept both: CRLF
- This causes compatibility and portability issues...

End-of-line characters

- In C, your code depends on the platform:
  
  printf( "Hello world!\n" ); // Unix/Linux/macOS
  printf("Hello world!\r""); // classic Mac OS
  printf("Hello world!\r\n"); // Microsoft

- In C++, the compiler deals with it:
  std::cout << "Hello world!" << std::endl;
String literals

- A sequence of characters is described as a *string of characters*
  - More simply, a *string*
- When we include "Hello world!" directly in our source code,
  we call this a *string literal*
  - That is literally the string of characters to be used
- A string encompasses all characters after the opening double quote
  up to the closing double quote, which **must** be on the same line

Escape sequences

- The escape character for C++ strings is also the backslash:
  ```
  std::cout << "She said \"Hello world!\"";
  She said "Hello world!"
  
  std::cout << "Look in C:\\Users\\dwharder";
  Look in C:\Users\dwharder
  
  std::cout << "Times: t0.1 s t23.4 s t56.789 s t0 s";
  Times: 0.1 s 23.4 s 56.789 s 0 s
  Tab stops
  ```

Floating-point literals

- We cannot store real numbers to arbitrary precision
  - π to 769 digits of precision:
    3.1415926535897932384626433832795028841971693993751058209749445
    923078164062862089986280348253421170679821480865132823066470938446
    095505822317253594081284811174502841027019385211055946642294895493038
    9166428810975659334461284756482337887831462712701909154568569234403486104
    5432664821339360726052914125732458700660631558141488153292920932892542680417153646
    78925916601132050488206465213414691941516694193572703657959919530218261738193
    26117931051158407446273962749567351857527249121229381830194812953676244065664
    308602199496495247372190720179866943702770535217762785238464781486766040513200
    05861271452635808723577134275775896917361787214644490012495543014654598537105079
    2279969289254201995601212902199686803441815981329747713099651870271134999998…
  - We can only store a finite number of digits of precision relative to a
decimal point
    - We call such representations **floating-point**
Floating-point literals

• Any sequence of decimal digits that has a decimal point (period) somewhere is considered a floating-point literal
  – Can be prefixed by either a + or -

- .42  0.42  4.2  +42.
+ .1023  -0.1023  -10.23  1023.
   .1048576  +0.1048576  +1048.576  -1048576.

Floating-point literals

• You can represent any of these numbers or an integer multiplied by $10^N$ where $N$ is any integer by appending eN:

- .42e5  -0.42e+3  4.2e-5  +42.e12  42e-3
+ .1023e5  +0.1023e+3  -10.23e-5  1023.e12  1023e-3
   .1048576e5  0.1048576e+3  +1048.576e-5  -1048576.e12  1048576e-3

• The most common use of this format is scientific notation:

  6.62607015e-34  2.99792458e8  6.02214076e23

Floating-point literals

• Printing floating-point numbers is different printing other literals:

```cpp
#include <iostream>

int main()
{
  std::cout << "Some floats: ";
  std::cout << 3;
  std::cout << std::endl;
  std::cout << 3.14;
  std::cout << std::endl;
  std::cout << 3.141592653589793;
  std::cout << std::endl;
  return 0;
}
```

Output:

```
Some floats: 
3
3.14
3.14159
```

Boolean literals

• The last category of literal in C++ are Boolean literals:

```cpp
#include <iostream>

int main()
{
  std::cout << true;
  std::cout << std::endl;
  std::cout << false;
  std::cout << std::endl;
  return 0;
}
```

Output:

```
1
0
```
Summary

• After this lesson, you now
  – Understand the idea of literal data in source code
  – Know how to include in your source code literal:
    • Integers
    • Characters
    • Strings
    • Floating-point numbers (reals)
    • Boolean

References

https://en.wikipedia.org/wiki/Literal_(computer_programming)
[2] cplusplus.com tutorial
http://www.cplusplus.com/doc/tutorial/constants/
[3] C++ reference
https://en.cppreference.com/w/c/language/integer_constant

Acknowledgments

Proof read by Dr. Thomas McConkey

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using Consolas.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see https://www.rbg.ca/ for more information.
Disclaimer

These slides are provided for the ECE 150 Fundamentals of Programming course taught at the University of Waterloo. The material in it reflects the authors’ best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.