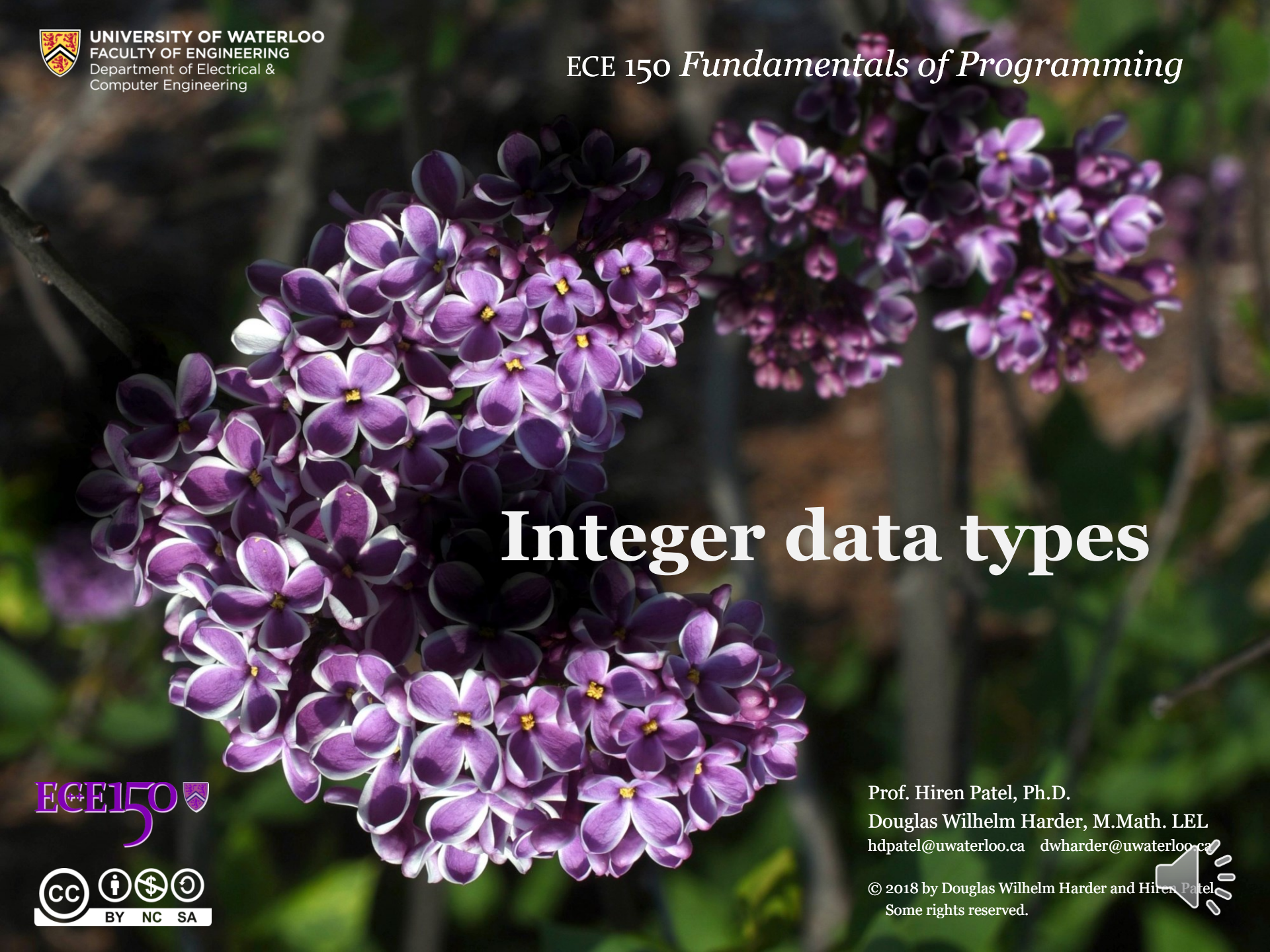


UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

Integer data types

ECE150

Prof. Hiren Patel, Ph.D.
Douglas Wilhelm Harder, M.Math. LEL
hdpatel@uwaterloo.ca dwharder@uwaterloo.ca

Outline

- In this lesson, we will:
  - Review main memory
  - Consider what we can store with *n* decimal digits and *n* bits
  - Describe the storage of `int`
  - Determine the problems with storing negative values
  - Introduce unsigned types

    `unsigned char` `unsigned short` `unsigned int` `unsigned long`
  - Do arithmetic with unsigned integers
  - Introduce 2s complement
  - Do arithmetic with signed integers

What is main memory?

- Main memory is generally volatile memory where any memory location can be accessed as quickly as any other
  - Such memory is called *random access*

- Main memory consists of billions of bits
  - The smallest grouping of bits is a byte consisting of 8 bits
    - All of main memory is divided into bytes

  - A computer with 4 GiB of main memory
    actually has 4 294 967 296 bytes
    - This translates into 34 359 738 368 bits

What can be stored?

- Suppose I only allow you to store three decimal digits:
  - What is the maximum number of values you can store?
- Of course, the answer is 000 through 999,
  or one thousand different numbers
  - This equals 10^3
- Suppose I only allow you to store three bits:
  - What is the maximum number of values you can store?
- We can store `000`, `001`, `010`, `011`, `100`, `101`, `110` and `111`,
  or eight different values
  - This equals 2^3

What can be stored?

- Thus, given n decimal digits, we can store 10^n different values
  - These values range from 0 to $10^n - 1$
  - For example, if $n = 10$,
    we can store values between 0 and 9999999999

- Thus, given n bits, we can store 2^n different values
  - These values range from 0 to $2^n - 1$
  - For example, if $n = 10$,
    we can store values between 0 and $2^{10} - 1 = 1023$
  - These are `0b0000000000` through `0b1111111111`

What is `int`?

- Up to now, we have been using the integer date type `int`
  - Question: how is this stored on the computer?

- Answer:
  - Every local variable or parameter of type `int` occupies `32` bits
  - The compiler decides where the `32` bits will be in main memory

- Recall that when stored in binary,
  a number is represented by a sequence of `0`s and `1`s
  - Allocate four bytes for any local variable or parameter declared to be of type `int`, and then interpret those bits

How is an integer stored?

- With 32 bits,

  we could store 32 coefficients of a binary number:

  $$b_{31}b_{30}b_{29}b_{28}b_{27}b_{26}\cdots b_3b_2b_1b_0$$

- The bit labeled b_k is the coefficient of 2^k
  - This is why we always start with the zeroeth bit on the right
  - If necessary, bits beyond the most significant 1 are zero

How is an integer stored?

- For example, in this program, the local variable `m` is stored as

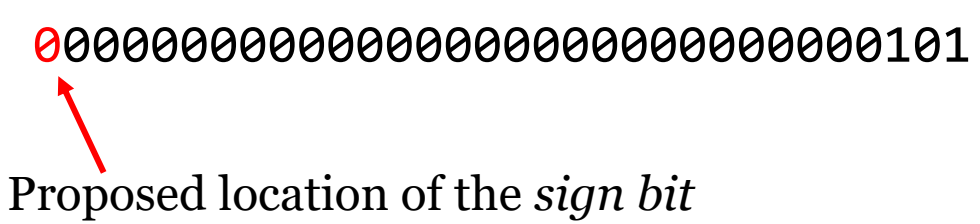
`00000000000000000000000000000101`

```
#include <iostream>

int main();

int main() {
    int m{5};
    std::cout << m << std::endl;

    return 0;
}
```

  - When printed, the 32 bits are interpreted as an integer

How is an integer stored?

- Problem: how do we store negative numbers?
  - We need to store either a "+" or a "–"
  - To do this, we could allocate one bit to store the sign:



  - Our convention could be:
    - If the sign bit is `0`, the number is positive
    - Otherwise, the sign bit is `1`, indicating the number is negative

- Recall, all these are stored in the computer as voltages in a circuit…
  - More in your course on digital circuits and digital computers

unsigned int

- Suppose you are counting events in an embedded system
  - In such cases, you never need negative numbers
- There is a type for this in C++:

```
unsigned int
```

  - Note that the compiler interprets this as a single type
    - The authors of C could have chosen `unsigned_int`, but they chose to add an additional keyword `unsigned`
- All `32` bits of an `unsigned int` are used for storing positive integers:
  - Values between 0 and $2^{32} - 1$, or 0 and 4294967295
  - Think of this as values between 0 and 4 billion

Wasted memory?

- Suppose you know you only need values no larger than 100 or 1000
  - This requires no more than 10 bits
  - Isn't this potentially wasted memory?
    - In an embedded system, this can cause significant issues:
      - More memory requires more cost and power
      - More power requires larger batteries or reduced battery life
      - More memory also produces more heat,
        which requires more cooling

Other integer types

- Thus, C++ has other types:

`unsigned short`	2 bytes or 16 bits	0 and $2^{16} - 1 = 65535$
`unsigned long`	8 bytes or 64 bits	0 and $2^{64} - 1 = 18.5$ quintillion

- Now, some compilers are…peculiar
  - The Microsoft Visual Studio compiler is one such compiler…

    `unsigned long` 4 bytes or 32 bits

  - This is the same as `unsigned int`!
    - To get 64 bits, you must use

      `unsigned long long`

Other integer types

- There is one final integer datatype:

  `unsigned char` 1 byte or 8 bits 0 and $2^8 - 1 = 255$

  – If you ever try to print such an integer,
  it will still try to interpret it as a letter

```
int main() {
    for ( unsigned char k{32}; k < 127; ++k ) {
        std::cout << "I am a char: " << k << std::endl;
    }

    std::cout << "...and not a truck." << std::endl;
}
```

Output:

```
I am a char:
I am a char: !
I am a char: "
I am a char: #
I am a char: $
I am a char: %
I am a char: &
I am a char: '
I am a char: (
I am a char: )
I am a char: *
I am a char: +
I am a char: ,
I am a char: -
I am a char: .
I am a char: /
I am a char: 0
I am a char: 1
      ⋮
I am a char: y
I am a char: z
I am a char: {
I am a char: |
I am a char: }
I am a char: ~
...and not a truck.
```

Other integer types

- Please check your compiler's specifications, or run this code:

```
#include <iostream>
int main();

int main() {
    std::cout << "char:       " << sizeof( char )      << std::endl;
    std::cout << "short:      " << sizeof( short )     << std::endl;
    std::cout << "int:        " << sizeof( int )       << std::endl;
    std::cout << "long:       " << sizeof( long )      << std::endl;
    std::cout << "long long:  " << sizeof( long long ) << std::endl;

    return 0;
}
```

Like `return`, `sizeof` is an operator that evaluates to the number of bytes of the type

Output on my compiler:

```
char:       1
short:      2
int:        4
long:       8
long long:  8
```

Other integer types

- We have now introduced five types that store positive integer values:

`unsigned char`	0 to $2^8 - 1$	`0b11111111`
`unsigned short`	0 to $2^{16} - 1$	`0b1111111111111111`
`unsigned int`	0 to $2^{32} - 1$	
`unsigned long`	0 to $2^{64} - 1$	
`unsigned long long`	0 to $2^{64} - 1$	

Unsigned arithmetic

- All arithmetic is performed modulo 2^n where n is the number of bits
  - The processor just ignores any additional carries:

```
int main() {
    unsigned short m{0b1000001101001101};
    unsigned short n{0b0111111000111110};

    std::cout << "  m   = " << m << std::endl;
    std::cout << "  n   = " << n << std::endl;
    m += n;
    std::cout << "m + n = " << m << std::endl;

    return 0;
}
```

Output:

```
  m   = 33613
  n   = 32318
m + n = 395
```

Arithmetic

- Why did that happen?

```
  1 1 1 1 1   1     1 1 1 1 1
    1000001101001101
+ 0111111000111110
  1000000110001011
```

  - Thus, we see that 33613 + 32318 = 65931
    but we have 1 + 2 + 8 + 128 + 256 = 395
  - Note that $395 + 2^{16} = 65931$

Arithmetic

- The same happens with multiplication:

```
int main() {
    unsigned short m{0b1000001101001101};
    unsigned short n{0b0111111000111110};

    std::cout << "  m   = " << m << std::endl;
    std::cout << "  n   = " << n << std::endl;
    m *= n;
    std::cout << "m * n = " << m << std::endl;

    return 0;
}
```

Output:

```
  m   = 33613
  n   = 32318
m * n = 45734
```

Arithmetic

- Why did that happen?

```
                1000001101001101
              × 0111111000111110
                10000011010011010
               100000110100110100
              1000001101001101000
             10000011010011010000
            100000110100110100000
           1000001101001101000000000
           1000001101001101000000000
          10000011010011010000000000
          10000011010011010000000000
         100000110100110100000000000
       + 100000110100110100000000000
         10000001011111110110010101001110
```

  - Thus, we see that $33613 \times 32318 = 1086304934$ but we have 45734
  - Note that $1086304934 = 16575 \times 2^{16} + 45734$

Arithmetic

- If adding, subtracting or multiplying two unsigned integer types and the result is no longer valid,

  we will say that a *carry* has occurred

- For example,
  - Adding two `unsigned short` and the sum is greater than $2^{16} - 1$
  - Subtracting a larger `unsigned short` from a smaller one
  - Multiplying two `unsigned short`

    and the product is greater than $2^{16} - 1$

Back to negative values

- If these are all unsigned types, then

```
signed char
short
int
long
long long
```

must be signed types

- How do we deal with negative numbers?
  - If the first bit is 1, the number is negative

Largest positive value

- If the first bit is zero, it is a positive value:
  - Thus, the largest positive value for each of these types are:

`signed char`	`01111111`	$2^7 - 1$	127
`short`	`0111111111111111`	$2^{15} - 1$	32767
`int`	`01111111···11111111`	$2^{31} - 1$	2 billion
`long`	`01111111···11111111`	$2^{63} - 1$	8 quintillion

Representing negative numbers

- Why not just using a "1" to indicate a negative number:
  - For example, if 5 is assigned to a variable of type `short`, we have

    `0000000000000101`
  - A short –5 would be stored as

    `1000000000000101`
- First problem:
  - There is now both 0 and –0:

    `0000000000000000`

    `1000000000000000`
- Second problem:
  - It's actually hard to do arithmetic…

2s complement

- The 2s complement notation is actually significantly better:
  - To store a negative number,

    take the positive value and
    - Flip all the bits
    - Add one

2s complement

- Calculating 2s complement:
  - Switch all the bits and add 1:

```
 0000000000000101
        ↓
 1111111111111010
+                1
 1111111111111011

 0000000010110000
        ↓
 1111111101001111
+                1
 1111111101010000
```

2s complement

- Calculating 2s complement:
  - Switch all the bits and add 1:

```
 0000000000000001
        ↓
 1111111111111110
+                1
 1111111111111111

 0111111111111111
        ↓
 1000000000000000
+                1
 1000000000000001
```

2s complement

- A quick and easy way to do this:
  - Switch all bits up to, but not including the right-most "1"

```
0000000010110000
        ↓
1111111101010000

0011010110000000
        ↓
1100101010000000

0000000010110000
        ↓
1111111101010000
```

2s complement

- The same is true for `int`:
  - Flip all bits up to, but not including the right-most "1"

```
00000000000000000000000101 10000
                           ↓
11111111111111111111111010 10000

00100000000000000010101 10000000
                        ↓
11011111111111111101010 10000000
```

2s complement

- Given a negative number, what is its absolute value?
  - Just take the 2s complement, again

- For example,
  - What is the value of this `int`?

    `11111111111111111111111111010110`
  - It is negative, so its positive value is:

    `00000000000000000000000000101010`
  - This number is 2 + 8 + 32 = 42
  - Thus, the original number stored –42

2s complement

- Question:
  - What is the value of:

    `1000000000000000`
  - The 2s complement of this number is

    `1000000000000000`

- Consequently, this stores -2^{15}
  - Thus, we have a slightly different range:

`char`	-2^{7} to $2^{7} - 1$
`short`	-2^{15} to $2^{15} - 1$
`int`	-2^{31} to $2^{31} - 1$
`long`	-2^{63} to $2^{63} - 1$
`long long`	-2^{63} to $2^{63} - 1$

Arithmetic

- How do you add two signed numbers that are in 2s complement
  - Add them as if they were positive integers

Unsigned arithmetic

- Perform addition as if the stored representations were `unsigned`
  - The processor just ignores any additional carries:

```
int main() {

    short m{-0b0111110010110011}; // 1000001101001101
    short n{ 0b0111111000111110};

    std::cout << "  m   = " << m << std::endl;
    std::cout << "  n   = " << n << std::endl;
    m += n;
    std::cout << "m + n = " << m << std::endl;

    return 0;
}
```

Output:

```
  m   = -31923
  n   = 32318
m + n = 395
```

Arithmetic

- Why did that happen?

```
  1 1 1 1 1   1     1 1 1 1 1
    1000001101001101
+ 0111111000111110
  10000000110001011
```

  - Thus, we see that –31923 + 32318 = 395

Unsigned arithmetic

- Perform addition as if the stored representations were `unsigned`
  - The processor just ignores any additional carries:

```
int main() {

    short m{ 0b0111110010110011};
    short n{-0b0111111000111110}; // 1000000111000010

    std::cout << "  m   = " << m << std::endl;
    std::cout << "  n   = " << n << std::endl;
    m += n;
    std::cout << "m + n = " << m << std::endl;

    return 0;
}
```

Output:

```
  m   = 31923
  n   = -32318
m + n = -395
```

Arithmetic

- Why did that happen?

```
          1 1         1
  0111110010110011
+ 1000000111000010
  1111111001110101
```

  - The result is negative, and the absolute value of the result is

    `110001011`

  - This equals 1 + 2 + 8 + 128 + 256 = 395
  - Thus, we see that 31923 + –32318 = –395

Unsigned arithmetic

- Perform addition as if the stored representations were `unsigned`
  - The processor just ignores any additional carries:

```
int main() {

    short m{-0b0111110010110011}; // 1000001101001101
    short n{ 0b0111110010110011};

    std::cout << "  m   = " << m << std::endl;
    std::cout << "  n   = " << n << std::endl;
    m += n;
    std::cout << "m + n = " << m << std::endl;

    return 0;
}
```

Output:

```
  m   = -31923
  n   = 31923
m + n = 0
```

Arithmetic

- Why did that happen?

```
  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1000001101001101
+   0111110010110011
  1 0000000000000000
```

  - Thus, we see that 31923 + –31923 = 0

Arithmetic

- If adding or multiplying two signed integer types and the result is no longer valid,

  we will say that an *overflow* has occurred

- For example,
  - Adding two positive `short` and the sum is greater than $2^{15} - 1$
  - Adding two negative `short` and the sum is less than -2^{15}
  - Adding a positive short to a negative short

    can never result in an overflow
  - Multiplying two `short` and

    the product is greater than $2^{15} - 1$ or less than -2^{15}

Arithmetic

- You are welcome to examine how fascinating 2s complement is
  - For example, try multiplying two integers with opposite signs
  - Try multiplying two negative integers

- To perform subtraction, e.g., `a - b`,
  just take the 2s complement of `b` and add the result to `a`

Summary

- Following this lesson, you now
  - You understand

    `signed char` `short` `int` `long`

    are stored with 1, 2, 4 and 8 bytes
  - Know that each stores a different range of values
    - Each has its unsigned equivalents
  - Know negative numbers are stored using 2s complement
  - Understand that all operations occur as if we ignore any carries beyond the most significant bit

References

[1] Wikipedia:
https://en.wikipedia.org/wiki/Binary_number
https://en.wikipedia.org/wiki/Hexadecimal
https://simple.wikipedia.org/wiki/Hexadecimal_numeral_system

Acknowledgements

Akshat Jawne and Lorena Rosati for noting the use of unsigned integers on Slide 30.

Sami El-Imam for noting had an incorrect sum on Slide 17.

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.