Outline

- In this lesson, we will:
  - Introduce comments
    - The need for comments
    - The economic benefit of comments
  - See how to comment functions
    - Documentary, functional, algorithmic and coding comments
  - Understand where to comment
  - Know how not to comment
  - Describe comment blocks

Comments

- Consider the following function:

```
double sinc_1_16( double x ) {
    return (0.5*x*x + 1.5)*x*x + 1;
}
```

- It's your first month on a co-op work placement:
  - You've been asked to find a bug
  - You track it down to a file containing this peculiar function

Comments

- You check Wikipedia...

Sinc function

In mathematics, physics and engineering, the **cardinal sine function** or **sinc function**, denoted by $\operatorname{sinc}(x)$, has two slightly different definitions.[1]

In mathematics, the historical **unnormalized sinc function** is defined for $x \neq 0$ by

$$\operatorname{sinc}(x) = \frac{\sin(x)}{x}.$$

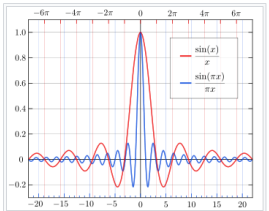
In digital signal processing and information theory, the **normalized sinc function** is commonly defined for $x \neq 0$ by

$$\operatorname{sinc}(x) = \frac{\sin(\pi x)}{\pi x}.$$

In either case, the value at $x = 0$ is defined to be the limiting value $\operatorname{sinc}(0) = 1$.

The normalization causes the definite integral of the function over the real numbers to equal 1 (whereas the same integral of the unnormalized sinc function has a value of π). As a further useful property, all of the zeros of the normalized sinc function are integer values of x.

The normalized sinc function is the Fourier transform of the rectangular function with no scaling. It is used in the concept of reconstructing a continuous bandlimited signal from uniformly spaced samples of that signal.



The normalized sinc (blue) and unnormalized sinc function (red) shown on the same scale.

Comments

- You know this job has something to do with signal processing, so you determine the formula is wrong, and you *fix* it:

```
double sinc_1_16( double x ) {
    return std::sin(3.141592653589793*x)/
                    (3.141592653589793*x);
}
```

- You run the tests, and the bug is fixed
  - You check in your code and you call it a day…
- The next morning, quality assurance reports that some overnight tests took 350% more time than the previous day and one test fails…

Comments

- Your manager explains to you that the function:
  - Approximates the cardinal sine function on the interval [–1.16, 1.16] with an error no greater than 0.021

```
double sinc_1_16( double x ) {
    return (0.5*x*x - 1.5)*x*x + 1;
}
```

  - It also removes the pole at $x = 0$:
    - If you ran your code with the argument `0.0`, you would get $\frac{0}{0}$
    - If you tried printing this out, you would see the output `-nan`
    - The problem is: sinc(0) = 1

Comments

- You discover newly-checked-in code used this function to approximate the sinc function at $x = 1.42$
  - This function was doing exactly what it was meant to do; calculate

$$\tfrac{1}{2}x^4 + \tfrac{3}{2}x^2 + 1$$

  - What is not obvious is "Why is this wrong?"

The economic benefit

- More time is spent on:
  - Debugging
  - Maintaining
  - Extending

  existing code than is ever spent on authoring it
- Without comments, it often takes future developers minutes if not hours trying to understand someone else's code:
  - If you don't comment your code, your developers won't either
  - If your developers don't comment their code, your costs increase
  - If your costs increase, your bonus or likelihood of continued employment decreases

How not to comment...

- Comments explain to the reader what the code is meant to do
  - In C++, in-function comments start with a `//` up to the end of the line

```
double sinc_1_16( double x ) {
    //            1   4    3   2
    // Calculate --- x  + --- x  + 1
    //            2        2
    return (0.5*x*x + 1.5)*x*x + 1;
}
```

Warner Bros.

- This is **the most** useless comment in the world:
  - It repeats in ASCII art what is already obvious

Function comments

```
////////////////////////////////////////////////////////////////////////////
// sinc_1_16
//
// @file      dsp.cpp
// @author    Douglas Wilhelm Harder
// @date      2018-06-19
// @version   1.0
//
// @param x    a value -1.16 <= x <= 1.16
// @return     an approximation of the sinc function on the given
//             interval with an absolute error no larger than 0.02
//
// @section    DESCRIPTION
// This approximation uses a clamped quartic spline that satisfies
// the following conditions:
//      p(-1) = sinc(-1) = 0   p'(-1) = sinc'(-1) =  1
//      p( 0) = sinc( 0) = 1   p'( 0) = sinc'( 0) =  0
//      p( 1) = sinc( 1) = 0   p'( 1) = sinc'( 1) = -1
//
// @todo Requires some optimization...
// @see <a href="https://en.wikipedia.org/wiki/Sinc_function">Wikipedia</a>
////////////////////////////////////////////////////////////////////////////
```

Documentation: the author, creation date, etc.

What the parameters are, what is returned

A description of why this function works

Function comments

- The purpose of comments is to inform the programmer reading the function to understand what is going on

- Comments could be used to describe the
  - Documentary
  - Functional
  - Algorithm
  - Explanatory

Documentary

- Documentation includes:
  - Who was the original author
  - When was the file first written
  - What is the current version number
  - What have been the significant changes made

- Example:

```
// @file      gcd.cpp
// @author    Hiren Patel
// @author    Douglas Wilhelm Harder
// @date      2018-06-19
// @version   1.3
// @since 1.3 Correctly deals with negative arguments
// @since 1.2 Uses 'long' and not 'unsigned long'
// @since 1.1 Fixed bug when one argument is 0
```

Functional

- Functionality includes:
  - What the parameters are
    - Any restrictions on the arguments
  - What is returned and how that is related to the parameters
  - Are there any side effects?

- Example:

```
// @param m    a long integer
// @param n    a long integer
// @returns    the greatest-common divisor (gcd) of the integers m and n
//              - the gcd will always be a positive integer >= 1
```


Algorithmic

- Most functions implement some form of algorithm
  - What is the algorithm being used
  - Are there any modifications?
  - Are there any optimizations that implemented here?
  - What steps, if any, are made specifically to deal with C++ types?
  - Additional details and comments

- Example:

```
// 1. If m or n is negative, make them positive--take the absolute value
// 2. If m = n, gcd(m, n) = m, so return m
// 3. If m < n, swap m and n so that m >= n
// 4. Repeat the following:
//     a. Find a and r such that m = a*n + r
//     b. If r = 0, then gcd( m, n ) = n
//     c. Otherwise, let m take the value n and let n take the value r
```


Explanatory

- In some cases, code can be clever but consequently opaque
  - Throughout the course, we will look at various techniques
  - Sometimes, its difficult to determine why code does what it does
  - If it's obvious, don't comment
  - If it's not obvious, make it clear

- Example:

```
unsigned int mod256( unsigned int n ) {
    return n % 256;
}

unsigned int mod256( unsigned int n ) {
    // The last 8 bits are the number modulo 256
    //  - this is much faster than the modulus operator
    return n & 255;
}
```


Comment location

- Comments describing the function should appear before the function, generally including:
  - Documentation, functionality and algorithm comments

```
////////////////////////////////////////////////////////////////////////
// function_name
//
// @author  Hiren Patel
// @date    ...
// @version ...
//
// @param x    adescription of x...
// @param y    adescription of y...
// @returns    a description of what is returned...
//
// Other comments...
////////////////////////////////////////////////////////////////////////

void function_name( ... ) {
```

Comment location

- In documenting the algorithm, enumerate the steps if there are more than one:

```
////////////////////////////////////////////////////////////////////////
// fastest_route(…)
//      .
//      .
//  Definition:
//       current distance:  the distance from the source to the vertex
//       total distance:    the current distance plus the Euclidean distance
//                          to the final vertex.
//
// Implementing the A* algorithm:
//    1. Allocation of the appropriate memory
//    2. Initializing of the arrays
//    3. While we have not yet reached the target vertex:
//          a. Find that unvisited vertex 'u' with minimum total distance
//          b. Flag it as visited
//          c. For each adjacent unvisited vertex 'v':
//                i.  Calculate the distance to 'u' plus the weight of
//                    the edge connecting 'u' to 'v'
//               ii. If that less than the current distance to 'v',
//                    updated that current distance.
////////////////////////////////////////////////////////////////////////
```


Comment location

- Comments within the function will usually be restricted to
  - A functional description of what the following code does
    - Parameter checking: Are the parameters valid?
    - The core implementation of the algorithm
  - With respect to the algorithm being implemented:
    - Identifying the steps of the algorithm
  - Describing interesting or non-standard aspects of the code

Comment location

```
double fastest_route(…) {
    // Parameter checking
    Some checks…

    // 1. Memory allocation
    Some code…
    // 2. Array initializations
    Some more code…

    // 3. Iterate until we have found the shortest distance to the target vertex by repeatedly finding
    //    the unvisited vertex with minimum distance to it updating distances to its neighbors
    while ( … ) {
        // 3a. Find that unvisited vertex 'u' with minimum total distance
        Find the vertex described…
        // 3b. If this is the target, we're done; otherwise mark it as visited
        Return or mark

        // 3c. For each adjacent vertex, see if we need to update the distance
        for ( … ) {
            // 3c(i)  Calculate the updated distance
            Even more code…
            // 3c(ii) Update the distance if appropriate
            Yet more code…
        }
    }
```


Comment ASCII art

- Often, programmers will engage in artistic attempts to bring attention to their comments
  - In a large file, there may be numerous functions, many with different purposes—it makes sense to organize the file

```
  ////////////////////////////////////////////////////////
 //                Function declarations                 //
////////////////////////////////////////////////////////
```

Don't make your comments hard to update

```
template <typename Type>
bool Beap<Type>::find( Type const &obj ) const {
    if ( empty() ) {
        return false;
    }

    int h = height();
    int posn = h*(h + 1)/2;                                  // Starting at the bottom left

    while ( true ) {
        if ( array[posn] < obj ) {
            if ( posn == (height() + 1)*(height() + 2)/2 - 1 ) {     // Move down and to the right
                return false;                                        //
            }                                                        //           x
                                                                     //         x   x
            if ( (posn == height()*(height() + 1)/2 - 1)             //       < a   x   x
                 && (size() != (height() + 1)*(height() + 2)/2) ) {  //     x   *   x   x
                return false;                                        //   x   x   x   x   x
            }
```

It is very difficult to maintain the correct alignment if new code is being added or code is moved around…

Don't make your comments hard to update

```
            if ( posn + h + 2 < size() ) {
                posn += h + 2;                                   // Move down to the right
                ++h;
            } else {
                ++posn;                                          // Move across to the right
                if ( posn == size() ) {
                    posn -= h;                                   // Move up to the right
                    --h;
                }
            }
        } else if ( array[posn] > obj ) {
            if ( posn == (h + 1)*(h + 2)/2 - 1 ) {               // Move down and to the right
                return false;                                    //
            } else {                                             //           x
                posn -= h;                                       //         *   x
                --h;                                             //       > a   x   x
            }                                                    //     x   x   x   x
        } else {                                                 //   x   x   x   x   x
            return true;
        }
    }
}
```


Don't make your comments hard to update

- No one wants to update the right-hand wall
  - No one will ever update these comments:

```
////////////////////////////////////////////////////////////////////////////
// fastest_route(…)                                                       //
//      .                                                                 //
//      .                                                                 //
//  Definition:                                                           //
//      current distance:  the distance from the source to the vertex     //
//      total distance:    the current distance plus the Euclidean        //
//                         distance to the final vertex.                  //
//                                                                        //
// Implementing the A* algorithm:                                         //
//   1. Allocation of the appropriate memory                              //
//   2. Initializing of the arrays                                        //
//   3. While we have not yet reached the target vertex:                  //
//        a. Find an unvisited vertex 'u' with least total distance       //
//        b. Flag it as visited                                           //
//        c. For each adjacent unvisited vertex 'v':                      //
//             i.  Calculate the distance to 'u' plus the weight of       //
//                 the edge connecting 'u' to 'v'                         //
//            ii. If that less than the current distance to 'v',          //
//                 updated that current distance.                         //
////////////////////////////////////////////////////////////////////////////
```


Comment blocks

- In C++, there is a second form of comment:
  - Everything between a starting `/*` and the next `*/` is a comment
  - `//`-style comments within comment blocks are ignored
  - If `//` are used for documentation, comment blocks can be used during debugging
  - `/* */`-style comments do not nest
- Much more prone to ASCII art:

```
/*        /******************
 *        *
 *        *
 *        *
 *        *
 */       ******************/
```

```
/************************
 * ******************** *
 * *  Section block   * *
 * ******************** *
 ************************/
      /*
      **
      **
      **
      **
      */
```

Summary

- Following this lesson, you now:
  - Understand the need for comments and how to comment
  - Have an understanding the economic benefits
  - Can differentiate between documentary, functional, algorithmic and coding comments
  - Understand that there are reasonable approaches to coding
  - Have seen examples of poor commenting practice can be frustrating
  - Have seen comment blocks

- Important: Commenting is an art-form and a skill, but it is a skill worth learning

References

[1] Bernhard Spuida, *The fine Art of Commenting*, http://www.icsharpcode.net/technotes/commenting20020413.pdf

[2] Wikipedia https://en.wikipedia.org/wiki/Comment_(computer_programming)

Acknowledgments

Proof read by Dr. Thomas McConkey

"Please STRONGLY emphasize this lecture. I see so much code from grad students which has utterly useless commenting."

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.